Oracle Java EE 7文档团队成员精准解说Java EE 7平台，为深入理解Java EE 7企业应用开发提供翔实指导

更简洁明确的概念介绍，更具体完整的应用示例，全面解读Java EE 7平台特性，手把手教你运行和测试示例，帮助你真正掌握Java EE 技术，成为Java EE企业应用开发的高手

PEARSON

HZ BOOKS 华章IT

Java 核心技术系列

Java EE 7 权威指南 卷2

（原书第5版）

The Java EE 7 Tutorial: Volume 2

(Fifth Edition)

埃里克·珍兆科（Eric Jendrock）
里卡多·塞维拉-纳瓦罗（Ricardo Cervera-Navarro）
[美] 伊恩·埃文斯（Ian Evans） 著
金姆·哈泽（Kim Haase）
威廉·马基特（William Markito）

苏金国 江健 等译

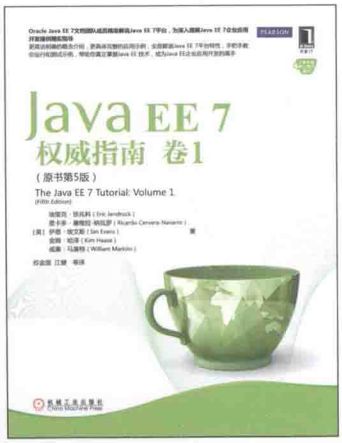
机械工业出版社
China Machine Press

Java EE 7
权威指南 卷2

（原书第5版）

本书是一个面向任务的实用指南，通过一系列实用示例介绍如何开发Java EE 7（Java平台企业版7）企业应用，由Oracle的Java EE文档小组成员倾力撰写，能够帮助初学者和中级Java程序员深入了解这个平台。

本书包括平台特性的全面描述，并提供使用最新版本NetBeans IDE和GlassFish服务器开源版的有关说明。书中介绍了EJB组件、Java持久化API、Java消息服务（JMS）API、Java EE安全、事务、资源适配器、Java EE拦截器、Java平台批处理应用以及Java EE并发工具。本书最后提供了3个案例研究，展示了如何综合使用多个Java EE 7 API。



Java EE 7权威指南：卷1（原书第5版）
书号：978-7-111-49760-8
定价：99.00元

上架指导：计算机/程序设计/Java
ISBN 978-7-111-49711-0
9 787111 497110 >
定价：99.00元

PEARSON
www.pearson.com

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

Java 核心技术系列

Java EE 7 权威指南 卷2

（原书第5版）

The Java EE 7 Tutorial: Volume 2
(Fifth Edition)

埃里克·珍兆科（Eric Jendrock）
里卡多·塞维拉-纳瓦罗（Ricardo Cervera-Navarro）
[美] 伊恩·埃文斯（Ian Evans） 著
金姆·哈泽（Kim Haase）
威廉·马基特（William Markito）

苏金国 江健 等译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Java EE 7 权威指南：卷 2（原书第 5 版）/（美）珍兆科（Jendrock, E.）等著；苏金国等译．—北京：机械工业出版社，2015.5
（Java 核心技术系列）
书名原文：The Java EE 7 Tutorial：Volume 2，Fifth Edition

ISBN 978-7-111-49711-0

I. J… II. ①珍… ②苏… III. Java 语言－程序设计－指南 IV. TP312-62

中国版本图书馆 CIP 数据核字（2015）第 061863 号

本书版权登记号：图字：01-2014-4434

Authorized translation from the English language edition, entitled *The Java EE 7 Tutorial: Volume 2, Fifth Edition*, 9780321980083 by Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Kim Haase, William Markito, published by Pearson Education, Inc., Copyright © 2014, Oracle and/or its affiliates.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2015.

本书中文简体字版由 Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括中国台湾地区和中国香港、澳门特别行政区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

Java EE 7 权威指南：卷 2（原书第 5 版）

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：关 敏　　责任校对：殷 虹
印　　刷：三河市宏图印务有限公司　　版　　次：2015 年 5 月第 1 版第 1 次印刷
开　　本：186mm×240mm 1/16　　印　　张：30.5
书　　号：ISBN 978-7-111-49711-0　　定　　价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066　　投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259　　读者信箱：hzjsj@hzbook.com

版权所有・侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

华章IT

HZBOOKS | Information Technology

The Translator's Words 译 者 序

Java 平台素以提倡开放为宗旨，基于这种开放性，在全行业共同推动之下，再加上 Java 社区和 GlassFish 社区的大力支持，Oracle 公司终于在 2013 年推出了 Java EE 7，而这距离 2008 年发布 Java EE 6 已经过去了 5 年之久。在这 5 年里，企业应用有了哪些发展？Java EE 又有了哪些变化？新增了哪些特性？对 Java 企业应用提供了哪些新的支持？在哪些领域有了新的探索？本书[⊖]可以很好地回答我们的这些疑问。

实际上，伴随着 Java EE 的发展脚步，从 2001 年《The J2EE Tutorial》第 1 版问世，到现在本书与大家见面，也经过了十余年的演进历程，其间 2004 年改版为《The J2EE™ Tutorial》第 2 版，2006 年出版了《Java EE 5 Tutorial》第 3 版，从《Java EE 6 Tutorial》第 4 版开始分为两册，分别介绍基本概念和高级主题，不过基本概念分册于 2010 年出版，而高级主题分册到 2013 年才姗姗到来，好在本书没有让我们久等，2014 年第 1 卷和第 2 卷同时出版，你手上的就是其中的第 2 卷。

在第 2 卷中，介绍了 EJB、Java 持久化、Java 消息服务、安全性等高级技术，还提供了 3 个综合使用多项 Java EE 7 技术的案例研究。在这本书中，你不仅能看到简洁明确的概念介绍，还会看到具体完整的实用示例，书中甚至会手把手地教你如何运行和测试这些示例，使你能真正掌握 Java EE 技术，成为 Java EE 企业应用开发的高手。

本书由苏金国、江健主译，程芳、吴忠望、张练达、陈峰、杨健康、姚勇、张莹、牛亚峰、高强、赵蓓、荆涛、刘卫宏参与了全书的修改整理，朱涛江、杜志秀、杨春生、刘亮、王宇、张伶、王志淋、刘跃邦完善了关键部分的翻译。不过由于水平有限，译文肯定有不当之处，敬请批评指正。

译者

⊖ 本书英文版和中文版都分两卷出版，你手上的这一本是卷 2。——编辑注

前　言 *Preface*

本书针对的是 Java 平台企业版 7（Java Platform，Enterprise Edition 7），即 Java EE 7，是开发企业应用的实用指南，全书将使用 GlassFish 服务器开源版（GlassFish Server Open Source Edition）来介绍。

GlassFish 服务器开源版是目前业界领先的开源项目和开放社区平台，可用来构建和部署下一代应用和服务。GlassFish 服务器开源版由 GlassFish 项目开源社区研发（https://glassfish.java.net/），是 Java EE 7 平台规范的第一个兼容实现。这个应用服务器具有轻量级、灵活和开源等特点，使得企业不仅可以充分利用 Java EE 7 规范引入的诸多新功能，还能通过更快捷、更优化的开发和部署流程扩展其已有功能。本书后面将把 GlassFish 服务器开源版简称为 GlassFish 服务器。

本书的读者对象

本书面向有兴趣开发和部署 Java EE 7 应用的程序员，讨论构成 Java EE 平台的有关技术，另外还会介绍如何开发 Java EE 组件并部署到 Java EE 软件开发包（Software Development Kit，SDK）。

阅读须知

学习本书之前，应当对 Java 编程语言有充分的了解并熟悉本书第 1 卷，为此，建议大家仔细通读《The Java Tutorials》(http://docs.oracle.com/javase/tutorial/)。

相关文档

本书第 1 卷介绍了本书未涵盖的 Java EE 7 技术，包括 JSF、Java Servlets、bean 验证、Java EE 上下文和依赖注入以及 Web 服务。

GlassFish 服务器文档集介绍了部署计划和系统安装的有关内容。GlassFish 服务器开源版

的相关文档可以从 https://glassfish.java.net/docs/ 得到。

可以在 http://docs.oracle.com/javaee/7/api/ 查阅 Java EE 7 API 规范，另外 Java EE 7 SDK 也提供了这个规范。

此外，Java EE 规范也很有用，参见 http://www.oracle.com/technetwork/java/javaee/tech/index.html。

要在 NetBeans 集成开发环境（Integrated Development Environment，IDE）创建企业应用，有关信息参见 https://netbeans.org/kb/。

要了解 GlassFish 服务器使用的 Java DB 数据库，有关信息请访问 http://www.oracle.com/technetwork/java/javadb/overview/。

GlassFish Samples 项目提供了一组示例应用，展示了诸多 Java EE 技术。GlassFish Samples 与 Java EE 软件开发包（SDK）捆绑发行，也可以从 GlassFish Samples 项目网页（https://glassfish-samples.java.net/）获得。

Oracle 无障碍计划

关于 Oracle 无障碍承诺的有关信息请访问 Oracle 无障碍计划网站（http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc）。

本书约定

下表描述了本书使用的排版约定。

约　定	含　义	示　例
粗体	粗体表示定义的术语	**缓存**（cache）是本地存储的一个副本
等宽字体	等宽字体指示文件和目录名、段落中的命令、URL、示例中的代码、屏幕上出现的文本或者输入的文本	编辑 `.login` 文件。 使用 `ls-a` 列出所有文件 `machine_name% you have mail.`
斜体	斜体表示占位变量（要为这些占位变量提供具体的值）	删除文件的命令是 `rm` *filename*

默认路径和文件名

下表描述了本书中使用的默认路径和文件名。

占位变量	描　述	默认值
as-install	表示 GlassFish 服务器或 SDK（包含 GlassFish 服务器）的安装主目录	Solaris 操作系统、Linux 操作系统和 Mac 操作系统上的安装目录为： *user's-home-directory*`/glassfish4/glassfish` Windows 上的安装目录为： *SystemDrive*`:\glassfish4\glassfish`

（续）

占位变量	描　述	默认值
as-install-parent	表示 GlassFish 服务器安装主目录的父目录	Solaris 操作系统、Linux 操作系统和 Mac 操作系统上的父目录为： *user's-home-directory*/glassfish4 Windows 上的父目录为：*SystemDrive*:\glassfish4
tut-install	表示安装 GlassFish 服务器或 SDK 并运行更新工具（Update Tool）之后 *Java EE Tutorial* 的安装主目录	*as-install-parent*/docs/javaee-tutorial
domain-dir	表示存储一个域的配置文件的目录	*as-install*/domains/domain1

致谢

首先要感谢 Java EE 规范的各位领导者：Linda DeMichiel、Bill Shannon、Emmanuel Bernard、Ed Burns、Shing Wai Chan、Kin-Man Chung、Danny Coward、Nigel Deakin、Rod Johnson、Roger Kitain、Jitendra Kotamraju、Anthony Lai、Bob Lee、Ron Monzillo、Rajiv Mordani、Pete Muir、Paul Parkinson、Santiago Pericas-Geertsen、Marek Potociar、Sivakumar Thyagarajan、Marina Vatkina 和 Chris Vignola。

另外还要感谢 Java EE 7 SDK 团队，特别是 Snjezana Sevo-Zenzerovic、Adam Leftik、Michael Chen 和 John Clingan。

Manfred Riem 和 JavaServer Faces 规范领导者提出的建议对于有关 JSF 技术的各章有很大帮助。

要特别感谢我们的经理 Alan Sommerer，感谢他一直以来的支持和影响。

还要感谢 Jordan Douglas 和 Dawn Tyler 制作并更新了本书的插图。感谢 Edna Elle 在工具方面提供了极大的帮助。感谢 Sheila Cepero 在很多方面帮助我们扫清障碍，使出版能够顺利进行。

最后，我们要向 Greg Doench、Elizabeth Ryan、Caroline Senay 和 Addison-Wesley 的制作团队致以最诚挚的谢意，感谢大家审阅我们的初稿，让本书得以顺利出版。

Contents 目　录

译者序
前　言

第一部分　引言

第 1 章　概述 …… 2
1.1　Java EE 7 平台新增特性 …… 3
1.2　Java EE 应用模型 …… 3
1.3　分布式多层应用 …… 4
1.3.1　安全 …… 4
1.3.2　Java EE 组件 …… 5
1.3.3　Java EE 客户端 …… 6
1.3.4　Web 组件 …… 7
1.3.5　业务组件 …… 8
1.3.6　企业信息系统层 …… 8
1.4　Java EE 容器 …… 9
1.4.1　容器服务 …… 9
1.4.2　容器类型 …… 10
1.5　Web 服务支持 …… 11
1.5.1　XML …… 11
1.5.2　SOAP 传输协议 …… 12
1.5.3　WSDL 标准格式 …… 12
1.6　Java EE 应用装配与部署 …… 12
1.7　开发角色 …… 13
1.7.1　Java EE 产品提供商 …… 13
1.7.2　工具提供商 …… 13
1.7.3　应用组件提供商 …… 13
1.7.4　应用装配人员 …… 14
1.7.5　应用部署人员和管理员 …… 14
1.8　Java EE 7 API …… 14
1.8.1　EJB 技术 …… 16
1.8.2　Java servlet 技术 …… 17
1.8.3　JSF 技术 …… 18
1.8.4　JSP 技术 …… 18
1.8.5　JSP 标准标记库 …… 19
1.8.6　Java 持久化 API …… 19
1.8.7　Java 事务 API …… 19
1.8.8　RESTful Web 服务 Java API …… 19
1.8.9　托管 bean …… 20
1.8.10　Java EE 的上下文和依赖注入 …… 20
1.8.11　Java 的依赖注入 …… 20
1.8.12　bean 验证 …… 20
1.8.13　Java 消息服务 API …… 20

1.8.14 Java EE 连接器架构 …… 21
1.8.15 JavaMail API …… 21
1.8.16 Java 容器授权契约 …… 21
1.8.17 Java 容器认证服务提供者接口 … 21
1.8.18 WebSocket Java API …… 22
1.8.19 JSON 处理 Java API …… 22
1.8.20 Java EE 的并发工具 …… 22
1.8.21 Java 平台的批处理应用 …… 22
1.9 Java 平台标准版 7 的 Java EE 7 API …… 23
1.9.1 Java 数据库互连 API …… 23
1.9.2 Java 命名和目录接口 API …… 23
1.9.3 JavaBeans 激活框架 …… 24
1.9.4 XML 处理 Java API …… 24
1.9.5 Java XML 绑定架构 …… 24
1.9.6 XML Web 服务 Java-API …… 24
1.9.7 带附件 SOAP Java API …… 25
1.9.8 Java 认证和授权服务 …… 25
1.9.9 Java 平台的公共注解 …… 25
1.10 GlassFish 服务器工具 …… 25

第 2 章 使用教程示例 …… 27

2.1 必要软件 …… 27
2.1.1 Java EE 7 软件开发包 …… 28
2.1.2 Java 平台标准版 …… 28
2.1.3 Java EE 7 教程组件 …… 28
2.1.4 NetBeans IDE …… 29
2.1.5 Apache Maven …… 30
2.2 启动和停止 GlassFish 服务器 …… 30
2.2.1 使用 NetBeans IDE 启动 GlassFish 服务器 …… 30
2.2.2 使用 NetBeans IDE 停止 GlassFish 服务器 …… 30
2.2.3 使用命令行启动 GlassFish 服务器 …… 30
2.2.4 使用命令行停止 GlassFish 服务器 …… 31
2.3 启动管理控制台 …… 31
2.3.1 使用 NetBeans IDE 启动管理控制台 …… 31
2.4 启动和停止 Java DB 服务器 …… 32
2.4.1 使用 NetBeans IDE 启动数据库服务器 …… 32
2.5 构建示例 …… 32
2.6 教程示例目录结构 …… 32
2.7 教程中的 Java EE 7 Maven 原型 …… 33
2.7.1 安装教程 Maven 原型 …… 33
2.8 获取教程最新更新 …… 34
2.8.1 使用 NetBeans IDE 更新教程 … 34
2.8.2 使用命令行更新教程 …… 34
2.9 调试 Java EE 应用 …… 34
2.9.1 使用服务器日志 …… 34
2.9.2 使用调试器 …… 35

第二部分 企业 bean

第 3 章 企业 bean …… 38

3.1 什么是企业 bean …… 38
3.1.1 企业 bean 的好处 …… 39
3.1.2 何时使用企业 bean …… 39
3.1.3 企业 bean 类型 …… 39

3.2 什么是会话 bean ····· 39
3.2.1 会话 bean 类型 ····· 40
3.2.2 何时使用会话 bean ····· 41
3.3 什么是消息驱动 bean ····· 41
3.3.1 消息驱动 bean 与会话 bean 的区别 ····· 41
3.3.2 何时使用消息驱动 bean ····· 42
3.4 访问企业 bean ····· 42
3.4.1 在客户端使用企业 bean ····· 43
3.4.2 确定远程或本地访问 ····· 44
3.4.3 本地客户端 ····· 45
3.4.4 远程客户端 ····· 46
3.4.5 Web 服务客户端 ····· 47
3.4.6 方法参数和访问 ····· 47
3.5 企业 bean 的内容 ····· 48
3.6 企业 bean 的命名约定 ····· 48
3.7 企业 bean 的生命周期 ····· 48
3.7.1 有状态会话 bean 的生命周期 ··· 48
3.7.2 无状态会话 bean 的生命周期 ··· 49
3.7.3 单例会话 bean 的生命周期 ····· 50
3.7.4 消息驱动 bean 的生命周期 ····· 50
3.8 有关企业 bean 的更多信息 ····· 51

第 4 章 企业 bean 入门 ····· 52
4.1 创建企业 bean ····· 52
4.1.1 编写企业 bean 类 ····· 53
4.1.2 创建 converter Web 客户端 ····· 53
4.1.3 运行 converter 示例 ····· 54
4.2 修改 Java EE 应用 ····· 55
4.2.1 修改类文件 ····· 55

第 5 章 运行企业 bean 示例 ····· 56
5.1 cart 示例 ····· 56
5.1.1 业务接口 ····· 57
5.1.2 会话 bean 类 ····· 57
5.1.3 @Remove 方法 ····· 61
5.1.4 辅助类 ····· 61
5.1.5 运行 cart 示例 ····· 61
5.2 一个单例会话 bean 示例：counter ····· 62
5.2.1 创建一个单例会话 bean ····· 63
5.2.2 counter 示例的架构 ····· 67
5.2.3 运行 counter 示例 ····· 69
5.3 一个 Web 服务示例：helloservice ··· 69
5.3.1 Web 服务端点实现类 ····· 69
5.3.2 无状态会话 bean 实现类 ····· 70
5.3.3 运行 helloservice 示例 ····· 71
5.4 使用定时器服务 ····· 72
5.4.1 创建基于日历的定时器表达式 ··· 72
5.4.2 编程定时器 ····· 74
5.4.3 自动定时器 ····· 76
5.4.4 取消和保存定时器 ····· 76
5.4.5 获得定时器信息 ····· 77
5.4.6 事务和定时器 ····· 77
5.4.7 timersession 示例 ····· 77
5.4.8 运行 timersession 示例 ····· 80
5.5 处理异常 ····· 81

第 6 章 使用嵌入式企业 bean 容器 ··· 82
6.1 嵌入式企业 bean 容器概述 ····· 82
6.2 开发嵌入式企业 bean 应用 ····· 82
6.2.1 运行嵌入式应用 ····· 83

6.2.2 创建企业 bean 容器……83
6.2.3 查找会话 bean 引用……84
6.2.4 关闭企业 bean 容器……85
6.3 standalone 示例应用……85
6.3.1 使用 NetBeans IDE 运行 standalone 示例应用……86
6.3.2 使用 Maven 运行 standalone 示例应用……87

第 7 章 在会话 bean 中使用异步方法调用……88

7.1 异步方法调用……88
7.1.1 创建异步业务方法……88
7.1.2 从企业 bean 客户端调用异步方法……89
7.2 async 示例应用……90
7.2.1 async-war 模块的架构……91
7.2.2 运行 async 示例……92

第三部分 持久化

第 8 章 Java 持久化 API 介绍……96

8.1 实体……96
8.1.1 实体类的需求……97
8.1.2 实体类中的持久化字段和属性……97
8.1.3 实体的主键……101
8.1.4 实体关系中的多重性……103
8.1.5 实体关系中的方向……103
8.1.6 实体中的可嵌入类……105
8.2 实体继承……106
8.2.1 抽象实体……106
8.2.2 映射超类……106
8.2.3 非实体超类……107
8.2.4 实体继承映射策略……107
8.3 管理实体……109
8.3.1 EntityManager 接口……109
8.3.2 持久化单元……113
8.4 查询实体……114
8.5 数据库模式创建……114
8.5.1 配置应用以创建或删除数据库表……115
8.5.2 使用 SQL 脚本加载数据……116
8.6 有关持久化的更多信息……116

第 9 章 运行持久化示例……118

9.1 order 应用……118
9.1.1 order 应用中的实体关系……119
9.1.2 order 应用中的主键……121
9.1.3 实体映射到多个数据库表……125
9.1.4 order 应用中的层叠操作……125
9.1.5 order 应用中的 BLOB 和 CLOB 数据库类型……126
9.1.6 order 应用中的时态类型……126
9.1.7 管理 order 应用的实体……127
9.1.8 运行 order 示例……129
9.2 roster 应用……129
9.2.1 roster 应用中的关系……130
9.2.2 roster 应用中的实体继承……131
9.2.3 roster 应用中的 Criteria 查询……132
9.2.4 roster 应用中的自动表生成……134

9.2.5 运行 roster 示例 …… 135
9.3 address-book 应用 …… 136
9.3.1 address-book 中的 bean 验证约束 …… 136
9.3.2 在 address-book 中为约束指定错误消息 …… 137
9.3.3 验证 JSF 应用的联系信息输入 …… 138
9.3.4 运行 address-book 示例 …… 139

第 10 章 Java 持久化查询语言 …… 140

10.1 查询语言术语 …… 140
10.2 使用 Java 持久化查询语言创建查询 …… 141
10.2.1 查询中的命名参数 …… 142
10.2.2 查询中的位置参数 …… 142
10.3 简化的查询语言语法 …… 142
10.3.1 选择语句 …… 142
10.3.2 更新和删除语句 …… 143
10.4 示例查询 …… 143
10.4.1 简单查询 …… 143
10.4.2 导航到相关实体的查询 …… 144
10.4.3 包含其他条件表达式的查询 …… 146
10.4.4 批处理更新和删除 …… 147
10.5 完整的查询语言语法 …… 148
10.5.1 BNF 符号 …… 148
10.5.2 Java 持久化查询语言的 BNF 文法 …… 148
10.5.3 FROM 子句 …… 152
10.5.4 路径表达式 …… 157
10.5.5 WHERE 子句 …… 158
10.5.6 SELECT 子句 …… 165
10.5.7 ORDER BY 子句 …… 167
10.5.8 GROUP BY 和 HAVING 子句 …… 167

第 11 章 使用 Criteria API 创建查询 …… 168

11.1 Criteria 和 Metamodel API 概述 …… 168
11.2 使用 Metamodel API 为实体类建模 …… 170
11.2.1 使用元模型类 …… 170
11.3 使用 Criteria API 和 Metamodel API 创建基本的类型安全查询 …… 171
11.3.1 创建 Criteria 查询 …… 171
11.3.2 查询根 …… 172
11.3.3 使用联接查询关系 …… 172
11.3.4 Criteria 查询中的路径导航 …… 173
11.3.5 限制 Criteria 查询结果 …… 173
11.3.6 管理 Criteria 查询结果 …… 175
11.3.7 执行查询 …… 176

第 12 章 创建和使用基于字符串的 Criteria 查询 …… 178

12.1 基于字符串的 Criteria API 查询概述 …… 178
12.2 创建基于字符串的查询 …… 178
12.3 执行基于字符串的查询 …… 179

第 13 章 使用锁定控制对实体数据的并发访问 …… 180

13.1 实体锁定和并发概述 …… 180

13.1.1 使用乐观锁定 …… 181
13.2 锁模式 …… 181
13.2.1 设置锁模式 …… 182
13.2.2 使用悲观锁定 …… 183

第 14 章 用实体图创建获取计划 …… 185

14.1 实体图基础 …… 185
14.1.1 默认实体图 …… 186
14.1.2 在持久化操作中使用实体图 …… 186
14.2 使用命名实体图 …… 187
14.2.1 对实体类应用命名实体图注解 …… 187
14.2.2 从命名实体图获得 EntityGraph 实例 …… 188
14.3 在查询操作中使用实体图 …… 189

第 15 章 对 Java 持久化 API 应用使用二级缓存 …… 190

15.1 二级缓存概述 …… 190
15.1.1 控制实体是否可以缓存 …… 191
15.2 指定缓存模式设置来提高性能 …… 192
15.2.1 设置缓存获取和存储模式 …… 192
15.2.2 通过编程方式控制二级缓存 …… 194

第四部分 消息传送

第 16 章 Java 消息服务概念 …… 198

16.1 JMS API 概述 …… 198
16.1.1 什么是消息传送 …… 198
16.1.2 什么是 JMS API …… 199
16.1.3 何时使用 JMS API …… 199
16.1.4 Java EE 平台如何使用 JMS API …… 200
16.2 基本 JMS API 概念 …… 201
16.2.1 JMS API 架构 …… 201
16.2.2 消息传送方式 …… 201
16.2.3 消息使用 …… 203
16.3 JMS API 编程模型 …… 204
16.3.1 JMS 受管理对象 …… 205
16.3.2 连接 …… 206
16.3.3 会话 …… 206
16.3.4 JMSContext 对象 …… 207
16.3.5 JMS 消息生产者 …… 208
16.3.6 JMS 消息消费者 …… 208
16.3.7 JMS 消息 …… 213
16.3.8 JMS 队列浏览器 …… 215
16.3.9 JMS 异常处理 …… 216
16.4 使用高级 JMS 特性 …… 216
16.4.1 控制消息确认 …… 217
16.4.2 为发送消息指定选项 …… 218
16.4.3 创建临时目的地 …… 220
16.4.4 使用 JMS 本地事务 …… 221
16.4.5 异步发送消息 …… 223
16.5 在 Java EE 应用中使用 JMS API …… 223
16.5.1 为 Java EE 应用创建资源 …… 223
16.5.2 在企业 bean 或 Web 组件中使用资源注入 …… 225
16.5.3 使用 Java EE 组件生成和同步接收消息 …… 226

16.5.4 使用消息驱动 bean 异步接收消息 …… 227
16.5.5 管理 JTA 事务 …… 229
16.6 关于 JMS 的更多信息 …… 231

第 17 章 Java 消息服务示例 …… 232

17.1 JMS 示例概述 …… 233
17.2 编写简单的 JMS 应用 …… 233
17.2.1 启动 JMS 提供者 …… 234
17.2.2 创建 JMS 受管理对象 …… 234
17.2.3 构建所有简单示例 …… 235
17.2.4 发送消息 …… 235
17.2.5 同步接收消息 …… 238
17.2.6 使用消息监听器完成异步消息传送 …… 240
17.2.7 浏览队列中的消息 …… 242
17.2.8 在同一个目的地运行多个消费者 …… 245
17.2.9 确认消息 …… 245
17.3 编写更高级的 JMS 应用 …… 247
17.3.1 使用持久订阅 …… 247
17.3.2 使用本地事务 …… 249
17.4 编写高性能和可扩展的 JMS 应用 …… 254
17.4.1 使用共享非持久订阅 …… 254
17.4.2 使用共享持久订阅 …… 256
17.5 使用一个简单 Web 应用发送和接收消息 …… 257
17.5.1 websimplemessage Facelets 页面 …… 257
17.5.2 websimplemessage 托管 bean … 258
17.5.3 运行 websimplemessage 示例 …… 259
17.6 使用消息驱动 bean 异步接收消息 …… 261
17.6.1 simplemessage 示例概述 …… 261
17.6.2 simplemessage 应用客户端 … 261
17.6.3 simplemessage 消息驱动 bean 类 …… 262
17.6.4 运行 simplemessage 示例 …… 263
17.7 从会话 bean 向 MDB 发送消息 … 264
17.7.1 为 clientsessionmdb 示例编写应用组件 …… 265
17.7.2 运行 clientsessionmdb 示例 … 267
17.8 使用实体联接两个 MDB 的消息 …… 268
17.8.1 clientmdbentity 示例应用概述 …… 269
17.8.2 为 clientmdbentity 示例编写应用组件 …… 270
17.8.3 运行 clientmdbentity 示例 … 272
17.9 使用 NetBeans IDE 创建 JMS 资源 …… 274
17.9.1 使用 NetBeans IDE 创建 JMS 资源 …… 274
17.9.2 使用 NetBeans IDE 删除 JMS 资源 …… 275

第五部分 安全

第 18 章 Java EE 平台安全介绍 …… 278

18.1 Java EE 安全概述 …… 278

18.1.1 简单的应用安全演示……279
18.1.2 安全机制特性……281
18.1.3 应用安全特点……281
18.2 安全机制……282
18.2.1 Java SE 安全机制……282
18.2.2 Java EE 安全机制……283
18.3 保护容器安全……285
18.3.1 使用注解指定安全信息……285
18.3.2 使用部署描述文件提供声明式安全……285
18.3.3 使用编程式安全……286
18.4 保护 GlassFish 服务器安全……286
18.5 使用安全域、用户、组和角色…286
18.5.1 什么是安全域、用户、组和角色……287
18.5.2 在 GlassFish 服务器中管理用户和组……289
18.5.3 建立安全角色……290
18.5.4 角色映射到用户和组……291
18.6 使用 SSL 建立安全连接……292
18.6.1 验证和配置 SSL 支持……293
18.7 有关安全的更多信息……293

第 19 章 Web 应用安全入门……295

19.1 Web 应用安全概述……295
19.2 保护 Web 应用安全……296
19.2.1 指定安全约束……297
19.2.2 指定认证机制……300
19.2.3 在部署描述文件中指定认证机制……302
19.2.4 声明安全角色……303
19.3 Web 应用使用编程式安全……304
19.3.1 通过编程方式认证用户……304
19.3.2 通过编程方式检查调用者身份……306
19.3.3 编程式安全的示例代码……306
19.3.4 声明和链接角色引用……308
19.4 示例：保护 Web 应用安全……309
19.4.1 设置系统来运行安全示例…309
19.4.2 hello2-basicauth 示例：对 servlet 使用基本认证……310
19.4.3 hello1-formauth 示例：对 JSF 应用使用基于表单的认证…312

第 20 章 企业应用安全入门……316

20.1 企业应用的基本安全任务……316
20.2 保护企业 bean 安全……316
20.2.1 使用声明式安全保护企业 bean 安全……318
20.2.2 通过编程方式保护企业 bean 安全……321
20.2.3 传播安全身份（Run-As）…323
20.2.4 部署安全企业 bean……324
20.3 示例：保护企业 bean 安全……324
20.3.1 cart-secure 示例：使用声明式安全保护企业 bean 安全…324
20.3.2 converter-secure 示例：使用编程式安全保护企业 bean 安全……328

第 21 章 Java EE 安全：高级主题…331

21.1 使用数字证书……331

21.1.1 创建服务器证书……332
21.1.2 向证书安全域增加用户……334
21.1.3 为 GlassFish 服务器使用一个不同的服务器证书……334
21.2 认证机制……335
21.2.1 客户端认证……335
21.2.2 相互认证……335
21.3 使用 JDBC 安全域完成用户认证……338
21.3.1 配置 JDBC 认证安全域……339
21.4 保护 HTTP 资源安全……340
21.5 保护应用客户端安全……343
21.5.1 使用登录模块……343
21.5.2 使用编程式登录……344
21.6 保护企业信息系统应用安全……344
21.6.1 容器托管登录……344
21.6.2 组件托管登录……345
21.6.3 配置资源适配器安全……345
21.6.4 将应用主体映射到 EIS 主体……346
21.7 使用部署描述文件配置安全……347
21.7.1 在部署描述文件中指定基本认证安全……347
21.7.2 在部署描述文件中指定非默认主体 – 角色映射……348
21.8 关于高级安全主题的更多信息……348

第六部分 Java EE 支持技术

第 22 章 事务……352
22.1 Java EE 应用中的事务……352
22.2 什么是事务……353
22.3 容器托管事务……353
22.3.1 事务属性……354
22.3.2 回滚容器托管事务……357
22.3.3 同步会话 bean 的实例变量 357
22.3.4 容器托管事务中不允许的方法……358
22.4 bean 托管事务……358
22.4.1 JTA 事务……358
22.4.2 不提交返回……359
22.4.3 bean 托管事务中不允许的方法……359
22.5 事务超时……359
22.5.1 设置事务超时……359
22.6 更新多个数据库……360
22.7 Web 组件中的事务……361
22.8 关于事务的更多信息……361

第 23 章 资源适配器和契约……362
23.1 什么是资源适配器……362
23.1.1 管理契约……363
23.1.2 通用工作上下文契约……364
23.1.3 出站和入站契约……364
23.2 元数据注解……365
23.3 公共客户端接口……366
23.4 对 Java EE 上下文和依赖注入（CDI）使用资源适配器……367
23.5 关于资源适配器的更多信息……368

第 24 章 资源适配器示例……369
24.1 trading 示例……369

24.1.1 使用出站资源适配器……370
24.1.2 实现出站资源适配器……372
24.1.3 运行 trading 示例……373
24.2 traffic 示例……374
24.2.1 使用入站资源适配器……375
24.2.2 实现入站资源适配器……376
24.2.3 运行 traffic 示例……378

第 25 章 使用 Java EE 拦截器……380

25.1 拦截器概述……380
25.1.1 拦截器类……381
25.1.2 拦截器生命周期……381
25.1.3 拦截器和 CDI……381
25.2 使用拦截器……381
25.2.1 拦截方法调用……382
25.2.2 拦截生命周期回调事件……384
25.2.3 拦截超时事件……385
25.2.4 为组件绑定拦截器……386
25.2.5 拦截器排序……387
25.3 interceptor 示例应用……388
25.3.1 运行 interceptor 示例……389

第 26 章 批处理……390

26.1 批处理介绍……391
26.1.1 批处理作业中的步骤……391
26.1.2 并行处理……392
26.1.3 状态和判定元素……392
26.1.4 批处理框架功能……393
26.2 Java EE 中的批处理……394
26.2.1 批处理框架……394
26.2.2 创建批处理应用……394
26.2.3 批处理作业的元素……395
26.2.4 属性和参数……395
26.2.5 作业实例和作业执行……395
26.2.6 批处理和退出状态……395
26.3 简单用例……396
26.3.1 块步骤……397
26.3.2 任务步骤……399
26.4 使用作业规范语言……399
26.4.1 job 元素……400
26.4.2 step 元素……401
26.4.3 flow 元素……406
26.4.4 split 元素……406
26.4.5 decision 元素……406
26.5 创建批处理工件……407
26.5.1 批处理工件接口……407
26.5.2 批处理工件中的依赖注入……409
26.5.3 从批处理运行时环境使用上下文对象……410
26.6 向批处理运行时环境提交作业……411
26.6.1 开始作业……411
26.6.2 检查作业的状态……411
26.6.3 在应用中调用批处理运行时环境……412
26.7 打包批处理应用……412
26.8 webserverlog 示例应用……412
26.8.1 webserverlog 示例应用架构……412
26.8.2 运行 webserverlog 示例应用……418
26.9 phonebilling 示例应用……419
26.9.1 phonebilling 示例应用架构……419
26.9.2 运行 phonebilling 示例应用……425
26.10 关于批处理的更多信息……426

第 27 章　Java EE 的并发工具 …… 427

27.1　并发基础 …… 427

27.1.1　线程和进程 …… 428

27.2　并发工具的主要组件 …… 428

27.3　并发和事务 …… 429

27.4　并发和安全 …… 430

27.5　jobs 并发示例 …… 430

27.5.1　运行 jobs 示例 …… 430

27.6　taskcreator 并发示例 …… 433

27.6.1　运行 taskcreator 示例 …… 435

27.7　关于并发工具的更多信息 …… 436

第七部分　案例研究

第 28 章　Duke 书店案例研究示例 … 438

28.1　Duke 书店的设计和架构 …… 438

28.2　Duke 书店接口 …… 439

28.2.1　Book Java 持久化 API 实体 … 439

28.2.2　Duke 书店中使用的企业 bean …… 440

28.2.3　Duke 书店中使用的 Facelets 页面和托管 bean …… 440

28.2.4　Duke 书店中使用的定制组件和其他定制对象 …… 441

28.2.5　Duke 书店中使用的属性文件 …… 442

28.2.6　Duke 书店中使用的部署描述文件 …… 443

28.3　运行 Duke 书店案例研究应用 …… 443

28.3.1　使用 NetBeans IDE 构建和部署 Duke 书店 …… 443

28.3.2　使用 Maven 构建和部署 Duke 书店 …… 443

28.3.3　运行 Duke 书店应用 …… 444

第 29 章　Duke 辅导案例研究示例 … 445

29.1　Duke 辅导应用的设计和架构 …… 445

29.2　主界面 …… 447

29.2.1　主界面中使用的 Java 持久化 API 实体 …… 447

29.2.2　主界面中使用的企业 bean … 448

29.2.3　主界面中使用的 WebSocket 端点 …… 448

29.2.4　主界面中使用的 Facelets 文件 …… 448

29.2.5　主界面中使用的辅助类 …… 449

29.2.6　属性文件 …… 449

29.2.7　Duke 辅导应用中使用的部署描述文件 …… 450

29.3　管理界面 …… 450

29.3.1　管理界面中使用的企业 bean …… 450

29.3.2　管理界面中使用的 Facelets 文件 …… 451

29.3.3　管理界面中使用的 CDI 托管 bean …… 451

29.3.4　管理界面中使用的辅助类 … 451

29.4　运行 Duke 辅导案例研究应用 … 452

29.4.1　运行 Duke 辅导应用 …… 452

第 30 章　Duke 综合案例研究示例 ··· 455

30.1　Duke 综合应用的设计和架构 ···· 456

30.1.1　events 工程 ····· 458

30.1.2　entities 工程 ····· 459

30.1.3　dukes-payment 工程 ····· 461

30.1.4　dukes-resources 工程 ····· 461

30.1.5　Duke 商店工程 ····· 461

30.1.6　Duke 货运工程 ····· 465

30.2　构建和部署 Duke 综合案例研究应用 ····· 467

30.2.1　使用 NetBeans IDE 构建和部署 Duke 综合应用 ····· 467

30.2.2　使用 Maven 构建和部署 Duke 综合应用 ····· 467

30.3　运行 Duke 综合应用 ····· 467

30.3.1　注册为 Duke 商店顾客 ····· 468

30.3.2　购买产品 ····· 468

30.3.3　批准产品发货 ····· 468

30.3.4　创建新产品 ····· 469

以电商为基础的网络营销在线视频软件,提供淘宝,电商,微商,网络运营,搜索优化等精品视频,每月2次稳定更新,官方网址 WWW.527XX.COM

以网络安全为基础的IT类资源软件,提供编程,逆向,白帽脚本,网络运营等高品质商业教程,提供电子书及各类IT类期刊杂志,每月1次更新,官方网址 WWW.176KU.COM

第一部分 *Part 1*

引　言

第一部分将介绍平台、教程和示例，包括以下各章：

- 第 1 章　概述
- 第 2 章　使用教程示例

Chapter 1 第 1 章

概　　述

本章将介绍 Java EE 企业应用开发。我们会在这里回顾开发的基础知识，学习 Java EE 体系结构和 API，熟悉一些重要的术语和概念，了解如何完成 Java EE 应用编程、装配和部署。

如今，开发人员越发认识到市场对分布性、事务性和可移植性应用的需求，这些应用能充分发挥服务器端技术在速度、安全性和可靠性方面的优势。企业应用为企业提供业务逻辑。这些应用要集中管理，而且通常要与其他企业软件交互。在信息技术时代，必须本着投资少、速度快、省资源的原则来设计、构建和开发企业应用。

随着 Java 平台企业版（Java Platform，Enterprise Edition，Java EE）的出现，Java 企业应用的开发变得前所未有地简单和快捷。Java EE 平台的目标是为开发人员提供一组强大的 API，同时缩短开发周期，降低应用复杂性，并改善应用的性能。

Java EE 平台通过 Java 社区进程（Java Community Process，JCP）来开发，所有 Java 技术都由 JCP 负责。由相关各方组成的专家组编制 Java 规范请求（Java Specification Request，JSR），定义各项 Java EE 技术。在 JCP 计划指导下，Java 社区的工作有助于确保 Java 技术标准的稳定性和跨平台兼容性。

Java EE 平台使用了简化的编程模型。XML 部署描述文件变得不再必不可少。实际上，开发人员可以把信息作为**注解**（annotation）直接输入到 Java 源文件中，而且 Java EE 服务器可以在部署和运行时配置组件。通常可以在程序数据中嵌入注解，而原先必须有单独的部署描述文件放置这些信息。利用注解，规范信息可以直接放在代码中，与受这些规范影响的程序元素放在一起。

在 Java EE 平台中，可以对组件需要的所有资源应用依赖注入（dependency injection），这样应用代码中就无需考虑资源的创建和查找。可以在 EJB 容器（Enterprise JavaBeans

Container)、Web 容器和应用客户端中使用依赖注入。另外，依赖注入允许 Java EE 容器使用注解自动插入其他所需组件或资源的引用。

本书将通过示例介绍开发企业应用时可用的 Java EE 平台特性。无论是新手还是经验丰富的企业应用开发人员，都会发现这些示例和讲解将构成一个丰富的知识库，对于创建你自己的解决方案非常有帮助。

本章将介绍以下内容：

- Java EE 7 平台新增特性
- Java EE 应用模型
- 分布式多层应用
- Java EE 容器
- Web 服务支持
- Java EE 应用装配与部署
- Java EE 7 API
- Java 平台标准版 7 的 Java EE 7 API
- GlassFish 服务器工具

1.1 Java EE 7 平台新增特性

Java EE 7 平台最重要的目标是简化开发过程，为 Java EE 平台中不同类型的组件提供一个通用基础。由于 Java EE 7 平台更多地使用了注解而减少了 XML 配置，另外更多地使用 POJO（Plain Old Java Object，简单 Java 对象），并采用简化的应用打包方式，使得生产效率得以提升，因此开发人员将从中受益。Java EE 7 平台包括以下新特性。

- 新技术，包括：
  - 面向 Java EE 平台的批处理应用（参见 1.8.21 节）
  - 面向 Java EE 的并发工具（参见 1.8.20 节）
  - 用于 JSON 处理（JSON-P）的 Java API（参见 1.8.19 节）
  - WebSocket Java API（参见 1.8.18 节）
- EJB 组件新特性（详细内容参见 1.8.1 节）
- servlet 新特性（详细内容参见 1.8.2 节）
- JavaServer Faces 组件新特性（详细内容参见 1.8.3 节）
- Java 消息服务（JMS）新特性（详细内容参见 1.8.13 节）

1.2 Java EE 应用模型

Java EE 应用模型以 Java 编程语言和 Java 虚拟机为前提，它们提供的可移植性、安全性

和开发效率已经得到广泛认可，这构成了应用模型的基础。Java EE 旨在帮助开发和部署企业级应用，为顾客、员工、供应商、合作伙伴以及对企业有要求或有贡献的其他各方面实现企业服务。这些应用本质上非常复杂，可能要访问各种不同来源的数据，并将应用分布在大量不同的客户端上。

为了更好地控制和管理这些应用，面向不同用户的业务逻辑置于模型的中间层。中间层是一个受企业 IT 部门严格控制的环境，通常在专用的服务器硬件上运行，有权访问企业的全部服务。

Java EE 应用模型定义了一种架构，可以把服务实现为多层应用，从而提供企业级应用所需的可扩展性、可访问性和可管理性。这个模型将实现多层服务的工作划分为以下几部分：

- 开发人员实现的业务和表现逻辑
- Java EE 平台提供的标准系统服务

开发人员可以依赖 Java EE 平台提供解决方案，来解决开发多层服务所面对的复杂的系统级问题。

1.3 分布式多层应用

Java EE 平台使用一个分布式多层应用模型来构建企业应用。应用逻辑根据功能划分为不同的组件，构成 Java EE 应用的应用组件安装在不同的主机上，这取决于应用组件在多层 Java EE 环境中属于哪一个层次。

图 1-1 显示了两个多层 Java EE 应用，它们划分为以下所列的多个层次。图 1-1 中所示的 Java EE 应用部分将在 1.3.2 节中详细介绍。

- 客户端层组件运行在客户端主机上。
- Web 层组件运行在 Java EE 服务器上。
- 业务层组件运行在 Java EE 服务器上。
- 企业信息系统（Enterprise Information System，EIS）层软件运行在 EIS 服务器上。

尽管 Java EE 应用可以包括图 1-1 所示的所有层次，不过通常认为 Java EE 多层应用是 3 层应用，这是因为它们分布在 3 个位置上：客户端主机、Java EE 服务器主机以及后端的数据库或遗留主机。以这种方式运行的 3 层应用扩展了标准的两层客户端 – 服务器模型，它在客户端应用与后端存储之间增加了一个多线程应用服务器。

1.3.1 安全

其他企业应用模型要求各个应用中要考虑平台特定的安全措施，而基于 Java EE 的安全环境则允许在部署时设定安全约束。利用 Java EE 平台，开发人员可以避免实现安全特性的复杂性，这就使得基于 Java EE 平台的应用可以移植到各种不同的安全实现。

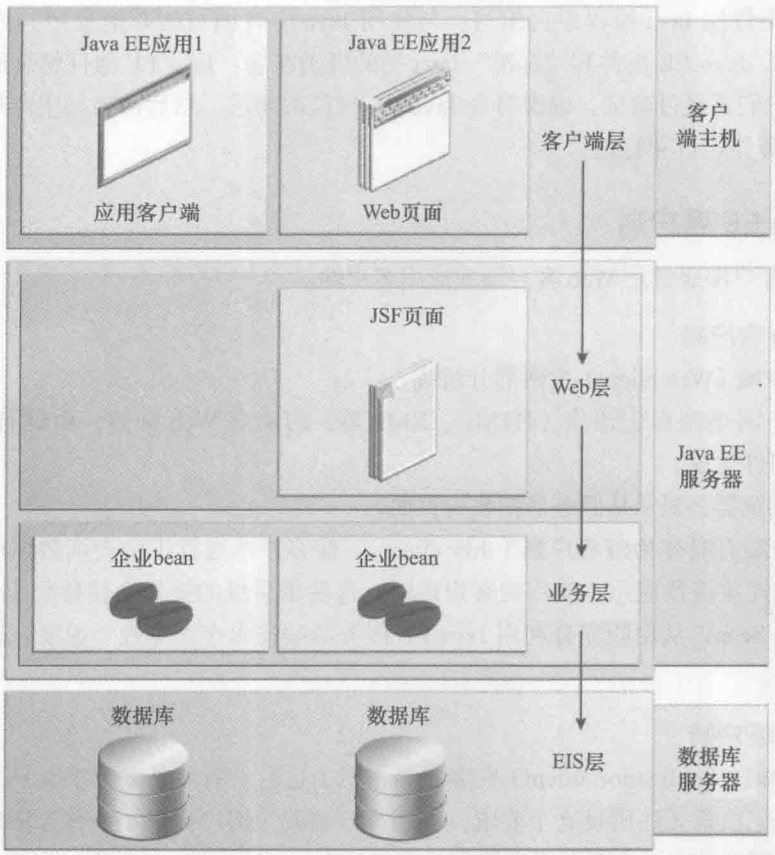


图 1-1 多层应用

Java EE 平台提供了标准的声明式访问控制规则，这些规则由开发人员定义，应用部署到服务器时会解释这些规则。Java EE 还提供了标准登录机制，使应用开发人员无需在自己的应用中实现这些机制。同样的应用可以在不同的安全环境中正常工作，而无需修改源代码。

1.3.2 Java EE 组件

Java EE 应用由组件构成。**Java EE 组件**（Java EE component）是可以装配到 Java EE 应用中并具备完整功能的软件单元，包括相关的类和文件，Java EE 组件可以与其他组件通信。

Java EE 规范定义了以下 Java EE 组件：

- 应用客户端和 applet 是运行在客户端的组件。
- Java servlet、JavaServer Faces 和 JavaServer Pages（JSP）技术组件是运行在服务器上的 Web 组件。
- EJB 组件（企业 bean）是运行在服务器上的业务组件。

Java EE 组件用 Java 编程语言编写，与使用 Java 语言编写的其他程序一样，也采用类似的方式编译。Java EE 组件和“标准”Java 类的区别在于：Java EE 组件将装配到一个 Java EE 应用中，它们要经过验证，确保符合 Java EE 规范的要求，然后部署到生产环境中，由一个 Java EE 服务器运行并管理。

1.3.3 Java EE 客户端

Java EE 客户端通常是 Web 客户端或应用客户端。

1.3.3.1 Web 客户端

Web 客户端（Web client）由两部分组成：

- 包含不同类型标记语言（HTML、XML 等）的动态 Web 页面，由运行在 Web 层的 Web 组件生成。
- Web 浏览器，显示从服务器接收的页面。

Web 客户端有时称为**瘦客户端**（thin client）。瘦客户端通常不会查询数据库、执行复杂的业务规则或连接遗留应用。使用瘦客户端时，这些重量级的操作会转移到 Java EE 服务器上执行的企业 bean，从而能充分利用 Java EE 服务器端技术在安全性、速度、服务和可靠性等方面的优势。

1.3.3.2 应用客户端

应用客户端（application client）在客户端主机上运行。有些任务要求有一个更丰富的用户界面，而标记语言无法满足这个要求，应用客户端就为用户提供了一种方法来处理这些任务。应用客户端通常有一个图形用户界面（graphical user interface，GUI），由 Swing API 或抽象窗口工具包（Abstract Window Toolkit，AWT）API 创建，不过，当然也可以有一个命令行界面。

应用客户端直接访问运行在业务层上的企业 bean。不过，如果确实有应用需求，应用客户端可以打开一个 HTTP 连接，与运行在 Web 层上的 servlet 建立通信。用非 Java 的其他语言编写的应用客户端可以与 Java EE 服务器交互，使得 Java EE 平台具备与遗留系统、客户端以及非 Java 语言互操作的能力。

1.3.3.3 applet

从 Web 层接收的 Web 页面可以包含嵌入的 applet。applet 是用 Java 编程语言编写的一个小客户端应用，在 Web 浏览器上安装的 Java 虚拟机中执行。不过，为了让 applet 能够在 Web 浏览器中成功执行，客户端系统可能需要 Java 插件，还可能需要一个安全策略文件。

Web 组件是创建 Web 客户端程序的首选 API，因为客户端系统上无需插件或安全策略文件。另外，由于 Web 组件提供了一种方法可以将应用程序开发与网页设计分开，因此 Web 组件支持更简洁、更模块化的应用设计。所以，Web 网页设计人员无需理解 Java 编程语言的语法也能很好地完成他们的工作。

1.3.3.4 JavaBeans 组件架构

服务器和客户端层也可以包含基于 JavaBeans 组件架构的组件（JavaBeans 组件），用来管理以下元素之间的数据流：

- 应用客户端或 applet 与 Java EE 服务器上运行的组件之间的数据流。
- 服务器组件与数据库之间的数据流。

根据 Java EE 规范，并不认为 JavaBeans 组件是 Java EE 组件。

JavaBeans 组件有属性，另外有访问这些属性的 `get` 和 `set` 方法。以这种方式使用的 JavaBeans 组件在设计和实现上通常都很简单，不过应当符合 JavaBeans 组件架构中指定的命名和设计约定。

1.3.3.5 Java EE 服务器通信

图 1-2 显示了构成客户端层的各个元素。客户端可以直接与运行在 Java EE 服务器上的业务层通信，如果客户端在浏览器中运行，还可以通过 Web 层中运行的 Web 页面或 servlet 进行通信。

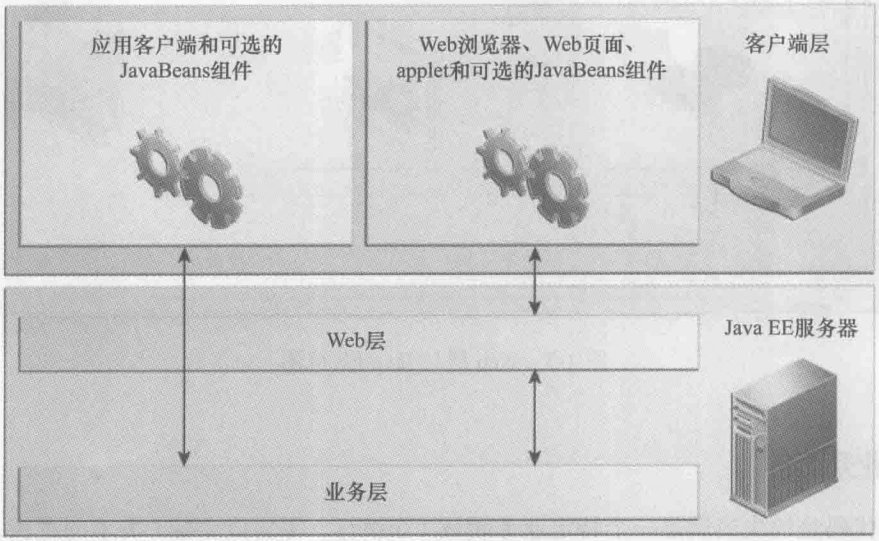


图 1-2 服务器通信

1.3.4 Web 组件

Java EE Web 组件可能是 servlet，或者是由 JSF 技术和（或）JSP 技术创建的 Web 页面（JSP 页面）。**servlet** 是 Java 编程语言编写的类，会动态地处理请求和构造响应。**JSP 页面**（JSP page）是基于文本的文档，可以作为 servlet 执行，不过允许采用一种更自然的方式创建静态内容。**JSF 技术**（JavaServer Faces technology）建立在 servlet 和 JSP 技术之上，为 Web 应用提供一个用户界面组件框架。

装配应用时，静态 HTML 页面及 applet 将与 Web 组件打包在一起，不过根据 Java EE 规范，并不认为它们是 Web 组件。类似于 HTML 页面，服务器端的工具类也可以与 Web 组件打包在一起，同样也不认为它们是 Web 组件。

如图 1-3 所示，与客户端层类似，Web 层也可以包含一个 JavaBeans 组件来管理用户输入，并将该输入发送到业务层上运行的企业 bean 进行处理。

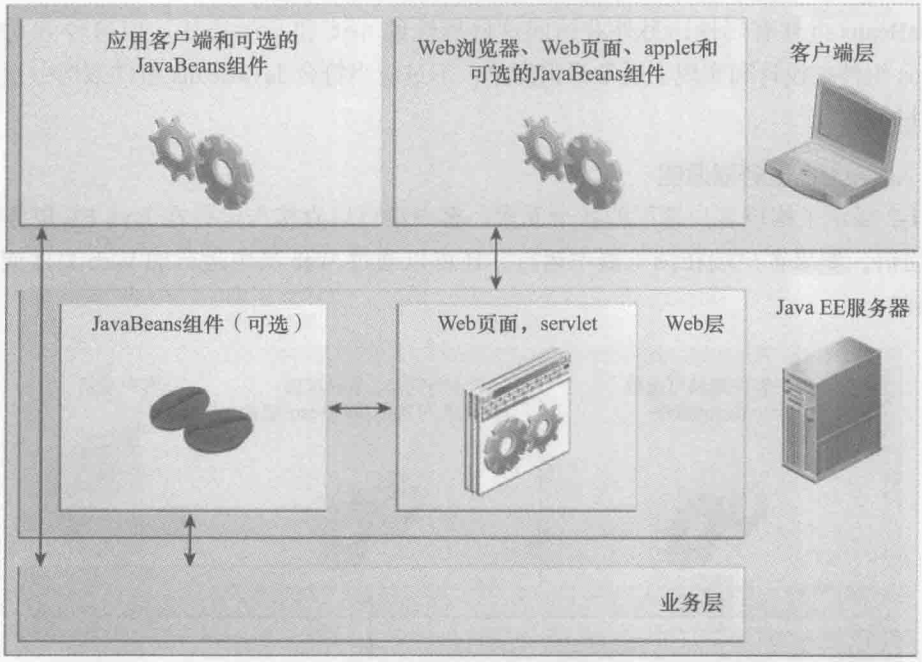


图 1-3 Web 层与 Java EE 应用

1.3.5 业务组件

业务代码是解决或满足一个特定业务领域（如银行、零售或金融）需求的逻辑，由业务层或 Web 层上运行的企业 bean 处理。图 1-4 展示了企业 bean 如何接收来自客户端程序的数据，进行处理（如果有必要），并将其发送到企业信息系统层进行存储。企业 bean 还会从数据存储中获取数据，进行处理（如果需要），再将其发送回客户端程序。

1.3.6 企业信息系统层

企业信息系统层处理企业信息软件（EIS），它包括企业基础设施系统，如企业资源计划（Enterprise Resource Planning，ERP）、主机事务处理、数据库系统以及其他遗留信息系统。例如，Java EE 应用组件可能需要访问企业信息系统来实现数据库连接。

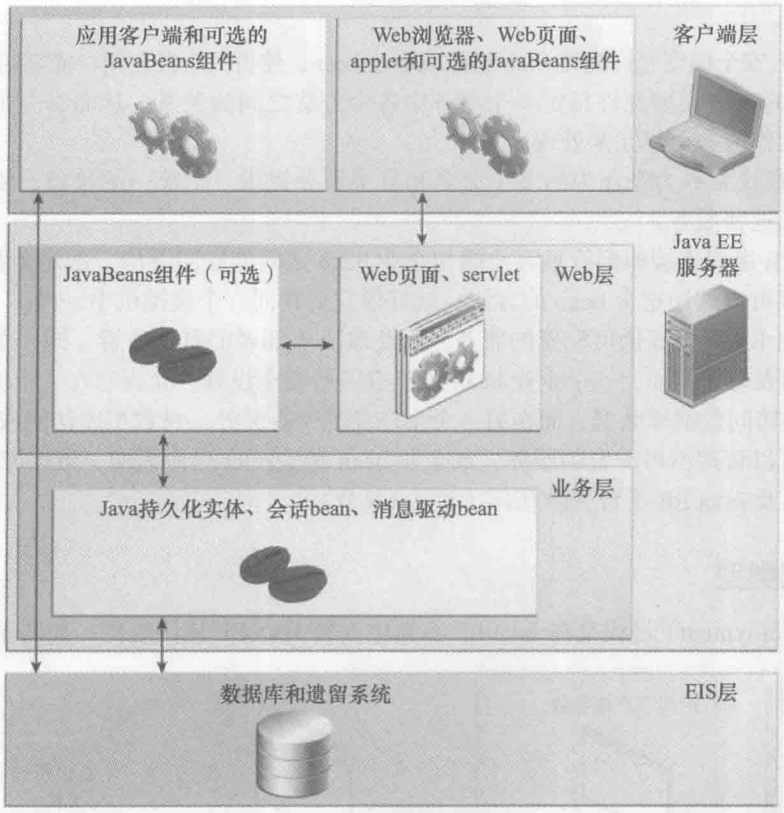


图 1-4 业务层和 EIS 层

1.4 Java EE 容器

通常情况下，瘦客户端多层应用程序很难编写，因为这需要大量复杂的代码来处理事务和状态管理、多线程、资源池以及其他复杂的底层细节。不过，利用基于组件而且平台独立的 Java EE 架构，编写应用会更为容易，因为业务逻辑可以组织为可重用的组件。另外，Java EE 服务器以容器形式为每一个组件类型提供了底层服务。由于你不需要自行开发这些服务，所以可以集中精力解决具体的业务问题。

1.4.1 容器服务

容器（Container）是组件与支持该组件的底层平台特定功能之间的接口。Web 企业 bean 或应用客户端组件执行之前，必须装配到一个 Java EE 模块并部署到其容器中。

装配过程中，需要为 Java EE 应用中的各个组件以及 Java EE 应用本身指定容器设置。容器设置会定制 Java EE 服务器提供的底层支持，包括安全、事务管理、Java 命名与目录接口（Java Naming and Directory Interface，JNDI）API 查找和远程连接等服务。以下是其中比

较重要的几项。

- Java EE 安全模型允许配置 Web 组件或企业 bean，使得只有授权用户能够访问系统资源。
- Java EE 事务模型允许指定一个事务中各个方法之间的关系，从而将一个事务中的所有方法作为一个单元来处理。
- JNDI 查找服务为企业中的多个命名和目录服务提供一个统一的接口，使应用组件可以访问这些服务。
- Java EE 远程连接模型管理客户端与企业 bean 之间的底层通信。创建企业 bean 之后，客户端可以调用这个 bean 的方法，就好像它们在同一个虚拟机中一样。

由于 Java EE 架构提供可配置的服务，因此取决于部署的具体位置，同一个应用中的组件会有不同的表现。例如，一个企业 bean 可能有一些安全设置，允许它在一个生产环境中以某种访问权限访问数据库数据，而在另一个生产环境中有另外一种数据库访问权限。

容器还可以管理不可配置的服务，如企业 bean 和 servlet 生命周期、数据库连接资源池、数据持久化以及 Java EE 平台 API 访问（参见 1.8 节）。

1.4.2 容器类型

部署（deployment）过程是在 Java EE 容器中安装 Java EE 应用组件，如图 1-5 所示。

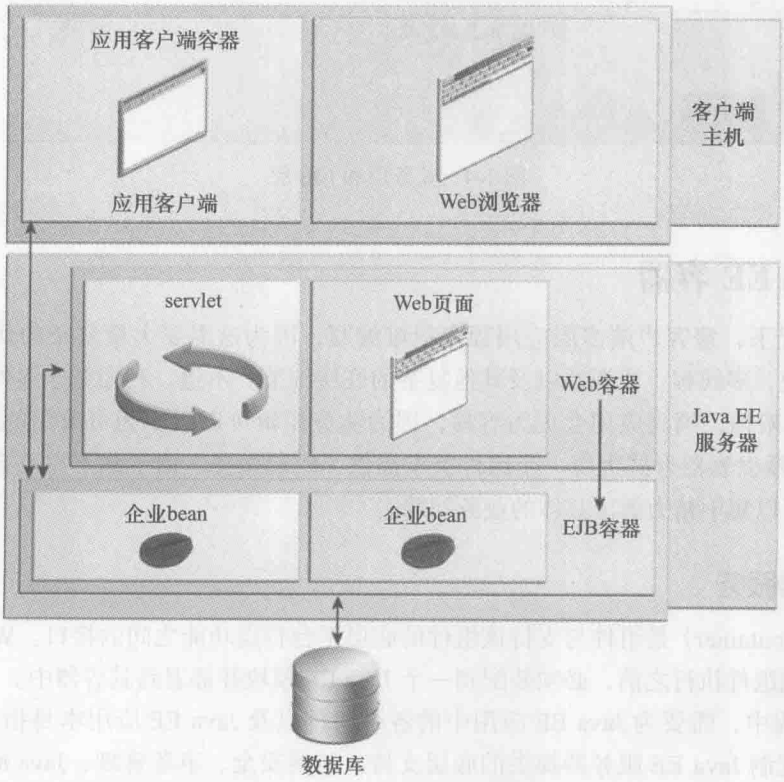


图 1-5 Java EE 服务器和容器

服务器和容器包括：

- **Java EE 服务器**：这是 Java EE 产品的运行时部分。Java EE 服务器提供了 EJB 容器和 Web 容器。
- **EJB 容器**：管理 Java EE 应用中企业 bean 的执行。企业 bean 及其容器在 Java EE 服务器上运行。
- **Web 容器**：管理 Java EE 应用中 Web 页面、servlet 和一些 EJB 组件的执行。Web 组件及其容器在 Java EE 服务器上运行。
- **应用客户端容器**：管理应用客户端组件的执行。应用客户端及其容器在客户端上运行。
- **applet 容器**：管理 applet 的执行。applet 容器包括一个 Web 浏览器和一个 Java 插件，都在客户端上运行。

1.5 Web 服务支持

Web 服务是基于 Web 的企业应用，使用开放的、基于 XML 的标准和传输协议，与调用该服务的客户端进行数据交换。Java EE 平台提供了我们所需的 XML API 和工具，从而能快速设计、开发、测试和部署可与其他 Web 服务及客户端互操作的 Web 服务（客户端可能运行在 Java 平台或非 Java 平台上）。

要用 Java EE XML API 编写 Web 服务和客户端，开发人员只需为方法调用传递参数数据，并处理返回的数据。对于面向文档的 Web 服务，需要发送和接收包含服务数据的文档。这里不需要底层编程，因为 XML API 实现会完成应用数据与基于 XML 的数据流之间的转换，从而能够在基于 XML 的标准化传输协议之上传送基于 XML 的数据流。这些基于 XML 的标准和协议将在接下来的小节中介绍。

由于数据会转换为标准化的基于 XML 的数据流，这使得用 Java EE XML API 编写的 Web 服务和客户端有充分的互操作性。这并不意味着所传输的数据都必须包含 XML 标记，因为传输的数据可以是纯文本、XML 数据或者任何类型的二进制数据，如音频、视频、地图、程序文件、计算机辅助设计（computer-aided design，CAD）文档，等等。下一节将介绍 XML，并解释业务各方如何以一种有意义的方式使用 XML 标记和模式（schema）来交换数据。

1.5.1 XML

可扩展标记语言 XML（Extensible Markup Language）是一种跨平台、可扩展、基于文本的数据表示标准。交换 XML 数据的双方可以创建自己的标记（tag）来描述数据，建立模式来指定哪些标记可以用在某种特定的 XML 文档中，并使用 XML 样式表管理数据的显示和处理。

例如，一个 Web 服务可以使用 XML 和一个模式生成价格清单，收到这些价格清单和模式的公司可以使用自己的定制样式表，以最能满足其需要的方式处理数据。下面来看几个例子。

- 一家公司用一个程序处理 XML 报价信息，将 XML 转换为 HTML，以便在其内部网发布价格清单。
- 一家合作伙伴公司可以将这个 XML 报价信息输入一个工具，来创建市场营销介绍。
- 另一家公司可以将 XML 报价信息读入一个应用进行处理。

1.5.2 SOAP 传输协议

客户端请求和 Web 服务响应都作为简单对象访问协议（Simple Object Access Protocol，SOAP）消息基于 HTTP 协议传输，使客户端和 Web 服务有充分的互操作性，即使它们在不同的平台上运行或者位于因特网的不同位置，也可以很好地实现信息交换。HTTP 协议是我们熟悉的请求 – 响应标准，用于在因特网上传送数据；SOAP 是一个基于 XML 的协议，同样遵循 HTTP 请求 – 响应模型。

传输消息中的 SOAP 部分包括以下内容：

- 定义一个基于 XML 的信封，描述消息的内容并解释如何处理消息。
- 包括一些基于 XML 的编码规则，描述消息中应用定义的数据类型的实例。
- 定义一个基于 XML 的约定，表示对远程服务的请求以及所得到的响应。

1.5.3 WSDL 标准格式

Web 服务描述语言（Web Services Description Language，WSDL）是用来描述网络服务的标准化 XML 格式。这个描述包括服务名、服务位置以及与服务通信的方式。WSDL 服务描述可以在 Web 上发布。GlassFish 服务器提供了一个工具，可以为使用远程过程调用与客户端通信的 Web 服务生成 WSDL 规范。

1.6 Java EE 应用装配与部署

Java EE 应用可以打包到一个或多个标准单元中，从而能部署到与 Java EE 平台兼容的任何系统上。每个单元包含以下部分：

- 一个或多个功能组件，如企业 bean、Web 页面、servlet 或 applet。
- 一个可选的部署描述文件，用来描述其内容。

一旦建立了 Java EE 单元，就可以进行部署了。部署通常需要使用平台提供的部署工具来指定与位置有关的特定信息，如可以访问这个单元的本地用户列表以及本地数据库名。一旦部署在本地平台上，就可以运行这个应用了。

1.7 开发角色

基于可重用的模块，可以将应用开发和部署过程划分到不同的角色，使不同的人或公司可以完成这个过程中不同部分的工作。

前两个角色是 Java EE 产品提供商和工具提供商，需要考虑购买和安装 Java EE 产品和工具。购买并安装软件之后，可以由应用组件提供商开发 Java EE 组件，由应用装配人员负责装配，并由应用部署人员完成部署。在大型的组织内部，每个角色可能由不同的个人或团队承担。这种工作划分是可行的，因为前一个角色会输出一个可移植的文件，这可以作为下一个角色的输入。例如，在应用组件开发阶段，一位企业 bean 软件开发人员会交付 EJB JAR 文件。作为应用装配人员，另一位开发人员可以将这些 EJB JAR 文件组合为一个 Java EE 应用，并保存到一个 EAR 文件中。作为应用部署人员，在顾客现场的系统管理员可以使用这个 EAR 文件将 Java EE 应用安装到一个 Java EE 服务器。

不同角色并不总是由不同的人来承担。例如，如果你在一家小型公司工作，或者要完成一个示例应用的原型开发，那么每一个阶段的工作可能都需要由你完成。

1.7.1 Java EE 产品提供商

Java EE 产品提供商是设计和实现 Java EE 平台 API 及相关特性的公司，使其他人员可以购买 Java EE 平台 API 以及 Java EE 规范中定义的其他特性。产品提供商通常是根据 Java EE 7 平台规范实现 Java EE 平台的应用服务器开发商。

1.7.2 工具提供商

工具提供商是创建开发工具、装配工具和打包工具的公司或个人，这些工具分别供组件提供商、装配人员和部署人员使用。

1.7.3 应用组件提供商

应用组件提供商是创建 Web 组件、企业 bean、applet 或应用客户端的公司或个人，以便在 Java EE 应用中使用。

1.7.3.1 企业 bean 开发人员

企业 bean 开发人员要完成以下任务来交付包含一个或多个企业 bean 的 EJB JAR 文件：

- 编写和编译源代码。
- 指定部署描述文件（可选）。
- 将 `.class` 文件和部署描述文件打包到 EJB JAR 文件。

1.7.3.2 Web 组件开发人员

Web 组件开发人员要完成以下任务来交付包含一个或多个 Web 组件的 WAR 文件：

- 编写和编译 servlet 源代码。
- 编写 JSF、JSP 和 HTML 文件。
- 指定部署描述文件（可选）。
- 将 .class 文件、.jsp 文件、.html 文件以及部署描述文件打包到 WAR 文件。

1.7.3.3 应用客户端开发人员

应用客户端开发人员要完成以下任务来交付包含应用客户端的 JAR 文件：

- 编写和编译源代码。
- 指定客户端的部署描述文件（可选）。
- 将 .class 文件和部署描述文件打包到 JAR 文件。

1.7.4 应用装配人员

应用装配人员是从组件提供商接收应用模块的公司或个人，并将这些应用模块装配到一个 Java EE 应用 EAR 文件中。装配人员和部署人员可以直接编辑部署描述文件，或者可以使用工具，这些工具会根据交互选择正确地增加 XML 标记。

应用装配人员要完成以下任务来交付包含 Java EE 应用的 EAR 文件：

- 将之前各阶段中创建的 EJB JAR 文件和 WAR 文件装配到一个 Java EE 应用（EAR）文件中。
- 指定 Java EE 应用的部署描述文件（可选）。
- 验证 EAR 文件的内容是否合法并符合 Java EE 规范。

1.7.5 应用部署人员和管理员

应用部署人员和管理员是配置和部署应用客户端、Web 应用、EJB 组件和 Java EE 应用的公司或个人，他们要管理运行 Java EE 组件和应用的计算和网络基础设施，还要监督运行时环境。具体职责包括设置事务控制和安全属性以及指定数据库连接。

配置过程中，部署人员遵循应用组件提供商提供的说明来处理外部依赖，并指定安全设置和事务属性。安装过程中，部署人员将应用组件移至服务器，并生成容器特定的类和接口。

部署人员或系统管理员要完成以下任务来安装和配置 Java EE 应用或 Java EE 组件：

- 为运行环境配置 Java EE 应用或组件。
- 验证 EAR、JAR 和 / 或 WAR 文件的内容是否合法并符合 Java EE 规范。
- 将 Java EE 应用或组件部署（安装）到 Java EE 服务器。

1.8 Java EE 7 API

图 1-6 显示了 Java EE 容器中的各种关系。

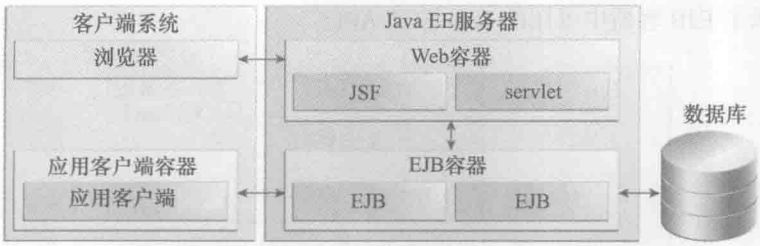


图 1-6 Java EE 容器

图 1-7 显示了 Web 容器中可用的 Java EE 7 API。

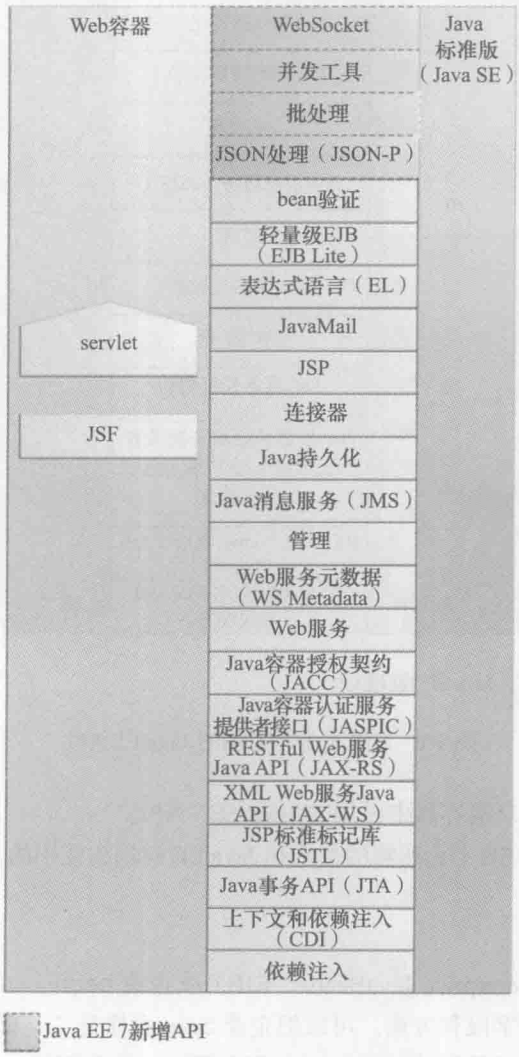


图 1-7 Web 容器中可用的 Java EE API

图 1-8 显示了 EJB 容器中可用的 Java EE 7 API。

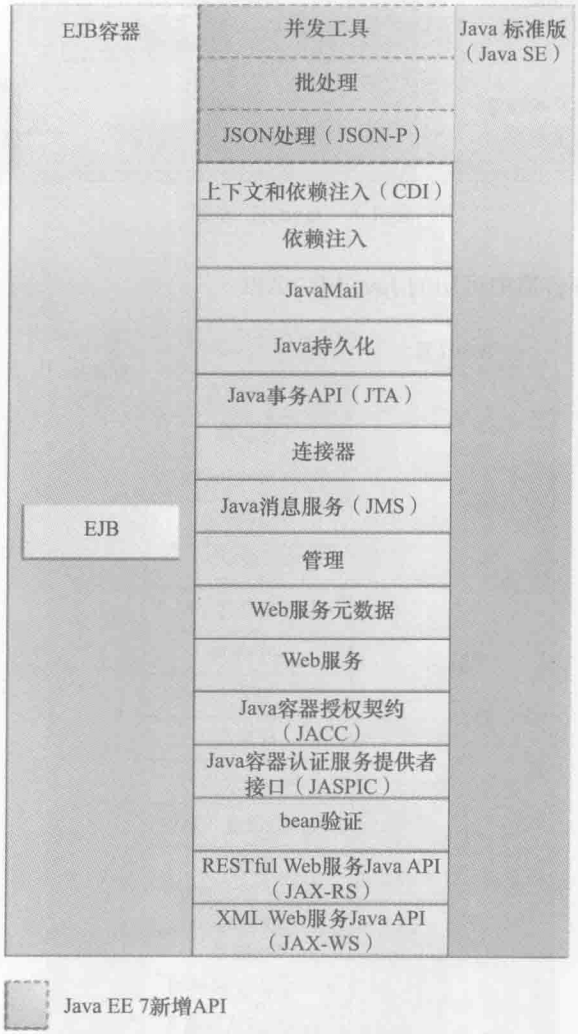


图 1-8 EJB 容器中可用的 Java EE API

图 1-9 显示了应用客户端容器中可用的 Java EE 7 API。

下面各小节将对 Java EE 平台所需的技术和 Java EE 应用中使用的 API 做一个简要的小结。

1.8.1 EJB 技术

企业 JavaBeans（Enterprise JavaBeans，EJB）或**企业 bean**（enterprise bean）是实现业务逻辑模块的代码，包含字段和方法。可以把企业 bean 看作是“基础构件”，可以单独使用，也可以与其他企业 bean 一同用来执行 Java EE 服务器上的业务逻辑。

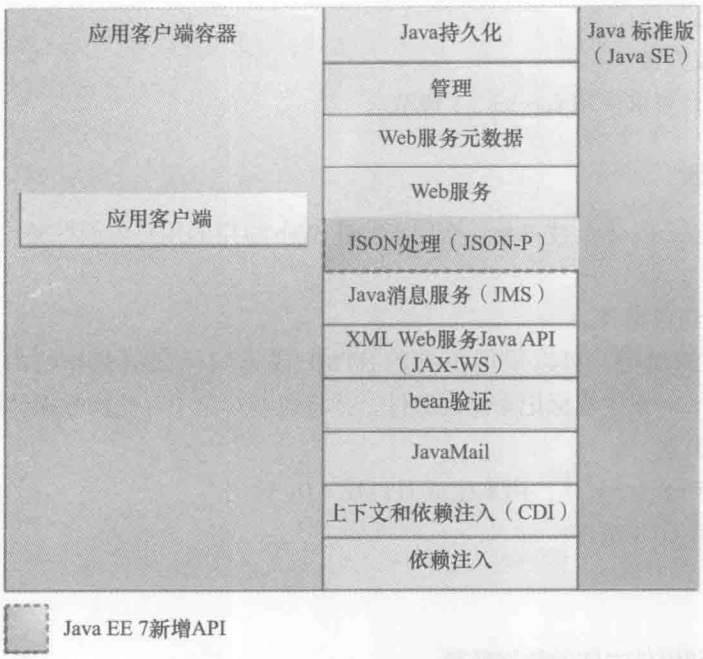


图 1-9 应用客户端容器中可用的 Java EE API

企业 bean 可以是会话 bean 或消息驱动 bean。

- **会话 bean**（session bean）表示与客户端的一个临时会话。客户端执行结束时，会话 bean 及其数据都会消失。
- **消息驱动 bean**（message-driven bean）结合了会话 bean 和消息监听器的特性，允许业务组件异步地接收消息。通常情况下，这些消息都是 Java 消息服务（Java Message Service，JMS）消息。

在 Java EE 7 平台上，新增了以下企业 bean 特性，包括：

- 轻量级 EJB（EJB Lite）中的异步本地会话 bean
- EJB Lite 中的非持久定时器

Java EE 7 平台要求使用 EJB 3.2 和 Interceptors 1.2 规范。Interceptor（拦截器）规范是 EJB 规范的一部分。

1.8.2 Java servlet 技术

Java servlet 技术允许定义特定于 HTTP 的 servlet 类。通过一个请求－响应编程模型，servlet 类可以扩展宿主应用所访问的服务器的能力。尽管 servlet 可以响应任意类型的请求，不过通常都用来扩展 Web 服务器上的应用。

在 Java EE 7 平台上，Java servlet 技术新增的特性包括：

- 非阻塞 I/O
- HTTP 协议升级

Java EE 7 平台要求使用 servlet 3.1 规范。

1.8.3 JSF 技术

JSF（JavaServer Faces）技术是一个用于构建 Web 应用的用户界面框架。JSF 技术主要包括以下主要组件：

- 一个 GUI 组件框架。
- 一个灵活的模型，可以采用不同的 HTML 或不同标记语言和技术完成组件呈现。`Renderer` 对象生成标记来呈现组件，并将模型对象中存储的数据转换为可以在视图中表现的类型。
- 一个标准 `RenderKit`，用来生成 HTML 4.01 标记。

以下特性支持 GUI 组件：

- 输入验证
- 事件处理
- 模型对象和组件之间的数据转换
- 托管模型对象创建
- 页面导航配置
- 表达式语言（Expression Language，EL）

通过使用标准 Java API 和基于 XML 的配置文件，可以得到所有这些功能。

在 Java EE 7 平台上，JSF 技术新增的特性包括：

- HTML5 友好的标记
- Face 流（Faces Flows）
- 资源库契约

Java EE 7 平台要求使用 JSF 2.2 和 EL 3.0 规范。

1.8.4 JSP 技术

利用 JSP（JavaServer Pages）技术，可以将 servlet 代码片段直接放在基于文本的文档中。JSP 页面是一个基于文本的文档，包含两类文本：

- 静态数据，可以采用任何基于文本的格式来表示，如 HTML 或 XML。
- JSP 元素，JSP 元素确定页面如何构造动态内容

关于 JSP 技术的更多信息参见《The Java EE 5 Tutorial》（`http://docs.oracle.com/javaee/5/tutorial/doc/`）。

Java EE 7 平台要求使用 JSP 2.3 规范来保证与较早版本的兼容性，不过建议在新开发的应用中显示技术采用 Facelets。

1.8.5 JSP 标准标记库

JSP 标准标记库（JavaServer Pages Standard Tag Library，JSTL）封装了很多 JSP 应用共有的核心功能。可以使用一组标准的标记，而不是在 JSP 应用中混合使用多个开发商提供的标记。基于这种标准化，可以将应用部署在支持 JSTL 的任何 JSP 容器上，而且这样更能保证标记实现得到优化。

JSTL 包括处理流控制的迭代器和条件标记、管理 XML 文档的标记、国际化标记、使用 SQL 访问数据库的标记以及完成一些常用功能的标记。

Java EE 7 平台要求使用 JSTL 1.2 规范。

1.8.6 Java 持久化 API

Java 持久化 API（Java Persistence API，JPA）是一个基于 Java 标准的持久化解决方案。持久化使用一种对象 / 关系映射方法在面向对象模型和关系数据库之间架起一座“桥梁”。Java 持久化 API 还可以用在 Java EE 环境之外的 Java SE 应用中。Java 持久化包括以下几个方面：

- Java 持久化 API
- 查询语言
- 对象 / 关系映射元数据

Java EE 7 平台要求使用 JPA 2.1 规范。

1.8.7 Java 事务 API

Java 事务 API（Java Transaction API，JTA）为事务界定提供了一个标准接口。Java EE 架构提供了一种默认的自动提交来处理事务提交和回滚。自动提交是指查看数据的所有其他应用所看到的都是数据库读写操作之后已更新的数据。不过，如果你的应用要完成两个单独的数据库访问操作，而且这两个操作相互依赖，可能就需要使用 JTA API 界定整个事务（包括两个操作）在哪里开始、回滚和提交。

Java EE 7 平台要求使用 JTA 1.2 规范。

1.8.8 RESTful Web 服务 Java API

RESTful Web 服务 Java API（Java API for RESTful Web Services，JAX-RS）为根据 REST 架构类型构建的 Web 服务定义了一个开发 API，这里的 REST 含义为表示性状态转移（Representational State Transfer）。JAX-RS 应用是一个 Web 应用，其中的类作为 servlet 连同必要的库打包在一个 WAR 文件中。

Java EE 7 平台要求使用 JAX-RS 2.0 规范。

1.8.9 托管 bean

托管 bean（Managed Bean）是有最低需求的轻量级容器托管对象（POJOs），支持很少的一组基本服务，如资源注入、生命周期回调和拦截器。相对于 JSF 技术中指定的托管 bean，这里的托管 bean 是一个推广，可以用在 Java EE 应用中的任何地方，而不仅限于 Web 模块中。

托管 bean 规范是 Java EE 7 平台规范的一部分（JSR 342）。Java EE 7 平台要求使用 Managed Beans 1.0 规范。

1.8.10 Java EE 的上下文和依赖注入

Java EE 的上下文和依赖注入（Contexts and Dependency Injection，CDI）定义了一组上下文服务，由 Java EE 容器提供，以便开发人员在 Web 应用中结合 JSF 技术使用企业 bean。CDI 原本设计用于有状态对象（stateful object），不过也有很多其他用途，使开发人员有充分的灵活性，可以用一种松耦合但类型安全的方式集成不同类型的组件。

Java EE 7 平台要求使用 CDI 1.1 规范。

1.8.11 Java 的依赖注入

Java 的依赖注入（Dependency Injection）定义了用于可注入类的一组标准的注解（和一个接口）。

CDI 在 Java EE 平台上提供了依赖注入支持。具体来讲，只能在支持 CDI 的应用中使用注入点。

Java EE 7 平台要求使用 Java 依赖注入（Dependency Injection for Java）1.0 规范。

1.8.12 bean 验证

bean 验证（Bean Validation）规范定义了一个元数据模型和 API，用于验证 JavaBeans 组件中的数据。不是将数据验证分布到多层，如浏览器和服务器端，而是可以只在一处定义验证约束，并由多个不同的层共享这些验证约束。

Java EE 7 平台要求使用 bean 验证（Bean Validation）1.1 规范。

1.8.13 Java 消息服务 API

Java 消息服务（Java Message Service，JMS）API 是一个消息传送标准，允许 Java EE 应用组件创建、发送、接收和读取消息。它支持松耦合、可靠、异步的分布式通信。

在这个平台上，JMS 的新特性包括：

- 一个新的简化 API，与之前的 API 相比，这是一个更简单的替代接口。这个 API 包括一个 `JMSContext` 对象，合并了 `Connection` 和 `Session` 的功能。
- 包含 `close` 方法的所有对象都要实现 `java.lang.Autocloseable` 接口，才能在 Java SE 7 `try`-with-resources 语句中使用。

Java EE 7 平台要求使用 JMS 2.0 规范。

1.8.14 Java EE 连接器架构

工具开发商和系统集成商使用 Java EE 连接器架构（Java EE Connector Architecture）来创建资源适配器，以支持访问企业信息系统，这些企业信息系统可以插入到任意的 Java EE 产品中。**资源适配器**（resource adapter）是一个软件组件，允许 Java EE 应用组件访问 EIS 的底层资源管理器，并与之交互。由于资源适配器特定于相应的资源管理器，所以每种类型的数据库或企业信息系统通常有一个不同的资源适配器。

Java EE 连接器架构还为基于 Java EE 平台的 Web 服务与已有的企业信息系统（可能同步或异步）提供了一种事务集成，具有高性能、安全、可扩展和基于消息等特点。利用 JAX-WS 和 Java EE 组件模型，通过 Java EE 连接器架构集成到 Java EE 平台的已有应用和 EIS 可以作为基于 XML 的 Web 服务对外提供。因此，JAX-WS 和 Java EE 连接器架构是实现企业应用集成（enterprise application integration，EAI）和端到端业务集成的补充技术。

Java EE 7 平台要求使用 Java EE 连接器架构 1.7 规范。

1.8.15 JavaMail API

Java EE 应用使用 JavaMail API 来发送 email 通知。JavaMail API 包括两部分：

- 应用组件用于发送邮件的应用级接口
- 服务提供者接口

Java EE 平台包括 JavaMail API，另外还有一个服务提供者，允许应用组件发送 Internet 邮件。

Java EE 7 平台要求使用 JavaMail 1.5 规范。

1.8.16 Java 容器授权契约

Java 容器授权契约（Java Authorization Contract for Containers，JACC）规范定义了 Java EE 应用服务器和授权策略提供者之间的一个契约。所有 Java EE 容器都支持这个契约。

JACC 规范定义了满足 Java EE 授权模型的 `java.security.Permission` 类。规范还定义了这些权限类实例上的操作与容器访问决策之间的绑定。它定义了策略提供者的语义，策略提供者将使用这些新的权限类来处理 Java EE 平台的授权需求，包括角色的定义和使用。

Java EE 7 平台要求使用 JACC 1.5 规范。

1.8.17 Java 容器认证服务提供者接口

Java 容器认证服务提供者接口（Java Authentication Service Provider Interface for Containers，JASPIC）规范定义了一个服务提供者接口（service provider interface，SPI），利用这个接口，

实现消息认证机制的认证提供者可以集成到客户端或服务器端的消息处理容器或运行时环境。通过这个接口集成的认证提供者会处理调用容器为它们提供的网络消息。认证提供者会转换发出的消息，使得接收消息的容器能够对各个消息的来源完成认证，而消息发送者也能对消息接收者进行认证。认证提供者会认证每个到来的消息，并向其调用容器返回所确定的身份，作为消息认证的结果。

Java EE 7 平台要求使用 JASPIC 1.1 规范。

1.8.18 WebSocket Java API

WebSocket 是一个应用协议，可以在 TCP 上为通信双方提供全双工通信。WebSocket Java API 允许 Java EE 应用使用注解创建端点，这些注解指定了端点的配置参数，并指定其生命周期回调方法。

WebSocket API 是 Java EE 7 平台新增的特性。Java EE 7 平台要求使用 WebSocket Java API 1.0 规范。

1.8.19 JSON 处理 Java API

JSON 是由 JavaScript 发展而来的一种基于文本的数据交换格式，广泛用于 Web 服务和其他连接性应用。JSON 处理 Java API（JSON-P）允许 Java EE 应用使用对象模型或流模型解析、转换和查询 JSON 数据。

JSON-P 是 Java EE 7 平台新增的特性。Java EE 7 平台要求使用 JSON-P 1.0 规范。

1.8.20 Java EE 的并发工具

Java EE 的并发工具（Concurrency Utilities）是一个标准 API，通过以下各类对象为 Java EE 应用组件提供异步功能：托管执行器服务、托管调度执行器服务、托管线程工厂和上下文服务。

Java EE 的并发工具是 Java EE 7 平台新增的特性。Java EE 7 平台要求使用 Java EE 并发工具（Concurrency Utilities for Java EE）1.0 规范。

1.8.21 Java 平台的批处理应用

批处理作业是无需用户交互也可以执行的任务。Java 平台的批处理应用（Batch Applications）规范是一个批处理框架，为 Java 应用中创建和运行批处理作业提供支持。这个批处理框架包括一个批处理运行时环境、一个基于 XML 的作业规范语言、一个与批处理运行时环境交互的 Java API 以及一个实现批处理工件（artifact）的 Java API。

Java 平台的批处理应用是 Java EE 7 平台新增的特性。Java EE 7 平台要求使用 Java 平台批处理应用（Batch Applications for the Java Platform）1.0 规范。

1.9 Java 平台标准版 7 的 Java EE 7 API

Java EE 7 平台需要的很多 API 包含在 Java 平台标准版 7（Java SE 7）中，在 Java EE 应用中也可以使用。

1.9.1 Java 数据库互连 API

Java 数据库互连（Java Database Connectivity，JDBC）API 允许由 Java 编程语言方法调用 SQL 命令。会话 bean 访问数据库时，就会在企业 bean 中使用 JDBC API。还可以使用 JDBC API 由一个 servlet 或 JSP 页面直接访问数据库，而无需通过企业 bean。

JDBC API 包括两部分：

- 应用组件访问数据库所用的应用级接口
- 将一个 JDBC 驱动程序关联到 Java EE 平台的服务提供者接口

Java SE 7 平台要求使用 JDBC 4.1 规范。

1.9.2 Java 命名和目录接口 API

Java 命名和目录接口（Java Naming and Directory Interface，JNDI）API 提供了命名和目录功能，允许应用访问多个命名和目录服务，如 LDAP、DNS 和 NIS。JNDI API 为应用提供了大量方法来完成标准的目录操作，如为对象关联属性以及使用对象属性搜索对象。通过使用 JNDI，Java EE 应用可以存储和获取任意类型的命名 Java 对象，使 Java EE 应用可以与很多遗留应用和系统并存。

利用 Java EE 命名服务，应用客户端、企业 bean 和 Web 组件可以访问 JNDI 命名环境。**命名环境**（naming environment）允许定制组件，而不需要访问或修改组件的源代码。由容器实现组件的环境，并将这个环境作为一个 JNDI **命名上下文**（naming context）提供给组件。

命名环境为组件、模块或应用中使用的对象以及由所有部署应用共享的对象提供了 4 个逻辑命名空间：`java:comp`、`java:module`、`java:app` 和 `java:global`。Java EE 组件可以访问系统提供的命名对象以及用户自定义的命名对象。系统提供的一些对象（如默认的 JDBC `DataSource` 对象、默认的 JMS 连接工厂以及 JTA `UserTransaction` 对象）的名字存储在 `java:comp` 命名空间。Java EE 平台允许组件对用户自定义对象命名，如企业 bean、环境项（environment entry）、JDBC `DataSource` 对象和消息传送目的地。

Java EE 组件还可以使用 JNDI 接口找到其环境命名上下文。组件可以创建一个 `javax.naming.InitialContext` 对象，在 `InitialContext` 中用 `java:comp/env` 名查找环境命名上下文。组件的命名环境直接存储在环境命名上下文中，或者存储在它的某个直接或间接子上下文中。

1.9.3 JavaBeans 激活框架

JavaBeans 激活框架（JavaBeans Activation Framework，JAF）在 JavaMail API 中使用。JAF 提供了一些标准服务，用于确定任意一个数据的类型、封装对数据的访问、发现可用的操作以及创建适当的 JavaBeans 组件来完成这些操作。

1.9.4 XML 处理 Java API

XML 处理 Java API（Java API for XML Processing，JAXP）是 Java SE 平台的一部分，支持使用文档对象模型（Document Object Model，DOM）、XML 简单 API（Simple API for XML，SAX）和可扩展样式表语言转换（Extensible Stylesheet Language Transformation，XSLT）处理 XML 文档。JAXP 允许应用不依赖特定的 XML 处理实现来解析和转换 XML 文档。

JAXP 还提供了命名空间支持，允许正确地使用模式（schema），否则模式可能会有命名冲突。本着灵活性这一设计初衷，JAXP 允许在应用中使用任何与 XML 兼容的解析器或 XSL 处理器，并支持万维网联盟（Worldwide Web Consortium，W3C）模式。有关 W3C 模式的信息可以参见 `http://www.w3.org/XML/Schema`。

1.9.5 Java XML 绑定架构

Java XML 绑定架构（Java Architecture for XML Binding，JAXB）提供了一种很方便的方法，可以将 XML 模式绑定到 Java 程序中的一个表示。JAXB 可以独立使用，也可以结合 JAX-WS 使用，无论哪一种情况，JAXB 都会为 Web 服务消息提供一个标准的数据绑定。所有 Java EE 应用客户端容器、Web 容器和 EJB 容器都支持 JAXB API。

Java EE 7 平台要求使用 JAXB 2.2 规范。

1.9.6 XML Web 服务 Java-API

XML Web 服务 Java-API（Java API for XML Web Service，JAX-WS）规范为使用 JAXB API 将 XML 数据绑定到 Java 对象的 Web 服务提供支持。JAX-WS 规范定义了访问 Web 服务的客户端 API，另外还定义了实现 Web 服务端点的技术。实现企业 Web 服务（Implementing Enterprise Web Service）规范描述了基于 JAX-WS 的服务和客户端的部署。EJB 和 Java servlet 规范也对这种部署的某些方面做了说明。基于 JAX-WS 的应用可以使用其中任意的部署模型来部署。

JAX-WS 规范描述了对消息处理器的支持，这些消息处理器可以处理消息请求和响应。一般来讲，这些消息处理器都在同一个容器中执行，与相关联的 JAX-WS 客户端或端点组件有相同的权限和执行上下文。类似于其关联的组件，这些消息处理器可以访问相同的 JNDI 命名空间。如果支持定制串行化工具和逆串行化工具，则会采用与消息处理器同样的方式来处理。

Java EE 7 平台要求使用 JAX-WS 2.2 规范。

1.9.7 带附件 SOAP Java API

带附件 SOAP Java API（SOAP with Attachments API for Java，SAAJ）是一个底层 API，JAX-WS 就以这个 API 为基础。SAAJ 支持生成和使用符合 SOAP 1.1 和 1.2 规范以及带附件 SOAP 说明的消息。大多数开发人员并不使用 SAAJ API，而会使用更高层的 JAX-WS API。

1.9.8 Java 认证和授权服务

Java 认证和授权服务（Java Authentication and Authorization Service，JAAS）为 Java EE 应用提供了一种方法，可以用来对运行该应用的特定用户或用户组进行认证和授权。

JAAS 是标准可插式认证模块（Pluggable Authentication Module，PAM）框架的一个 Java 编程语言版本，它扩展了 Java 平台的安全架构，可以支持基于用户的授权。

1.9.9 Java 平台的公共注解

注解（annotation）支持 Java 平台的声明式编程。

Java EE 7 平台要求使用 Java 平台的公共注解（Common Annotations for the Java Platform）1.2 规范。

1.10 GlassFish 服务器工具

GlassFish 服务器是 Java EE 7 平台的一个兼容实现。除了支持以上各小节介绍的所有 API，GlassFish 服务器还包括很多 Java EE 工具，这些工具不属于 Java EE 7 平台，不过可以为开发人员提供很多便利。

这一节将简要总结 GlassFish 服务器包括的工具。要了解如何启动和停止 GlassFish 服务器、启动管理控制台以及启动和停止 Java DB 服务器，可以参见第 2 章的有关说明。

GlassFish 服务器包含表 1-1 所列的工具。本书中会给出其中很多工具的基本用法信息。有关的详细信息请参见 GUI 工具中的联机帮助。

表 1-1 GlassFish 服务器工具

工 具	描 述
管理控制台	这是一个基于 Web 的 GlassFish 服务器 GUI 管理工具。用于停止 GlassFish 服务器以及管理用户、资源和应用
asadmin	这是一个 GlassFish 服务器命令行管理工具。用于启动和停止 GlassFish 服务器以及管理用户、资源和应用
appclient	这是一个命令行工具，用来启动应用客户端容器，并调用打包在应用客户端 JAR 文件中的客户端应用

（续）

工　具	描　述
capture-schema	这是一个命令行工具，用于从数据库抽取模式信息，生成一个模式文件，GlassFish 服务器可以利用这个模式文件实现容器托管持久化
package-appclient	这是一个命令行工具，用于打包应用客户端容器库和 JAR 文件
Java DB 数据库	Java DB 服务器的一个副本
xjc	这是一个命令行工具，用于将源 XML 模式转换或绑定为用 Java 编程语言编写的一组 JAXB 内容类
schemagen	这是一个命令行工具，用于为 Java 类中引用的各个命名空间创建一个模式文件
wsimport	这是一个命令行工具，用于为给定的 WSDL 文件生成 JAX-WS 可移植工件。生成之后，这些工件可以与 WSDL 和模式文档一同打包在一个 WAR 文件中（还包括端点实现），然后完成部署
wsgen	这是一个命令行工具，用于读取一个 Web 服务端点类，并生成所需的所有 JAX-WS 可移植工件来完成 Web 服务的部署和调用

第 2 章 Chapter 2

使用教程示例

本章将介绍安装、构建和运行本书示例所需要了解的所有知识，包括以下内容：

- 必要软件
- 启动和停止 GlassFish 服务器
- 启动管理控制台
- 启动和停止 Java DB 服务器
- 构建示例
- 教程示例目录结构
- 教程中的 Java EE 7 Maven 原型
- 获取教程最新更新
- 调试 Java EE 应用

2.1 必要软件

运行本书示例要求有以下软件：

- Java 平台标准版（见 2.1.2 节）
- Java EE 7 软件开发包（见 2.1.1 节）
- Java EE 7 教程组件（见 2.1.3 节）
- NetBeans IDE（见 2.1.4 节）
- Apache Maven（见 2.1.5 节）

2.1.1 Java EE 7 软件开发包

本书将采用 GlassFish 服务器开源版本 4 作为教程示例的构建和运行时环境。要构建、部署和运行这些示例，需要一个 GlassFish 服务器，还可以有一个 NetBeans IDE（可选）。要得到 GlassFish 服务器，必须安装 Java EE 7 软件开发包（Software Development Kit，SDK），SDK 可以从 `http://www.oracle.com/technetwork/java/javaee/downloads/` 下载。务必要下载 Java EE 7 SDK，而不是 Java EE 7 Web Profile SDK。Java EE 7 SDK 有包含以及不包含 Java 平台标准版 7 开发包的不同版本。

2.1.1.1 SDK 安装提示

安装 SDK 过程中要完成以下工作：

- 允许安装程序下载和配置更新工具（Update Tool）。如果通过一个防火墙访问因特网，还需要提供代理主机和端口。
- 配置 GlassFish 服务器管理用户名为 `admin`，不用指定密码。这是默认设置。
- 接受 Admin 端口和 HTTP 端口的默认端口值，分别为 4848 和 8080。
- 不要选择为域创建操作系统服务的复选框。

如果愿意，可以选中“创建后启动域”复选框，不过并不要求必须选中这个复选框。

这个教程用 *as-install-parent* 表示安装 GlassFish 服务器的目录。例如，Microsoft Windows 上的默认安装目录为 `C:\glassfish4`，所以 *as-install-parent* 就是 `C:\glassfish4`。GlassFish 服务器本身安装在 *as-install*，这是 *as-install-parent* 下的 `glassfish` 目录。所以，在 Microsoft Windows 上 *as-install* 就是 `C:\glassfish4\glassfish`。

安装 GlassFish 服务器之后，将以下目录增加到 `PATH`，这样使用命令时就不必再指定完整的路径：

```
as-install-parent/bin
as-install/bin
```

2.1.2 Java 平台标准版

要构建、部署和运行这些示例，需要有 Java 平台标准版 7 开发包（JDK 7）。有些 Java EE 7 SDK 版本包含 JDK 7。可以从以下地址单独下载 JDK 7 软件：

`http://www.oracle.com/technetwork/java/javase/downloads`.

2.1.3 Java EE 7 教程组件

教程示例的源代码包含在教程组件中。需要使用更新工具（Update Tool）来得到教程组件。

2.1.3.1 使用更新工具得到教程组件

1. 完成以下任意一个动作启动更新工具。

- 在命令行输入命令 `updatetool`。
- 在 Windows 系统上，从 Start（开始）菜单选择 All Programs（所有程序），然后选择 Java EE 7 SDK，再选择 Start Update Tool（启动更新工具）。

2. 展开 Java EE 7 SDK 节点。
3. 选择 Available Updates（可用更新）。
4. 从列表中选择 Java EE 7 Tutorial 复选框。
5. 单击 Install（安装）。
6. 接受许可协议。

安装之后，已安装组件列表中会出现 Java EE 7 Tutorial。这个工具安装在 *as-install-parent*/`docs/javaee-tutorial` 目录中，本书将这个安装目录称为 *tut-install*。该目录包含两个子目录：`docs` 和 `examples`。`examples` 目录中的各个子目录分别对应本书中讨论的各项技术。

后续步骤

Java EE 7 Tutorial 组件的更新会定期发布。关于如何获得这些更新的详细信息参见 2.8 节。

2.1.4　NetBeans IDE

NetBeans 集成开发环境（integrated development environment，IDE）是开发 Java 应用（也包括企业应用）的一个免费、开源的 IDE。NetBeans IDE 支持 Java EE 平台。可以在 NetBeans IDE 中构建、打包、部署和运行教程示例。

要运行教程示例，需要最新版本的 NetBeans IDE。可以从以下地址下载 NetBeans IDE：`https://www.netbeans.org/downloads/`。一定要下载 Java EE bundle。

2.1.4.1　不通过 GlassFish 服务器安装 NetBeans IDE

安装 NetBeans IDE 时，不要安装提供 NetBeans IDE 的 GlassFish 服务器版本。完成以下步骤跳过 GlassFish 服务器的安装：

1. 在 NetBeans IDE 安装程序向导的第一个页面上，取消选择对应 GlassFish 服务器的复选框，并单击 OK。
2. 接受许可协议和 Junit 许可协议。有些教程示例会用到 Junit 库，所以需要安装 Junit。
3. 继续安装 NetBeans IDE。

2.1.4.2　增加 GlassFish 服务器作为使用 NetBeans IDE 的服务器

要在 NetBeans IDE 中运行教程示例，必须在 NetBeans IDE 中增加 GlassFish 服务器作为服务器。按照以下说明为 NetBeans IDE 增加 GlassFish 服务器。

1. 从 Tools（工具）菜单选择 Servers（服务器）。
2. 在服务器向导页面中，点击 Add Server（增加服务器）。
3. 在 Choose Server（选择服务器）下面选择 GlassFish Server（GlassFish 服务器）并单击

Next（下一步）。

4. 在 Server Location（服务器位置）下面浏览到 Java EE 7 SDK 所在位置，并单击 Next（下一步）。

5. 在 Domain Location（域位置）下面选择 Register Local Domain（注册本地域）。

6. 单击 Finish（完成）。

2.1.5 Apache Maven

Maven 是一个基于 Java 技术的构建工具，由 Apache 软件基金会（Apache Software Foundation）开发，用于构造、打包和部署教程示例。要从命令行运行教程示例，需要使用 Maven 3.0 或更高版本。如果还没有安装 Maven，可以从以下地址安装：

```
http://maven.apache.org
```

要确保将 *maven-install*/bin 目录增加到 PATH。

如果使用 NetBeans IDE 构建和运行示例，则需要知道 NetBeans IDE 已经自带安装了 Maven。

2.2 启动和停止 GlassFish 服务器

可以使用 NetBeans IDE 或命令行启动和停止 GlassFish 服务器。

2.2.1 使用 NetBeans IDE 启动 GlassFish 服务器

1. 单击 Services（服务）标签页。

2. 展开 Servers（服务器）。

3. 右键单击 GlassFish 服务器实例并选择 Start（启动）。

2.2.2 使用 NetBeans IDE 停止 GlassFish 服务器

要使用 NetBeans IDE 停止 GlassFish 服务器，右键单击 GlassFish 服务器实例并选择 Stop（停止）。

2.2.3 使用命令行启动 GlassFish 服务器

要从命令行启动 GlassFish 服务器，打开一个终端窗口或者命令行提示窗口，执行以下命令：

```
asadmin start-domain --verbose
```

domain（域）是由一个管理服务器管理的一个或多个 GlassFish 服务器实例。与域关联的元素包括：

- GlassFish 服务器的端口号：默认为 8080。
- 管理服务器的端口号：默认为 4848。
- 管理用户名和密码：默认用户名为 admin，默认情况下不需要密码。

安装 GlassFish 服务器时要指定这些值。本教程中的示例假设你选择了默认端口，另外选择了默认用户名，而且未指定密码。

如果没有参数，start-domain 命令会启动默认域，也就是 domain1。--verbose 标志会使所有日志记录和调试输出出现在终端窗口或命令行提示窗口上。这个输出也会记入服务器日志，服务器日志位于 *domain-dir*/logs/server.log。

或者，在 Windows 上，可以从 Start（开始）菜单选择 All Programs（所有程序），然后选择 Java EE 7 SDK，再选择 Start Application Server（启动应用服务器）。

2.2.4　使用命令行停止 GlassFish 服务器

要停止 GlassFish 服务器，首先打开一个终端窗口或命令行提示窗口，并执行以下命令：

```
asadmin stop-domain domain1
```

或者，在 Windows 上，可以从 Start（开始）菜单选择 All Programs（所有程序），然后选择 Java EE 7 SDK，再选择 Stop Application Server（停止应用服务器）。

2.3　启动管理控制台

要管理 GlassFish 服务器以及用户、资源和 Java EE 应用，需要使用管理控制台工具。调用管理控制台之前，必须先运行 GlassFish 服务器。启动管理控制台时，首先打开浏览器访问 http://localhost:4848/。

或者在 Windows 上，可以从 Start（开始）菜单选择 All Programs（所有程序），然后选择 Java EE 7 SDK，再选择 Administration Console（管理控制台）。

2.3.1　使用 NetBeans IDE 启动管理控制台

1. 单击 Services（服务）标签页。
2. 展开 Servers（服务器）。
3. 右键单击 GlassFish 服务器实例并选择 View Domain Admin Console（查看域管理控制台）。

说明　NetBeans IDE 会用你的默认 Web 浏览器打开管理控制台。

2.4 启动和停止 Java DB 服务器

GlassFish 服务器包含有 Java DB 数据库服务器。

要从命令行启动 Java DB 服务器，首先打开一个终端窗口或命令行提示窗口，并执行以下命令：

```
asadmin start-database
```

要从命令行停止 Java DB 服务器，需打开一个终端窗口或命令行提示窗口，并执行以下命令：

```
asadmin stop-database
```

关于 GlassFish 服务器自带 Java DB 的有关信息，参见 http://www.oracle.com/technetwork/java/javadb/overview/。

2.4.1 使用 NetBeans IDE 启动数据库服务器

使用 NetBeans IDE 启动 GlassFish 服务器时，数据库服务器会自动启动。不过，如果需要手动地启动数据库服务器，则可以遵循以下步骤。

1. 单击 Services（服务）标签页。
2. 展开 Databases（数据库）。
3. 右键单击 Java DB 并选择 Start Server（启动服务器）。

后续步骤

要使用 NetBeans IDE 停止数据库，可以右键单击 Java DB 并选择 Stop Server（停止服务器）。

2.5 构建示例

教程示例都附带一个面向 NetBeans IDE 或 Maven 的配置文件。可以用 NetBeans IDE 或 Maven 来构建、打包、部署和运行这些示例。各章都提供了构建示例的详细说明。

2.6 教程示例目录结构

为了便于迭代式开发，并保证应用的源文件与已编译文件分开，教程示例将使用 Maven 应用目录结构。

每个应用模块有以下结构：

- pom.xml：Maven 构建文件
- src/main/java：模块的 Java 源文件

- `src/main/resources`：模块的配置文件（Web 应用除外）
- `src/main/webapp`：Web 页面、样式表、标记文件和图像（仅 Web 应用）
- `src/main/webapp/WEB-INF`：Web 应用的配置文件（仅 Web 应用）

如果一个示例将多个应用模块打包在一个 EAR 文件中，则其子模块目录使用以下命名约定：

- *example-name*-`app-client`：应用客户端
- *example-name*-`ejb`：企业 bean JAR 文件
- *example-name*-`war`：Web 应用
- *example-name*-`ear`：企业应用
- *example-name*-`common`：包含组件、类和其他模块所用文件的库 JAR

随示例发布的 Maven 构建文件（`pom.xml`）包含构建目标，将应用编译并装配到 `target` 目录，并将归档文件部署到 GlassFish 服务器。

2.7　教程中的 Java EE 7 Maven 原型

有些章节介绍了如何使用 Maven 原型构建示例应用。**原型**（Archetypes）是用来生成一个特定 Maven 工程的模板。本教程包含很多生成 Java EE 7 工程的 Maven 原型。

2.7.1　安装教程 Maven 原型

基于原型创建新工程之前，必须将所包括的相应 Maven 原型安装到你的本地 Maven 存储库中。可以使用 NetBeans IDE 或 Maven 来安装这些原型。

2.7.1.1　使用 NetBeans IDE 安装教程 Maven 原型

1. 从 File（文件）菜单选择 Open Project（打开工程）。
2. 在 Open Project（打开工程）对话框中导航至：

```
tut-install/examples
```

3. 选择 `archetypes` 文件夹。
4. 单击 Open Project（打开工程）。
5. 在 Projects（工程）标签页中，右键单击 `archetypes` 工程并选择 Build（构建）。

2.7.1.2　使用 Maven 安装教程 Maven 原型

1. 在终端窗口中导航至：

```
tut-install/examples/archetypes/
```

2. 输入以下命令：

```
mvn install
```

2.8 获取教程最新更新

可以使用 Java EE 7 SDK 附带的更新工具（Update Tool）查看教程的更新情况。

2.8.1 使用 NetBeans IDE 更新教程

1. 在 NetBeans IDE 中打开 Services（服务）标签页，并展开 Servers（服务器）。

2. 右键单击 GlassFish 服务器实例并选择 View Domain Update Center（查看域更新中心），显示更新工具（Update Tool）。

3. 在树中选择 Available Updates（可用更新），显示一个更新包列表。

4. 查找 Java EE 7 Tutorial（javaee-tutorial）包的更新。

5. 如果教程有一个更新版本，则选择 Java EE 7 Tutorial（javaee-tutorial）并单击 Install（安装）。

2.8.2 使用命令行更新教程

1. 打开一个终端窗口，输入以下命令来显示更新工具：

```
updatetool
```

2. 在树中选择 Available Updates（可用更新），显示一个更新包列表。

3. 查找 Java EE 7 Tutorial（javaee-tutorial）包的更新。

4. 如果教程有一个更新版本，则选择 Java EE 7 Tutorial（javaee-tutorial）并单击 Install（安装）。

2.9 调试 Java EE 应用

这一节介绍如何确定导致应用部署或执行中出现错误的原因。

2.9.1 使用服务器日志

调试应用时，一种方法是查看位于 *domain-dir*/`logs/server.log` 的服务器日志。这个日志包含了 GlassFish 服务器和应用的输出。可以用 `System.out.println` 和 Java 日志 API 记录应用中所有 Java 类的消息，Java 日志 API（Java Logging API）参见 `http://docs.oracle.com/javase/7/docs/technotes/guides/logging/`，另外可以用 `ServletContext.log` 方法记录 Web 组件的消息。

如果使用 NetBeans IDE，那么日志输出除了记入服务器日志外，还会出现在输出窗口。

如果用 `--verbose` 标志启动 GlassFish 服务器，则所有日志记录和调试输出都会出现在终端窗口或命令行提示窗口以及服务器日志中。如果在后台启动 GlassFish 服务器，则调试信息只会记入日志。可以用文本编辑器或者管理控制台日志阅读器来查看服务器日志。

2.9.1.1　使用管理控制台日志阅读器

1. 选择 GlassFish 服务器节点。
2. 单击 View Log Files（查看日志文件）。

   日志阅读器会打开并显示最近的 40 个日志。
3. 要显示其他日志，可以遵循以下步骤。

   a. 单击 Modify Search（修改搜索）。

   b. 根据你想要查看的日志指定相应的约束条件。

   c. 单击日志阅读器上方的 Search（搜索）。

2.9.2　使用调试器

GlassFish 服务器支持 Java 平台调试器架构（Java Platform Debugger Architecture，JPDA）。利用 JPDA，可以适当地配置 GlassFish 服务器以使用 socket 传递调试信息。

2.9.2.1　使用调试器调试应用

1. 使用管理控制台按照以下步骤在 GlassFish 服务器中启用调试：

   a. 展开 Configurations（配置）节点，然后展开 server-config（服务器配置）节点。

   b. 选择 JVM Settings（JVM 设置）节点。默认的调试选项设置为：

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=9009
```

可以看到，默认的调试器 socket 端口为 9009。可以把它改为未由 GlassFish 服务器或其他服务使用的某个端口。

   c. 选择 Debug Enabled（启用调试）复选框。

   d. 单击 Save（保存）。
2. 停止 GlassFish 服务器，然后重启。

第二部分 *Part 2*

企业 bean

第二部分研究企业 JavaBeans 组件（Enterprise JavaBeans component），包括以下各章：

- 第 3 章　企业 bean
- 第 4 章　企业 bean 入门
- 第 5 章　运行企业 bean 示例
- 第 6 章　使用嵌入式企业 bean 容器
- 第 7 章　在会话 bean 中使用异步方法调用

Chapter 3 第 3 章

企业 bean

企业 bean 是实现企业 JavaBeans 技术（Enterprise JavaBeans，EJB）的 Java EE 组件。企业 bean 在 EJB 容器中运行，这是 GlassFish 服务器中的一个运行时环境（参见 1.4.2 节）。尽管对应用开发人员是透明的，不过 EJB 容器为企业 bean 提供了大量系统级服务，如事务和安全性。利用这些服务，可以快速地构建和部署企业 bean，企业 bean 则构成了事务性 Java EE 应用的核心。

本章将介绍以下内容：

- 什么是企业 bean
- 什么是会话 bean
- 什么是消息驱动 bean
- 访问企业 bean
- 企业 bean 的内容
- 企业 bean 的命名约定
- 企业 bean 的生命周期
- 关于企业 bean 的更多信息

3.1 什么是企业 bean

企业 bean 用 Java 编程语言编写，这是封装了应用业务逻辑的服务器端组件。**业务逻辑**（business logic）就是实现应用用途的代码。例如，在一个库存控制应用中，企业 bean 可能会在名为 `checkInventoryLevel` 和 `orderProduct` 的方法中实现业务逻辑。通过调用这

些方法，客户端可以访问这个应用提供的库存服务。

3.1.1 企业bean的好处

企业bean可以简化大型分布式应用的开发，这有多方面的原因。首先，由于EJB容器为企业bean提供了系统级服务，bean开发人员可以集中精力解决业务问题。由EJB容器负责系统级服务，而不需要bean开发人员考虑这些问题，如事务管理和安全授权。

其次，由于应用的业务逻辑包含在bean中，而不是包含在客户端，所以客户端开发人员可以集中精力考虑客户端的表现。客户端开发人员不需要编写实现业务规则或访问数据库的例程。因此，客户端更瘦，这个优点对于在小型设备上运行的客户端程序来说尤为重要。

第三，由于企业bean是可移植的组件，因此应用装配工具可以由现有的bean构建新的应用。假设都使用标准API，则这些应用可以在任何兼容的Java EE服务器上运行。

3.1.2 何时使用企业bean

如果你的应用有以下某个需求，就应该考虑使用企业bean。

- 应用必须是可扩展的。为了适应不断增长的用户数，可能需要将一个应用的组件分布在多个机器上。应用的企业bean不仅可以运行在不同的机器上，而且它们的位置对客户端来说仍保持透明。
- 事务必须确保数据完整性。企业bean支持事务，这是一种管理共享对象并发访问的机制。
- 应用有多种不同类型的客户端。只需要简单的几行代码，远程客户端就可以轻松地找到企业bean。这些客户端可以很瘦、有不同类型，而且可以数量众多。

3.1.3 企业bean类型

表3-1总结了两类企业bean。后面几小节将更详细地讨论这两类企业bean。

表3-1 企业bean类型

企业bean类型	用　途
会话bean	为客户端完成一个任务，或者可能实现一个Web服务
消息驱动bean	作为某个特定消息类型的监听器，如Java消息服务API（Java Message Service API）

3.2 什么是会话bean

会话bean封装了可能由客户端通过本地、远程或Web服务客户端视图调用的业务逻辑。要访问部署在服务器上的一个应用，客户端要调用会话bean的方法。会话bean会为其客户端完成工作，通过在服务器中执行业务任务，从而向客户端隐藏复杂性。

会话bean不是持久化的（也就是说，它的数据不会保存到一个数据库中）。有关的代码

示例参见第 5 章。

3.2.1 会话 bean 类型

会话 bean 有 3 种类型：有状态会话 bean、无状态会话 bean 和单例会话 bean。

3.2.1.1 有状态会话 bean

对象的状态由其实例变量的值组成。在一个**有状态会话 bean** 中，实例变量表示一个唯一客户端 /bean 会话的状态。由于客户端与其 bean 交互（“对话”），所以这个状态通常称为**对话状态**（conversational state）。

顾名思义，会话 bean 与交互式会话类似。会话 bean 不能共享；它只能有一个客户端，这类似于交互式会话只能有一个用户。客户端终止时，其会话 bean 也会终止，不再与这个客户端关联。

状态在客户端 /bean 会话期间保留。如果客户端删除了这个 bean，会话就会结束，状态也会消失。不过，状态的这种暂时性并不是问题，因为客户端与 bean 之间的对话结束时，也就没有必要再保留状态了。

3.2.1.2 无状态会话 bean

无状态会话 bean 不会维持与客户端的会话状态。客户端调用一个无状态 bean 的方法时，bean 的实例变量可能包含特定于该客户端但仅适用于此次调用的状态。方法完成时，客户端特定的状态不应保留。不过，客户端可以改变池中无状态 bean 的实例变量状态，这个状态会保留到池中无状态 bean 的下一次调用。除了方法调用期间，无状态 bean 的所有实例都是等价的，这就允许 EJB 容器将一个实例分配到任意客户端。也就是说，一个无状态会话 bean 的状态适用于所有客户端。

由于无状态会话 bean 可以支持多个客户端，所以它们可以为需要大量客户端的应用提供更好的可扩展性。一般地，对于支持相同数目的客户端，应用需要的无状态会话 bean 要少于有状态会话 bean。

无状态会话 bean 可以实现 Web 服务，但是有状态会话 bean 不可以。

3.2.1.3 单例会话 bean

单例会话 bean 在每个应用中只实例化一次，而且只在应用的生命周期期间存在。单例会话 bean 是为以下情况设计的，即一个企业 bean 实例会由多个客户端共享，并由它们并发访问。

单例会话 bean 提供的功能与无状态会话 bean 类似，但是也有所不同，因为一个应用中只有一个单例会话 bean，这不同于无状态会话 bean 池，其中任意一个 bean 都可以响应客户端请求。类似于无状态会话 bean，单例会话 bean 可以实现 Web 服务端点。

单例会话 bean 会维持客户端调用之间的状态，不过不要求在服务器崩溃或宕机后仍维持状态。

使用单例会话bean的应用可以指定应用一旦启动就实例化单例bean，这就允许这个单例bean为应用完成初始化任务。单例bean还可以在应用关闭时完成清理任务，因为这个单例将在应用生命周期的整个过程中完成操作。

3.2.2 何时使用会话bean

如果满足以下任意一个条件，就适合使用有状态会话bean。

- bean的状态表示bean与一个特定客户端之间的交互。
- bean需要跨方法调用保存一个客户端的有关信息。
- bean在客户端和应用其他组件之间协调，为客户端呈现一个简化的视图。
- 在后台，bean管理多个企业bean的工作流。

为了改善性能，可以选择无状态会话bean，它有以下特性：

- bean的状态不包括某个特定客户端的数据。
- 在一个方法调用中，bean完成所有客户端的一个通用任务。例如，可以使用一个无状态会话bean发送一个email来确认一个在线订单。
- bean实现了一个Web服务。

单例会话bean适用于以下情况。

- 需要在整个应用中共享状态。
- 多个线程会并发访问一个企业bean。
- 企业bean要在应用启动和关闭时完成任务。
- bean实现一个Web服务。

3.3 什么是消息驱动bean

消息驱动bean是允许Java EE应用异步处理消息的企业bean。这种bean通常作为一个JMS消息监听器，这类似于事件监听器，不过会接收JMS消息而不是事件。消息可能由任何Java EE组件发送（应用客户端、另一个企业bean或Web组件），或者由JMS应用或未使用Java EE技术的系统发送。消息驱动bean可以处理JMS消息或其他类型的消息。

3.3.1 消息驱动bean与会话bean的区别

消息驱动bean和会话bean之间最明显的差异是客户端不通过接口访问消息驱动bean。接口在3.4节中介绍。与会话bean不同，消息驱动bean只有一个bean类。

在很多方面，消息驱动bean很像是一个无状态会话bean。

- 消息驱动bean的实例不保留特定客户端的数据或会话状态。
- 消息驱动bean的所有实例都是等价的，这允许EJB容器将一个消息分配至任意的消息驱动bean实例。容器可以将这些实例入池，从而并发地处理消息流。

- 单个消息驱动 bean 可以处理来自多个客户端的消息。

消息驱动 bean 实例的实例变量可以包含处理不同客户端消息期间的状态，如 JMS API 连接、打开的数据库连接或一个企业 bean 对象的对象引用。

客户端组件不会查找消息驱动 bean 并直接调用其方法。实际上，客户端会通过其他方式（例如，通过 JMS）来访问消息驱动 bean，将消息发送到消息目的地（其消息驱动 bean 类是 `MessageListener`）。可以在部署时使用 GlassFish 服务器资源指定消息驱动 bean 的目的地。

消息驱动 bean 有以下特点

- 接收到一个客户端消息时才执行
- 异步调用
- 生存期相当短
- 不表示数据库中直接共享的数据，不过可以访问和更新这些数据
- 具有事务性
- 无状态

一个消息到达时，容器调用消息驱动 bean 的 `onMessage` 方法来处理这个消息。正常情况下，`onMessage` 方法会把这个消息强制转换为 5 种 JMS 消息类型之一，并根据应用的业务逻辑进行处理。`onMessage` 方法可以调用辅助方法，或者可以调用一个会话 bean 来处理消息中的信息，也可以把它存储到一个数据库中。

消息可以传送到一个事务上下文中的一个消息驱动 bean，所以 `onMessage` 方法中的所有操作都作为事务的一部分。如果消息处理回滚，这个消息就会重新传送。有关的更多信息参见 17.6 节和第 22 章。

3.3.2 何时使用消息驱动 bean

会话 bean 允许发送 JMS 消息并同步地接收，而不能异步接收。为了避免对服务器资源的绑定，不要使用服务器端组件中的阻塞式同步接收；一般地，JMS 消息不应同步发送或接收。要异步地接收消息，可以使用消息驱动 bean。

3.4 访问企业 bean

说明 这一节的内容只适用于会话 bean 而不适用于消息驱动 bean。因为它们的编程模型不同，所以消息驱动 bean 没有接口或定义客户端访问的非接口视图。

客户端要访问企业 bean，可以通过一个非接口视图，或者通过一个业务接口。企业 bean 的**非接口视图**（no-interface view）为客户端提供企业 bean 实现类的公共方法。使用企业 bean

非接口视图的客户端可以调用这个企业bean实现类或该实现类的任何超类的所有公共方法。**业务接口**（business interface）是一个标准的Java编程语言接口，包含企业bean的业务方法。

客户端只能通过bean业务接口中定义的方法或者通过有非接口视图的企业bean的公共方法来访问会话bean。业务接口或非接口视图定义了企业bean的客户端视图。企业bean的所有其他方面（方法实现和部署设置）都对客户端隐藏。

良好设计的接口和非接口视图可以简化Java EE应用的开发和维护。清晰的接口和非接口视图不仅可以对客户端隐藏EJB层的复杂性，还允许企业bean在内部改变而不会影响客户端。例如，如果修改了一个会话bean业务方法的实现，则不需要为此修改客户端代码。不过，如果要改变接口中的方法定义，可能还需要修改客户端代码。因此，一定要仔细设计接口和非接口视图，使客户端与企业bean中的修改隔离，这一点很重要。

会话bean可以有多个业务接口。会话bean应实现其业务接口（不过这一点不要求）。

3.4.1 在客户端使用企业bean

企业bean的客户端包含企业bean实例的一个引用，可能通过**依赖注入**来包含（使用Java编程语言注解）；也可能通过**JNDI查找**来包含（使用Java命令和目录接口语法查找企业bean实例）。

依赖注入是得到一个企业bean引用的最简单的方法。在Java EE服务器托管环境中运行的客户端、JSF Web应用、JAX-RS Web服务、其他企业bean或Java EE应用客户端都支持使用`javax.ejb.EJB`注解实现依赖注入。

在Java EE服务器托管环境之外运行的应用（如Java SE应用）必须完成一个显式的查找。JNDI支持一个全局语法，可以识别Java EE组件来简化这个显式查找。

3.4.1.1 可移植JNDI语法

有3个JNDI命名空间用于可移植JNDI查找：`java:global`、`java:module`和`java:app`。

- `java:global` JNDI命名空间提供了一种可移植的方式，可以使用JNDI查找（JNDI lookup）来寻找远程企业bean。JNDI地址有以下形式：

  `java:global[/`*应用名*`]/`*模块名*`/`*企业bean名*`[/`*接口名*`]`

这里的“应用名”和“模块名”默认为去掉文件扩展名的应用和模块名。只有当应用打包在一个EAR中才需要有应用名。只有当企业bean实现了多个业务接口才需要有接口名。

- `java:module`命名空间用于查找打包在同一个模块中的本地企业bean。使用`java:module`命名空间的JNDI地址采用以下形式：

  `java:module/`*企业bean名*`/[`*接口名*`]`

只有当企业bean实现了多个业务接口才需要有接口名。

- `java:app` 命名空间用于查找打包在同一个应用中的本地企业 bean。也就是说，企业 bean 打包在包含多个 Java EE 模块的 EAR 文件中。使用 `java:app` 命名空间的 JNDI 地址采用以下形式：

  `java:app[/模块名]/企业 bean 名[/接口名]`

模块名是可选的。只有当企业 bean 实现了多个业务接口才需要有接口名。

例如，如果一个企业 bean（`MyBean`）打包在 Web 应用归档 `myApp.war` 中，模块名为 `myApp`。可移植 JNDI 名为 `java:module/MyBean`。使用 `java:global` 命名空间的等价 JNDI 名是 `java:global/myApp/MyBean`。

3.4.2 确定远程或本地访问

设计 Java EE 应用时，首先要确定的一点是企业 bean 允许的客户端访问类型：远程、本地还是 Web 服务。

允许本地还是远程访问取决于以下因素：

- **相关 bean 的紧耦合或松耦合**：紧耦合的 bean 会相互依赖。例如，如果一个处理销售订单的会话 bean 调用另一个会话 bean，它通过 email 将一个确认消息发送给顾客，这两个 bean 就是紧耦合的。紧耦合的 bean 最适合选择本地访问。因为它们联合在一起构成一个逻辑单元，通常会经常相互调用，如果选择本地访问，就能获得由此带来的性能提升。
- **客户端类型**：如果一个企业 bean 会由应用客户端访问，就应当允许远程访问。在生产环境中，客户端几乎总是在运行 GlassFish 服务器以外的其他机器上运行。如果一个企业 bean 的客户端是 Web 组件或其他企业 bean，则访问类型取决于你希望组件如何分布。
- **组件分布**：Java EE 应用是可扩展的，因为服务器端组件可以分布在多个机器上。例如，在一个分布式应用中，可能会在一个服务器上运行 Web 组件，而在另一个服务器上部署它们所要访问的企业 bean。在这种分布式情况下，企业 bean 应当允许远程访问。
- **性能**：由于网络延迟等因素，远程调用可能比本地调用要慢。另一方面，如果将组件分布在不同的服务器上，可能会改善应用的总体性能。这些都是一般性的说法；在不同的操作环境中性能可能会有变化。不过一定要记住，你的应用设计可能会影响性能。

如果不确定企业 bean 应当采用哪一种访问类型，就选择远程访问。这会提供更大的灵活性。将来，你可以分布组件来适应应用不断增长的需求。

虽然不太常见，但是企业 bean 确实可以同时允许远程和本地访问。如果是这种情况，必须显式地指定（为 bean 的业务接口装饰 `@Remote` 或 `@Local` 注解），或者 bean 类必须显式地指定业务接口（使用 `@Remote` 和 `@Local` 注解）。同一个业务接口不能同时作为本地和远程业务接口。

3.4.3 本地客户端

本地客户端有以下特点：

- 必须在它访问的企业 bean 所在的同一个应用中运行。
- 可以是一个 Web 组件或另一个企业 bean。
- 对于本地客户端，它访问的企业 bean 的位置不是透明的。

企业 bean 的非接口视图是一个本地视图。企业 bean 实现类的公共方法会提供给访问企业 bean 非接口视图的本地客户端。使用非接口视图的企业 bean 不实现业务接口。

本地业务接口（local business interface）定义了 bean 的业务和生命期方法。如果 bean 的业务接口没有装饰 `@Local` 或 `@Remote`，而且如果 bean 类没有使用 `@Local` 或 `@Remote` 来指定接口，则业务接口默认为本地接口。

要构建一个只允许本地访问的企业 bean，可以（但不要求）采用以下任意一种做法：

- 创建一个不实现业务接口的企业 bean 实现类，指示这个 bean 为客户端提供一个非接口视图。例如：

```
@Session
public class MyBean { ... }
```

- 用注解指示企业 bean 的业务接口为 `@Local` 接口。例如：

```
@Local
public interface InterfaceName { ... }
```

- 用 `@Local` 装饰 bean 类来指定接口，并指定接口名。例如：

```
@Local(InterfaceName.class)
public class BeanName implements InterfaceName  { ... }
```

3.4.3.1 使用非接口视图访问本地企业 bean

客户端要访问一个提供了本地非接口视图的企业 bean，可以通过依赖注入或 JNDI 查找来完成。

- 要通过依赖注入得到一个企业 bean 非接口视图的引用，可以使用 `javax.ejb.EJB` 注解，并指定企业 bean 的实现类：

```
@EJB
ExampleBean exampleBean;
```

- 要通过 JNDI 查找得到一个企业 bean 非接口视图的引用，可以使用 `javax.naming.InitialContext` 接口的 `lookup` 方法：

```
ExampleBean exampleBean = (ExampleBean)
        InitialContext.lookup("java:module/ExampleBean");
```

对于使用非接口视图的企业 bean，客户端不使用 `new` 操作符来得到这个 bean 的新实例。

3.4.3.2 访问实现业务接口的本地企业 bean

客户端要访问一个实现本地业务接口的企业 bean，可以通过依赖注入或 JNDI 查找来完成。

- 要通过依赖注入得到一个企业 bean 本地业务接口的引用，可以使用 javax.ejb.EJB 注解，并指定企业 bean 的本地业务接口名：

```
@EJB
Example example;
```

- 要通过 JNDI 查找得到一个企业 bean 本地业务接口的引用，可以使用 javax.naming.InitialContext 接口的 lookup 方法：

```
ExampleLocal example = (ExampleLocal)
        InitialContext.lookup("java:module/ExampleLocal");
```

3.4.4 远程客户端

企业 bean 的远程客户端有以下特性：

- 可以与它访问的企业 bean 在不同的机器和不同的 JVM 上运行（但不要求在不同的 JVM 上运行）。
- 可以是一个 Web 组件、应用客户端或另一个企业 bean。
- 对于远程客户端，企业 bean 的位置是透明的。
- 企业 bean 必须实现一个业务接口。也就是说，远程客户端不能通过非接口视图来访问企业 bean。

要创建一个允许远程访问的企业 bean，必须采用以下做法：

- 用 @Remote 注解装饰企业 bean 的业务接口：

```
@Remote
public interface InterfaceName { ... }
```

- 或者用 @Remote 装饰 bean 类，指定一个业务接口（或多个业务接口）：

```
@Remote(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

远程接口（remote interface）定义这个 bean 特定的业务方法和生命期方法。例如，名为 BankAccountBean 的 bean 的远程接口中，可能包含名为 deposit 和 credit 的业务方法。图 3-1 显示了接口如何控制一个企业 bean 的客户端视图。

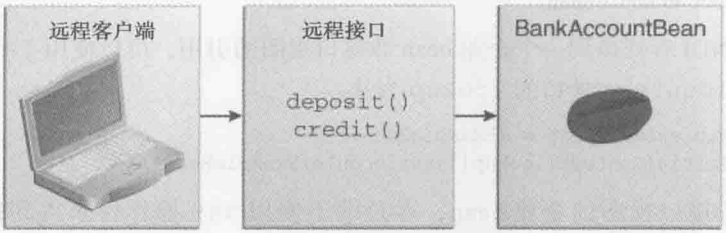


图 3-1 远程访问的企业 bean 的接口

客户端要访问一个实现远程业务接口的企业bean，可以通过依赖注入或JNDI查找来完成。

- 要通过依赖注入得到一个企业bean远程业务接口的引用，可以使用javax.ejb.EJB注解，并指定企业bean的远程业务接口名：

```
@EJB
Example example;
```

- 要通过JNDI查找得到一个企业bean远程业务接口的引用，可以使用javax.naming.InitialContext接口的lookup方法：

```
ExampleRemote example = (ExampleRemote)
        InitialContext.lookup("java:global/myApp/ExampleRemote");
```

3.4.5 Web服务客户端

Web服务客户端可以采用两种方式访问Java EE应用。首先，客户端可以访问用JAX-WS创建的Web服务（关于JAX-WS的更多信息，参见本书第1卷第28章）。其次，Web服务客户端可以调用一个无状态会话bean的业务方法。Web服务客户端不能访问消息bean。

假设使用了正确的协议（SOAP、HTTP、WSDL），则任何Web服务客户端都可以访问无状态会话bean，而不论这个客户端是否用Java编程语言编写。客户端甚至不“知道”由哪种技术实现这个服务：可能是无状态会话bean、JAX-WS或另外某种技术。另外，企业bean和Web组件可以是Web服务的客户端。基于这种灵活性，允许你集成Java EE应用和Web服务。

Web服务客户端要通过一个无状态会话bean的Web服务端点实现类访问这个bean。默认地，Web服务客户端可以访问bean类中的所有公共方法。@WebMethod注解可以用来定制Web服务方法的行为。如果使用了@WebMethod注解装饰bean类的方法，那么只有用@WebMethod装饰的那些方法会提供给Web服务客户端。

有关的代码示例，请参见5.3节。

3.4.6 方法参数和访问

访问类型会影响客户端调用的bean方法的参数。下面各小节不仅适用于方法参数，也适用于方法返回值。

3.4.6.1 隔离

远程调用的参数比本地调用的参数更为隔离。对于远程调用，客户端和bean会处理参数对象的不同副本。如果客户端改变了对象的值，则bean中副本的值并不改变。客户端不小心修改了数据时，这个隔离层有助于保护bean。

在本地调用中，客户端和bean可以修改同一个参数对象。一般地，不能依赖本地调用的这种副作用。可能将来你会希望分布部署组件，把本地调用替换为远程调用。

与远程客户端一样，Web服务客户端与实现Web服务的bean也会处理不同的参数副本。

3.4.6.2 访问数据的粒度

由于远程调用很可能比本地调用要慢，所以远程方法中的参数要相对粗粒度。粗粒度对象比细粒度对象包含更多的数据，所以需要更少的访问调用。出于同样的原因，Web 服务客户端调用的方法参数也应当是粗粒度的。

3.5 企业 bean 的内容

要开发一个企业 bean，必须提供以下文件：

- **企业 bean 类**：实现企业 bean 的业务方法以及生命周期回调方法。
- **业务接口**：定义企业 bean 类实现的业务方法。如果企业 bean 提供了一个本地非接口视图，则不必提供业务接口。
- **辅助类**：企业 bean 类需要的其他类，如异常和工具类。

将以上所列的编程工件打包到 EJB JAR 文件（存储企业 bean 的独立模块）或者打包到一个 Web 应用归档（WAR）模块。更多信息参见本书第 1 卷中的 5.2.1 节和 5.2.2 节。

3.6 企业 bean 的命名约定

由于企业 bean 由多个部分组成，因此遵循应用的命名约定很有用。表 3-2 总结了本教程中示例 bean 的命名约定。

表 3-2 企业 bean 的命名约定

项	语 法	示 例
企业 bean 名	*name*Bean	AccountBean
企业 bean 类	*name*Bean	AccountBean
业务接口	*name*	Account

3.7 企业 bean 的生命周期

企业 bean 在其生命期或生命周期中会经过多个阶段。每个类型的企业 bean（有状态会话、无状态会话、单例会话或消息驱动）都有不同的生命周期。

下面的描述会提到一些方法，这些方法将在下面两章的代码示例中详细解释。如果你还不了解企业 bean，可以跳过这一节，先运行那些代码示例。

3.7.1 有状态会话 bean 的生命周期

图 3-2 展示了一个有状态会话 bean 在其生命期中经历的不同阶段。首先客户端获得一个有状态会话 bean 的引用，由此开始生命周期。容器完成依赖注入（如果有），然后调用有

@PostConstruct 注解的方法（如果有）。现在 bean 已经准备就绪，客户端可以调用它的业务方法了。

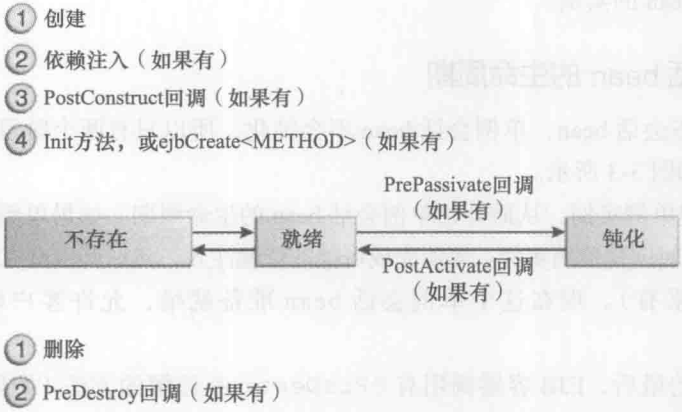


图 3-2 有状态会话 bean 的生命周期

在就绪阶段，EJB 容器可能决定将 bean 解除激活或**钝化**（passivate），把它从内存移至辅存。（一般地，EJB 容器使用一种最近最少使用算法来选择要钝化的 bean）。EJB 容器在钝化前会调用有 @PrePassivate 注解的方法（如果有）。如果一个 bean 正处于钝化阶段，而客户端调用了它的业务方法，则 EJB 容器会激活这个 bean，调用有 @PostActivate 注解的方法（如果有），然后把它移至就绪阶段。

在生命周期的最后，客户端调用一个有 @Remove 注解的方法，EJB 容器调用有 @PreDestroy 注解的方法（如果有）。然后就可以垃圾回收这个 bean 的实例了。

你编写的代码只控制一个生命周期方法调用：有 @Remove 注解的方法。图 3-2 中的所有其他方法都由 EJB 容器调用。有关的更多信息参见第 23 章。

3.7.2 无状态会话 bean 的生命周期

由于无状态会话 bean 永远不会钝化，所以它的生命周期只有两个阶段：不存在（nonexistent）和准备调用业务方法。图 3-3 展示了一个无状态会话 bean 的阶段。

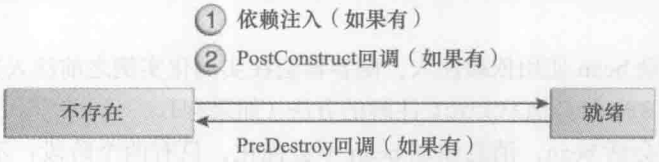


图 3-3 无状态或单例会话 bean 的生命周期

EJB 容器通常会创建并维护一个无状态会话 bean 池，从而开始无状态会话 bean 的生命周期。容器会完成依赖注入（如果有），然后调用有 @PostConstruct 注解的方法（如果存

在)。现在 bean 已经准备就绪，允许客户端调用它的业务方法。

在生命周期的最后，EJB 容器调用有 @PreDestroy 注解的方法（如果存在）。然后可以垃圾回收这个 bean 的实例。

3.7.3 单例会话 bean 的生命周期

类似于无状态会话 bean，单例会话 bean 不会钝化，所以只有两个阶段：不存在和准备调用业务方法，如图 3-3 所示。

EJB 容器创建单例实例，从而开始单例会话 bean 的生命周期。如果单例有 @Startup 注解，会在应用部署时创建单例实例。容器完成可能的依赖注入，然后调用有 @PostConstruct 注解的方法（如果有）。现在这个单例会话 bean 准备就绪，允许客户端调用它的业务方法。

在生命周期的最后，EJB 容器调用有 @PreDestroy 注解的方法（如果存在）。现在可以垃圾回收这个单例会话 bean。

3.7.4 消息驱动 bean 的生命周期

图 3-4 展示了消息驱动 bean 生命周期中的各个阶段。

图 3-4 消息驱动 bean 的生命周期

EJB 容器通常会创建一个消息驱动 bean 实例池。对于每个实例，EJB 容器会完成以下任务。

1. 如果消息驱动 bean 使用依赖注入，则容器会在实例化实例之前注入这些引用。

2. 容器调用有 @PostConstruct 注解的方法（如果有）。

类似于无状态会话 bean，消息驱动 bean 不会钝化，只有两个阶段：不存在和准备接收消息。

在生命周期的最后，容器调用有 @PreDestroy 注解的方法（如果有）。现在可以垃圾回收这个 bean 实例。

3.8 有关企业bean的更多信息

关于EJB技术的更多信息，参见：

- Enterprise JavaBeans 3.2规范：

  `http://www.jcp.org/en/jsr/detail?id=345`

- Enterprise JavaBeans 3.2规范项目：

  `https://java.net/projects/ejb-spec/`

Chapter 4

第 4 章

企业 bean 入门

这一章介绍如何开发、部署和运行一个简单的 Java EE 应用 converter，这个应用使用一个 EJB 来实现业务逻辑。converter 的作用是计算日元、欧元和美元之间的汇率。converter 应用包括一个完成计算的企业 bean 和一个 Web 客户端。

下面给出这一章的大致步骤：

1. 创建企业 bean:ConverterBean。
2. 创建 Web 客户端。
3. 在服务器上部署 converter。
4. 使用浏览器运行 Web 客户端。

在继续读下面的内容之前，一定要确保已经完成了以下工作：

- 已经阅读第 1 章
- 熟悉企业 bean（参见第 3 章）
- 已经启动服务器（参见 2.2 节）

本章将介绍以下内容：

- 创建企业 bean
- 修改 Java EE 应用

4.1 创建企业 bean

这个例子中的企业 bean 是一个无状态会话 bean，名为 ConverterBean。ConverterBean 的源代码位于 *tut-install*/examples/ejb/converter/src/main/java/ 目录。

创建 ConverterBean 需要以下步骤：

1. 编写 bean 的实现类（已提供源代码）。
2. 编译源代码。

4.1.1 编写企业 bean 类

这个例子的企业 bean 类名为 ConverterBean。这个类实现了两个业务方法：dollarToYen 和 yenToEuro。由于这个企业 bean 类没有实现业务接口，因此它提供了一个本地非接口视图。得到 ConverterBean 引用的客户端都可以使用这个企业 bean 类中的公共方法。ConverterBean 类的源代码如下所示：

```
package javaeetutorial.converter.ejb;

import java.math.BigDecimal;
import javax.ejb.Stateless;

@Stateless
public class ConverterBean {
    private final BigDecimal yenRate = new BigDecimal("83.0602");
    private final BigDecimal euroRate = new BigDecimal("0.0093016");

    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }

    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = yen.multiply(euroRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
```

需要注意装饰企业 bean 类的 @Stateless 注解。这个注解使容器知道 ConverterBean 是一个无状态会话 bean。

4.1.2 创建 converter Web 客户端

Web 客户端包含在以下 servlet 类（位于 *tut-install*/examples/ejb/converter/src/main/java/ 目录）：

```
converter/web/ConverterServlet.java
```

Java servlet 是一个响应 HTTP 请求的 Web 组件。

ConverterServlet 类使用依赖注入得到 ConverterBean 的一个引用。这里对私有成员变量 converter 增加了 javax.ejb.EJB 注解，这个成员变量的类型为 ConverterBean。ConverterBean 提供了一个本地非接口视图，所以企业 bean 实现类也使用这个变量类型：

```
@WebServlet(urlPatterns="/")
public class ConverterServlet extends HttpServlet {
  @EJB
  ConverterBean converter;
  ...
}
```

用户输入转换成日元和欧元的金额时，会从请求参数获取数额，然后调用 ConverterBean.dollarToYen 和 ConverterBean.yenToEuro 方法：

```
...
try {
  String amount = request.getParameter("amount");
  if (amount != null && amount.length()> 0) {
    // convert the amount to a BigDecimal from the request parameter
    BigDecimal d = new BigDecimal(amount);
    // call the ConverterBean.dollarToYen() method to get the amount
    // in Yen
    BigDecimal yenAmount = converter.dollarToYen(d);

    // call the ConverterBean.yenToEuro() method to get the amount
    // in Euros
    BigDecimal euroAmount = converter.yenToEuro(yenAmount);
    ...
  }
  ...
}
```

然后向用户显示结果。

4.1.3 运行 converter 示例

现在可以编译这个企业 bean 类（ConverterBean.java）和 servlet 类（ConverterServlet.java），并把已编译的类打包到一个 WAR 文件。可以使用 NetBeans IDE 或 Maven 构建、打包、部署和运行 converter 示例。

4.1.3.1 使用 NetBeans IDE 运行 converter 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中导航至：

tut-install/examples/ejb

4. 选择 converter 文件夹。
5. 单击 Open Project。
6. 在 Projects 标签页中，右键单击 converter 工程并选择 Build（构建）。
7. 打开 Web 浏览器访问以下 URL：

http://localhost:8080/converter

8. 在 Servlet ConverterServlet 页面上，在输入域中输入 100，并单击 Submit（提交）。这会打开第二个页面，显示转换后的值。

4.1.3.2　使用 Maven 运行 converter 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 在一个终端窗口中导航至：

```
tut-install/examples/ejb/converter/
```

3. 输入以下命令：

```
mvn install
```

这个命令会编译企业 bean 和 servlet 的源文件，将工程打包到一个 WAR 模块（converter.war），并将这个 WAR 部署到服务器。有关 Maven 的更多信息参见 2.5 节。

4. 打开 Web 浏览器访问以下 URL：

```
http://localhost:8080/converter
```

5. 在 Servlet ConverterServlet 页面上，在输入域中输入 100，并单击 Submit（提交）。这会打开第二个页面，显示转换后的值。

4.2　修改 Java EE 应用

GlassFish 服务器支持迭代开发。只要对 Java EE 应用做出修改，就必须重新部署应用。

4.2.1　修改类文件

要修改企业 bean 中的一个类文件，需要修改源代码，重新编译，再重新部署这个应用。例如，要更新 ConverterBean 类的 dollarToYen 业务方法中的汇率，可以采用以下步骤。

修改 ConverterServlet 的过程是一样的。

1. 编辑 ConverterBean.java 并保存文件。
2. 重新编译源文件。

- 要在 NetBeans IDE 中重新编译 ConverterBean.java，右键单击 converter 工程并选择 Run（运行）。

这会重新编译 ConverterBean.java 文件，替换构建目录中原来的类文件，并把应用重新部署到 GlassFish 服务器。

- 使用 Maven 重新编译 ConverterBean.java。
  a. 在一个终端窗口中，进入 *tut-install*/examples/ejb/converter/ 目录。
  b. 输入以下命令：

```
mvn install
```

这个命令会重新打包和部署应用。

Chapter 5 第 5 章

运行企业 bean 示例

本章介绍一些 EJB 示例。会话 bean 提供了一种简单而强大的方式来封装应用中的业务逻辑。可以从远程 Java 客户端、Web 服务客户端以及同一个服务器中运行的其他组件访问这些会话 bean。

在第 4 章中，你已经构建了一个名为 ConverterBean 的无状态会话 bean。本章将分析另外 4 个会话 bean 的源代码：

- CartBean：这是一个有状态会话 bean，将由一个远程客户端访问。
- CounterBean：这是一个单例会话 bean。
- HelloServiceBean：这是一个无状态会话 bean，实现了一个 Web 服务。
- TimerSessionBean：这是一个无状态会话 bean，用来设置一个定时器。

本章将介绍以下内容：

- cart 示例
- 一个单例会话 bean 示例：counter
- 一个 Web 服务示例：helloservice
- 使用定时器服务
- 处理异常

5.1 cart 示例

cart 示例表示在线商店里的一个购物车，使用一个有状态会话 bean 管理这个购物车的操作。这个 bean 的客户端可以向购物车中增加一本书，删除一本书，或者可以获取购物车中

的内容。装配这个 cart 需要以下代码：

- 会话 bean 类（CartBean）
- 远程业务接口（Cart）

所有会话 bean 都需要一个会话 bean 类。所有允许远程访问的企业 bean 都必须有一个远程业务接口。要满足一个特定应用的需要，企业 bean 可能还需要一些辅助类。CartBean 会话 bean 使用两个辅助类 BookException 和 IdVerifier，这两个类将在 5.1.4 节中介绍。

这个例子的源代码位于 *tut-install*/examples/ejb/cart/ 目录。

5.1.1 业务接口

Cart 业务接口是一个纯 Java 接口，定义了 bean 类中实现的所有业务方法。如果这个 bean 类实现了一个接口，则认为这个接口是一个业务接口。业务接口是一个本地接口，除非有 javax.ejb.Remote 注解。因此作为一个本地接口，javax.ejb.Local 注解是可选的。

bean 类可以实现多个接口。在这种情况下，业务接口必须显式地增加 @Local 或 @Remote 注解，或者使用 @Local 或 @Remote 装饰 bean 类来指定。不过，确定 bean 类是否实现多个接口时，不考虑以下接口：

- java.io.Serializable
- java.io.Externalizable
- javax.ejb 包中定义的所有接口

Cart 业务接口的源代码如下所示：

```
package javaeetutorial.cart.ejb;

import cart.util.BookException;
import java.util.List;
import javax.ejb.Remote;

@Remote
public interface Cart {
    public void initialize(String person) throws BookException;
    public void initialize(String person, String id) throws BookException;
    public void addBook(String title);
    public void removeBook(String title) throws BookException;
    public List<String> getContents();
    public void remove();
}
```

5.1.2 会话 bean 类

这个例子的会话 bean 类名为 CartBean。与所有有状态会话 bean 类似，CartBean 类必须满足以下需求：

- 这个类必须有 @Stateful 注解。
- 这个类要实现业务接口中定义的业务方法。

有状态会话 bean 可能还要完成以下工作：

- 实现业务接口，这是一个纯 Java 接口。实现 bean 的业务接口是一个很好的做法。
- 实现可选的生命周期回调方法（这些方法有 @PostConstruct、@PreDestroy、@PostActivate 和 @PrePassivate 注解）。
- 实现有 @Remove 注解的业务方法（可选）。

CartBean 类的源代码如下所示：

```
package javaeetutorial.cart.ejb;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javaeetutorial.cart.util.BookException;
import javaeetutorial.cart.util.IdVerifier;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful
public class CartBean implements Cart {
    String customerId;
    String customerName;
    List<String> contents;

    @Override
    public void initialize(String person) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }
        customerId = "0";
        contents = new ArrayList<>();
    }

    @Override
    public void initialize(String person, String id)
                throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }

        IdVerifier idChecker = new IdVerifier();
        if (idChecker.validate(id)) {
```

```
            customerId = id;
        } else {
            throw new BookException("Invalid id: " + id);
        }
        contents = new ArrayList<>();
    }

    @Override
    public void addBook(String title) {
        contents.add(title);
    }

    @Override
    public void removeBook(String title) throws BookException {
        boolean result = contents.remove(title);
        if (result == false) {
            throw new BookException("\"" + title + " not in cart.");
        }
    }

    @Override
    public List<String> getContents() {
        return contents;
    }

    @Remove
    @Override
    public void remove() {
        contents = null;
    }
}
```

5.1.2.1　生命周期回调方法

通过增加以下注解，可以将 bean 类中的方法声明为一个生命周期回调方法：

- `javax.annotation.PostConstruct`：完成所有依赖注入之后，而且在对企业 bean 调用第一个业务方法之前，容器会对新构建的 bean 实例调用有 `@PostConstruct` 注解的方法。
- `javax.annotation.PreDestroy`：完成有 `@Remove` 注解的方法之后而且在容器删除企业 bean 实例之前，会调用有 `@PreDestroy` 注解的方法。
- `javax.ejb.PostActivate`：容器将 bean 从辅存转移到活动状态之后，会调用有 `@PostActivate` 注解的方法。
- `javax.ejb.PrePassivate`：容器钝化企业 bean 之前会调用有 `@PrePassivate` 注解的方法，这说明容器会临时从环境中删除这个 bean，并把它保存到辅存。

生命周期回调方法必须返回 `void`，而且不能有参数。

5.1.2.2 业务方法

会话 bean 的主要作用就是为客户端运行业务任务。客户端由依赖注入或 JNDI 查找得到对象引用后，会调用对象引用上的业务方法。从客户端的角度来讲，业务方法看起来就在本地运行，尽管实际上它们是在会话 bean 中远程运行的。下面的代码段显示了 CartClient 程序如何调用业务方法：

```
cart.initialize("Duke DeEarl", "123");
...
cart.addBook("Bel Canto");
...
List<String> bookList = cart.getContents();
...
cart.removeBook("Gravity's Rainbow");
```

CartBean 类用以下代码实现了这个业务方法：

```
@Override
public void addBook(String title) {
   contents.add(title);
}
@Override
public void removeBook(String title) throws BookException {
   boolean result = contents.remove(title);
   if (result == false) {
      throw new BookException("\"" + title + "not in cart.");
   }
}

@Override
public List<String> getContents() {
   return contents;
}
```

业务方法的签名必须符合以下规则：

- 方法名不能以 ejb 开头，以避免与 EJB 架构定义的回调方法冲突。例如，业务方法名不能为 ejbCreate 或 ejbActivate。
- 访问控制修饰符必须是 public。
- 如果 bean 允许通过一个远程业务接口完成远程访问，则参数和返回类型必须是 Java 远程方法调用（RMI）API 允许的合法类型。
- 如果 bean 是一个 JAX-WS Web 服务端点，则有 @WebMethod 注解的方法的参数和返回类型必须是 JAX-WS 允许的合法类型。
- 如果 bean 是一个 JAX-RS 资源，则资源方法的参数和返回类型必须是 JAX-RS 允许的合法类型。
- 修饰符不能是 static 或 final。

throws 子句可以包含为应用定义的异常。例如，如果一本书不在购物车中，则删除这

本书的 removeBook 方法会抛出一个 BookException 异常。

要指示一个系统级问题，如无法连接到一个数据库，业务方法要抛出一个 javax.ejb.EJBException 异常。容器不会包装应用异常，如 BookException。由于 EJBException 是 RuntimeException 的一个子类，所以不需要在业务方法的 throws 子句中包含这个异常。

5.1.3 @Remove 方法

有状态会话 bean 类中如果包含增加了 javax.ejb.Remove 注解的业务方法，则企业 bean 客户端可以调用这个 @Remove 方法删除 bean 实例。容器将在 @Remove 方法完成之后删除企业 bean（这包括正常完成和非正常完成）。

在 CartBean 中，remove 方法是一个 @Remove 方法：

```
@Remove
@Override
public void remove() {
    contents = null;
}
```

5.1.4 辅助类

CartBean 会话 bean 有两个辅助类：BookException 和 IdVerifier。BookException 由 removeBook 方法抛出，IdVerifier 在某个 create 方法中验证 customerId。辅助类可以放在包含企业 bean 类的 EJB JAR 文件中；如果企业 bean 打包在一个 WAR 中，辅助类也可以放在 WAR 文件中；或者可以在一个包含 EJB JAR 的 EAR 文件、WAR 文件或单独的库 JAR 文件中。在 cart 中，这两个辅助类包含在应用客户端和 EJB JAR 使用的一个库 JAR 中。

5.1.5 运行 cart 示例

现在可以编译远程接口（Cart.java）、企业 bean 类（CartBean.java）、客户端类（CartClient.java）和辅助类（BookException.java 和 IdVerifier.java）。

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 cart 应用。

5.1.5.1 使用 NetBeans IDE 运行 cart 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

   tut-install/examples/ejb

4. 选择 cart 文件夹。

5. 选择 Open Required Projects（打开所需工程）复选框。

6. 单击 Open Project。

7. 在 Projects 标签页中，右键单击 cart 工程并选择 Build（构建）。

这会构建并将应用打包到 cart.ear，位于 *tut-install*/examples/ejb/cart/cart-ear/target/，并把这个 EAR 文件部署到 GlassFish 服务器实例。

可以在 Output（输出）标签页看到 cart-app-client 应用客户端的输出：

```
...
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
```

5.1.5.2 使用 Maven 运行 cart 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在一个终端窗口中导航至：

```
tut-install/examples/ejb/cart/
```

3. 输入以下命令：

```
mvn install
```

这个命令会编译并将应用打包到一个 EAR 文件 cart.ear，这个文件位于 target 目录，将这个 EAR 部署到 GlassFish 服务器实例。

然后，获取并运行客户端桩。这等价于运行以下命令：

```
appclient -client cart-ear/target/cart-earClient.jar
```

客户端 JAR cart-earClient.jar 包含应用客户端类、辅助类 BookException 和 Cart 业务接口。

运行客户端时，应用客户端容器会注入应用客户端类中声明的所有组件引用，在这里就是 Cart 企业 bean 的引用。

可以在终端窗口中看到 cart-app-client 应用客户端的输出：

```
...
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
```

5.2 一个单例会话 bean 示例：counter

counter 示例展示了如何创建一个单例会话 bean。

5.2.1　创建一个单例会话 bean

javax.ejb.Singleton 注解用于指定企业 bean 实现类是一个单例会话 bean：

```
@Singleton
public class SingletonBean { ... }
```

5.2.1.1　初始化单例会话 bean

EJB 容器负责确定何时初始化一个单例会话 bean 实例，除非单例会话 bean 实现类有 javax.ejb.Startup 注解。这种情况有时会称为**积极初始化**（eager initialization），EJB 容器在应用启动时必须初始化单例会话 bean。单例会话 bean 在 EJB 容器向应用中的企业 bean 发送请求之前初始化。这就允许单例会话 bean 完成一些工作，如应用的启动任务。

下面的单例会话 bean 会存储应用的状态，并完成积极初始化：

```
@Startup
@Singleton
public class StatusBean {
  private String status;

  @PostConstruct
  void init {
    status = "Ready";
  }
  ...
}
```

有时会用多个单例会话 bean 来初始化一个应用的数据，因此必须以一种特定的顺序初始化。在这些情况下，可以使用 javax.ejb.DependsOn 注解来声明单例会话 bean 的启动依赖关系。@DependsOn 注解的 value 属性是一个或多个字符串，指定了目标单例会话 bean 的名字。如果 @DependsOn 中指定了多个依赖的单例 bean，则 EJB 容器不一定会按目标单例会话 bean 列出的顺序来初始化这些 bean。

下面的单例会话 bean PrimaryBean 应当最先启动：

```
@Singleton
public class PrimaryBean { ... }
```

SecondaryBean 依赖于 PrimaryBean：

```
@Singleton
@DependsOn("PrimaryBean")
public class SecondaryBean { ... }
```

这样可以保证 EJB 容器在 SecondaryBean 之前先初始化 PrimaryBean。

下面的单例会话 bean TertiaryBean 依赖于 PrimaryBean 和 SecondaryBean：

```
@Singleton
@DependsOn({"PrimaryBean", "SecondaryBean"})
public class TertiaryBean { ... }
```

SecondaryBean 显式要求 PrimaryBean 在它之前初始化，这是通过它自己的 @DependsOn 注解指定的。在这种情况下，EJB 容器会首先初始化 PrimaryBean，再初始化 SecondaryBean，最后再初始化 TertiaryBean。

不过，如果 SecondaryBean 没有显式依赖于 PrimaryBean，则 EJB 容器可能会先初始化 PrimaryBean 或 SecondaryBean。也就是说，EJB 容器可能会以下面的顺序初始化这些单例 bean：SecondaryBean,PrimaryBean,TertiaryBean。

5.2.1.2 管理单例会话 bean 中的并发访问

单例会话 bean 设计用来实现**并发访问**，在这些情况下，多个客户端需要同时访问一个会话 bean 的单个实例。单例的客户端只需要一个单例引用来调用它提供的业务方法，而不需要担心其他客户端可能会同时调用同一个单例上的业务方法。

创建一个单例会话 bean 时，可以采用两种方式来控制对这个单例业务方法的并发访问：容器托管并发和 bean 托管并发。

javax.ejb.ConcurrencyManagement 注解用来指定单例的容器托管或 bean 托管并发。对于 @ConcurrencyManagement，必须将类型属性设置为 javax.ejb.ConcurrencyManagementType.CONTAINER 或 javax.ejb.ConcurrencyManagementType.BEAN。如果单例实现类上没有 @ConcurrencyManagement 注解，则会使用默认为容器托管并发的 EJB 容器。

容器托管并发

如果单例使用**容器托管并发**（container-managed concurrency），则 EJB 容器会控制对这个单例业务方法的客户端访问。可以使用 javax.ejb.Lock 注解和一个 javax.ejb.LockType 类型来指定单例业务方法或 @Timeout 方法的访问级别。LockType 枚举类型为 READ 和 WRITE。

如果一个单例的业务方法或超时方法允许由多个客户端并发地访问或共享，则可以对这个方法增加 @Lock(LockType.READ) 注解。客户端调用一个单例会话 bean 的业务方法或超时方法时，如果要对其他客户端锁定这个单例会话 bean，则为这个业务方法或超时方法增加 @Lock(LockType.WRITE) 注解。一般地，客户端修改单例的状态时要使用 @Lock(LockType.WRITE) 注解。

为单例类增加 @Lock 注解时，会指定这个单例的所有业务方法和超时方法将使用指定的锁类型，除非用一个方法级 @Lock 注解显式地设置了锁类型。如果单例类上没有 @Lock 注解，就会对所有业务方法和超时方法应用默认的锁类型 @Lock(LockType.WRITE)。

下面的例子显示了如何对一个使用容器托管并发的单例使用 @ConcurrencyManagement、@Lock(LockType.READ) 和 @Lock(LockType.WRITE) 注解。

尽管默认情况下单例使用容器托管并发，不过也可以在单例的类层次上增加 @ConcurrencyManagement(CONTAINER) 注解，以显式地设置并发管理类型：

```
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class ExampleSingletonBean {
  private String state;

  @Lock(LockType.READ)
  public String getState() { return state; }
  @Lock(LockType.WRITE)
  public void setState(String newState) { state = newState; }
}
```

多个客户端可以同时访问 getState 方法，因为它有 @Lock(LockType.READ) 注解。不过，调用 setState 方法时，ExampleSingletonBean 中的所有方法都对其他客户端锁定，因为 setState 有 @Lock(LockType.WRITE) 注解。这样可以避免两个客户端试图同时改变 ExampleSingletonBean 的 state 变量。

以下单例中的 getData 和 getStatus 方法的类型为 READ，setStatus 方法的类型为 WRITE：

```
@Singleton
@Lock(LockType.READ)
public class SharedSingletonBean {
  private String data;
  private String status;

  public String getData() {
    return data;
  }

  public String getStatus() {
    return status;
  }
  @Lock(LockType.WRITE)
  public void setStatus(String newStatus) {
    status = newStatus;
  }
}
```

如果一个方法的锁类型是 WRITE，则这个单例所有方法的客户端访问都会被阻塞，直到当前客户端完成其方法调用，或者出现访问超时。出现访问超时时，EJB 容器会抛出一个 javax.ejb.ConcurrentAccessTimeoutException。javax.ejb.AccessTimeout 注解用来指定出现访问超时之前所经过的时间（毫秒数）。如果在单例的类层次上增加这个注解，则 @AccessTimeout 将指定单例中所有方法的访问超时值，除非一个方法用自己的 @AccessTimeout 注解显式地覆盖这个默认值。

@AccessTimeout 注解可以应用到 @Lock(LockType.READ) 和 @Lock(LockType.WRITE) 方法。@AccessTimeout 注解有一个必不可少的元素 value，还有一个可选的元

素 unit。默认地，value 按毫秒数指定。要改变 Value 的单位，可以将 unit 设置为以下某个 java.util.concurrent.TimeUnit 常量：NANOSECONDS、MICROSECONDS、MILLISECONDS 或 SECONDS。

以下单例的默认访问超时值为 120 000 毫秒，即 2 分钟。doTediousOperation 方法覆盖了这个默认的访问超时值，将超时值设置为 360 000 毫秒，或 6 分钟：

```
@Singleton
@AccessTimeout(value=120000)
public class StatusSingletonBean {
  private String status;

  @Lock(LockType.WRITE)
  public void setStatus(String new Status) {
    status = newStatus;
  }

  @Lock(LockType.WRITE)
  @AccessTimeout(value=360000)
  public void doTediousOperation {
    ...
  }
}
```

下面的单例默认访问超时值为 60 秒，这里使用 TimeUnit.SECONDS 常量来指定：

```
@Singleton
@AccessTimeout(value=60, unit=TimeUnit.SECONDS)
public class StatusSingletonBean { ... }
```

bean 托管并发

使用 **bean 托管并发**（bean-managed concurrency）的单例允许并发访问单例中的所有业务和超时方法。单例的开发者要负责确保单例的状态对所有客户端是同步的。创建采用 bean 托管并发的单例时，开发者可以使用 Java 编程语言同步原语，如 synchronization 和 volatile，以避免并发访问中的错误。

如果在单例的类层次上增加一个 @ConcurrencyManagement 注解，而且类型设置为 ConcurrencyManagementType.BEAN，这就指定了 bean 托管并发：

```
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
@Singleton
public class AnotherSingletonBean { ... }
```

5.2.1.3 处理单例会话 bean 中的错误

如果 EJB 容器初始化单例会话 bean 时遇到一个错误，则会撤销这个单例实例。

与其他企业 bean 不同，一旦初始化一个单例会话 bean 实例，如果这个单例的业务方法或生命周期方法导致系统异常，单例实例便不会撤销。这样可以确保整个应用生命周期中使用同一个单例实例。

5.2.2　counter 示例的架构

counter 示例包括一个单例会话 bean CounterBean 以及一个 JSF Facelets Web 前端。

CounterBean 是一个简单的单例，只有一个方法 getHits，它返回一个整数，表示 Web 页面已访问的次数。下面是 CounterBean 的代码：

```
package javaeetutorial.counter.ejb;

import javax.ejb.Singleton;

/**
 * CounterBean is a simple singleton session bean that records the number
 * of hits to a web page.
 */
@Singleton
public class CounterBean {
    private int hits = 1;

    // Increment and return the number of hits
    public int getHits() {
        return hits++;
    }
}
```

@Singleton 注解标志 CounterBean 为一个单例会话 bean。CounterBean 使用一个本地非接口视图。

CounterBean 使用 EJB 容器用于单例的默认元数据值，以简化单例实现类的编写。这个类没有 @ConcurrencyManagement 注解，所以会应用默认的容器托管并发。类或业务方法上没有 @Lock 注解，所以会对仅有的业务方法 getHits 应用默认的 @Lock(WRITE)。

下面这个 CounterBean 版本在功能上等价于前面的版本：

```
package javaeetutorial.counter.ejb;

import javax.ejb.Singleton;
import javax.ejb.ConcurrencyManagement;
import static javax.ejb.ConcurrencyManagementType.CONTAINER;
import javax.ejb.Lock;
import javax.ejb.LockType.WRITE;

/**
 * CounterBean is a simple singleton session bean that records the number
 * of hits to a web page.
 */
@Singleton
@ConcurrencyManagement(CONTAINER)
public class CounterBean {
    private int hits = 1;

    // Increment and return the number of hits
```

```
    @Lock(WRITE)
    public int getHits() {
        return hits++;
    }
}
```

counter 的 Web 前端包括一个 JSF 托管 bean Count.java，这个 bean 会在 Facelets XHTML 文件 template.xhtml 和 index.xhtml 中使用。Count JSF 托管 bean 通过依赖注入得到 CounterBean 的引用。Count 定义了一个 JavaBeans 属性 hitCount。从 XHTML 文件调用 getHitCount 获取方法时，会调用 CounterBean 的 getHits 方法来返回页面的当前访问数。

下面是 Count 托管 bean 类：

```
@Named
@ConversationScoped
public class Count implements Serializable {
    @EJB
    private CounterBean counterBean;
    private int hitCount;

    public Count() {
        this.hitCount = 0;
    }
    public int getHitCount() {
        hitCount = counterBean.getHits();
        return hitCount;
    }

    public void setHitCount(int newHits) {
        this.hitCount = newHits;
    }
}
```

template.xhtml 和 index.xhtml 文件用来呈现一个 Facelets 视图，显示这个视图的访问数。index.xhtml 文件使用一个 EL 语句 #{count.hitCount} 访问 Count 托管 bean 的 hitCount 属性。下面是 index.xhtml 的内容：

```
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <ui:composition template="/template.xhtml">
        <ui:define name="title">
            This page has been accessed #{count.hitCount} time(s).
        </ui:define>
        <ui:define name="body">
            Hooray!
        </ui:define>
    </ui:composition>
</html>
```

5.2.3　运行 counter 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 counter 示例。

5.2.3.1　使用 NetBeans IDE 运行 counter 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/ejb
```

4. 选择 counter 文件夹。
5. 单击 Open Project。
6. 在 Projects（工程）标签页中，右键单击 counter 工程并选择 Run（运行）。

Web 浏览器会打开 URL http://localhost:8080/counter，显示访问数。

7. 重新加载页面，会看到访问数增加。

5.2.3.2　使用 Maven 运行 counter 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 在一个终端窗口中导航至：

```
tut-install/examples/ejb/counter/
```

3. 输入以下命令：

```
mvn install
```

这会构建并将 counter 部署到 GlassFish 服务器实例。

4. 在 Web 浏览器中输入以下 URL：

```
http://localhost:8080/counter
```

5. 重新加载页面，会看到访问数增加。

5.3　一个 Web 服务示例：helloservice

这个例子展示了一个简单的 Web 服务，它会根据从客户端接收的信息生成一个响应。HelloServiceBean 是一个无状态会话 bean，实现了一个方法：sayHello。这个方法与本书第 1 卷 28.1.5 节中介绍的客户端调用的 sayHello 方法一致。

5.3.1　Web 服务端点实现类

HelloServiceBean 是端点实现类，通常是企业 bean Web 服务端点的主要编程工件。Web 服务端点实现类有以下需求：

- 实现类必须有 javax.jws.WebService 或 javax.jws.WebServiceProvider 注解。
- 实现类可以通过 @WebService 注解的 endpointInterface 元素显式引用一个 SEI，但不要求必须这样做。如果 @WebService 中没有指定 endpointInterface，则会为实现类隐式地定义一个 SEI。
- 实现类的业务方法必须是公共的，而且不能声明为 static 或 final。
- 为 Web 服务客户端提供的业务方法必须有 javax.jws.WebMethod 注解。
- 为 Web 服务客户端提供的业务方法必须有 JAXB 兼容的参数和返回类型。更多信息参见本书第 1 卷 28.2 节中给出的 JAXB 默认数据类型绑定列表。
- 实现类不能声明为 final，也不能是 abstract。
- 实现类必须有一个默认的公共构造函数。
- 端点类必须有 @Stateless 注解。
- 实现类不能定义 finalize 方法。
- 实现类可以在其生命周期事件回调方法上使用 javax.annotation.PostConstruct 或 javax.annotation.PreDestroy 注解。

容器会在实现类开始响应 Web 服务客户端之前调用 @PostConstruct 方法。

容器由操作删除端点之前会调用 @PreDestroy 方法。

5.3.2 无状态会话 bean 实现类

HelloServiceBean 类实现了 sayHello 方法，这个方法有 @WebMethod 注解。HelloServiceBean 类的源代码如下所示：

```
package javaeetutorial.helloservice.ejb;

import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;

@Stateless
@WebService
public class HelloServiceBean {
    private final String message = "Hello, ";
    public void HelloServiceBean() {}

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

5.3.3　运行 helloservice 示例

可以使用 NetBeans IDE 或 Maven 构建、打包和部署 helloservice 示例。然后可以使用管理控制台测试 Web 服务端点方法。

5.3.3.1　使用 NetBeans IDE 构建、打包和部署 helloservice 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 从 File（文件）菜单选择 Open Project（打开工程）。

3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/ejb
```

4. 选择 helloservice 文件夹。

5. 单击 Open Project。

6. 在 Projects 标签页中，右键单击 helloservice 工程并选择 Build（构建）。

这会构建并把应用打包到 helloservice.ear，这个文件位于 *tut-install*/examples/ejb/helloservice/target/，然后将这个 EAR 文件部署到 GlassFish 服务器。

5.3.3.2　使用 Maven 构建、打包和部署 helloservice 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在一个终端窗口中导航至：

```
tut-install/examples/ejb/helloservice/
```

3. 输入以下命令：

```
mvn install
```

这会编译源文件，并将应用打包到一个 EJB JAR 文件，该文件位于 *tut-install*/examples/ejb/helloservice/target/helloservice.jar。然后将这个 EJB JAR 文件部署到 GlassFish 服务器。

部署时，GlassFish 服务器会生成 Web 服务调用所需的其他工件，包括 WSDL 文件。

5.3.3.3　无客户端情况下测试服务

GlassFish 服务器管理控制台允许测试一个 Web 服务端点的方法。要测试 HelloServiceBean 的 sayHello 方法，可以遵循以下步骤。

1. 在一个 Web 浏览器中打开以下 URL 以打开管理控制台：

```
http://localhost:4848/
```

2. 在导航树中选择 Applications（应用）节点。

3. 在 Applications 表中，单击 helloservice 链接。

4. 在 Modules and Components（模块和组件）表中，单击 View Endpoint（查看端点）链接。

5. 在 Web 服务端点信息页面上，单击 Tester（测试工具）链接：

/HelloServiceBeanService/HelloServiceBean?Tester

6. 在 Web 服务测试链接页面上，单击不安全链接（指定端口 8080 的链接）。

7. 在 HelloServiceBeanService Web 服务测试工具页面上，在 Methods（方法）下输入一个名字作为 sayHello 方法的参数。

8. 单击 sayHello。

这会打开 sayHello 方法调用页面。在返回的方法下面，可以看到端点的响应。

5.4 使用定时器服务

对业务工作流建模的应用通常依赖于定时通知。除了有状态会话 bean 以外，企业 bean 容器的定时器服务允许为所有其他类型的企业 bean 调度定时通知。可以根据一个日历日历发生调度定时通知：在指定的时间发生，在一段时间之后发生，或者按指定的间隔发生。例如，可以设置定时器在 5 月 23 日上午 10：30 启动，30 天后启动，或者每 12 小时启动一次。

企业 bean 定时器可以是编程定时器或自动定时器。**编程定时器**（Programmatic timer）通过显式地调用 TimerService 接口的某个定时器创建方法来设置。**自动定时器**（Automatic timer）则会在成功部署一个包含调度方法的企业 bean 时创建，这个调度方法有一个 javax.ejb.Schedule 或 javax.ejb.Schedules 注解。

5.4.1 创建基于日历的定时器表达式

可以根据基于日历的调度来设置定时器，这要使用一种类似于 UNIX cron 工具的语法描述。编程定时器和自动定时器都可以使用基于日历的定时器表达式。表 5-1 显示了基于日历的定时器属性。

表 5-1 基于日历的定时器属性

属 性	描 述	默认值	允许值和示例
second	一分钟内的 1 秒或数秒	0	0 到 59。例如：second="30"
minute	一小时内的 1 分钟或数分钟	0	0 到 59。例如：minute="15"
hour	一天内的 1 小时或数小时	0	0 到 23。例如：hour="13"
dayOfWeek	一周内的 1 天或数天	*	0 到 7（0 和 7 都表示星期天）。例如：dayOfWeek="3"。 Sun,Mon,Tue,Wed,Thu,Fri,Sat。 例如：dayOfWeek="Mon"

（续）

属　性	描　述	默认值	允许值和示例
dayOfMonth	一月内的 1 天或数天	*	1 到 31。例如：dayOfMonth="15"。 -7 到 -1（负数表示月末之前的第 *n* 天）。 例如：dayOfMonth="-3"。 Last。例如：dayOfMonth="Last"。 [1st, 2nd, 3rd, 4th, 5th, Last][Sun, Mon, Tue,Wed, Thu, Fri, Sat]。例如：dayOfMonth="2nd Fri"
month	一年内的 1 月或数月	*	1 到 12。例如：month="7"。 Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,Nov, Dec。例如：month="July"
year	一个特定的日历年	*	一个 4 位数字的日历年。例如： year="2011"

5.4.1.1　在日历表达式中指定多个值

可以在日历表达式中指定多个值，如下面各小节所述。

日历表达式中使用通配符

将一个属性设置为星号（*），表示这个属性的所有允许值。

下面的表达式表示每一分钟：

```
minute="*"
```

下面的表达式表示一周中的每一天：

```
dayOfWeek="*"
```

指定值列表

要为一个属性指定两个或多个值，可以使用一个逗号（,）来分隔这些值。值范围可以作为列表的一部分，不过，列表中不允许有通配符和间隔。

列表中重复的值会被忽略。

下面的表达式将日期设置为星期二（Tuesday）和星期四（Thursday）：

```
dayOfWeek="Tue, Thu"
```

下面的表达式表示上午 4:00，从上午 9:00 到下午 5:00 的每个整点（使用了一个范围）和下午 10:00：

```
hour="4,9-17,22"
```

指定值范围

可以使用一个短横线（-）为属性指定一个值范围（包含这个范围的两个极值）。范围的成员不能是通配符、列表或间隔。形如 x-x 的范围等价于单值表达式 x。形如 x-y 的范围（其中 x 大于 y）等价于表达式 x-*maximumvalue*, *minimumvalue*-y。也就是说，这个表达式从 x 开始，再转回到第一个允许值，直到 y 结束。

下面的表达式表示上午 9:00 到下午 5:00：

```
hour="9-17"
```

下面的表达式表示从星期五到星期一：

```
dayOfWeek="5-1"
```

下面的表达式表示当月第 25 天到月末以及月初直到当月第 5 天：

```
dayOfMonth="25-5"
```

这等价于以下表达式：

```
dayOfMonth="25-Last,1-5"
```

指定间隔

斜线（/）会为一个属性指定一个起点和一个间隔，用来指定一分钟、一小时或一天中每 N 秒、分钟或小时。对于一个形如 x/y 的表达式，x 表示起点，y 表示间隔。指定间隔时，x 位置上可以使用通配符，这等价于将 x 设置为 0。

只能为 second、minute 和 hour 属性设置间隔。

下面的表达式表示一小时中每 10 分钟：

```
minute="*/10"
```

这等价于：

```
minute="0,10,20,30,40,50"
```

下面的表达式表示从中午 12 点开始每 2 小时为一个间隔：

```
hour="12/2"
```

5.4.2 编程定时器

编程定时器到期（启动）时，容器会调用 bean 实现类中的超时方法（有 @Timeout 注解的方法）。@Timeout 方法包含处理定时事件的业务逻辑。

5.4.2.1 @Timeout 方法

企业 bean 类中有 @Timeout 注解的方法必须返回 void，可以有一个 javax.ejb.Timer 对象作为唯一的参数（可选）。这些方法不能抛出应用异常：

```
@Timeout
public void timeout(Timer timer) {
    System.out.println("TimerBean: timeout occurred");
}
```

5.4.2.2 创建编程定时器

要创建一个定时器，bean 要调用 TimerService 接口的某个 create 方法。这些方法支持创建基于单个动作、间隔或日历的定时器。

对于单个动作或间隔定时器，定时器的到期时间可以表述为一个时间段或一个绝对时间。时间段表述为触发超时事件之前的毫秒数。要指定一个绝对时间，可以创建一个 java.util.Date 对象，并把它传递到 TimerService.createSingleActionTimer 或 TimerService.createTimer 方法。

下面的代码设置一个编程定时器将在 1 分钟（60 000 毫秒）后到期：

```
long duration = 60000;
Timer timer =
    timerService.createSingleActionTimer(duration, new TimerConfig());
```

下面的代码设置一个编程定时器将在 2015 年 5 月 1 日下午 12:05 到期，这个日期指定为一个 java.util.Date：

```
SimpleDateFormatter formatter =
    new SimpleDateFormatter("MM/dd/yyyy 'at' HH:mm");
Date date = formatter.parse("05/01/2015 at 12:05");
Timer timer = timerService.createSingleActionTimer(date, new TimerConfig());
```

对于基于日历的定时器，定时器的到期时间表述为一个 javax.ejb.ScheduleExpression 对象，这个对象要作为参数传递给 TimerService.createCalendarTimer 方法。ScheduleExpression 类表示基于日历的定时器表达式，包含的方法与 5.4.1 节中所述的属性对应。

下面的代码使用 ScheduleExpression 辅助类创建一个编程定时器：

```
ScheduleExpression schedule = new ScheduleExpression();
schedule.dayOfWeek("Mon");
schedule.hour("12-17, 23");
Timer timer = timerService.createCalendarTimer(schedule);
```

关于这些方法签名的详细内容，参见 TimerService API 文档（http://docs.oracle.com/javaee/7/api/javax/ejb/TimerService.html）。

5.4.7 节中介绍的 bean 创建了一个定时器，如下所示：

```
Timer timer = timerService.createTimer(intervalDuration,
        "Created new programmatic timer");
```

在 timersession 示例中，调用 createTimer 的方法将在一个业务方法中得到调用，这个业务方法则由一个客户端调用。

定时器默认为持久的。如果服务器崩溃或瘫痪，则持久定时器会保存，服务器重启时再次激活。如果一个持久定时器在服务器关机时到期，则服务器重启时容器会调用 @Timeout 方法。

要创建非持久编程定时器，可以调用 TimerConfig.setPersistent(false)，并将这个 TimerConfig 对象传递到某个定时器创建方法。

createTimer 方法的 Date 和 long 参数表示时间（单位为毫秒）。不过，由于定时器服务并不面向实时应用，所以 @Timeout 方法回调可能不会以毫秒级的精度发生。定时器服

务用于业务应用，而业务应用通常按小时甚至更长的时间段来度量时间。

5.4.3 自动定时器

如果企业 bean 包含有 @Schedule 或 @Schedules 注解的方法，则部署这些企业 bean 时，EJB 容器会创建自动定时器。一个企业 bean 可以有多个自动超时方法，这与编程定时器不同，编程定时器只允许企业 bean 类中有一个指定 @Timeout 注解的方法。

可以通过注解或者通过 ejb-jar.xml 部署描述文件配置自动定时器。

在一个企业 bean 上增加一个 @Schedule 注解时，将把这个方法标识为一个基于日历调度的超时方法，日历调度在 @Schedule 属性中指定。

@Schedule 注解包含的元素与 5.4.1 节中的日历表达式对应，另外还包含 persistent、info 和 timezone 元素。

可选的 persistent 元素取一个布尔值，用来指定一个自动定时器在服务器重启或崩溃之后是否仍保留。默认地，所有自动定时器都是持久的。

可选的 timezone 元素用来指定这个自动定时器与一个特定的时区关联。如果设置了这个元素，那么它会相对于指定的时区来计算所有定时器表达式，而不论 EJB 容器在哪个时区运行。默认地，所有自动定时器设置都相对于服务器的默认时区。

可选的 info 元素用来设置定时器的一个信息描述。以后可以使用 Timer.getInfo 来获取定时器的信息。

下面的超时方法使用 @Schedule 设置一个定时器将在每个星期天午夜到期：

```
@Schedule(dayOfWeek="Sun", hour="0")
public void cleanupWeekData() { ... }
```

@Schedules 注解用来为一个给定的超时方法指定多个基于日历的定时器表达式。

下面的超时方法使用 @Schedules 注解来设置多个基于日历的定时器表达式。第一个表达式设置定时器在每个月的最后一天到期。第二个表达式设置定时器在每个星期五的下午 11:00 到期：

```
@Schedules ({
    @Schedule(dayOfMonth="Last"),
    @Schedule(dayOfWeek="Fri", hour="23")
})
public void doPeriodicCleanup() { ... }
```

5.4.4 取消和保存定时器

定时器可以由以下事件取消：

- 一个单事件定时器到期时，EJB 容器会调用相关联的超时方法，然后取消这个定时器。
- bean 调用 Timer 接口的 cancel 方法时，容器会取消定时器。

如果在一个已经取消的定时器上调用一个方法，容器会抛出 javax.ejb.NoSuch-

ObjectLocalException 异常。

要保存一个 Timer 对象以备将来引用，可以调用它的 getHandle 方法，将 TimerHandle 对象保存在一个数据库中（TimerHandle 对象是可串行化的）。要重新实例化 Timer 对象，可以从数据库获取这个句柄，并对这个句柄调用 getTimer。对于远程或 Web 服务接口中定义的方法，不能传递 TimerHandle 对象作为这些方法的参数。换句话说，远程客户端和 Web 服务客户端不能访问一个 bean 的 TimerHandle 对象。不过，本地客户端没有这个限制。

5.4.5　获得定时器信息

除了定义 cancel 和 getHandle 方法外，Timer 接口还定义了一些方法来获得定时器的有关信息：

```
public long getTimeRemaining();
public java.util.Date getNextTimeout();
public java.io.Serializable getInfo();
```

getInfo 方法返回的对象是 createTimer 调用的最后一个参数。例如，在上一节的 createTimer 代码段中，这个信息参数是一个 String 对象，值为 created timer。

要获取一个 bean 的所有活动定时器，可以调用 TimerService 接口的 getTimers 方法。getTimers 方法会返回一个 Timer 对象集合。

5.4.6　事务和定时器

企业 bean 通常在一个事务中创建定时器。如果这个事务回滚，则定时器创建也会回滚。类似地，如果一个 bean 在事务中取消了一个定时器，则这个事务回滚时，定时器取消也会回滚。在这种情况下，定时器的时间段会重置，就好像取消从未发生过一样。

在使用容器托管事务的 bean 中，@Timeout 方法通常有 Required 或 RequiresNew 事务属性来保持事务完整性。利用这些属性，EJB 容器会在调用 @Timeout 方法之前开始新事务。如果这个事务回滚，则容器会至少再调用一次 @Timeout 方法。

5.4.7　timersession 示例

这个例子的源代码位于 *tut-install*/examples/ejb/timersession/src/main/java/ 目录。

TimerSessionBean 是一个单例会话 bean，将展示如何设置一个自动定时器和一个编程定时器。在下面的 TimerSessionBean 源代码清单中，setTimer 和 @Timeout 方法用于设置一个编程定时器。创建 bean 时，由容器注入一个 TimerService 实例。由于它的业务方法 setTimer 提供到 TimerSessionBean 的本地非接口视图，所以可以由客户端调用。在这个例子中，客户端调用 setTimer，并指定间隔为 8000 毫秒（即 8 秒）。

setTimer 方法通过调用 TimerService 的 createTimer 方法创建一个新的定时器。既然设置了定时器，定时器到期时（约 8 秒后），EJB 容器便将调用 TimerSessionBean 的 programmaticTimeout 方法：

```
...
    public void setTimer(long intervalDuration) {
        logger.log(Level.INFO,
                "Setting a programmatic timeout for {0} milliseconds from now.",
                intervalDuration);
        Timer timer = timerService.createTimer(intervalDuration,
                "Created new programmatic timer");
    }

    @Timeout
    public void programmaticTimeout(Timer timer) {
        this.setLastProgrammaticTimeout(new Date());
        logger.info("Programmatic timeout occurred.");
    }
...
```

TimerSessionBean 还有一个自动定时器和超时方法 automaticTimeout。这个自动定时器设置为每 1 分钟到期，这是在 @Schedule 注解中使用一个基于日历的定时器表达式设置的：

```
...
    @Schedule(minute = "*/1", hour = "*", persistent = false)
    public void automaticTimeout() {
        this.setLastAutomaticTimeout(new Date());
        logger.info("Automatic timeout occured");
    }
...
```

TimerSessionBean 还有两个业务方法：getLastProgrammaticTimeout 和 getLastAutomaticTimeout。客户端调用这些方法分别得到编程定时器和自动定时器最后一次超时的日期和时间。

下面是 TimerSessionBean 类的源代码：

```
package javaeetutorial.timersession.ejb;

import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.ejb.Schedule;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerService;

@Singleton
```

```
@Startup
public class TimerSessionBean {
    @Resource
    TimerService timerService;

    private Date lastProgrammaticTimeout;
    private Date lastAutomaticTimeout;

    private static final Logger logger =
            Logger.getLogger("timersession.ejb.TimerSessionBean");

    public void setTimer(long intervalDuration) {
        logger.log(Level.INFO,
                "Setting a programmatic timeout for {0} milliseconds from now.",
                intervalDuration);
        Timer timer = timerService.createTimer(intervalDuration,
                "Created new programmatic timer");
    }

    @Timeout
    public void programmaticTimeout(Timer timer) {
        this.setLastProgrammaticTimeout(new Date());
        logger.info("Programmatic timeout occurred.");
    }

    @Schedule(minute = "*/1", hour = "*", persistent = false)
    public void automaticTimeout() {
        this.setLastAutomaticTimeout(new Date());
        logger.info("Automatic timeout occured");
    }

    public String getLastProgrammaticTimeout() {
        if (lastProgrammaticTimeout != null) {
            return lastProgrammaticTimeout.toString();
        } else {
            return "never";
        }
    }

    public void setLastProgrammaticTimeout(Date lastTimeout) {
        this.lastProgrammaticTimeout = lastTimeout;
    }

    public String getLastAutomaticTimeout() {
        if (lastAutomaticTimeout != null) {
            return lastAutomaticTimeout.toString();
        } else {
            return "never";
        }
    }
```

```
    public void setLastAutomaticTimeout(Date lastAutomaticTimeout) {
        this.lastAutomaticTimeout = lastAutomaticTimeout;
    }
}
```

说明 GlassFish 服务器默认的最小超时值为 1000 毫秒（即 1 秒）。如果需要设置小于 1000 毫秒的超时值，需要在管理控制台中改变 Minimum Delivery Interval（最小间隔）设置的值。要修改这个最小超时值，在管理控制台中展开 Configurations（配置），然后展开 server-config（服务器配置），选择 EJB Container（EJB 容器），再单击 EJB Timer Service（EJB 定时器服务）标签页。在 Minimum Delivery Interval（最小传送间隔）下输入一个新的超时值，并单击 Save（保存）。受虚拟机的限制，minimum-delivery-interval-in-millis 可取的最小值约为 10 毫秒。

5.4.8 运行 timersession 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 timersession 示例。

5.4.8.1 使用 NetBeans IDE 运行 timersession 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/ejb
```

4. 选择 timersession 文件夹。
5. 单击 Open Project。
6. 从 Run（运行）菜单选择 Run Project（运行工程）。

这会构建并将应用打包到一个 WAR 文件，这个 WAR 文件位于 *tut-install*/examples/ejb/timersession/target/timersession.war，将这个 WAR 文件部署到 GlassFish 服务器实例，然后运行 Web 客户端。

5.4.8.2 使用 Maven 构建、打包和部署 timersession 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 在一个终端窗口中导航至：

```
tut-install/examples/ejb/timersession/
```

3. 输入以下命令：

```
mvn install
```

这会构建并将应用打包到一个 WAR 文件，这个 WAR 文件位于 *tut-install*/examples/

`ejb/timersession/target/timersession.war`，将这个 WAR 文件部署到 GlassFish 服务器实例。

5.4.8.3　运行 Web 客户端

1. 打开 Web 浏览器访问以下 URL：

```
http://localhost:8080/timersession
```

2. 单击 Set Timer（设置定时器）来设置一个编程定时器。

3. 稍等一会，然后单击浏览器的 Refresh（刷新）按钮。

你会看到编程定时器和自动定时器最后一次超时的日期和时间。

要查看出现超时时所记录的消息，可以打开位于 *domain-dir*`/logs/` 的 `server.log` 文件查看。

5.5　处理异常

企业 bean 抛出的异常可以分为两类：系统异常和应用异常。

系统异常（system exception）指示支持应用的服务出现了一个问题。例如，无法得到一个外部资源的连接，或者无法找到一个注入的资源。如果遇到一个系统级问题，企业 bean 会抛出 `javax.ejb.EJBException`。由于 `EJBException` 是 `RuntimeException` 的一个子类，所以不用在方法声明的 `throws` 子句中指定。如果抛出一个系统异常，EJB 容器可能会撤销这个 bean 实例。因此，系统异常不能由 bean 的客户端程序来处理，而需要系统管理员的干预。

应用异常（application exception）指示企业 bean 业务逻辑中存在一个错误。应用异常通常是你自己编写的异常，如 `CartBean` 示例中业务方法抛出的 `BookException`。企业 bean 抛出一个应用异常时，容器不会把它包装在另一个异常中。客户端应该能够处理它接收的所有应用异常。

如果一个事务中出现一个系统异常，则 EJB 容器会回滚事务。不过，如果在事务中抛出一个应用异常，则容器不会回滚事务。

Chapter 6 第 6 章

使用嵌入式企业 bean 容器

本章介绍如何使用嵌入式企业 bean 容器在 Java SE 环境中而不是 Java EE 服务器中运行企业 bean 应用，包括以下内容：

- 嵌入式企业 bean 容器概述
- 开发嵌入式企业 bean 应用
- standalone 示例应用

6.1 嵌入式企业 bean 容器概述

嵌入式企业 bean 容器用于从 Java SE 环境中执行的客户端代码访问企业 bean 组件。容器和客户端代码都在同一个虚拟机中执行。嵌入式企业 bean 容器通常用来测试企业 bean，而不需要将这些企业 bean 部署到服务器上。

Java EE 服务器中企业 bean 容器提供的大多数服务在嵌入式企业 bean 容器中也同样提供，包括注入、容器托管事务和安全性。企业 bean 组件在嵌入式环境和 Java EE 环境中的执行很相似，因此同样的企业 bean 可以很容易地在独立应用和网络应用中重用。

6.2 开发嵌入式企业 bean 应用

所有嵌入式企业 bean 容器都支持表 6-1 所列的特性。

表 6-1　可嵌入容器支持的企业 bean 特性

企业 bean 特性	描　述
本地会话 bean	提供本地非接口视图的无状态、有状态和单例会话 bean。所有方法访问都是同步的。会话 bean 不能作为 Web 服务端点
事务	容器托管和 bean 托管的事务
安全	声明式和编程式安全
拦截器	会话 bean 的类级和方法级拦截器
部署描述文件	可选的 ejb-jar.xml 部署描述文件，与 Java EE 服务器中的企业 bean 容器有相同的覆盖规则

容器提供者可以支持企业 bean 的全部特性，不过如果使用了未在表 6-1 中列出的企业 bean 特性，则使用嵌入式容器的应用将不可移植，如定时器服务、会话 bean 作为 Web 服务端点或者远程业务接口。

6.2.1　运行嵌入式应用

嵌入式容器、企业 bean 组件和客户端都使用相同的类路径在同一个虚拟机中执行。因此，开发人员可以像运行典型的 Java SE 应用一样运行一个使用嵌入式容器的应用，如下所示：

```
java -classpath mySessionBean.jar:containerProviderRuntime.jar:myClient.jar \
com.example.ejb.client.Main
```

在上面的例子中，mySessionBean.jar 是一个 EJB JAR，包含一个本地无状态会话 bean；containerProviderRuntime.jar 是企业 bean 提供者提供的一个 JAR 文件，其中包含嵌入式容器所需的运行时类；myClient.jar 是一个 JAR 文件，其中包含一个 Java SE 应用，它会通过嵌入式容器调用会话 bean 中的业务方法。

在 GlassFish 服务器中，包含嵌入式容器相应类的运行时 JAR 是 glassfish-embedded-all.jar。

6.2.2　创建企业 bean 容器

javax.ejb.embedded.EJBContainer 抽象类表示企业 bean 容器的一个实例，包含一些创建容器实例的工厂方法。EJBContainer.create EJBContainer 方法用于创建和初始化嵌入式容器实例。

下面的代码段展示了如何创建一个嵌入式容器，这个嵌入式容器将使用容器提供者的默认设置初始化：

```
EJBContainer ec = EJBContainer.createEJBContainer();
```

默认地，嵌入式容器会在虚拟机类路径中搜索企业 bean 模块：包含 META-INF/ejb-jar.xml 部署描述文件的目录、包含指定有某个企业 bean 组件注解（如 @Stateless）

的类文件的目录，或者包含一个 ejb-jar.xml 部署描述文件或有企业 bean 注解的类文件的 JAR 文件。所有匹配项都认为是同一个应用中的企业 bean 模块。一旦在类路径中找到所有合法的企业 bean 模块，容器便开始初始化这些模块。createEJBContainer 方法成功返回时，客户端应用可以得到由嵌入式容器找到的所有企业 bean 模块的客户端视图引用。

EJBContainer.createEJBContainer 方法还有另一个版本，取属性和设置的一个 Map 来定制可嵌入容器实例：

```
Properties props = new Properties();
props.setProperty(...);
...
EJBContainer ec = EJBContainer.createEJBContainer(props);
```

6.2.2.1 显式指定要初始化的企业 bean 模块

开发人员可以明确指定嵌入式容器要初始化哪些企业 bean 模块。要显式地指定嵌入式容器初始化的企业 bean 模块，需要设置 EJBContainer.MODULES 属性。

这些模块可能位于运行嵌入式容器和客户端代码的虚拟机类路径中，也可能在虚拟机类路径之外。

要指定虚拟机类路径中的模块，可将 EJBContainer.MODULES 设置为一个 String 来指定一个模块名，或者设置为一个包含模块名的 String 数组。嵌入式容器在虚拟机类路径中搜索与指定名字匹配的企业 bean 模块：

```
Properties props = new Properties();
props.setProperty(EJBContainer.MODULES, "mySessionBean");
EJBContainer ec = EJBContainer.createEJBContainer(props);
```

要指定虚拟机类路径之外的企业 bean 模块，可将 EJBContainer.MODULES 设置为一个 java.io.File 对象或一个 File 对象数组。每个 File 对象指示一个 EJB JAR 文件，或者指示一个目录，其中包含展开的 EJB JAR 文件：

```
Properties props = new Properties();
File ejbJarFile = new File(...);
props.setProperty(EJBContainer.MODULES, ejbJarFile);
EJBContainer ec = EJBContainer.createEJBContainer(props);
```

6.2.3 查找会话 bean 引用

要在一个使用嵌入式容器的应用中查找会话 bean 引用，可以使用 EJBContainer 实例获取一个 javax.naming.Context 对象。调用 EJBContainer.getContext 方法来获取 Context 对象：

```
EJBContainer ec = EJBContainer.createEJBContainer();
Context ctx = ec.getContext();
```

然后可以使用可移植 JNDI 语法得到会话 bean 的引用，这个语法在 3.4.1.1 节中已经详细介绍。例如，要得到 MySessionBean 的一个引用（这是一个有非接口视图的本地会话 bean），可使用以下代码：

```
MySessionBean msb = (MySessionBean)
            ctx.lookup("java:global/mySessionBean/MySessionBean");
```

6.2.4　关闭企业 bean 容器

从客户端调用 EJBContainer 实例的 close 方法来关闭嵌入式容器：

```
EJBContainer ec = EJBContainer.createEJBContainer();
...
ec.close();
```

尽管不要求客户端关闭 EJBContainer 实例，不过这样做可以释放嵌入式容器占用的资源。如果运行客户端应用的虚拟机比客户端应用的生命期长，则这一点尤其重要。

6.3　standalone 示例应用

standalone 示例应用展示了如何在一个 JUnit 测试类中创建嵌入式企业 bean 容器的一个实例，并调用一个会话 bean 业务方法。通过在一个单元测试中测试企业 bean 的业务方法，开发人员可以独立于其他应用层执行应用的业务逻辑，例如表现层，而且不需要将应用部署到一个 Java EE 服务器上。

standalone 示例有两个主要组件：StandaloneBean 是一个无状态会话 bean，另外 StandaloneBeanTest 是一个 JUnit 测试类，相当于使用嵌入式容器的 StandaloneBean 的客户端。

StandaloneBean 是一个简单的会话 bean，提供了一个本地非接口视图，包括一个业务方法 returnMessage，这个方法会返回 String "Greetings!"：

```
@Stateless
public class StandaloneBean {

    private static final String message = "Greetings!";

    public String returnMessage() {
        return message;
    }

}
```

StandaloneBeanTest 调用 StandaloneBean.returnMessage，并测试返回的消息是否正确。首先，在 setUp 方法中创建一个嵌入式容器实例和初始上下文，setUp 方法

有一个 org.junit.Before 注解，指示这个方法应当在测试方法之前执行：

```
@Before
public void setUp() {
    ec = EJBContainer.createEJBContainer();
    ctx = ec.getContext();
}
```

testReturnMessage 方法有一个 org.junit.Test 注解，指示这个方法包含一个单元测试，它通过 Context 实例得到 StandaloneBean 的一个引用，再调用 StandaloneBean.returnMessage。接下来使用一个 JUnit 断言 assertEquals 将这个结果与期望的结果进行比较。如果 StandaloneBean.returnMessage 返回的字符串等于 "Greetings!"，则通过测试：

```
@Test
public void testReturnMessage() throws Exception {
    logger.info("Testing standalone.ejb.StandaloneBean.returnMessage()");
    StandaloneBean instance = (StandaloneBean)
            ctx.lookup("java:global/classes/StandaloneBean");
    String expResult = "Greetings!";
    String result = instance.returnMessage();
    assertEquals(expResult, result);
}
```

最后，tearDown 方法有一个 org.junit.After 注解，指示这个方法应当在所有单元测试都已经运行之后再执行，这个方法将关闭嵌入式容器实例：

```
@After
public void tearDown() {
    if (ec != null) {
        ec.close();
    }
}
```

6.3.1 使用 NetBeans IDE 运行 standalone 示例应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

tut-install/examples/ejb

4. 选择 standalone 文件夹并单击 Open Project。
5. 在 Projects（工程）标签页中，右键单击 standalone 工程并选择 Test（测试）。

这会执行 JUnit 测试类 StandaloneBeanTest。Output（输出）标签页会显示测试的进度和输出日志。

6.3.2　使用 Maven 运行 standalone 示例应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在一个终端窗口中导航至：

```
tut-install/examples/ejb/standalone/
```

3. 输入以下命令：

```
mvn install
```

这个命令会编译并将应用打包到一个 JAR 文件，然后执行 JUnit 测试类 `StandaloneBeanTest`。

Chapter 7 第 7 章

在会话 bean 中使用异步方法调用

本章讨论如何在会话 bean 中实现异步业务方法以及如何从企业 bean 客户端调用这些方法，包括以下内容：

- 异步方法调用
- async 示例应用

7.1 异步方法调用

会话可以实现**异步方法**（asynchronous method），这表示在会话 bean 实例上调用这个业务方法之前，控制已经由企业 bean 容器返回给客户端。然后客户端可以使用 Java SE 并发 API 获取结果，取消调用，并检查异常。异步方法通常用于长时间运行的操作、处理器密集的任务以及后台任务来提高应用吞吐量，或者如果不要求立即得到方法调用结果，异步方法还可以改善应用响应时间。

会话 bean 客户端调用一个一般的非异步业务方法时，方法完成前不会将控制返回给客户端。不过，调用异步方法的客户端会立即让企业 bean 容器返回控制。这就允许客户端在完成方法调用期间执行其他任务。如果这个方法返回一个结果，结果将是 `java.util.concurrent.Future<V>` 接口的一个实现，这里“V”是结果值类型。`Future<V>` 接口定义了客户端用来检查计算是否完成、等待调用完成、获取最终结果和取消调用的方法。

7.1.1 创建异步业务方法

为一个业务方法增加 `javax.ejb.Asynchronous` 注解以标记这个方法是一个异步方

法，或者在类级应用 @Asynchronous，这会标记会话 bean 的所有业务方法都是异步方法。提供 Web 服务的会话 bean 方法不能是异步的。

异步方法必须返回 void，或者返回 Future<V> 接口的一个实现。返回 void 的异步方法不能声明应用异常，不过，如果返回 Future<V>，则可以声明应用异常。例如：

```
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
}
```

这个方法会处理订单的支付，并将状态作为一个 String 返回。尽管支付的处理要花费很长时间，但客户端可以继续正常工作，处理最终完成时再显示结果。

javax.ejb.AsyncResult<V> 类是 Future<V> 接口的一个具体实现，作为一个辅助类提供，用来返回异步结果。AsyncResult 有一个构造函数，以结果作为参数，可以很容易地创建 Future<V> 实现。例如，processPayment 方法可以使用 AsyncResult 将状态作为一个 String 返回：

```
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    String status = ...;
    return new AsyncResult<>(status);
}
```

这个结果返回给企业 bean 容器，而不是直接返回给客户端，企业 bean 容器允许客户端使用这个结果。会话 bean 可以调用 javax.ejb.SessionContext.wasCancelled 方法来检查客户端是否请求取消这个调用。例如：

```
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    if (SessionContext.wasCancelled()) {
        // clean up
    } else {
        // process the payment
    }
    ...
}
```

7.1.2　从企业 bean 客户端调用异步方法

就像调用非异步业务方法一样，会话 bean 客户端采用同样的方式调用异步方法。如果异步方法返回一个结果，则一旦调用方法，客户端就会接收到一个 Future<V> 实例。这个实例可以用来获取最终结果、取消调用、检查调用是否已经完成、检查处理期间是否抛出异常以及检查调用是否已经取消。

7.1.2.1 从异步方法调用获取最终结果

客户端可以使用某个 Future<V>.get 方法获取结果。如果处理调用的会话 bean 还没有完成处理，则调用 get 方法会导致客户端暂停执行，直到调用完成。在调用一个 get 方法之前，应当使用 Future<V>.isDone 方法确定处理是否已经完成。

get() 方法将结果作为 Future<V> 实例类型值中指定的类型返回。例如，则调用 Future<String>.get() 会返回一个 String 对象。如果方法调用已经取消，则调用 get() 会抛出一个 java.util.concurrent.CancellationException。如果方法调用导致会话 bean 处理期间抛出一个异常，则调用 get() 将抛出一个 java.util.concurrent.ExecutionException。抛出 ExecutionException 的原因可以通过调用 ExecutionException.getCause 方法来获得。

get(long timeout,java.util.concurrent.TimeUnit unit) 方法与 get() 方法类似，不过允许客户端设置一个超时值。如果超出这个超时值，会抛出一个 java.util.concurrent.TimeoutException 异常。关于指定超时值的时间单位，更多信息参见 TimeUnit 类的 API 文档。

7.1.2.2 取消异步方法调用

在 Future<V> 实例上调用 cancel(boolean mayInterruptIfRunning) 方法可以取消这个方法调用。如果成功地取消了调用，cancel 方法会返回 true；如果方法调用无法取消，则返回 false。

调用无法取消时，可以使用 mayInterruptIfRunning 参数来提示运行这个方法调用的会话 bean 实例，告诉它客户端试图取消这个调用。如果 mayInterruptIfRunning 设置为 true，则会话 bean 实例调用 SessionContext.wasCancelled 会返回 true。如果 mayInterruptIfRunning 设置为 false，则会话 bean 实例调用 SessionContext.wasCancelled 会返回 false。

Future<V>.isCancelled 方法用来检查在异步方法调用完成之前是否已经通过调用 Future<V>.cancel 取消了方法调用。如果调用已经取消，则 isCancelled 方法返回 true。

7.1.2.3 检查异步方法调用的状态

如果会话 bean 实例完成了方法调用处理，则 Future<V>.isDone 方法返回 true。如果异步方法调用正常完成、被取消或导致一个异常，则 isDone 方法返回 true。也就是说，isDone 只指示会话 bean 是否已经结束处理调用。

7.2 async 示例应用

async 示例展示了如何在会话 bean 上定义一个异步业务方法，并从一个 Web 客户端调

用这个方法。这个例子包含两个模块：

- 一个 Web 应用（async-war），其中包含一个无状态会话 bean 和一个 JSF 接口。MailerBean 无状态会话 bean 定义了一个异步方法 sendMessage，这个方法使用 JavaMail API 向指定的 email 地址发送一个 email。
- 一个辅助的 Java SE 程序（async-smtpd），这个程序模拟一个 SMTP 服务器，在 TCP 端口 25 监听 SMTP 请求，并将 email 消息打印到标准输出（而不是真正发送这些消息）。

下面的小节将给出 async-war 模块的架构。

7.2.1　async-war 模块的架构

async-war 模块包括一个无状态会话 bean MailerBean，还有一个 JSF Web 应用前端，在 XHTML 文件中使用 Facelets 标记为用户显示一个表单，要求用户输入 email 接收者的 email 地址。email 最后发出时会更新 email 的状态。

MailerBean 会话 bean 注入一个 JavaMail 资源，用来向用户指定的地址发送一个 email 消息。这个消息使用 JavaMail API 来创建、修改和发送。会话 bean 如下所示：

```
@Named
@Stateless
public class MailerBean {
    @Resource(name="mail/myExampleSession")
    private Session session;
    private static final Logger logger =
            Logger.getLogger(MailerBean.class.getName());

    @Asynchronous
    public Future<String> sendMessage(String email) {
        String status;
        try {
            Message message = new MimeMessage(session);
            message.setFrom();
            message.setRecipients(Message.RecipientType.TO,
                    InternetAddress.parse(email, false));
            message.setSubject("Test message from async example");
            message.setHeader("X-Mailer", "JavaMail");
            DateFormat dateFormatter = DateFormat
                    .getDateTimeInstance(DateFormat.LONG, DateFormat.SHORT);
            Date timeStamp = new Date();
            String messageBody = "This is a test message from the async "
                    + "example of the Java EE Tutorial. It was sent on "
                    + dateFormatter.format(timeStamp)
                    + ".";
            message.setText(messageBody);
            message.setSentDate(timeStamp);
            Transport.send(message);
```

```
            status = "Sent";
            logger.log(Level.INFO, "Mail sent to {0}", email);
        } catch (MessagingException ex) {
            logger.severe("Error in sending message.");
            status = "Encountered an error: " + ex.getMessage();
            logger.severe(ex.getMessage());
        }
        return new AsyncResult<>(status);
    }
}
```

注入的 JavaMail 资源可以通过 GlassFish 服务器管理控制台来配置，也可以通过 GlassFish 服务器管理命令或者通过打包在应用中的一个资源配置文件来配置。运行时 GlassFish 服务器管理员可以修改资源配置，从而使用一个不同的邮件服务器或传输协议。

Web 客户端包括一个 Facelets 模板 `template.xhtml`；两个 Facelets 客户端 `index.xhtml` 和 `response.xhtml`；一个 JSF 托管 bean `MailerManagedBean`。`index.xhtml` 文件包含一个表单来输入目标 email 地址。用户提交表单时，会调用 `MailerManagedBean.send` 方法。这个方法使用一个注入的 `MailerBean` 会话 bean 实例来调用 `MailerBean.sendMessage`。结果发送到 `response.xhtml` Facelets 视图。

7.2.2 运行 async 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 `async` 示例。

7.2.2.1 使用 NetBeans IDE 运行 async 示例应用

1. 确保已启动 GlassFish 服务器。

2. 从 File（文件）菜单选择 Open Project（打开工程）。

3. 在 Open Project 对话框中，导航至：

tut-install/examples/ejb

4. 选择 `async` 文件夹，再选择 Open Required Projects（打开所需工程），并单击 Open Project。

5. 在 Projects（工程）标签页中，右键单击 `async-smtpd` 工程并选择 Run（运行）。

SMTP 服务器模拟程序开始接收连接。async-smptd 输出标签页会显示以下消息：

```
[Test SMTP server listening on port 25]
```

6. 在 Projects（工程）标签页中，右键单击 `async-war` 工程并选择 Build（构建）。

这个命令会使用一个 GlassFish 服务器管理命令配置 JavaMail 资源，并构建、打包和部署 `async-war` 模块。

7. 在 Web 浏览器窗口中打开以下 URL：

http://localhost:8080/async-war

8. 在 Web 浏览器窗口中输入一个 email 地址，并单击 Send email（发送 email）。

`MailerBean` 无状态 bean 使用 JavaMail API 向 SMTP 服务器模拟程序发送一个 email。NetBeans IDE 中的 async-smptd 输出窗口会显示得到的 email 消息，包括其首部。

9. 要停止这个 SMTP 服务器模拟程序，单击 NetBeans IDE 状态条右侧的 X 按钮。

10. 删除 JavaMail 会话资源。

   a. 在 Services（服务）标签页中，展开 Servers（服务器）节点，然后展开 GlassFish Server 节点。

   b. 展开 Resources（资源）节点，再展开 JavaMail Sessions（JavaMail 会话）节点。

   c. 右键单击 mail/myExampleSession 并选择 Unregister（取消注册）。

7.2.2.2　使用 Maven 运行 async 示例应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在一个终端窗口中导航至：

```
tut-install/examples/ejb/async/async-smtpd/
```

3. 输入以下命令来构建和打包 SMTP 服务器模拟程序：

```
mvn install
```

4. 输入以下命令启动这个 STMP 服务器模拟程序：

```
mvn exec:java
```

会出现以下消息：

```
[Test SMTP server listening on port 25]
```

保持这个终端窗口打开。

5. 在一个新的终端窗口中访问：

```
tut-install/examples/ejb/async/async-war
```

6. 输入以下命令来配置 JavaMail 资源，并构建、打包和部署 `async-war` 模块：

```
mvn install
```

7. 在 Web 浏览器窗口中打开以下 URL：

```
http://localhost:8080/async-war
```

8. 在 Web 浏览器窗口中输入一个 email 地址，并单击 Send email。

`MailerBean` 无状态 bean 使用 JavaMail API 向 SMTP 服务器模拟程序发送一个 email。第一个终端窗口会显示得到的 email 消息，包括其首部。

9. 要停止这个 SMTP 服务器模拟程序，将执行命令启动 STMP 服务器模拟程序的那个终端窗口关闭。

10. 要删除 JavaMail 会话资源，键入以下命令：

```
asadmin delete-javamail-resource mail/myExampleSession
```

第三部分 *Part 3*

持 久 化

第三部分讨论 Java 持久化 API，包括以下各章：

- 第 8 章　Java 持久化 API 介绍
- 第 9 章　运行持久化示例
- 第 10 章　Java 持久化查询语言
- 第 11 章　使用 Criteria API 创建查询
- 第 12 章　创建和使用基于字符串的 Criteria 查询
- 第 13 章　使用锁定控制对实体数据的并发访问
- 第 14 章　用实体图创建获取计划
- 第 15 章　对 Java 持久化 API 应用使用二级缓存

Chapter 8 第 8 章

Java 持久化 API 介绍

Java 持久化 API（Java Persistence API）为 Java 开发人员提供了一个对象 / 关系映射功能，可以在 Java 应用中管理关系数据。Java 持久化包括 4 个方面：

- Java 持久化 API
- 查询语言
- Java 持久化 Criteria API
- 对象 / 关系映射元数据

本章将介绍以下内容：

- 实体
- 实体继承
- 管理实体
- 查询实体
- 数据库模式创建
- 关于持久化的更多信息

8.1 实体

实体（entity）是一个轻量级持久化域对象。一般地，实体表示关系数据库中的一个表，各个实体实例对应于这个表中的一行。实体的主要编程工件是实体类，不过也可以使用辅助类。

实体的持久化状态通过持久化字段或持久化属性来表示。这些字段或属性使用对象 / 关系映射注解将实体和实体关系映射到底层数据存储中的关系数据。

8.1.1　实体类的需求

实体类必须满足以下需求：

- 实体类必须有 javax.persistence.Entity 注解。
- 实体类必须有一个公共或保护的无参数构造函数，另外还可以有其他构造函数。
- 实体类不能声明为 final。方法或持久化实例变量都不能声明为 final。
- 如果一个实体实例作为一个分离对象按值传递，例如通过一个会话 bean 的远程业务接口传递，那么这个类必须实现 Serializable 接口。
- 实体可以扩展实体和非实体类，非实体类也可以扩展实体类。
- 持久化实例变量必须声明为私有、保护或包私有，而且只能由实体类的方法直接访问。客户端必须通过存取方法或业务方法来访问实体的状态。

8.1.2　实体类中的持久化字段和属性

实体的持久化状态可以通过实体的实例变量或属性来访问。这些字段或属性必须是以下 Java 类型：

- Java 基本类型
- java.lang.String
- 其他可串行化类型，包括：
  - Java 基本类型的包装器类型
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.TimeStamp
  - 用户自定义可串行化类型
  - byte[]
  - Byte[]
  - char[]
  - Character[]
- 枚举类型
- 其他实体和 / 或实体集合
- 可嵌入的类

实体可以使用持久化字段、持久化属性或二者的组合。如果对实体的实例变量应用了映

射注解，则实体会使用持久化字段。如果对实体的 JavaBeans 属性的获取方法应用了映射注解，实体会使用持久化属性。

8.1.2.1 持久化字段

如果实体类使用持久化字段，则持久化运行时环境会直接访问实体类实例变量。没有 javax.persistence.Transient 注解的字段或者未标识为 Java transient 的所有字段都会持久存储到数据库中。对象 / 关系映射注解必须应用于实例变量。

8.1.2.2 持久化属性

如果实体使用持久化属性，则这个实体必须遵循 JavaBeans 组件的方法约定。JavaBeans 属性使用获取方法和设置方法，这些方法通常按实体类的实例变量名来命名。如果有类型为 *Type* 的实体，则对于这个实体的每一个持久化属性 *property*，会有一个获取方法 get*Property* 和设置方法 set*Property*。如果这个属性是一个 Boolean 型，则可以使用 is*Property* 而不是 get*Property*。例如，如果一个 Customer 实体使用持久化属性，而且有一个私有实例变量（名为 firstName），这个类就会定义一个 getFirstName 和 setFirstName 方法来获取和设置 firstName 实例变量的状态。

单值持久化属性的方法签名如下：

```
Type getProperty()
void setProperty(Type type)
```

持久化属性的对象 / 关系映射注解必须应用于获取方法。映射注解不能应用于有 @Transient 注解或标识为 transient 的字段或属性。

8.1.2.3 实体字段和属性中使用集合

不论实体使用持久化字段还是持久化属性，集合值持久化字段和属性都必须使用得到支持的 Java 集合接口。可以使用以下集合接口：

- java.util.Collection
- java.util.Set
- java.util.List
- java.util.Map

如果实体类使用持久化字段，则前述方法签名中的类型必须是以上集合类型之一。也可以使用这些集合类型的泛型类型。例如，如果有一个持久化属性，其中包含一组电话号码，则 Customer 实体将有以下方法：

```
Set<PhoneNumber> getPhoneNumbers() { ... }
void setPhoneNumbers(Set<PhoneNumber>) { ... }
```

如果一个实体的字段或属性由一个基本类型或可嵌入类的集合组成，则对这个字段或属性使用 javax.persistence.ElementCollection 注解。

@ElementCollection 的两个属性是 targetClass 和 fetch。targetClass 属

性指定了基本类型或可嵌入类的类名，如果字段或属性使用 Java 泛型类型定义，则这个属性是可选的。可选的 fetch 属性用来指定采用懒惰方式还是积极方式获取集合，为此要分别使用 javax.persistence.FetchType 常量 LAZY 或 EAGER。默认地，会采用懒惰方式获取集合。

下面的实体 Person 有一个持久化字段 nicknames，这是一个 String 类集合，可以采用积极方式获取。targetClass 元素不是必要的，因为这里使用泛型来字义这个字段：

```
@Entity
public class Person {
    ...
    @ElementCollection(fetch=EAGER)
    protected Set<String> nickname = new HashSet();
    ...
}
```

实体元素和关系的集合可以由 java.util.Map 集合表示。Map 由一个键和一个值组成。使用 Map 元素或关系时，遵循以下规则：

- Map 键或值可以是一个基本 Java 类型、可嵌入类或一个实体。
- Map 值是一个可嵌入类或基本类型时，使用 @ElementCollection 注解。
- Map 值是一个实体时，使用 @OneToMany 或 @ManyToMany 注解。
- 只在双向关系的一端使用 Map 类型。

如果 Map 的键类型是一个 Java 基本类型，则使用注解 javax.persistence.MapKeyColumn 来设置键的列映射。默认地，@MapKeyColumn 的 name 属性形式为 *RELATIONSHIP-FIELD/PROPERTY-NAME*_KEY。例如，如果引用的关系字段名是 image，则默认 name 属性为 IMAGE_KEY。

如果 Map 的键类型是一个实体，则使用 javax.persistence.MapKeyJoinColumn 注解。如果需要多列来设置映射，则要使用注解 javax.persistence.MapKeyJoinColumns 来包含多个 @MapKeyJoinColumn 注解。如果没有 @MapKeyJoinColumn，则映射列名默认设置为 *RELATIONSHIP-FIELD/PROPERTY-NAME*_KEY。例如，如果关系字段名为 employee，则默认的 name 属性为 EMPLOYEE_KEY。

如果关系字段或属性中没有使用 Java 泛型类型，就必须使用 javax.persistence.MapKeyClass 注解显式地设置键类。

如果 Map 键是 Map 值实体的主键或者持久化字段或属性，则使用 javax.persistence.MapKey 注解。@MapKeyClass 和 @MapKey 注解不能在同一个字段或属性上使用。

如果 Map 值是一个 Java 基本类型或可嵌入类，它会映射为底层数据库中的一个集合表。如果没有使用泛型类型，则 @ElementCollection 注解的 targetClass 属性必须设置为 Map 值的类型。

如果 Map 值是一个实体，而且是一个多对多或一对多关系的一部分，则它会映射为底层数据库中的一个联接表。也可以使用 @JoinColumn 注解映射使用 Map 的单向一对多关系。

如果实体是一个一对多 / 多对一双向关系的一部分，它会映射到表示 Map 值的实体表中。如果没有使用泛型类型，则 @OneToMany 和 @ManyToMany 注解的 targetEntity 属性必须设置为 Map 值的类型。

8.1.2.4 验证持久化字段和属性

JavaBeans 验证（bean 验证）Java API 提供了一种验证应用数据的机制。bean 验证已经集成到 Java EE 容器中，允许在一个企业应用的任意层中使用相同的验证逻辑。

bean 验证约束可以应用到持久化实体类、可嵌入类和所映射的超类。默认地，如果持久化字段或属性有 bean 验证约束注解，则持久化提供者要在 PrePersist、PreUpdate 和 PreRemove 生命周期事件之后立即对有这些持久化字段或属性的实体自动完成验证。

bean 验证约束是应用到 Java 类字段或属性的注解。bean 验证提供了一组约束，另外提供了一个 API 来定义定制约束。定制约束可以是默认约束的特定组合，或者也可以是未使用默认约束的新约束。每个约束都至少关联一个验证器类，这个验证器类会验证加约束字段或属性的值。定制约束开发人员还必须为约束提供一个验证器类。

bean 验证约束要应用到持久化类的持久化字段或属性。增加 bean 验证约束时，要使用与持久化类相同的访问策略。也就是说，如果持久化类使用字段访问，则在类的字段上应用 bean 验证约束注解。如果类使用属性访问，则在获取方法上应用约束。

本书第 1 卷中的表 21-1 列出了 bean 验证的内置约束，这些约束在 javax.validation.constraints 包中定义。

本书第 1 卷表 21-1 中列出的所有内置约束都有对应的注解 *ConstraintName*.List，用于对相同字段或属性上相同类型的多个约束分组。例如，以下持久化字段有两个 @Pattern 约束：

```
@Pattern.List({
    @Pattern(regexp="..."),
    @Pattern(regexp="...")
})
```

下面的实体类 Contact 在其持久化字段上应用了 bean 验证约束：

```
@Entity
public class Contact implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
            + "[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
            + "(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9]"
```

```
            + "(?:[a-z0-9-]*[a-z0-9])?",
            message = "{invalid.email}")
    protected String email;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(javax.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

firstName 和 lastName 字段上的 @NotNull 注解指定这些字段是必需的。如果 firstName 或 lastName 还未初始化，则要创建一个新的 Contact 实例，bean 验证就会抛出一个验证错误。类似地，如果之前创建的 Contact 实例已被修改，使 firstName 或 lastName 为 null，也会抛出一个验证错误。

email 字段应用了一个 @Pattern 约束，而且有一个复杂的正则表达式，可以匹配大多数合法的 email 地址。如果 email 的值与这个正则表达式不匹配，则会抛出一个验证错误。

homePhone 和 mobilePhone 字段有相同的 @Pattern 约束。这个正则表达式匹配美国和加拿大的 10 位电话号码，形式为 (*xxx*) *xxx*-*xxxx*。

birthday 字段有一个 @Past 约束，确保 birthday 的值是过去的一个时间。

8.1.3　实体的主键

每个实体有一个唯一的对象标识符。例如，顾客实体可以由一个顾客号来标识。这个唯一标识符或**主键**（primary key）允许客户端找到一个特定的实体实例。每个实体必须有一个主键。实体可以有一个简单主键或一个复合主键。

简单主键使用 javax.persistence.Id 注解来指示主键属性或字段。

主键由多个属性组成时，就会使用复合主键，这些属性对应一组持久化属性或字段。复合主键必须在一个主键类中定义。复合主键使用 javax.persistence.EmbeddedId 和 javax.persistence.IdClass 注解指示。

主键或复合主键的属性或字段必须是以下 Java 类型：

- Java 基本类型
- Java 基本包装器类型
- java.lang.String
- java.util.Date（时态类型应当是 DATE）
- java.sql.Date

- java.math.BigDecimal
- java.math.BigInteger

主键中不能使用浮点数类型。如果使用生成的主键，则只有整数类型是可移植的。

主键类必须满足以下需求：

- 主键类的访问控制修饰符必须是 public。
- 如果使用基于属性的访问，则主键类的属性必须是 public 或 protected。
- 主键类必须有一个公共的默认构造函数。
- 主键类必须实现 hashCode() 和 equals(Object other) 方法。
- 主键类必须是可串行化的。
- 复合主键必须表示并映射到实体类的多个字段或属性，或者必须表示和映射为一个可嵌入类。
- 如果一个类映射到实体类的多个字段或属性，则主键类中主键字段或属性的名称与类型必须与实体类的相应名称和类型匹配。

以下主键类是一个复合键，customerOrder 和 itemId 字段共同唯一标识一个实体：

```
public final class LineItemKey implements Serializable {
    private Integer customerOrder;
    private int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer order, int itemId) {
        this.setCustomerOrder(order);
        this.setItemId(itemId);
    }

    @Override
    public int hashCode() {
        return ((this.getCustomerOrder() == null
                ? 0 : this.getCustomerOrder().hashCode())
                ^ ((int) this.getItemId()));
    }

    @Override
    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((this.getCustomerOrder() == null
                ? other.getCustomerOrder() == null : this.getCustomerOrder()
                .equals(other.getCustomerOrder()))
```

```
            && (this.getItemId() == other.getItemId()));
    }

    @Override
    public String toString() {
        return "" + getCustomerOrder() + "-" + getItemId();
    }
    /* Getters and setters */
}
```

8.1.4 实体关系中的多重性

多重性有以下类型：

- **一对一**：每个实体实例与另一个实体的单个实例关联。例如，要对一个仓库建模，其中每个储箱包含一个部件，储箱（StorageBin）和部件（Widget）就有一种一对一的关系。一对一关系在相应的持久化属性或字段上使用javax.persistence.OneToOne注解。
- **一对多**：一个实体实例可以与其他实体的多个实例关联。例如，一个销售订单可以有多个商品。在订单（order）应用中，顾客订单（CustomerOrder）就与商品（ineItem）有一种一对多的关系。一对多关系在相应的持久化属性或字段上使用javax.persistence.OneToMany注解。
- **多对一**：一个实体的多个实例可以与另一个实体的一个实例关联。这种多重性与一对多关系正相反。在刚才提到的例子中，从LineItem到CustomerOrder的关系就是多对一关系。多对一关系在相应的持久化属性或字段上使用javax.persistence.ManyToOne注解。
- **多对多**：实体实例可以与多个实例互相关联。例如，每门大学课程都有多个学生，每个学生也可以选多门课程。因此，在一个注册应用中，课程（Course）和学生（Student）就是一种多对多的关系。多对多关系在相应的持久化属性或字段上使用javax.persistence.ManyToMany注解。

8.1.5 实体关系中的方向

关系的方向可以是双向，也可以是单向。双向关系有一个所属方和一个接受方。单向关系只有一个所属方。关系的所属方确定了持久化运行时环境如何对数据库中的关系完成更新。

8.1.5.1 双向关系

在一个**双向关系**（bidirectional relationship）中，每个实体有一个关系字段或属性指向另一个实体。通过这个关系字段或属性，实体类的代码可以访问其关联的对象。如果一个实体有一个关联字段，则称这个实体“知道”它的关联对象。例如，如果CustomerOrder知道

它有什么 LineItem 实例，而且如果 LineItem 知道它属于哪个 CustomerOrder，它们就有一种双向关系。

双向关系必须遵循以下规则：

- 双向关系的接受方必须通过 @OneToOne、@OneToMany 或 @ManyToMany 注解的 mappedBy 元素指示其所属方。mappedBy 元素指示关系所有者实体的属性或字段。
- 多对一双向关系的多方不能定义 mappedBy 元素。多方往往是这个关系的所属方。
- 对于一对一双向关系，所属方对应包含相应外键的一方。
- 对于多对多双向关系，双方都可以是所属方。

8.1.5.2 单向关系

在一个**单向关系**（unidirectional relationship）中，只有一个实体有一个关系字段或属性指示另一个实体。例如，LineItem 有一个关系字段指示 Product，但 Product 没有对应 LineItem 的关系字段或属性。换句话说，LineItem 知道 Product，但 Product 并不知道哪些 LineItem 实例指向它。

8.1.5.3 查询和关系方向

Java 持久化查询语言和 Criteria API 查询通常用于在关系中导航。关系的方向确定了一个查询是否可以从一个实体导航到另一个实体。例如，一个查询可以从 LineItem 导航到 Product，但是不能反方向导航。对于 CustomerOrder 和 LineItem，查询在两个方向上都可以导航，因为这两个实体有一种双向关系。

8.1.5.4 层叠操作和关系

使用关系的实体通常会依赖于关系中另一个实体是否存在。例如，一个商品是一个订单的一部分；如果删除这个订单，这个商品也应当被删除。这称为一种层叠删除关系。

javax.persistence.CascadeType 枚举类型定义了关系注解 cascade 元素中应用的层叠操作。表 8-1 列出了实体的这些层叠操作。

表 8-1 实体的层叠操作

层叠操作	描 述
ALL	所有层叠操作将应用到父实体的关联实体。All 等价于指定 cascade={DETACH,MERGE,PERSIST,REFRESH,REMOVE}
DETACH	如果父实体从持久化上下文中分离，关联实体也要分离
MERGE	如果父实体合并到持久化上下文，关联实体也要合并
PERSIST	如果父实体在持久化上下文中保持，关联实体也要保持
REFRESH	如果父实体在当前持久化上下文中刷新，关联实体也要刷新
REMOVE	如果父实体从当前持久化上下文中删除，关联实体也要被删除

层叠删除关系使用 @OneToOne 和 @OneToMany 关系的 cascade=REMOVE 元素规范指定。例如：

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<CustomerOrder> getOrders() { return orders; }
```

8.1.5.5 关系中的孤删除

一对一或一对多关系中的一个目标实体从关系中删除时，通常需要完成目标实体的层叠删除操作。这种目标实体被认为是“孤实体”，orphanRemoval 属性可以用来指定应删除孤实体。例如，如果一个订单有多个商品，则从订单中删除其中一个商品时，所删除的商品就被认为是一个孤实体。如果 orphanRemoval 设置为 true，从订单中删除商品时，就要删除相应的商品实体。

@OneToMany 和 @oneToOne 中的 orphanRemoval 属性取一个布尔值，默认为 false。下面的例子会在删除 customer 实体时对 order 孤实体完成层叠删除操作：

```
@OneToMany(mappedBy="customer", orphanRemoval="true")
public List<CustomerOrder> getOrders() { ... }
```

8.1.6 实体中的可嵌入类

可嵌入类用来表示一个实体的状态，但是它们本身不是持久化的，这一点与实体类不同。一个可嵌入类的实例会共享其所有者实体的标识（即拥有这个可嵌入类的实体）。可嵌入类只作为另一个实体的状态存在。实体可以有单值或集合值可嵌入类属性。

可嵌入类与实体类有相同的规则，不过要有 javax.persistence.Embeddable 注解而不是 @Entity。

下面的可嵌入类 ZipCode 包含字段 zip 和 plusFour：

```
@Embeddable
public class ZipCode {
    String zip;
    String plusFour;
    ...
}
```

Address 实体使用了这个可嵌入类：

```
@Entity
public class Address {
    @Id
    protected long id;
    String street1;
    String street2;
    String city;
    String province;
    @Embedded
    ZipCode zipCode;
    String country;
    ...
}
```

如果实体包含可嵌入类作为其持久化状态的一部分，这个实体可以对相应字段或属性增加 javax.persistence.Embedded 注解，但不要求一定这么做。

可嵌入类本身也可以使用其他可嵌入类来表示它们的状态，它们还可以包含基本 Java 编程语言类型或其他可嵌入类的集合。另外，可嵌入类还可以包含与其他实体或实体集合的关系。如果可嵌入类有这样一种关系，则这种关系是从目标实体或实体集合到拥有可嵌入类的实体之间的关系。

8.2 实体继承

实体支持类继承、多态关联和多态查询。实体类可以扩展非实体类，非实体类也可以扩展实体类。实体类可以是抽象类和具体类。

roster 示例应用展示了实体继承，见 9.2.2 节。

8.2.1 抽象实体

通过用 @Entity 装饰抽象类，可以将这个抽象类声明为一个实体。抽象实体与具体实体很相似，不过不能实例化。

与具体实体一样，抽象实体可以查询。如果查询的目标是一个抽象实体，则这个查询会作用于这个抽象实体的所有具体子类：

```
@Entity
public abstract class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
}
```

8.2.2 映射超类

实体可以继承包含持久化状态和映射信息的超类（而不是继承实体）。也就是说，这个超类没有用 @Entity 注解装饰，不会由 Java 持久化提供者映射为一个实体。超类最常用于有多个实体类共同的状态和映射信息时。

用 javax.persistence.MappedSuperclass 注解装饰类来指定映射超类：

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}
```

映射超类不能查询，也不能用在EntityManager或Query操作中。EntityManager或Query操作中必须使用映射超类的实体子类。映射超类不能作为实体关系的目标。映射超类可以是抽象或具体的。

映射超类在底层数据库中没有相应的表。要由从映射超类继承的实体来定义表映射。例如，在前面的代码示例中，底层表有FULLTIMEEMPLOYEE和PARTTIMEEMPLOYEE，但是没有EMPLOYEE表。

8.2.3 非实体超类

实体可以有非实体超类，这些超类可以是抽象或具体的。非实体超类的状态不是持久化的，实体类从非实体超类继承的所有状态都不是持久化的。非实体超类不能用于EntityManager或Query操作中。非实体超类中的所有映射或关系注解都会被忽略。

8.2.4 实体继承映射策略

通过使用javax.persistence.Inheritance注解装饰类层次体系中的根类，可以对Java持久化提供者如何将继承的实体映射到底层数据库进行配置。可以使用下面的映射策略将实体数据映射到底层数据库：

- 每个类层次体系映射到一个表
- 每个具体实体类映射到一个表
- “联接”策略，一个子类特定的字段或属性与父类共有的字段或属性会映射到不同的表要配置这个策略，需要将@Inheritance的strategy元素设置为javax.persistence.InheritanceType枚举类型中定义的一个选项：

```
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};
```

如果实体层次体系的根类上没有指定@Inheritance注解，则使用默认策略InheritanceType.SINGLE_TABLE。

8.2.4.1 每个类层次体系映射一个表的策略

这种策略对应默认的InheritanceType.SINGLE_TABLE，层次体系中的所有类都映射到数据库中的一个表。这个表有一个**鉴别器列**（discriminator column），其中包含的值用来标识这一行表示的实例属于哪个子类。

鉴别器列的元素见表8-2，可以在实体类层次体系的根上使用javax.persistence.DiscriminatorColumn注解来指定鉴别器列。

表8-2 @DiscriminatorColumn元素

类 型	名	描 述
String	name	列名用作为鉴别器列。默认为DTYPE。这个元素是可选的
DiscriminatorType	discriminatorType	列类型用作为鉴别器列。默认为DiscriminatorType.STRING。这个元素是可选的
String	columnDefinition	创建鉴别器列时使用的SQL片段。默认值由持久化提供者生成，特定于具体实现。这个元素是可选的
String	length	基于String的鉴别器类型的列长度。对于非String鉴别器类型会忽略这个元素。默认为31。这个元素是可选的

javax.persistence.DiscriminatorType枚举类型用来设置数据库中鉴别器列的类型，要把@DiscriminatorColumn的discriminatorType元素设置为已定义的一种类型。DiscriminatorType定义如下：

```
public enum DiscriminatorType {
    STRING,
    CHAR,
    INTEGER
};
```

如果在实体层次体系的根上没有指定@DiscriminatorColumn，但需要一个鉴别器列，则持久化提供者会假设一个默认的列名DTYPE，列类型为DiscriminatorType.STRING。

对于类层次体系中的各个实体，可以用javax.persistence.DiscriminatorValue注解来设置其鉴别器列中输入的值。@DiscriminatorValue只能用来装饰具体的实体类。

在一个使用鉴别器列的类层次体系中，如果一个实体上没有指定@DiscriminatorValue，则持久化提供者会提供一个默认的特定于具体实现的值。如果@DiscriminatorColumn

的 discriminatorType 元素是 DiscriminatorType.STRING，默认值就是实体名。

这个策略可以很好地支持整个实体类层次体系中实体和查询之间的多态关系。不过，这个策略要求包含子类状态的列为 null。

8.2.4.2　每个具体类映射一个表的策略

这个策略对应 InheritanceType.TABLE_PER_CLASS，在这个策略中，每个具体类映射到数据库中一个单独的表。类中的所有字段或属性（包括继承的字段或属性）都映射到这个类对应的数据库表中的列。

这个策略对于多态关系的支持很弱，对于覆盖整个实体类层次体系的查询，通常要对各个子类使用 SQL UNION 查询或者单独的 SQL 查询。

对这种策略的支持是可选的，并不是所有 Java 持久化 API 提供者都支持这个策略。GlassFish 服务器中默认的 Java 持久化 API 提供者就不支持这个策略。

8.2.4.3　联接子类策略

这个策略对应 InheritanceType.JOINED，在这个策略中，类层次体系中的根由一个表表示，每个子类都有一个单独的表，其中只包含这个子类特定的字段。也就是说，子类表不包含对应继承字段或属性的列。子类表还有一个或多个列表示其主键，这将是超类表主键的外键。

这个策略可以很好地支持多态关系，但是在实例化实体子类时需要完成一个或多个联接操作。对于庞大的类层次体系，这可能导致性能低下。类似地，覆盖整个类层次体系的查询要求在子类表之间完成联接操作，这也会导致性能下降。

一些 Java 持久化 API 提供者（包括 GlassFish 服务器中的默认提供者）在使用这种联接子类策略时要求有一个对应根实体的鉴别器列。如果你的应用中没有使用自动表创建，那么一定要确保为鉴别器列默认值正确地建立了数据库表，或者使用 @DiscriminatorColumn 注解匹配数据库模式。有关鉴别器列的更多信息，参见 8.2.4.1 节。

8.3　管理实体

实体由实体管理器来管理，实体管理器由 javax.persistence.EntityManager 实例表示。每个 EntityManager 实例与一个持久化上下文关联：这是存在于一个特定数据库中的一组托管实体实例。持久化上下文定义了在哪个作用域下创建、保持和删除特定的实体实例。EntityManager 接口定义了与持久化上下文交互所用的方法。

8.3.1　EntityManager 接口

EntityManager API 可以创建和删除持久化实体实例、按实体的主键查找实体，还允许在实体上运行查询。

8.3.1.1 容器托管实体管理器

使用**容器托管实体管理器**时，EntityManager 实例的持久化上下文会由容器自动传播到在一个 JTA（Java 事务 API）事务中使用这个 EntityManager 实例的所有应用组件。

JTA 事务通常包括跨应用组件的调用。要完成一个 JTA 事务，这些组件通常需要访问一个持久化上下文。利用 javax.persistence.PersistenceContext 注解将 EntityManager 注入到应用组件时会发生这种情况。持久化上下文会随当前 JTA 事务自动传播，映射到同一个持久化单元的 EntityManager 引用允许访问该事务中的持久化上下文。通过自动传播持久化上下文，应用组件不需要在一个事务中相互传递 EntityManager 实例引用来完成修改。Java EE 容器会管理容器托管实体管理器的生命周期。

要得到一个 EntityManager 实例，需要将实体管理器注入应用组件：

```
@PersistenceContext
EntityManager em;
```

8.3.1.2 应用托管实体管理器

对于**应用托管实体管理器**则相反，持久化上下文不会传播到应用组件，EntityManager 实例的生命周期由应用管理。

应用需要在一个特定的持久化单元中访问持久化上下文，但持久化上下文未随 JTA 事务跨 EntityManager 实例传播，此时就要使用应用托管实体管理器。在这种情况下，每个 EntityManager 会创建一个新的、独立的持久化上下文。EntityManager 及其关联的持久化上下文将由应用显式地创建和撤销。不能直接注入 EntityManager 实例时也可以使用应用托管实体管理器，因为 EntityManager 实例不是线程安全的。EntityManagerFactory 实例则是线程安全的。

在这种情况下，应用使用 javax.persistence.EntityManagerFactory 的 createEntityManager 方法来创建 EntityManager 实例。

要得到一个 EntityManager 实例，首先必须得到一个 EntityManagerFactory 实例，为此要利用 javax.persistence.PersistenceUnit 注解将它注入到应用组件：

```
@PersistenceUnit
EntityManagerFactory emf;
```

然后从 EntityManagerFactory 实例得到一个 EntityManager：

```
EntityManager em = emf.createEntityManager();
```

应用托管实体管理器不会自动传播 JTA 事务上下文。这些应用需要手动地访问 JTA 事务管理器，完成实体操作时要增加事务界定信息。javax.transaction.UserTransaction 接口定义了开始、提交和回滚事务的方法。通过创建一个有 @Resource 注解的实例变量来注入一个 UserTransaction 实例：

```
@Resource
UserTransaction utx;
```

开始一个事务时，要调用 UserTransaction.begin 方法。所有实体操作都完成时，调用 UserTransaction.commit 方法来提交事务。UserTransaction.rollback 方法用于回滚当前事务。

下面的例子显示了如何在一个使用应用托管实体管理器的应用中管理事务：

```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
    utx.begin();
    em.persist(SomeEntity);
    em.merge(AnotherEntity);
    em.remove(ThirdEntity);
    utx.commit();
} catch (Exception e) {
    utx.rollback();
}
```

8.3.1.3 使用 EntityManager 查找实体

EntityManager.find 方法用于在数据库中根据实体的主键查找实体：

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, CustomerOrder newOrder) {
    Customer cust = em.find(Customer.class, custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```

8.3.1.4 管理实体实例的生命周期

可以利用一个 EntityManager 实例调用实体的操作来管理实体实例。实体实例有以下 4 种状态：新建、托管、分离或删除。

- 新建实体实例没有持久化标识，还未与一个持久化上下文关联。
- 托管实体实例有一个持久化标识，并与一个持久化上下文关联。
- 分离实体实例有一个持久化标识，目前未与一个持久化上下文关联。
- 删除实体实例有一个持久化标识，与一个持久化上下文关联，已经计划从数据库中删除。

8.3.1.5 保持实体实例

通过调用 persist 方法或者通过从关联实体（在关系注解中设置了 cascade=PERSIST 或 cascade=ALL 元素）调用的一个层叠 persist 操作，新建实体实例将变为托

管和保持状态。这说明，与persist操作关联的事务完成时，实体的数据会存储在数据库中。如果实体已经是托管的，则会忽略persist操作，尽管persist操作会层叠至相关的实体（即关系注解中cascade元素设置为PERSIST或ALL的实体）。如果在一个已经删除的实体实例上调用persist，则实体会变为托管状态。如果实体已经分离，则persist会抛出一个IllegalArgumentException，或者事务提交失败。下面的方法完成了一个persist操作：

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(CustomerOrder order, Product product,
        int quantity) {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}
```

persist操作会传播到与调用实体相关的所有实体，即关系注解中cascade元素设置为ALL或PERSIST的实体：

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

8.3.1.6 删除实体实例

通过调用remove方法或者通过从关联实体（在关系注解中设置了cascade=REMOVE或cascade=ALL元素）调用的一个层叠remove操作，可以将托管实体实例删除。如果在一个新建实体上调用remove方法，则会忽略这个remove操作，尽管remove会层叠至相关实体（即关系注解中cascade元素设置为REMOVE或ALL的实体）。如果在一个分离的实体上调用remove，则remove会抛出一个IllegalArgumentException，或者事务提交失败。如果在一个已经删除的实体上调用，则remove会被忽略。事务完成时，或者完成flush操作时，实体的数据会从数据库中删除。

在下面的例子中，与订单关联的所有LineItem实体也会被删除，因为CustomerOrder.getLineItems在关系注解中设置了cascade=ALL：

```
public void removeOrder(Integer orderId) {
    try {
        CustomerOrder order = em.find(CustomerOrder.class, orderId);
        em.remove(order);
    }
    ...
}
```

8.3.1.7 将实体数据同步到数据库

与实体关联的事务提交时，持久化实体的状态会同步到数据库。如果一个托管实体与另一个托管实体有一种双向关系，那么数据会根据这个关系的所属方持久存储。

要求托管实体同步到数据库时，可以调用 EntityManager 实例的 flush 方法。如果实体与另一个实体关联，而且关系注解中将 cascade 元素设置为 PERSIST 或 ALL，则相关的实体数据会在调用 flush 时与数据库同步。

如果删除了实体，则调用 flush 会从数据库删除实体数据。

8.3.2 持久化单元

持久化单元定义了一个集合，包含应用中由 EntityManager 实例管理的所有实体类。这组实体类表示了数据库中包含的数据。

持久化单元由 persistence.xml 配置文件定义。下面是一个示例 persistence.xml 文件：

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description>This unit manages orders and customers.
            It does not rely on any vendor-specific features and can
            therefore be deployed to any persistence provider.
        </description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.CustomerOrder</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

这个文件定义了一个名为 OrderManagement 的持久化单元，它使用一个支持 JTA 的数据源 jdbc/MyOrderDB。jar-file 和 class 元素用来指定托管持久化类：实体类、可嵌入类和映射超类。jar-file 元素指定打包的持久化单元（其中包含托管持久化类）可见的 JAR 文件，class 元素显式指定托管持久化类。

jta-data-source（对应支持 JTA 的数据源）和 non-jta-data-source（对应不支持 JTA 的数据源）元素指定容器使用的数据源的全局 JNDI 名。

如果 JAR 文件或目录中的 META-INF 目录包含 persistence.xml，则这个 JAR 文件或目录称为持久化单元的根。持久化单元的作用域由持久化单元的根确定。每个持久化单元必须由一个对持久化单元作用域唯一的名字来标识。

持久化单元可以打包到一个 WAR 或 EJB JAR 文件，或者可以打包为一个 JAR 文件，然后再包含到一个 WAR 或 EAR 文件中。

- 如果将持久化单元打包为一个 EJB JAR 文件中的一组类，则 persistence.xml 应当放在 EJB JAR 的 META-INF 目录中。

- 如果将持久化单元打包为一个 WAR 文件中的一组类，则 persistence.xml 应当放在这个 WAR 文件的 WEB-INF/classes/META-INF 目录中。
- 如果将持久化单元打包到一个 JAR 文件中，这个 JAR 文件再包含在一个 WAR 或 EAR 文件中，则 JAR 文件应当放在以下位置：
  - WAR 的 WEB-INF/lib 目录
  - EAR 文件的库目录

说明 在 Java 持久化 API 1.0 中，JAR 文件与持久化单元的根可以位于 EAR 文件的根目录。不过现在已经不再支持这种做法。可移植应用应当使用 EAR 文件的库目录作为持久化单元的根。

8.4 查询实体

Java 持久化 API 提供了以下方法来查询实体：

- Java 持久化查询语言（Java Persistence query language，JPQL）是一个简单的基于字符串的语言，类似于 SQL，用来查询实体及实体的关系。更多信息参见第 10 章。
- Criteria API 使用 Java 语言 API 来创建类型安全的查询，用于查询实体及实体的关系。更多信息参见第 11 章。

JPQL 和 Criteria API 都各有自己的优缺点。

简单地讲，JPQL 查询通常比 Criteria 查询更简洁、更可读。熟悉 SQL 的开发人员会发现 JPQL 的语法很容易学习。JPQL 查询可以在实体类中使用 Java 编程语言注解定义，或者在应用的部署描述文件中定义。不过，JPQL 查询不是类型安全的，从实体管理器获取查询结果时需要一个强制类型转换。这说明，编译时可能无法捕获到类型转换错误。另外 JPQL 查询不支持不定数目的参数。

Criteria 查询允许在应用的业务层定义查询。尽管使用 JPQL 动态查询也可以做到这一点，不过 Criteria 查询提供了更好的性能，因为 JPQL 动态查询每一次调用时都必须解析。Criteria 查询是类型安全的，因此不要求像 JPQL 查询那样完成类型强制转换。Criteria API 实际上就是一个 Java 编程语言 API，不需要开发人员学习另一种查询语言的语法。Criteria 查询通常比 JPQL 查询更麻烦，需要开发人员创建多个对象，要在这些对象上完成操作之后再向实体管理器提交查询。

8.5 数据库模式创建

应用部署期间，可以使用应用部署描述文件中的标准属性配置持久化提供者，来自动创建数据库表、在表中加载数据以及删除数据库表。这些功能通常在开发阶段使用，而不针对

生产数据库。

下面是一个 persistence.xml 部署描述文件的例子，指定了在部署应用时，提供者要利用所提供的一个脚本删除所有数据库工件、用所提供的一个脚本创建工件以及从所提供的一个脚本加载数据：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="examplePU" transaction-type="JTA">
    <jta-data-source>java:global/ExampleDataSource</jta-data-source>
    <properties>
      <property name="javax.persistence.schema-generation.database.action"
                value="drop-and-create"/>
      <property name="javax.persistence.schema-generation.create-source"
                value="script"/>
      <property
           name="javax.persistence.schema-generation.create-script-source"
           value="META-INF/sql/create.sql" />
      <property name="javax.persistence.sql-load-script-source"
                value="META-INF/sql/data.sql" />
      <property name="javax.persistence.schema-generation.drop-source"
                value="script" />
      <property
             name="javax.persistence.schema-generation.drop-script-source"
             value="META-INF/sql/drop.sql" />
    </properties>
  </persistence-unit>
</persistence>
```

8.5.1　配置应用以创建或删除数据库表

javax.persistence.schema-generation.database.action 属性用来指定部署应用时持久化提供者要采取的动作。如果这个属性未设置，则持久化提供者不会创建或删除任何数据库工件。

表 8-3　模式创建动作

设　置	描　述
none	不发生任何模式创建或删除
create	提供者会在应用部署时创建数据库工件。这些工件在应用重新部署后仍保持不变
drop-and-create	将删除数据库中的工件，另外提供者会在部署时创建数据库工件
drop	应用部署时会删除数据库中的所有工件

在这个例子中，应用部署时，持久化提供者会删除所有剩余的数据库工件，然后再创建工件：

```
<property name="javax.persistence.schema-generation.database.action"
          value="drop-and-create"/>
```

默认地，会使用持久化单元中的对象 / 关系元数据创建数据库工件。还可以提供脚本，提供者使用这个脚本来创建和删除数据库工件。

javax.persistence.schema-generation.create-source 和 javax.persistence.schema-generation.drop-source 属性控制提供者如何创建或删除数据库工件。

表 8-4 创建和删除源属性的设置

设 置	描 述
metadata	使用应用中的对象 / 关系元数据来创建或删除数据库工件
script	使用所提供的一个脚本来创建或删除数据库工件
metadata-then-script	组合使用对象 / 关系元数据以及用户提供的一个脚本来创建或删除数据库工件
script-then-metadata	组合使用用户提供的一个脚本以及对象 / 关系元数据来创建或删除数据库工件

在这个例子中，持久化提供者将使用打包在应用中的一个脚本来创建数据库工件：

```
<property name="javax.persistence.schema-generation.create-source"
          value="script"/>
```

如果在 create-source 或 drop-source 中指定了一个脚本，则需要使用 javax.persistence.schema-generation.create-script-source 或 javax.persistence.schema-generation.drop-script-source 属性指定这个脚本的位置。脚本的位置是相对于持久化单元根的一个相对位置：

```
<property name="javax.persistence.schema-generation.create-script-source"
          value="META-INF/sql/create.sql" />
```

在上面的例子中，create-script-source 设置为一个名为 create.sql 的 SQL 文件，这个文件位于相对于持久化单元根的 META-INF/sql 目录下。

8.5.2 使用 SQL 脚本加载数据

如果希望应用加载之前使用数据来填充数据库表，则需要在 javax.persistence.sql-load-script-source 属性中指定一个加载脚本的位置。这个属性中指定的位置是相对于持久化单元根的一个相对位置。

在这个例子中，加载脚本是一个名为 data.sql 的文件，这个文件位于相对于持久化单元根的 META-INF/sql 目录中：

```
<property name="javax.persistence.sql-load-script-source"
          value="META-INF/sql/data.sql" />
```

8.6 有关持久化的更多信息

关于 Java 持久化 API 的更多信息，参见：

- Java Persistence 2.1 API 规范：

  `http://jcp.org/en/jsr/detail?id=338`

- EclipseLink，这是 GlassFish 服务器中的 Java 持久化 API 实现：

  `http://www.eclipse.org/eclipselink/jpa.php`

- EclipseLink 小组博客：

  `http://eclipselink.blogspot.com/`

- EclipseLink wiki 文档：

  `http://wiki.eclipse.org/EclipseLink`

Chapter 9

第 9 章

运行持久化示例

本章将解释如何使用 Java 持久化 API。这里重点强调 3 个示例的源代码和设置。第一个例子 order 是一个使用有状态会话 bean 来管理一个订单系统相关实体的应用。第二个例子 roster 是一个管理社区运动系统的应用。第三个例子 address-book 是一个存储联系数据的 Web 应用。本章假定你已经熟悉第 8 章中介绍的概念。

本章将介绍以下内容：

- order 应用
- roster 应用
- address-book 应用

9.1 order 应用

order 应用是一个简单的库存和订单应用，用来维护一个部件编目以及对这些部件下订单。这个应用包含一些实体，分别表示部件、供货商、订单和商品。这些实体使用一个有状态会话 bean 来访问，这个 bean 中包含应用的业务逻辑。应用部署时将由一个简单的单例会话 bean 创建初始实体。由一个 Facelets Web 应用管理数据，并显示编目中的数据。

订单中包含的信息可以划分为元素。订单号是什么？订单中包含哪些部件？这个部件又由哪些部件构成？这个部件由谁制造？部件的规格是什么？这个部件有没有图纸？订单系统包含了所有这些元素，order 应用则是这样一个系统的简化版本。

order 应用由一个 WAR 模块组成，其中包括企业 bean 类、实体、支持类以及 Facelets XHTML 和类文件。

对于 order，Java DB 数据库中的数据库模式如图 9-1 所示。

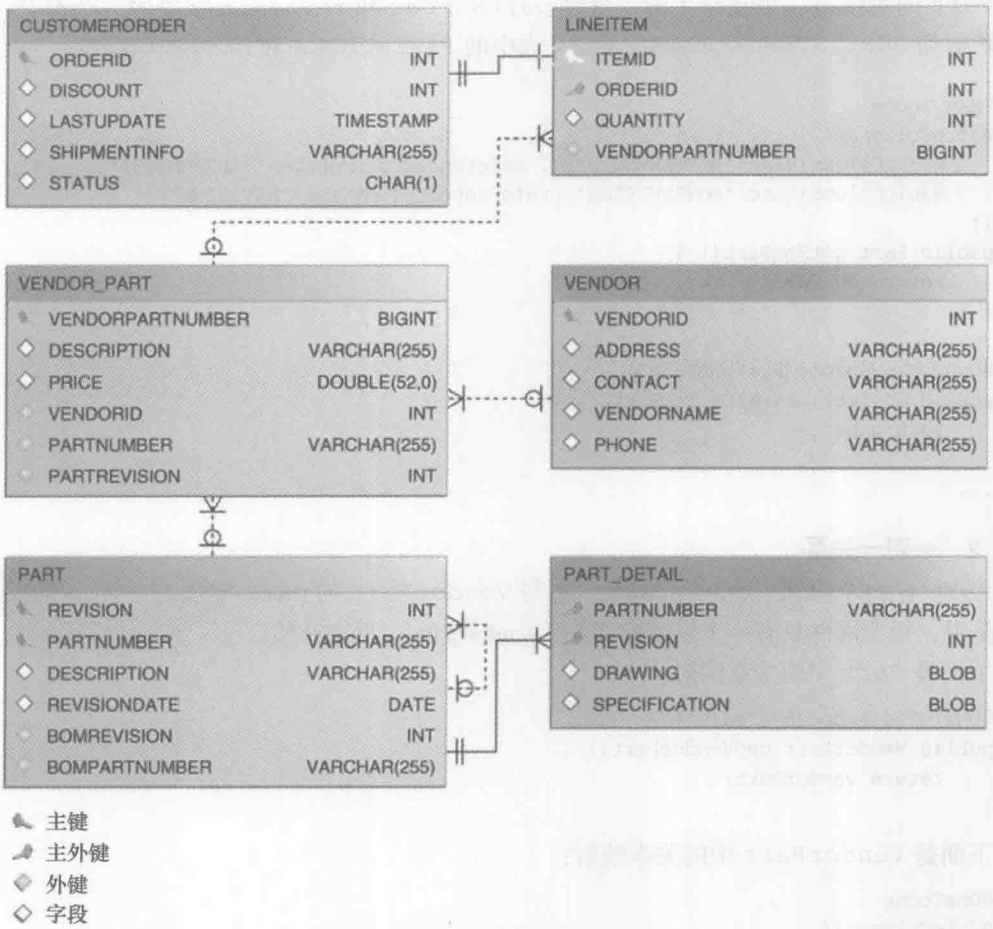


图 9-1　order 应用的数据库模式

说明　在这个图中，为简单起见，表名省略了 PERSISTENCE_ORDER_ 前缀。

9.1.1　order 应用中的实体关系

order 应用展示了多种不同类型的实体关系，包括自引用、一对一、一对多、多对一和单向关系。

9.1.1.1　自引用关系

自引用（self-referential）**关系**出现在同一个实体的关系字段之间。Part 有一个字段 bomPart，它与同样在 Part 中的字段 parts 有一对多的关系。也就是说，一个部件可以

由多个部件组成，每个部件只有一个材料单部件。

Part 的主键是一个复合主键，这是 partNumber 和 revision 字段的一个组合。这个键映射到 PERSISTENCE_ORDER_PART 表中的 PARTNUMBER 和 REVISION 列：

```
...
@ManyToOne
@JoinColumns({
    @JoinColumn(name="BOMPARTNUMBER", referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="BOMREVISION", referencedColumnName="REVISION")
})
public Part getBomPart() {
    return bomPart;
}
...
@OneToMany(mappedBy="bomPart")
public Collection<Part> getParts() {
    return parts;
}
...
```

9.1.1.2 一对一关系

Part 有一个字段 vendorPart，它与 VendorPart 的 part 字段有一对一的关系。也就是说，每个部件只有一个供应商部件（vendor part），反之亦然。

下面是 Part 中的关系映射：

```
@OneToOne(mappedBy="part")
public VendorPart getVendorPart() {
    return vendorPart;
}
```

下面是 VendorPart 中的关系映射：

```
@OneToOne
@JoinColumns({
    @JoinColumn(name="PARTNUMBER", referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="PARTREVISION", referencedColumnName="REVISION")
})
public Part getPart() {
    return part;
}
```

需要说明的是，由于 Part 使用了一个复合主键，因此将用 @JoinColumns 注解将 PERSISTENCE_ORDER_VENDOR_PART 表中的列映射到 PERSISTENCE_ORDER_PART 中的列。PERSISTENCE_ORDER_VENDOR_PART 表的 PARTREVISION 列指示 PERSISTENCE_ORDER_PART 的 REVISION 列。

9.1.1.3 一对多关系映射到重叠主键和外键

CustomerOrder 有一个字段 lineItems，它与 LineItem 的字段 customerOrder 有一对多关系。也就是说，每个订单中包含一个或多个商品。

LineItem 使用一个复合主键，由 orderId 和 itemId 字段组成。这个复合主键映射到 PERSISTENCE_ORDER_LINEITEM 表中的 ORDERID 和 ITEMID 列。ORDERID 是 PERSISTENCE_ORDER_CUSTOMERORDER 表中 ORDERID 列的一个外键。这说明 ORDERID 列映射了两次：一次作为主键字段 orderId；另一次作为一个关系字段 order。

下面是 CustomerOrder 中的关系映射：

```
@OneToMany(cascade=ALL, mappedBy="customerOrder")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

下面是 LineItem 中的关系映射：

```
@Id
@ManyToOne
@JoinColumn(name="ORDERID")
public CustomerOrder getCustomerOrder() {
    return customerOrder;
}
```

9.1.1.4　单向关系

LineItem 有一个字段 vendorPart，它与 VendorPart 有一种单向多对一关系。也就是说，在这个关系中目标实体中没有字段：

```
@JoinColumn(name="VENDORPARTNUMBER")
@ManyToOne
public VendorPart getVendorPart() {
    return vendorPart;
}
```

9.1.2　order 应用中的主键

order 应用使用了多种不同类型的主键：单值主键、生成主键和复合主键。

9.1.2.1　生成主键

VendorPart 使用了一个生成主键值。也就是说，应用没有为实体指定主键值，而是依赖于持久化提供者来生成主键值。@GeneratedValue 注解用来指定一个实体将使用生成主键。

在 VendorPart 中，下面的代码指定了生成主键值的设置：

```
@TableGenerator(
    name="vendorPartGen",
    table="PERSISTENCE_ORDER_SEQUENCE_GENERATOR",
    pkColumnName="GEN_KEY",
    valueColumnName="GEN_VALUE",
    pkColumnValue="VENDOR_PART_ID",
    allocationSize=10)
@Id
@GeneratedValue(strategy=GenerationType.TABLE, generator="vendorPartGen")
```

```
public Long getVendorPartNumber() {
    return vendorPartNumber;
}
```

@TableGenerator 注解与 @GeneratedValue 的 strategy=TABLE 元素结合使用。也就是说，生成主键所用的策略是使用数据库中的一个表。@TableGenerator 注解用来配置生成器表的设置。name 元素设置生成器名，在这里就是 VendorPart 中的 vendorPartGen。

PERSISTENCE_ORDER_SEQUENCE_GENERATOR 表包含两列 GEN_KEY 和 GEN_VALUE，将存储生成的主键值。这个表可以用来生成其他实体的主键，所以 pkColumnValue 元素设置为 VENDOR_PART_ID 以将这个实体的生成主键与其他实体的生成主键区分开。allocationSize 元素指定分配主键值时的递增量。在这里，每个 VendorPart 主键的递增量为 10。

主键字段 vendorPartNumber 类型为 Long，因为生成主键字段必须是一个整数类型。

9.1.2.2 复合主键

复合主键由多个字段组成，满足 8.1.3 节中描述的需求。要使用一个复合主键，必须创建一个包装器类。

在 order 中，两个实体使用了复合主键：Part 和 LineItem。

- Part 使用 PartKey 包装器类。Part 的主键是部件号和版本号的一个组合。PartKey 封装了这个主键。
- LineItem 使用 LineItemKey 类。LineItem 的主键是订单号和商品号的一个组合。LineItemKey 封装了这个主键。

下面是 LineItemKey 复合主键包装器类：

```
package javaeetutorial.order.entity;

import java.io.Serializable;

public final class LineItemKey implements Serializable {

    private Integer customerOrder;
    private int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer order, int itemId) {
        this.setCustomerOrder(order);
        this.setItemId(itemId);
    }
    @Override
    public int hashCode() {
        return ((this.getCustomerOrder() == null
```

```
                ? 0 : this.getCustomerOrder().hashCode())
                ^ ((int) this.getItemId()));
    }

    @Override
    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((this.getCustomerOrder() == null
                ? other.getCustomerOrder == null : this.getOrderId()
                .equals(other.getCustomerOrder()))
                && (this.getItemId == oother.getItemId()));
    }

    @Override
    public String toString() {
        return "" + getCustomerOrder() + "-" + getItemId();
    }

    public Integer getCustomerOrder() {
        return customerOrder;
    }

    public void setCustomerOrder(Integer order) {
        this.customerOrder = order;
    }

    public int getItemId() {
        return itemId;
    }

    public void setItemId(int itemId) {
        this.itemId = itemId;
    }
}
```

@IdClass 注解用于指定实体类中的主键类。在 LineItem 中，@IdClass 如下使用：

```
@IdClass(LineItemKey.class)
@Entity
...
public class LineItem implements Serializable { ... }
```

LineItem 的两个字段有 @Id 注解，标志这些字段作为复合主键的一部分：

```
@Id
public int getItemId() {
```

```
    return itemId;
}
...
@Id
@ManyToOne
@JoinColumn(name="ORDERID")
public CustomerOrder getCustomerOrder() {
    return customerOrder;
}
```

对于 customerOrder，还使用了 @JoinColumn 注解来指定表中的列名，并指定这个列是一个重叠的外键，指向 PERSISTENCE_ORDER_CUSTOMERORDER 表的 ORDERID 列（参见 9.1.1.3 节）。也就是说，customerOrder 将由 CustomerOrder 实体设置。

在 LineItem 的构造函数中，商品号（LineItem.itemId）使用 CustomerOrder.getNextId 方法来设置：

```
public LineItem(CustomerOrder order, int quantity, VendorPart vendorPart) {
    this.customerOrder = order;
    this.itemId = order.getNextId();
    this.quantity = quantity;
    this.vendorPart = vendorPart;
}
```

CustomerOrder.getNextId 统计当前的商品数，将商品数增 1，再返回这个数：

```
@Transient
public int getNextId() {
    return this.lineItems.size() + 1;
}
```

Part 需要在构成 Part 复合主键的两个字段上增加 @Column 注解，因为 Part 的复合主键是重叠主键 / 外键：

```
@IdClass(PartKey.class)
@Entity
...
public class Part implements Serializable {
    ...
    @Id
    @Column(nullable=false)
    public String getPartNumber() {
        return partNumber;
    }
    ...
    @Id
    @Column(nullable=false)
    public int getRevision() {
        return revision;
    }
    ...
}
```

9.1.3　实体映射到多个数据库表

Part 的字段映射到多个数据库表：PERSISTENCE_ORDER_PART 和 PERSISTENCE_ORDER_PART_DETAIL。PERSISTENCE_ORDER_PART_DETAIL 表包含部件的规格和图纸。@SecondaryTable 注解用来指定辅表：

```
...
@Entity
@Table(name="PERSISTENCE_ORDER_PART")
@SecondaryTable(name="PERSISTENCE_ORDER_PART_DETAIL", pkJoinColumns={
    @PrimaryKeyJoinColumn(name="PARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @PrimaryKeyJoinColumn(name="REVISION",
        referencedColumnName="REVISION")
})
public class Part implements Serializable { ... }
```

PERSISTENCE_ORDER_PART_DETAIL 和 PERSISTENCE_ORDER_PART 有相同的主键值。@SecondaryTable 的 pkJoinColumns 元素用来指定 PERSISTENCE_ORDER_PART_DETAIL 的主键列是 PERSISTENCE_ORDER_PART 的外键。@PrimaryKeyJoinColumn 注解设置主键列名，并指定这一列指示主表中哪一列。在这里，PERSISTENCE_ORDER_PART_DETAIL 和 PERSISTENCE_ORDER_PART 的主键列名是一样的：都是 PARTNUMBER 和 REVISION。

9.1.4　order 应用中的层叠操作

与其他实体有关系的实体通常会依赖于关系中另一个实体的存在。例如，一个商品属于某个订单；如果这个订单被删除，那么这个商品也应当被删除。这称为一个层叠删除关系。

在 order 中，实体关系中有两个层叠删除依赖关系。如果与一个 LineItem 关联的 CustomerOrder 被删除，那么这个 LineItem 也应当被删除。如果与一个 VendorPart 关联的 Vendor 被删除，那么这个 VendorPart 也应当被删除。

指定实体关系的层叠操作时，要在关系的接受方（非所属方）设置 cascade 元素。CustomerOrder.lineItems 中，这个 cascade 元素设置为 ALL。这说明所有持久化操作（删除、更新等）都将由订单层叠至商品。

下面是 CustomerOrder 中的关系映射：

```
@OneToMany(cascade=ALL, mappedBy="customerOrder")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

下面是 LineItem 中的关系映射：

```
@Id
@ManyToOne
@JoinColumn(name="ORDERID")
```

```
public CustomerOrder getCustomerOrder() {
    return customerOrder;
}
```

9.1.5 order 应用中的 BLOB 和 CLOB 数据库类型

数据库表 PARTDETAIL 有一个 DRAWING 列，类型为 BLOB。BLOB 表示二进制大对象，用于存储二进制数据，如图像。DRAWING 列映射到 java.io.Serializable 类型的字段 Part.drawing。@Lob 注解用来指示这个字段是一个大对象：

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public Serializable getDrawing() {
    return drawing;
}
```

PERSISTENCE_ORDER_PART_DETAIL 也有一个类型为 CLOB 的列 SPECIFICATION。CLOB 表示字符大对象，用于存储太大而无法在 VARCHAR 列中存储的字符串数据。SPECIFICATION 映射到类型为 java.lang.String 的字段 Part.specification。这里也使用了 @Lob 注解来指示这个字段是一个大对象：

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public String getSpecification() {
    return specification;
}
```

这两个字段都使用了 @Column 注解，并将 table 元素设置为辅表。

9.1.6 order 应用中的时态类型

CustomerOrder.lastUpdate 持久化属性的类型为 java.util.Date，映射到 PERSISTENCE_ORDER_CUSTOMERORDER.LASTUPDATE 数据库字段，其 SQL 类型为 TIMESTAMP。为了确保这些类型之间正确的映射，必须使用 @Temporal 注解，并在 @Temporal 的元素中指定正确的时态类型。@Temporal 的元素类型为 javax.persistence.TemporalType。可取值包括：

- DATE，映射到 java.sql.Date
- TIME，映射到 java.sql.Time
- TIMESTAMP，映射到 java.sql.Timestamp

下面是 CustomerOrder 中相关的部分：

```
@Temporal(TIMESTAMP)
public Date getLastUpdate() {
    return lastUpdate;
}
```

9.1.7 管理 order 应用的实体

RequestBean 有状态会话 bean 包含业务逻辑，并管理 order 的实体。RequestBean 使用 @PersistenceContext 注解来获取一个实体管理器实例，这个实例将在 RequestBean 的业务方法中用来管理 order 的实体：

```
@PersistenceContext
private EntityManager em;
```

这个 EntityManager 实例是一个容器托管实体管理器，所以容器会负责管理 order 实体有关的所有事务。

9.1.7.1 创建实体

RequestBean.createPart 业务方法创建一个新的 Part 实体。EntityManager.persist 方法将新创建的实体持久存储到数据库中：

```
Part part = new Part(partNumber,
        revision,
        description,
        revisionDate,
        specification,
        drawing);
em.persist(part);
```

ConfigBean 单例会话 bean 用来初始化 order 中的数据。ConfigBean 有 @Startup 注解，指示部署 order 时 EJB 容器要创建 ConfigBean。createData 方法增加了 @PostConstruct 注解，并通过调用 RequestBean 的业务方法创建 order 使用的初始实体。

9.1.7.2 查找实体

RequestBean.getOrderPrice 业务方法根据 orderId 返回一个给定订单的价格。EntityManager.find 方法用于从数据库获取这个实体：

```
CustomerOrder order = em.find(CustomerOrder.class, orderId);
```

EntityManager.find 的第一个参数是实体类，第二个参数是主键。

9.1.7.3 设置实体关系

RequestBean.createVendorPart 业务方法创建与一个特定 Vendor 关联的 VendorPart。EntityManager.persist 方法用于将新创建的 VendorPart 实体持久存储到数据库，VendorPart.setVendor 和 Vendor.setVendorPart 方法用于将 VendorPart 与 Vendor 关联：

```
PartKey pkey = new PartKey();
pkey.setPartNumber(partNumber);
pkey.setRevision(revision);

Part part = em.find(Part.class, pkey);
```

```
VendorPart vendorPart = new VendorPart(description, price, part);
em.persist(vendorPart);

Vendor vendor = em.find(Vendor.class, vendorId);
vendor.addVendorPart(vendorPart);
vendorPart.setVendor(vendor);
```

9.1.7.4 使用查询

RequestBean.adjustOrderDiscount 业务方法用来更新所有订单的折扣。这个方法使用 findAllOrders 命名查询，这个命名查询在 CustomerOrder 中定义：

```
@NamedQuery(
    name="findAllOrders",
    query="SELECT co FROM CustomerOrder co " +
          "ORDER BY co.orderId"
)
```

EntityManager.createNamedQuery 方法用于运行查询。由于查询返回所有订单的一个 List，因此要使用 Query.getResultList 方法：

```
List orders = em.createNamedQuery(
        "findAllOrders")
        .getResultList();
```

RequestBean.getTotalPricePerVendor 业务方法返回一个特定供货商所有部件的总价格。这个方法使用了一个命名参数 id，这在 VendorPart 中的命名查询 findTotal-VendorPartPricePerVendor 中定义：

```
@NamedQuery(
    name="findTotalVendorPartPricePerVendor",
    query="SELECT SUM(vp.price) " +
    "FROM VendorPart vp " +
    "WHERE vp.vendor.vendorId = :id"
)
```

运行查询时，使用 Query.setParameter 方法将命名参数 id 设置为 vendorId 的值，作为 RequestBean.getTotalPricePerVendor 的参数：

```
return (Double) em.createNamedQuery(
        "findTotalVendorPartPricePerVendor")
        .setParameter("id", vendorId)
        .getSingleResult();
```

这里使用了 Query.getSingleResult 方法，因为这个查询返回单个值。

9.1.7.5 删除实体

RequestBean.removeOrder 业务方法用于从数据库中删除一个给定的订单。这个方法使用 EntityManager.remove 方法从数据库中删除实体：

```
CustomerOrder order = em.find(CustomerOrder.class, orderId);
em.remove(order);
```

9.1.8　运行 order 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 order 应用。首先要在 Java DB 服务器中创建数据库表。

9.1.8.1　使用 NetBeans IDE 运行 order 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 如果数据库服务器还未运行，则按照 2.4 节中的说明启动该服务器。
3. 从 File（文件）菜单选择 Open Project（打开工程）。
4. 在 Open Project 对话框中，导航至：

```
tut-install/examples/persistence
```

5. 选择 order 文件夹。
6. 单击 Open Project。
7. 在 Projects 标签页中，右键单击 order 工程并选择 Run（运行）。

NetBeans IDE 打开一个 Web 浏览器指向以下 URL：

```
http://localhost:8080/order/
```

9.1.8.2　使用 Maven 运行 order 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 如果数据库服务器还未运行，则按照 2.4 节中的说明启动该服务器。
3. 在一个终端窗口中导航至：

```
tut-install/examples/persistence/order/
```

4. 输入以下命令：

```
mvn install
```

这会编译源文件，并将应用打包到一个 WAR 文件，这个文件位于 *tut-install*/examples/persistence/order/target/order.war。然后将这个 WAR 文件部署到 GlassFish 服务器实例。

5. 要创建和更新订单数据，打开 Web 浏览器访问以下 URL：

```
http://localhost:8080/order/
```

9.2　roster 应用

roster 应用为娱乐运动联盟的球员维护球队花名册。这个应用有 4 个部分：Java 持久化 API 实体（Player、Team 和 League），一个有状态会话 bean（RequestBean）、一个应用客户端（RosterClient）以及 3 个辅助类（PlayerDetails、TeamDetails 和

LeagueDetails)。

从功能上讲，roster 与 order 应用很类似，另外增加了 order 没有的 3 个新特性：多对多关系、实体继承和部署时的自动表创建。

对于 roster 应用，Java DB 数据库中的数据库模式如图 9-2 所示。

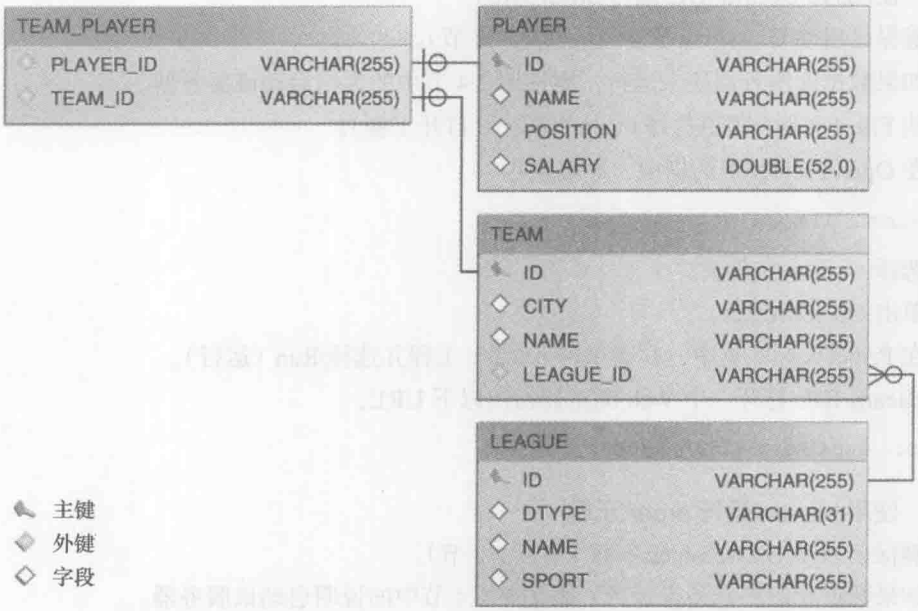


图 9-2 roster 应用的数据库模式

说明 在这个图中，为简单起见，表名省略了 PERSISTENCE_ROSTER_ 前缀。

9.2.1 roster 应用中的关系

娱乐运动系统有以下关系：

- 一个球员可以参加多个球队。
- 一个球队可以有多个球员。
- 一个球队只能参加一个联盟。
- 一个联盟可以有多个球队。

在 roster 中，这个系统反映为 Player、Team 和 League 实体之间的以下关系：

- 球员（Player）与球队（Team）之间有一种多对多关系。
- 球队（Team）与联盟（League）之间有一种多对一关系。

9.2.1.1 roster 中的多对多关系

Player 和 Team 之间的多对多关系使用 @ManyToMany 注解来指定。在 Team.java

中，用 @ManyToMany 注解装饰 getPlayers 方法：

```
@ManyToMany
@JoinTable(
    name="PERSISTENCE_ROSTER_TEAM_PLAYER",
    joinColumns=
        @JoinColumn(name="TEAM_ID", referencedColumnName="ID"),
    inverseJoinColumns=
        @JoinColumn(name="PLAYER_ID", referencedColumnName="ID")
)
public Collection<Player> getPlayers() {
    return players;
}
```

@JoinTable 注解指定一个数据库表将球员 ID 与球队 ID 关联。指定 @JoinTable 的实体是关系的所有者，所以在 Team 实体与 Player 实体的关系中，Team 实体是这个关系的所有者。由于 roster 使用了部署时自动表创建，所以容器会创建一个名为 PERSISTENCE_ROSTER_TEAM_PLAYER 的联接表。

在 Team 与 Player 的关系中，Player 是关系的接受方或非所属方。与一对一关系和多对一关系一样，要用关系注解中的 mappedBy 元素指示非所属方。由于 Player 和 Team 之间的关系是双向的，所以选择哪个实体作为这个关系的所有者是任意的。

在 Player.java 中，用 @ManyToMany 注解来装饰 getTeams 方法：

```
@ManyToMany(mappedBy="players")
public Collection<Team> getTeams() {
    return teams;
}
```

9.2.2　roster 应用中的实体继承

roster 应用展示了如何使用实体继承，参见 8.2 节中的介绍。

roster 中的 League 实体是一个抽象实体，有两个具体子类：SummerLeague 和 WinterLeague。因为 League 是一个抽象类，所以不能实例化：

```
@Entity
@Table(name = "PERSISTENCE_ROSTER_LEAGUE")
public abstract class League implements Serializable { ... }
```

实际上，创建联盟时，客户端会使用 SummerLeague 或 WinterLeague。SummerLeague 和 WinterLeague 继承了 League 中定义的持久化属性，另外只增加了一个构造函数来验证 sport 参数与这个季节联盟允许的运动类型是否匹配。例如，下面给出了 SummerLeague 实体：

```
@Entity
public class SummerLeague extends League implements Serializable {

    /** Creates a new instance of SummerLeague */
```

```
    public SummerLeague() {
    }

    public SummerLeague(String id, String name, String sport)
            throws IncorrectSportException {
        this.id = id;
        this.name = name;
        if (sport.equalsIgnoreCase("swimming") ||
                sport.equalsIgnoreCase("soccer") ||
                sport.equalsIgnoreCase("basketball") ||
                sport.equalsIgnoreCase("baseball")) {
            this.sport = sport;
        } else {
            throw new IncorrectSportException("Sport is not a summer sport.");
        }
    }
}
```

roster 应用使用默认映射策略 InheritanceType.SINGLE_TABLE，所以 @Inheritance 注解不是必要的。如果希望使用一个不同的映射策略，则可以用 @Inheritance 装饰 League，并在 strategy 元素中指定映射策略：

```
@Entity
@Inheritance(strategy=JOINED)
@Table(name="PERSISTENCE_ROSTER_LEAGUE")
public abstract class League implements Serializable { ... }
```

roster 应用使用了默认鉴别器列名，所以 @DiscriminatorColumn 注解不是必要的。因为 roster 中使用了自动表生成，所以持久化提供者会在 PERSISTENCE_ROSTER_LEAGUE 表中创建一个名为 DTYPE 的鉴别器列，其中将存储用于创建联盟的继承实体的名字。如果希望鉴别器列使用一个不同的名字，则可以用 @DiscriminatorColumn 装饰 League，并设置 name 元素：

```
@Entity
@DiscriminatorColumn(name="DISCRIMINATOR")
@Table(name="PERSISTENCE_ROSTER_LEAGUE")
public abstract class League implements Serializable { ... }
```

9.2.3 roster 应用中的 Criteria 查询

roster 应用使用了 Criteria API 查询，这与 order 中使用 JPQL 查询不同。Criteria 查询是 Java 编程语言编写的类型安全的查询，在 roster 的业务层定义，具体在 RequestBean 有状态会话 bean 中定义。

9.2.3.1 roster 应用中的元模型类

元模型类用于对实体的属性建模，Criteria 查询使用元模型类导航到实体的属性。roster 中的每个实体类都有一个相应的元模型类，在编译时生成，与实体的包名相同，后

面追加一个下划线字符（_）。例如，roster.entity.Player 实体有一个相应的元模型类 roster.entity.Player_。

实体类中的每个持久化字段或属性在实体的元模型类中都有一个相应的属性。对于 Player 实体，相应的元模型类如下：

```
@StaticMetamodel(Player.class)
public class Player_ {
    public static volatile SingularAttribute<Player, String> id;
    public static volatile SingularAttribute<Player, String> name;
    public static volatile SingularAttribute<Player, String> position;
    public static volatile SingularAttribute<Player, Double> salary;
    public static volatile CollectionAttribute<Player, Team> teams;
}
```

9.2.3.2　在 RequestBean 中获得一个 CriteriaBuilder 实例

CriteriaBuilder 接口定义了一些方法，可以用来创建 criteria 查询对象，还可以创建表达式来修改这些查询对象。RequestBean 使用一个 @PostConstruct 方法 init 来创建 CriteriaBuilder 实例：

```
@PersistenceContext
private EntityManager em;
private CriteriaBuilder cb;

@PostConstruct
private void init() {
    cb = em.getCriteriaBuilder();
}
```

EntityManager 实例在运行时注入，然后通过调用 getCriteriaBuilder 使用这个 EntityManager 对象创建 CriteriaBuilder 实例。CriteriaBuilder 实例在 @PostConstruct 方法中创建，以确保 EntityManager 实例已经由企业 bean 容器注入。

9.2.3.3　在 RequestBean 的业务方法中创建 Criteria 查询

RequestBean 中的很多业务方法定义了 Criteria 查询。业务方法 getPlayersByPosition 返回一个球队中某个特定位置上的球员列表：

```
public List<PlayerDetails> getPlayersByPosition(String position) {
    logger.info("getPlayersByPosition");
    List<Player> players = null;

    try {
        CriteriaQuery<Player> cq = cb.createQuery(Player.class);
        if (cq != null) {
            Root<Player> player = cq.from(Player.class);

            // set the where clause
            cq.where(cb.equal(player.get(Player_.position), position));
```

```
            cq.select(player);
            TypedQuery<Player> q = em.createQuery(cq);
            players = q.getResultList();
        }
        return copyPlayersToDetails(players);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}
```

创建查询对象时，要调用 CriteriaBuilder 对象的 createQuery 方法，并将类型设置为 Player，因为这个查询会返回一个球员列表。

查询根是一个基实体，查询将从这个基实体导航，来查找实体的属性和相关实体，可以调用查询对象的 from 方法创建查询根。这会设置查询的 FROM 子句。

通过在查询对象上调用 where 方法可以设置 WHERE 子句，这会根据一个表达式的条件限制查询的结果。CriteriaBuilder.equal 方法会完成两个表达式的比较。在 getPlayersByPosition 中，Player_ 元模型类的 position 属性（通过调用查询根的 get 方法来访问）要与传入 getPlayersByPosition 的 position 参数比较。

查询的 SELECT 子句通过调用查询对象的 select 方法来设置。这个查询将返回 Player 实体，所以查询根对象作为参数传入 select。

调用 EntityManager.createQuery 后，查询对象就准备就绪可以执行了，EntityManager.createQuery 返回一个 TypedQuery<T> 对象，T 为查询类型，在这里就是 Player。这个有类型的查询对象将用来执行查询，具体在调用 getResultList 方法时执行，并且会返回一个 List<Player> 集合。

9.2.4 roster 应用中的自动表生成

在部署时，GlassFish 服务器会自动删除和创建 roster 使用的数据库表。为此，要在 persistence.xml 中 将 javax.persistence.schema-generation.database.action 属性设置为 drop-and-create：

```
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="em" transaction-type="JTA">
    <jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    <properties>
      <property name="javax.persistence.schema-generation.database.action"
                value="drop-and-create"/>
    </properties>
  </persistence-unit>
</persistence>
```

9.2.5 运行 roster 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 roster 应用。

9.2.5.1 使用 NetBeans IDE 运行 roster 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 如果数据库服务器还未运行，则按照 2.4 节中的说明启动该服务器。
3. 从 File（文件）菜单选择 Open Project（打开工程）。
4. 在 Open Project 对话框中，导航至：

```
tut-install/examples/persistence
```

5. 选择 roster 文件夹。
6. 选择 Open Required Projects（打开所需工程）复选框。
7. 单击 Open Project。
8. 在 Projects（工程）标签页中，右键单击 roster 工程并选择 Build（构建）。

这会编译、打包 EAR 并将其部署到 GlassFish 服务器。

可以在 Output（输出）标签页中看到应用客户端的以下部分输出：

```
List all players in team T2:
P6 Ian Carlyle goalkeeper 555.0
P7 Rebecca Struthers midfielder 777.0
P8 Anne Anderson forward 65.0
P9 Jan Wesley defender 100.0
P10 Terry Smithson midfielder 100.0

List all teams in league L1:
T1 Honey Bees Visalia
T2 Gophers Manteca
T5 Crows Orland

List all defenders:
P2 Alice Smith defender 505.0
P5 Barney Bold defender 100.0
P9 Jan Wesley defender 100.0
P22 Janice Walker defender 857.0
P25 Frank Fletcher defender 399.0
```

9.2.5.2 使用 Maven 运行 roster 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 如果数据库服务器还未运行，则按照 2.4 节中的说明启动该服务器。
3. 在一个终端窗口中导航至：

```
tut-install/examples/persistence/roster/roster-ear/
```

4. 输入以下命令：

```
mvn install
```

这会编译源文件，并将应用打包到一个 EAR 文件，这个文件位于 *tut-install*/examples/persistence/roster/target/roster.ear。然后将 EAR 文件部署到 GlassFish 服务器。GlassFish 服务器再在部署时根据 persistence.xml 指定的设置删除并重新创建数据库表。

成功地部署 EAR 之后，会获取客户端桩，使用 GlassFish 服务器提供的 appclient 应用运行应用客户端。

接下来可以看到输出，开头部分如下：

```
[echo] running application client container.
[exec] List all players in team T2:
[exec] P6 Ian Carlyle goalkeeper 555.0
[exec] P7 Rebecca Struthers midfielder 777.0
[exec] P8 Anne Anderson forward 65.0
[exec] P9 Jan Wesley defender 100.0
[exec] P10 Terry Smithson midfielder 100.0

[exec] List all teams in league L1:
[exec] T1 Honey Bees Visalia
[exec] T2 Gophers Manteca
[exec] T5 Crows Orland

[exec] List all defenders:
[exec] P2 Alice Smith defender 505.0
[exec] P5 Barney Bold defender 100.0
[exec] P9 Jan Wesley defender 100.0
[exec] P22 Janice Walker defender 857.0
[exec] P25 Frank Fletcher defender 399.0
```

9.3 address-book 应用

address-book 示例应用是一个简单的 Web 应用，用来存储联系数据。这个应用使用了一个实体类 Contact，其中使用了 JavaBeans 验证（bean 验证）Java API 来验证存储在实体持久化属性中的数据，参见 8.1.2.4 节中的介绍。

9.3.1 address-book 中的 bean 验证约束

Contact 实体在持久化属性上使用了 @NotNull、@Pattern 和 @Past 约束。

@NotNull 约束标志属性是一个必要字段。在持久存储或修改实体之前，这个属性必须设置为一个非 null 值。如果持久存储或修改实体时这个属性为 null，则 bean 验证会抛出一个验证错误。

@Pattern 约束定义了一个正则表达式，要求在持久存储或修改实体之前属性值必须匹配。这个约束在 address-book 中有两个不同的用法。

- email 字段上有一个 @Pattern 注解，其中声明的正则表达式匹配形式为*用户名@域名.顶级域*的 email 地址，只允许包含可用于 email 地址的合法字符。例如，username@example.com 可以通过验证，firstname.lastname@mail.example.com 也可以通过验证。不过，firstname,lastname@example.com 在本地名中包含一个非法的逗号字符，所以验证失败。
- mobilePhone 和 homePhone 字段都有一个 @Pattern 注解约束，定义了一个正则表达式，要求匹配形式为 (*xxx*) *xxx-xxxx* 的电话号码。

生日（birthday）字段要应用 @Past 约束，这必须是过去的一个日期（java.util.Date）。

下面是 Contact 实体类的相关部分：

```
@Entity
public class Contact implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
            + "[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
            + "(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9]"
            + "(?:[a-z0-9-]*[a-z0-9])?",
            message = "{invalid.email}")
    protected String email;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(javax.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

9.3.2 在 address-book 中为约束指定错误消息

Contact 实体中的一些约束指定了一个可选的消息：

```
@Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
        message = "{invalid.phonenumber}")
protected String homePhone;
```

@Pattern 约束中这个可选的消息元素会覆盖默认的验证消息。可以直接指定这个消息：

```
@Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
        message = "Invalid phone number!")
protected String homePhone;
```

不过，Contact 中的约束是 *tut-install*/examples/persistence/address-book/src/java/ 目录中资源包 ValidationMessages.properties 里的字符串。这样可以将验证消息放在一个属性文件中，从而能很容易地找到消息。被覆盖的 bean 验证消息必须放在默认包中名为 ValidationMessages.properties 的资源包属性文件中，对于本地化资源包，则形如 ValidationMessages_*locale-prefix*.properties。例如，ValidationMessages_es.properties 是西班牙语本地化环境中使用的资源包。

9.3.3 验证 JSF 应用的联系信息输入

address-book 应用使用一个 JSF Web 前端，允许用户输入联系信息。尽管 JSF 提供了表单输入验证机制，可以使用 Facelets XHTML 文件中的标记来完成验证，不过 address-book 没有使用这些验证标记。JSF 托管 bean（在这里就是 Contact 实体）中的 bean 验证约束会在表单提交时自动触发验证。

下面的代码段取自 Create.xhtml Facelets 文件，展示了用来创建新 Contact 实例的输入表单的一部分：

```
<h:form>
    <table columns="3" role="presentation">
        <tr>
            <td><h:outputLabel value="#{bundle.CreateContactLabel_firstName}"
                               for="firstName" /></td>
            <td><h:inputText id="firstName"
                             value="#{contactController.selected.firstName}"
                             title="#{bundle.CreateContactTitle_firstName}"/>
            </td>
            <td><h:message for="firstName" /></td>
        </tr>
        <tr>
            <td><h:outputLabel value="#{bundle.CreateContactLabel_lastName}"
                               for="lastName" /></td>
            <td><h:inputText id="lastName"
                             value="#{contactController.selected.lastName}"
                             title="#{bundle.CreateContactTitle_lastName}" />
            </td>
            <td><h:message for="lastName" /></td>
        </tr>
        ...
    </table>
</h:form>
```

<h:inputText> 标记 firstName 和 lastName 都绑定到无状态会话 bean ContactController 中 Contact 实体实例 selected 的属性。每个 <h:inputText> 标记有一个关联的 <h:message> 标记，可以显示验证错误消息。不过，这个表单不需要任何

JSF 验证标记。

9.3.4　运行 address-book 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 `address-book` 应用。

9.3.4.1　使用 NetBeans IDE 运行 address-book 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 如果数据库服务器还未运行，则按照 2.4 节中的说明启动该服务器。
3. 从 File（文件）菜单选择 Open Project（打开工程）。
4. 在 Open Project 对话框中，导航至：

```
tut-install/examples/persistence
```

5. 选择 `address-book` 文件夹。
6. 单击 Open Project。
7. 在 Projects（工程）标签页中，右键单击 `address-book` 工程并选择 Run（运行）。

应用部署后，Web 浏览器窗口会显示以下 URL：

```
http://localhost:8080/address-book/
```

8. 单击 Show All Contact Items（显示所有联系信息项），然后单击 Create New Contact（创建新联系信息）。在域中输入值，再单击 Save（保存）。

如果输入的某个值违反了 `Contact` 中的约束，就会在这个输入域旁边用红色显示一个错误消息。

9.3.4.2　使用 Maven 运行 address-book 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 如果数据库服务器还未运行，则按照 2.4 节中的说明启动该服务器。
3. 在一个终端窗口中导航至：

```
tut-install/examples/persistence/address-book/
```

4. 输入以下命令：

```
mvn install
```

这会编译 `address-book` 应用并装配为一个 WAR。然后将 WAR 文件部署到 GlassFish 服务器。

5. 打开一个 Web 浏览器窗口，输入以下 URL：

```
http://localhost:8080/address-book/
```

6. 单击 Show All Contact Items（显示所有联系信息项），然后单击 Create New Contact（创建新联系信息）。在输入域中输入值，然后单击 Save（保存）。

如果输入的某个值违反了 `Contact` 中的约束，就会在这个输入域旁边用红色显示一个错误消息。

Chapter 10

第 10 章

Java 持久化查询语言

Java 持久化查询语言可以定义如何查询实体及其持久化状态。利用这个查询语言，可以编写可移植的查询，而不论底层数据库是什么。

Java 持久化查询语言使用实体的抽象持久化模式（包括其关系）表示数据模型，并根据这个数据模型定义操作符和表达式。查询的作用域涵盖打包到同一个持久化单元的相关实体的抽象模式。这个查询语言使用一种类 SQL 的语法，根据实体抽象模式类型和相互之间的关系来选择对象或值。

本章以前几章介绍的内容为基础。有关概念参见第 8 章。有关的代码示例参见第 9 章。

本章将介绍以下内容：

- 查询语言术语
- 使用 Java 持久化查询语言创建查询
- 简化的查询语言语法
- 示例查询
- 完整的查询语言语法

10.1 查询语言术语

下面定义了本章中用到的一些术语。

- **抽象模式**（Abstract schema）：这是查询所处理的持久化模式抽象（包括持久化实体、其状态和关系）。查询语言将持久化模式抽象上的查询转换为在实体映射的数据库模式上执行的查询。

- **抽象模式类型**（Abstract schema type）：抽象模式中实体的持久化属性的类型。也就是说，一个实体中的各个持久化字段或属性在抽象模式中都有一个相应的同类型状态字段。一个实体的抽象模式类型由实体类和 Java 语言注解提供的元数据信息得出。
- **巴科斯范式**（Backus-Naur Form，BNF）：这是描述高级语言语法的一种记法。本章中的语法图都采用 BNF 记法。
- **导航**（Navigation）：遍历查询语言表达式中的关系。导航操作符是一个点号（.）。
- **路径表达式**（Path expression）：导航到实体状态或关系字段的一个表达式。
- **状态字段**（State Field）：实体的一个持久化字段。
- **关系字段**（Relationship Field）：实体的一个持久化字段，其类型是相关实体的抽象模式类型。

10.2　使用 Java 持久化查询语言创建查询

通过使用 Java 持久化查询语言查询，`EntityManager.createQuery` 和 `EntityManager.createNamedQuery` 方法可以用来查询数据库。

`createQuery` 方法用来创建**动态查询**（dynamic query），这是直接在应用业务逻辑中定义的查询：

```
public List findWithName(String name) {
return em.createQuery(
    "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

`createNamedQuery` 方法用来创建**静态查询**（static query），也就是通过使用 `javax.persistence.NamedQuery` 注解在元数据中定义的查询。`@NamedQuery` 的 `name` 元素指定了将用于 `createNamedQuery` 方法的查询名。`@NamedQuery` 的 `query` 元素是如下查询：

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

下面是 `createNamedQuery` 的一个例子，这里使用了 `@NamedQuery`：

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

10.2.1 查询中的命名参数

命名参数是有冒号（:）前缀的查询参数。查询中的命名参数通过下面的方法绑定到一个实参：

```
javax.persistence.Query.setParameter(String name, Object value)
```

在下面的例子中，通过调用 Query.setParameter，将 findWithName 业务方法的 name 实参与查询中的 :custName 命名参数绑定：

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", name)
        .getResultList();
}
```

命名参数是区分大小写的，在动态查询和静态查询中都可以使用。

10.2.2 查询中的位置参数

查询中也可以使用位置参数而不是命名参数。位置参数有一个问号（?）前缀，后面是参数在查询中的位置号。Query.setParameter(integer position,Object value) 方法可以用来设置参数值。

在下面的例子中，重写了前面的 findWithName 业务方法，这里会使用输入参数：

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")
        .setParameter(1, name)
        .getResultList();
}
```

输入参数从 1 开始编号。输入参数是区分大小写的，在动态查询和静态查询中都可以使用。

10.3 简化的查询语言语法

这一节简要介绍查询语言的语法，以便更顺利地学习 10.4 节。等你掌握了足够的知识，希望更详细地学习语法时，可以再来学习 10.5 节。

10.3.1 选择语句

选择查询有 6 个子句：SELECT、FROM、WHERE、GROUP BY、HAVING 和 ORDER BY。SELECT 和 FROM 子句是必要的，而 WHERE、GROUP BY、HAVING 和 ORDER BY 子句是可选的。下面是选择查询的高层 BNF 语法：

```
QL_statement ::= select_clause from_clause
  [where_clause][groupby_clause][having_clause][orderby_clause]
```

这个 BNF 语法定义了以下子句。

- SELECT 子句定义查询返回的对象或值的类型。
- FROM 子句通过声明一个或多个**标识变量**（identification variable）来定义查询的范围，可以在 SELECT 和 WHERE 子句中引用这些标识变量。标识变量表示以下元素：
  - 实体的抽象模式名
  - 集合关系的一个元素
  - 单值关系的一个元素
  - 集合的一个成员（集合作为一对多关系中的多方）
- WHERE 子句是一个条件表达式，可以限制由查询获取的对象或值。尽管这个子句是可选的，不过大多数查询都有一个 WHERE 子句。
- GROUP BY 子句根据一组属性对查询结果分组。
- HAVING 子句与 GROUP BY 子句结合使用，根据一个条件表达式进一步限制查询结果。
- ORDER BY 子句将查询返回的对象或值按指定的顺序排序。

10.3.2 更新和删除语句

更新和删除语句提供了对实体集合的批处理操作。这些语句语法如下：

```
update_statement :: = update_clause [where_clause]
delete_statement :: = delete_clause [where_clause]
```

更新和删除子句确定要更新或删除的实体的类型。WHERE 子句可以用来限制更新或删除操作的范围。

10.4 示例查询

下面的查询选自 roster 应用的 Player 实体，参见 9.2 节。

10.4.1 简单查询

如果你对查询语言还不熟悉，那么这些简单查询可以帮助你入门。

10.4.1.1 基本选择查询

```
SELECT p
FROM Player p
```

- **获取的数据**：所有球员。
- **描述**：这个 FROM 子句声明一个标识变量 p，忽略了可选的关键字 AS。如果包括 AS 关键字，这个子句可以写为：

```
FROM Player AS p
```

Player 元素是 Player 实体的抽象模式名。

- **参见**：10.5.3.2 节。

10.4.1.2 删除重复值

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = ?1
```

- 获取的数据：查询参数所指定位置上的球员。
- 描述：DISTINCT 关键字可以删除重复的值。WHERE 子句通过检查 position(Player 实体的一个持久化字段）来限制所获取的球员。?1 元素指示查询的输入参数。
- 参见：10.5.5.2 节和 10.5.6.2 节。

10.4.1.3 使用命名参数

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = :position AND p.name = :name
```

- **获取的数据**：有指定位置和名字的球员。
- **描述**：position 和 name 元素是 Player 实体的持久化字段。WHERE 子句将这些字段的值与使用 Query.setNamedParameter 方法设置的查询命名参数进行比较。查询语言用一个冒号（:）后面跟一个标识符来表示一个命名输入参数。第一个输入参数是 :position，第二个是 :name。

10.4.2 导航到相关实体的查询

在查询语言中，表达式可以遍历或导航到相关实体。这些表达式正是 Java 持久化查询语言和 SQL 之间的主要区别。查询可以导航到相关实体，SQL 则要联接表。

10.4.2.1 关系的简单查询

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
```

- **获取的数据**：一个球队的所有球员。
- **描述**：FROM 子句声明了两个标识变量：p 和 t。p 变量表示 Player 实体，t 变量表示相关的 Team 实体。t 的声明引用了前面声明的 p 变量。IN 关键字指示 teams 是相关实体的一个集合。p.teams 表达式从一个 Player 导航到它的相关 Team。p.teams 表达式中的点号是导航操作符。

也可以用 JOIN 语句将这个查询写为：

```
SELECT DISTINCT p
FROM Player p JOIN p.teams t
```

这个查询还可以重写为：

```
SELECT DISTINCT p
FROM Player p
WHERE p.team IS NOT EMPTY
```

10.4.2.2 导航到单值关系字段

可以使用 JOIN 子句导航到一个单值关系字段：

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = 'soccer' OR l.sport ='football'
```

在这个例子中，这个查询会返回足球或橄榄球联盟的所有球队。

10.4.2.3 用输入参数遍历关系

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) AS t
WHERE t.city = :city
```

- **获取的数据**：指定城市里球队中的球员。
- **描述**：这个查询与前面的例子很类似，不过增加了一个输入参数。FROM 子句中的 AS 关键字是可选的。在 WHERE 子句中，持久化变量 city 前面的点号是一个分隔符，而不是一个导航操作符。严格地讲，表达式可以导航到关系字段（关联的实体）而不能导航到持久化字段。要访问一个持久化字段，表达式就要使用点号作为分隔符。

表达式不能对集合关系字段导航（或作进一步限定）。在表达式语法中，集合值字段是一个终结符。由于 teams 字段是一个集合，所以 WHERE 子句不能指定 p.teams.city（这是一个非法的表达式）。

- **参见**：10.5.4 节。

10.4.2.4 遍历多个关系

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league = :league
```

- **获取的数据**：属于指定联盟的球员。
- **描述**：这个查询中的表达式对两个关系导航。p.teams 表达式导航 Player-Team 关系，而 t.league 表达式导航 Team-League 关系。

在其他例子中，输入参数是 String 对象；但在这个例子中，参数是一个类型为 League 的对象。这个类型与 WHERE 子句的比较表达式中的 league 关系字段匹配。

10.4.2.5 根据关联字段导航

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

- **获取的数据**：参加指定运动的球员。
- **描述**：sport 持久化字段属于 League 实体。要得到 sport 字段，查询必须首先从 Player 实体导航到 Team(p.teams)，然后从 Team 导航到 League 实体(t.league)。因为这不是一个集合，所以 league 关系字段后面可以跟 sport 持久化字段。

10.4.3 包含其他条件表达式的查询

每个 WHERE 子句都必须指定一个条件表达式，而且条件表达式可以有很多种。在前面的例子中，条件表达式都是用于检查相等性的比较表达式。下面的例子展示了一些其他类型的条件表达式。关于条件表达式的全面描述，参见 10.5.5 节。

10.4.3.1 LIKE 表达式

```
SELECT p
FROM Player p
WHERE p.name LIKE 'Mich%'
```

- **获取的数据**：名字以 "Mich" 开头的所有球员。
- **描述**：LIKE 表达式使用通配符来搜索与通配模式匹配的字符串。在这里，查询使用 LIKE 表达式和 % 通配符查找所有名字以字符串 "Mich" 开头的球员。例如，"Michael" 和 "Michelle" 都与这个通配模式匹配。
- **参见**：10.5.5.7 节。

10.4.3.2 IS NULL 表达式

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

- **获取的数据**：所有与联盟没有关联的球队。
- **描述**：IS NULL 表达式可以用来检查两个实体之间是否设置了一个关系。在这里，查询检查球队是否与联盟有关联，并返回没有关联联盟的球队。
- **参见**：10.5.5.8 节和 10.5.5.14 节。

10.4.3.3 IS EMPTY 表达式

```
SELECT p
FROM Player p
WHERE p.teams IS EMPTY
```

- **获取的数据**：不属于任何球队的球员。
- **描述**：Player 实体的 teams 关系字段是一个集合。如果一个球员不属于任何球队，teams 集合就为空，这个条件表达式为 TRUE。
- **参见**：10.5.5.9 节。

10.4.3.4　BETWEEN 表达式

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

- **获取的数据**：工资在指定工资范围内的球员。
- **描述**：这个 BETWEEN 表达式有 3 个算术表达式：一个持久化字段（p.salary）和两个输入参数（:lowerSalary 和 :higherSalary）。下面的表达式与这个 BETWEEN 表达式等价：

```
p.salary >= :lowerSalary AND p.salary <= :higherSalary
```

- **参见**：10.5.5.5 节。

10.4.3.5　比较操作符

```
SELECT DISTINCT p1
FROM Player p1, Player p2
WHERE p1.salary > p2.salary AND p2.name = :name
```

- **获取的数据**：比指定名字的那个球员工资高的所有球员。
- **描述**：FROM 子句声明了两个有相同类型（Player）的标识变量（p1 和 p2）。这两个标识变量都是必需的，因为 WHERE 子句要将一个球员（p2）的工资与其他球员（p1）的工资进行比较。
- **参见**：10.5.3.2 节。

10.4.4　批处理更新和删除

下面的例子展示了如何在查询中使用 UPDATE 和 DELETE 表达式。UPDATE 和 DELETE 根据 WHERE 子句中设置的一个或多个条件处理多个实体。UPDATE 和 DELETE 查询中的 WHERE 子句与 SELECT 查询中的规则相同。

10.4.4.1　更新查询

```
UPDATE Player p
SET p.status = 'inactive'
WHERE p.lastPlayed < :inactiveThresholdDate
```

- **描述**：如果球员参加的最后一场比赛在 inactiveThresholdDate 指定的日期之前进行，则这个查询将把这组球员的状态设置为 inactive。

10.4.4.2　删除查询

```
DELETE
FROM Player p
WHERE p.status = 'inactive'
AND p.teams IS EMPTY
```

- **描述**：这个查询会删除所有不在任何球队中而且为非活动状态的球员。

10.5 完整的查询语言语法

这一节讨论 Java 持久化 API 2.0 规范定义的查询语言语法，参见 http://jcp.org/en/jsr/detail?id=317。下面的很多内容都是对规范的解释或者直接引用。

10.5.1 BNF 符号

表 10-1 给出了这一章中使用的 BNF 符号。

表 10-1 BNF 符号小结

符 号	描 述
::=	符号左边的元素由右边的构造定义
*	前面的构造可以出现 0 次或多次
{...}	大括号中的构造分组在一起
[...]	中括号中的构造是可选的
|	异或
BOLDFACE	关键字；尽管在 BNF 图中这是大写的，但关键字并不区分大小写
空白符	空白符可以是一个空格、水平制表符或换行符

10.5.2 Java 持久化查询语言的 BNF 文法

下面给出查询语言的完整 BNF 图：

```
QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
    [having_clause] [orderby_clause]
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
from_clause ::=
    FROM identification_variable_declaration
        {, {identification_variable_declaration |
            collection_member_declaration}}*
identification_variable_declaration ::=
        range_variable_declaration { join | fetch_join }*
range_variable_declaration ::= abstract_schema_name [AS]
        identification_variable
join ::= join_spec join_association_path_expression [AS]
        identification_variable
fetch_join ::= join_specFETCH join_association_path_expression
association_path_expression ::=
        collection_valued_path_expression |
        single_valued_association_path_expression
join_spec::= [LEFT [OUTER] |INNER] JOIN
join_association_path_expression ::=
        join_collection_valued_path_expression |
        join_single_valued_association_path_expression
```

```
join_collection_valued_path_expression::=
    identification_variable.collection_valued_association_field
join_single_valued_association_path_expression::=
        identification_variable.single_valued_association_field
collection_member_declaration ::=
        IN (collection_valued_path_expression) [AS]
        identification_variable
single_valued_path_expression ::=
        state_field_path_expression |
        single_valued_association_path_expression
state_field_path_expression ::=
    {identification_variable |
    single_valued_association_path_expression}.state_field
single_valued_association_path_expression ::=
    identification_variable.{single_valued_association_field.}*
    single_valued_association_field
collection_valued_path_expression ::=
    identification_variable.{single_valued_association_field.}*
    collection_valued_association_field
state_field ::=
    {embedded_class_state_field.}*simple_state_field
update_clause ::=UPDATE abstract_schema_name [[AS]
    identification_variable] SET update_item {, update_item}*
update_item ::= [identification_variable.]{state_field |
    single_valued_association_field} = new_value
new_value ::=
     simple_arithmetic_expression |
    string_primary |
    datetime_primary |
    boolean_primary |
    enum_primary simple_entity_expression |
    NULL
delete_clause ::= DELETE FROM abstract_schema_name [[AS]
    identification_variable]
select_clause ::= SELECT [DISTINCT] select_expression {,
    select_expression}*
select_expression ::=
    single_valued_path_expression |
    aggregate_expression |
    identification_variable |
    OBJECT(identification_variable) |
    constructor_expression
constructor_expression ::=
    NEW constructor_name(constructor_item {,
    constructor_item}*)
constructor_item ::= single_valued_path_expression |
    aggregate_expression
aggregate_expression ::=
    {AVG |MAX |MIN |SUM} ([DISTINCT]
        state_field_path_expression) |
```

```
    COUNT ([DISTINCT] identification_variable |
        state_field_path_expression |
        single_valued_association_path_expression)
where_clause ::= WHERE conditional_expression
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression
having_clause ::= HAVING conditional_expression
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression [ASC |DESC]
subquery ::= simple_select_clause subquery_from_clause
    [where_clause] [groupby_clause] [having_clause]
subquery_from_clause ::=
    FROM subselect_identification_variable_declaration
        {, subselect_identification_variable_declaration}*
subselect_identification_variable_declaration ::=
    identification_variable_declaration |
    association_path_expression [AS] identification_variable |
    collection_member_declaration
simple_select_clause ::= SELECT [DISTINCT]
    simple_select_expression
simple_select_expression::=
    single_valued_path_expression |
    aggregate_expression |
    identification_variable
conditional_expression ::= conditional_term |
    conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND
    conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary ::= simple_cond_expression |(
    conditional_expression)
simple_cond_expression ::=
    comparison_expression |
    between_expression |
    like_expression |
    in_expression |
    null_comparison_expression |
    empty_collection_comparison_expression |
    collection_member_expression |
    exists_expression
between_expression ::=
    arithmetic_expression [NOT] BETWEEN
        arithmetic_expressionAND arithmetic_expression |
    string_expression [NOT] BETWEEN string_expression AND
        string_expression |
    datetime_expression [NOT] BETWEEN
        datetime_expression AND datetime_expression
in_expression ::=
    state_field_path_expression [NOT] IN (in_item {, in_item}*
    | subquery)
```

```
in_item ::= literal | input_parameter
like_expression ::=
    string_expression [NOT] LIKE pattern_value [ESCAPE
        escape_character]
null_comparison_expression ::=
    {single_valued_path_expression | input_parameter} IS [NOT]
        NULL
empty_collection_comparison_expression ::=
    collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_expression
    [NOT] MEMBER [OF] collection_valued_path_expression
exists_expression::= [NOT] EXISTS (subquery)
all_or_any_expression ::= {ALL |ANY |SOME} (subquery)
comparison_expression ::=
    string_expression comparison_operator {string_expression |
    all_or_any_expression} |
    boolean_expression {= |<> } {boolean_expression |
    all_or_any_expression} |
    enum_expression {= |<> } {enum_expression |
    all_or_any_expression} |
    datetime_expression comparison_operator
        {datetime_expression | all_or_any_expression} |
    entity_expression {= |<> } {entity_expression |
    all_or_any_expression} |
    arithmetic_expression comparison_operator
        {arithmetic_expression | all_or_any_expression}
comparison_operator ::= = |> |>= |< |<= |<>
arithmetic_expression ::= simple_arithmetic_expression |
    (subquery)
simple_arithmetic_expression ::=
    arithmetic_term | simple_arithmetic_expression {+ |- }
        arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term {* |/ }
    arithmetic_factor
arithmetic_factor ::= [{+ |- }] arithmetic_primary
arithmetic_primary ::=
    state_field_path_expression |
    numeric_literal |
    (simple_arithmetic_expression) |
    input_parameter |
    functions_returning_numerics |
    aggregate_expression
string_expression ::= string_primary | (subquery)
string_primary ::=
    state_field_path_expression |
    string_literal |
    input_parameter |
    functions_returning_strings |
    aggregate_expression
datetime_expression ::= datetime_primary | (subquery)
```

```
datetime_primary ::=
    state_field_path_expression |
    input_parameter |
    functions_returning_datetime |
    aggregate_expression
boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::=
    state_field_path_expression |
    boolean_literal |
    input_parameter
enum_expression ::= enum_primary | (subquery)
enum_primary ::=
    state_field_path_expression |
    enum_literal |
    input_parameter
entity_expression ::=
    single_valued_association_path_expression |
        simple_entity_expression
simple_entity_expression ::=
    identification_variable |
    input_parameter
functions_returning_numerics::=
    LENGTH(string_primary) |
    LOCATE(string_primary, string_primary[,
        simple_arithmetic_expression]) |
    ABS(simple_arithmetic_expression) |
    SQRT(simple_arithmetic_expression) |
    MOD(simple_arithmetic_expression,
        simple_arithmetic_expression) |
    SIZE(collection_valued_path_expression)
functions_returning_datetime ::=
    CURRENT_DATE |
    CURRENT_TIME |
    CURRENT_TIMESTAMP
functions_returning_strings ::=
    CONCAT(string_primary, string_primary) |
    SUBSTRING(string_primary,
        simple_arithmetic_expression,
        simple_arithmetic_expression)|
    TRIM([[trim_specification] [trim_character] FROM]
        string_primary) |
    LOWER(string_primary) |
    UPPER(string_primary)
trim_specification ::= LEADING | TRAILING | BOTH
```

10.5.3 FROM 子句

FROM 子句通过声明标识变量来定义查询的范围。

10.5.3.1　标识符

标识符是由一个或多个字符组成的序列。第一个字符必须是 Java 编程语言标识符中的合法首字符（字母、$、_），本章后面将把 Java 编程语言简称为“Java”。这个序列中后面的各个字符必须是 Java 标识符中合法的非首字符（字母、数字、$、_）。（有关的详细信息，参见 `Character` 类的 `isJavaIdentifierStart` 和 `isJavaIdentifierPart` 方法的 Java SE API 文档。）问号（?）是查询语言中的一个保留字符，不能用在标识符中。

查询语言标识符是区分大小写的，但有两个例外情况：

- 关键字
- 标识变量

标识符不能与查询语言关键字相同。下面给出查询语言关键字列表：

```
ABS
ALL
AND
ANY
AS
ASC
AVG
BETWEEN
BIT_LENGTH
BOTH
BY
CASE
CHAR_LENGTH
CHARACTER_LENGTH
CLASS
COALESCE
CONCAT
COUNT
CURRENT_DATE
CURRENT_TIMESTAMP
DELETE
DESC
DISTINCT
ELSE
EMPTY
END
ENTRY
ESCAPE
EXISTS
FALSE
FETCH
FROM
```

```
GROUP
HAVING
IN
INDEX
INNER
IS
JOIN
KEY
LEADING
LEFT
LENGTH
LIKE
LOCATE
LOWER
MAX
MEMBER
MIN
MOD
NEW
NOT
NULL
NULLIF
OBJECT
OF
OR
ORDER
OUTER
POSITION
SELECT
SET
SIZE
SOME
SQRT
SUBSTRING
SUM
THEN
TRAILING
TRIM
TRUE
TYPE
UNKNOWN
UPDATE
UPPER
VALUE
WHEN
WHERE
```

不建议使用 SQL 关键字作为标识符，因为关键字列表可能还要扩展，将来有可能包含其他 SQL 保留字。

10.5.3.2 标识变量

标识变量是 FROM 子句中声明的一个标识符。尽管 SELECT 和 WHERE 子句可以引用标识变量，但不能声明标识变量。所有标识变量都必须在 FROM 子句中声明。

由于标识变量是一个标识符，所以它与标识符有相同的命名约定和限制，只不过标识变量不区分大小写。例如，标识变量不能与查询语言关键字相同（更多命名规则参见 10.5.3.1 节）。另外，在一个给定的持久化单元中，标识变量名不能与任何实体或抽象模式的名字相同。

FROM 子句可以包含多个声明，用逗号分隔。一个声明可以引用之前（左边）已声明的另一个标识变量。在下面的 FROM 子句中，变量 t 引用了之前声明的变量 p：

```
FROM Player p, IN (p.teams) AS t
```

即使未在 WHERE 子句中使用，标识变量的声明也可能会影响查询的结果。例如，比较下面两个查询。下面这个查询会返回所有球员，而不论他们是否属于一个球队：

```
SELECT p
FROM Player p
```

与之不同，由于声明了 t 标识变量，因此下面的查询会获取属于某个球队的所有球员：

```
SELECT p
FROM Player p, IN (p.teams) AS t
```

下面这个查询返回的结果与上一个查询相同，不过 WHERE 子句使它更易读：

```
SELECT p
FROM Player p
WHERE p.teams IS NOT EMPTY
```

标识变量总是指示单个值的引用，其类型为声明中所使用表达式的类型。有两种不同类型的声明：范围变量和集合成员。

10.5.3.3 范围变量声明

要声明一个标识变量为抽象模式类型，可以指定一个范围变量声明。换句话说，标识变量可以完全涵盖一个实体的抽象模式类型。在下面的例子中，名为 p 的标识变量表示抽象模式 Player：

```
FROM Player p
```

范围变量声明可以包括可选的 AS 操作符：

```
FROM Player AS p
```

要得到对象，查询通常使用路径表达式在关系中导航。不过，对于那些不能通过导航得到的对象，可以使用范围变量声明来指定一个起点或查询根。

如果查询要比较相同抽象模式类型的多个值，则 FROM 子句必须为这个抽象模式声明多个标识变量：

```
FROM Player p1, Player p2
```

关于这一类查询的例子，参见 10.4.3.5 节。

10.5.3.4 集合成员声明

在一个一对多关系中，多方包括实体的一个集合。标识变量可以表示这个集合中的一个成员。要访问一个集合成员，变量声明中的路径表达式可以对抽象模式中的关系导航。（关于路径表达式的更多信息，参见 10.5.4 节。）由于路径表达式可以建立在另一个路径表达式基础上，因此导航可以遍历多个关系。更多信息参见 10.4.2.4 节。

集合成员声明必须包括 IN 操作符，不过可以省略可选的 AS 操作符。

在下面的例子中，抽象模式 Player 表示的实体有一个名为 teams 的关系字段。标识变量 t 表示 teams 集合的一个成员：

```
FROM Player p, IN (p.teams) t
```

10.5.3.5 联接

JOIN 操作符用于遍历实体间的关系，其功能与 IN 操作符类似。

在下面的例子中，这个查询在顾客和订单之间的关系上完成联接：

```
SELECT c
FROM Customer c JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000
```

INNER 关键字是可选的：

```
SELECT c
FROM Customer c INNER JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000
```

这些例子等价于下面的查询，这里使用了 IN 操作符：

```
SELECT c
FROM Customer c, IN (c.orders) o
WHERE c.status = 1 AND o.totalPrice > 10000
```

还可以联接一个单值关系：

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = :sport
```

LEFT JOIN 或 LEFT OUTER JOIN 会获取一组实体，这里可以空缺联接条件中的匹配值。OUTER 关键字是可选的：

```
SELECT c.name, o.totalPrice
FROM CustomerOrder o LEFT JOIN o.customer c
```

FETCH JOIN 作为一个联接操作有一个副作用，即运行查询时会返回关联的实体。在下

面的例子中，这个查询会返回一组部门，另外作为副作用，还会返回部门中关联的员工，尽管 SELECT 子句并没有显式获取这些员工：

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

10.5.4 路径表达式

在查询语言的语法中，路径表达式是非常重要的构造，原因有很多。首先，路径表达式定义了在抽象模式关系中的导航路径。这些路径定义会同时影响查询的范围和结果。其次，路径表达式可以出现在查询的任意主要子句中（SELECT、DELETE、HAVING、UPDATE、WHERE、FROM、GROUP BY 和 ORDER BY）。最后一点，尽管查询语言很大一部分是 SQL 的子集，但路径表达式有所不同，这是 SQL 没有的扩展内容。

10.5.4.1 路径表达式示例

在下面的例子中，WHERE 子句包含一个单值路径表达式（single_valued_path_expression）；p 是一个标识变量，salary 是 Player 的一个持久化字段：

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

在这里，WHERE 子句也包含一个单值路径表达式（single_valued_path_expression）；t 是一个标识变量，league 是一个单值关系字段，sport 是 league 的一个持久化字段：

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

在下面的例子中，WHERE 子句包含一个集合值路径表达式（collection_valued_path_expression）；p 是一个标识变量，teams 指定了一个集合值关系字段：

```
SELECT DISTINCT p
FROM Player p
WHERE p.teams IS EMPTY
```

10.5.4.2 表达式类型

路径表达式的类型就是最后一个元素所表示的对象的类型，可以是以下类型：

- 持久化字段
- 单值关系字段
- 集合值关系字段

例如，表达式 p.salary 的类型是 double，因为最后一个持久化字段（salary）的类型是 double。

在表达式 p.teams 中，最后一个元素是一个集合值关系字段（teams）。这个表达式的类型就是抽象模式类型 Team 的集合。由于 Team 是 Team 实体对应的抽象模式名，所以这个类型映射到 Team 实体。关于抽象模式类型映射的更多信息，参见 10.5.6.1 节。

10.5.4.3 导航

路径表达式允许查询导航到关联的实体。表达式末尾的元素确定了是否允许导航。如果表达式包含一个单值关系字段，那么可以继续对与这个字段关联的对象导航。不过，表达式不能对持久化字段或集合值关系字段导航。例如，表达式 p.teams.league.sport 是不合法的，因为 teams 是一个集合值关系字段。要访问 sport 字段，FROM 子句可以为 teams 字段定义一个名为 t 的标识变量：

```
FROM Player AS p, IN (p.teams) t
WHERE t.league.sport = 'soccer'
```

10.5.5 WHERE 子句

WHERE 子句指定一个条件表达式，用来限制查询返回的值。查询会返回数据库中保证这个条件表达式为 TRUE 的所有相应值。尽管通常都会指定 WHERE 子句，不过这个子句是可选的。如果省略了 WHERE 子句，查询就会返回所有值。WHERE 子句的高层语法如下所示：

```
where_clause ::= WHERE conditional_expression
```

10.5.5.1 直接量

有 4 种类型的直接量：字符串、数值、布尔值和枚举。

- **字符串直接量**：字符串直接量用单引号括起：

  ```
  'Duke'
  ```

  如果一个字符串直接量包含一个单引号，则要使用两个单引号来指示：

  ```
  'Duke''s'
  ```

  类似于 Java String，查询语言中的字符串直接量使用 Unicode 字符编码。
- **数值直接量**：有两种类型的数值直接量：绝对数值直接量和近似数值直接量。
  - 绝对数值直接量是没有小数点的数值，如 57，-233 和 +12。采用 Java 整数语法，绝对数值直接量支持 Java long 范围内的数。
  - 近似数值直接量是采用科学记数法的数值，如 57.，-85.7 和 +2.1。采用 Java 浮点直接量的语法，近似数值直接量支持 Java double 范围内的数。
- **布尔直接量**：布尔直接量是 TRUE 或 FALSE。这些关键字不区分大小写。
- **枚举直接量**：Java 持久化查询语言支持使用枚举直接量（采用 Java 枚举直接量语法）。enum 类名必须指定为一个完全限定类名：

  ```
  SELECT e
  FROM Employee e
  WHERE e.status = com.example.EmployeeStatus.FULL_TIME
  ```

10.5.5.2　输入参数

输入参数可以是命名参数或位置参数。

- 命名输入参数由一个冒号（:）后面跟一个字符串来指定；例如 :name。
- 位置输入参数由一个问号（?）后面跟一个整数来指定。例如，第一个输入参数是 ?1，第二个是 ?2，依此类推。

输入参数适用以下规则：

- 只能用在 WHERE 或 HAVING 子句中。
- 位置参数必须编号，从整数 1 开始。
- 命名参数和位置参数不能在一个查询中混合使用。
- 命名参数是区分大小写的。

10.5.5.3　条件表达式

WHERE 子句包括一个条件表达式，相同优先级则从左到右计算。可以通过使用小括号来改变计算的顺序。

10.5.5.4　操作符和优先级

表 10-2 按优先级递减的顺序列出了查询语言操作符。

表 10-2　查询语言操作符优先级顺序

类　型	优先级顺序
导航	.（点号）
算术	+-（一元）
	* /（乘除）
	+-（加减）
比较	=
	>
	>=
	<
	<=
	<>（不等于）
	[NOT] BETWEEN
	[NOT] LIKE
	[NOT] IN
	IS [NOT] NULL
	IS [NOT] EMPTY
	[NOT] MEMBER OF
逻辑	NOT
	AND
	OR

10.5.5.5 BETWEEN 表达式

BETWEEN 表达式确定一个算术表达式是否在一个值范围中。

下面的两个表达式是等价的：

```
p.age BETWEEN 15 AND 19
p.age >= 15 AND p.age <= 19
```

下面两个表达式也是等价的：

```
p.age NOT BETWEEN 15 AND 19
p.age < 15 OR p.age > 19
```

如果一个算术表达式有一个 NULL 值，那么 BETWEEN 表达式的值未知。

10.5.5.6 IN 表达式

IN 表达式确定一个字符串是否属于一个字符串直接量集合，或者一个数是否属于一个数值集合。

路径表达式必须有一个字符串或数值。如果路径表达式有一个 NULL 值，则 IN 表达式的值则未知。

在下面的例子中，如果国家是 UK，则表达式为 TRUE，而如果国家是 Peru，那么这个表达式为 FALSE：

```
o.country IN ('UK', 'US', 'France')
```

还可以使用输入参数：

```
o.country IN ('UK', 'US', 'France', :country)
```

10.5.5.7 LIKE 表达式

LIKE 表达式确定一个通配模式与一个字符串是否匹配。

路径表达式必须有一个字符串或数值。如果这个值为 NULL，则 LIKE 表达式的值未知。模式值是一个可以包含通配字符的字符串直接量。下划线 (_) 通配字符表示任何单个字符。百分号 (%) 通配字符表示 0 个或多个字符。ESCAPE 子句指定模式值中对应通配字符的转义字符。表 10-3 显示了一些示例 LIKE 表达式。

表 10-3 LIKE 表达式示例

表达式	TRUE	FALSE
address.phone LIKE '12%3'	'123' '12993'	'1234'
asentence.word LIKE 'l_se'	'lose'	'loose'
aword.underscored LIKE '_%' ESCAPE '\'	'_foo'	'bar'
address.phone NOT LIKE '12%3'	'1234'	'123' '12993'

10.5.5.8 NULL 比较表达式

NULL 比较表达式测试一个单值路径表达式或一个输入参数是否为 NULL 值。通常，会

使用 NULL 比较表达式来检查是否已设置一个单值关系：

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

这个查询选择所有未设置联盟关系的球队。需要注意，下面的查询与它并不等价：

```
SELECT t
FROM Team t
WHERE t.league = NULL
```

使用等号（=）与 NULL 比较时总会返回一个未知值，即使确实未设置关系，也会返回未知值。第二个查询总是返回一个空结果。

10.5.5.9 空集合比较表达式

IS [NOT] EMPTY 比较表达式检查一个集合值路径表达式是否不包含任何元素。换句话说，它会检查一个集合值关系是否已设置。

如果集合值路径表达式为 NULL，空集合比较表达式就有一个 NULL 值。

下面给出一个例子，会查找所有未包含任何商品的订单：

```
SELECT o
FROM CustomerOrder o
WHERE o.lineItems IS EMPTY
```

10.5.5.10 集合成员表达式

[NOT] MEMBER [OF] 集合成员表达式确定一个值是否是一个集合的成员。值和集合成员必须有相同的类型。

如果集合值或单值路径表达式是未知的，集合成员表达式就是未知的。如果集合值路径表达式指定了一个空集合，那么集合成员表达式为 FALSE。

OF 关键字是可选的。

下面的例子检查一个商品是否属于一个订单：

```
SELECT o
FROM CustomerOrder o
WHERE :lineItem MEMBER OF o.lineItems
```

10.5.5.11 子查询

子查询可以用在查询的 WHERE 或 HAVING 子句中。子查询必须用小括号括起。

下面的例子会查找所有下过 10 个以上订单的顾客：

```
SELECT c
FROM Customer c
WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

子查询可以包含 EXISTS、ALL 和 ANY 表达式。

- **EXISTS 表达式**：[NOT] EXISTS 表达式结合子查询使用，只有当子查询结果包括一个或多个值时才为 true；否则，这个表达式为 false。

下面的例子查找配偶也是员工的所有员工：

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
    SELECT spouseEmp
    FROM Employee spouseEmp
    WHERE spouseEmp = emp.spouse)
```

- **ALL 和 ANY 表达式**：ALL 表达式结合子查询使用，如果子查询返回的所有值都为 true 或者如果子查询为空，这个表达式则为 true。

ANY 表达式结合子查询使用，如果子查询返回的一些值为 true，则 ANY 表达式为 true。如果子查询结果为空，或者如果返回的所有值都是 false，则 ANY 表达式为 false。SOME 关键字与 ANY 是同义词。

ALL 和 ANY 表达式结合 =、<、<=、>、>= 和 <> 比较操作符使用。

下面的例子查找比员工部门经理工资还要高的所有员工：

```
SELECT emp
FROM Employee emp
WHERE emp.salary > ALL (
    SELECT m.salary
    FROM Manager m
    WHERE m.department = emp.department)
```

10.5.5.12 函数表达式

查询语言包括很多字符串、算术和日期 / 时间函数，可以在查询的 SELECT、WHERE 或 HAVING 子句中使用。这些函数在表 10-4、表 10-5 和表 10-6 中列出。

在表 10-4 中，start 和 length 参数类型为 int，分别指定 String 参数中的位置。字符串中第一个位置指定为 1。

表 10-4 字符串表达式

函数语法	返回类型
CONCAT(String, String)	String
LENGTH(String)	int
LOCATE(String, String [, start])	int
SUBSTRING(String, start, length)	String
TRIM([[LEADING|TRAILING|BOTH] char) FROM] (String)	String
LOWER(String)	String
UPPER(String)	String

CONCAT 函数将两个字符串连接为一个字符串。

LENGTH 函数返回字符串的长度（即字符数），将作为一个整数返回。

LOCATE 函数返回给定字符串在一个字符串中的位置。这个函数会返回出现这个字符

串的第一个位置，将作为一个整数返回。第一个参数是要查找的字符串。第二个参数指定了要在哪个字符串中查找。第三个参数可选，这是一个整数，表示起始的字符串位置。默认地，LOCATE 从字符串开头开始查找。字符串的起始位置为 1。如果无法找到字符串，则 LOCATE 返回 0。

SUBSTRING 函数返回给定字符串（由第一个参数指定）的一个子字符串，子字符串根据起始位置和长度来确定。

TRIM 函数从一个字符串开头和 / 或末尾截断指定字符。如果没有指定字符，TRIM 将从字符串中删除空格或空白符。如果指定了 LEADING（可选），则 TRIM 只从字符串中删除前导字符。如果指定了 TRAILING（可选），则 TRIM 只从字符串中删除末尾字符。默认为 BOTH，这会从字符串中删除前导字符以及末尾字符。

LOWER 和 UPPER 函数分别将一个字符串转换为小写或大写。

在表 10-5 中，number 参数可以是 int、float 或 double。

表 10-5　算术表达式

函数语法	返回类型
ABS(number)	int, float 或 double
MOD(int, int)	int
SQRT(double)	double
SIZE(Collection)	int

ABS 函数取一个数值表达式，返回与参数类型相同的一个数。

MOD 函数返回第一个参数除以第二个参数的余数。

SQRT 函数返回一个数的平方根。

SIZE 函数返回给定集合中元素的个数（作为整数返回）。

表 10-6 中，日期 / 时间函数返回数据库服务器上的日期、时间或时间戳。

表 10-6　日期 / 时间表达式

函数语法	返回类型
CURRENT_DATE	java.sql.Date
CURRENT_TIME	java.sql.Time
CURRENT_TIMESTAMP	java.sql.Timestamp

10.5.5.13　case 表达式

case 表达式根据一个条件来改变，这类似于 Java 编程语言中的 case 关键字。CASE 关键字指示一个 case 表达式的开始，case 表达式以 END 关键字结束。WHEN 和 THEN 关键字定义各个条件，ELSE 关键字定义所有其他条件都不满足时的默认条件。

下面的查询根据 Person 实体的子类型选择一个人名和一个条件字符串。如果子类型是 Student，则返回字符串 kid。如果子类型是 Guardian 或 Staff，就返回字符串

adult。如果实体是 Person 的其他某个子类型，则返回字符串 unknown：

```
SELECT p.name
CASE TYPE(p)
    WHEN Student THEN 'kid'
    WHEN Guardian THEN 'adult'
    WHEN Staff THEN 'adult'
    ELSE 'unknown'
END
FROM Person p
```

下面的查询为不同类型的顾客设置一个折扣。金卡顾客可以得到 20% 的折扣，银卡顾客得到 15% 的折扣，铜卡顾客得到 10% 的折扣，所有其他顾客可以得到 5% 的折扣：

```
UPDATE Customer c
SET c.discount =
    CASE c.level
        WHEN 'Gold' THEN 20
        WHEN 'SILVER' THEN 15
        WHEN 'Bronze' THEN 10
        ELSE 5
    END
```

10.5.5.14 NULL 值

如果引用目标不在持久化库中，则目标为 NULL。对于包含 NULL 的条件表达式，查询语言使用 SQL92 定义的语义。简单地说，这包括：

- 如果一个比较操作或算术操作有一个未知值，则得到一个 NULL 值。
- 两个 NULL 值不相等。比较两个 NULL 值会得到一个未知值。
- IS NULL 测试会把 NULL 持久化字段或单值关系字段转换为 TRUE。IS NOT NULL 测试则将它们转换为 FALSE。
- 布尔操作符和条件测试使用表 10-7 和表 10-8 定义的三值逻辑。(在这些表中，T 表示 TRUE，F 表示 FALSE，U 表示未知。)

表 10-7 AND 操作符逻辑

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

表 10-8 OR 操作符逻辑

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

10.5.5.15 相等性语义

在查询语言中，只有相同类型的值可以比较。不过，这个规则有一个例外：绝对和近似数值可以比较。在这样一个比较中，所需完成的类型转换要遵循 Java 数值提升规则。

查询语言将比较的值看作是 Java 类型，而不认为它们表示底层数据库中的类型。例如，一个可能为 NULL 或整数的持久化字段必须指定为一个 Integer 对象，而不能指定为 int 基本类型。这个指定是必要的，因为 Java 对象可以为 NULL，而基本类型不能为 NULL。

只有当两个字符串包含相同的字符序列时，它们才相等。末尾的空格很重要；例如，字符串 'abc' 和 'abc' 就不相等。

对于同一个抽象模式类型的两个实体，只有当它们的主键有相同的值时才认为这两个实体相等。表 10-9 显示了非操作符的逻辑，表 10-10 给出了条件测试的真值。

表 10-9　NOT 操作符逻辑

NOT 值	值
T	F
F	T
U	U

表 10-10　条件测试

条件测试	T	F	U
表达式 IS TRUE	T	F	F
表达式 IS FALSE	F	T	F
表达式未知	F	F	T

10.5.6　SELECT 子句

SELECT 子句定义了查询返回的对象或值的类型。

10.5.6.1　返回类型

SELECT 子句的返回类型由子句中包含的选择表达式的结果类型来定义。如果使用了多个表达式，查询的结果将是一个 Object[]，数组中的元素对应 SELECT 子句中表达式的顺序，类型则对应各个表达式的结果类型。

SELECT 子句不能指定集合值表达式。例如，SELECT 子句 p.teams 就是不合法的，因为 teams 是一个集合。不过，以下查询中的子句是合法的，因为 t 是 teams 集合中的一个元素：

```
SELECT t
FROM Player p, IN (p.teams) t
```

下面的查询中，SELECT 子句有多个表达式：

```
SELECT c.name, c.country.name
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

这个查询返回一个 Object[] 元素列表：第一个数组元素是一个指示顾客名字的字符串，第二个数组元素是一个指示顾客所在国家名的字符串。

查询的结果可能是一个聚集函数的结果，如表 10-11 所示。

表 10-11 Select 语句中的聚集函数

函数名	返回类型	描 述
AVG	Double	返回字段的平均值
COUNT	Long	返回结果总数
MAX	字段类型	返回结果集中的最大值
MIN	字段类型	返回结果集中的最小值
SUM	Long（对于整型字段） Double（对于浮点数字段） BigInteger（对于 BigInteger 字段） BigDecimal（对于 BigDecimal 字段）	返回结果集中所有值的总和

如果选择查询的 SELECT 子句中包含聚集函数（AVG，COUNT，MAX，MIN 或 SUM），则适用以下规则：

- 如果没有可以应用这个函数的值，则 AVG、MAX、MIN 和 SUM 函数返回 null。
- 如果没有可以应用这个函数的值，则 COUNT 函数返回 0。

下面的例子会返回平均订单数量：

```
SELECT AVG(o.quantity)
FROM CustomerOrder o
```

下面的例子返回 Roxane Coss 订购的商品的总价格：

```
SELECT SUM(l.price)
FROM CustomerOrder o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

下面的例子返回订单总数：

```
SELECT COUNT(o)
FROM CustomerOrder o
```

下面的例子返回 Hal Incandenza 的订单中有价格的商品总数：

```
SELECT COUNT(l.price)
FROM CustomerOrder o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Incandenza' AND c.firstname = 'Hal'
```

10.5.6.2 DISTINCT 关键字

DISTINCT 关键字可以删除重复的返回值。如果查询返回一个 java.util.Collection，这个集合中允许有重复的值，则必须指定 DISTINCT 关键字来删除重复值。

10.5.6.3 构造器表达式

利用构造器表达式，可以返回存储查询结果元素的 Java 实例，而不是一个 Object[]。

下面的查询为与 WHERE 子句匹配的每个 Customer 创建一个 CustomerDetail 实例。CustomerDetail 存储顾客名和顾客所在的国家名。所以这个查询会返回 CustomerDetail 实例的一个 List：

```
SELECT NEW com.example.CustomerDetail(c.name, c.country.name)
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

10.5.7 ORDER BY 子句

顾名思义，ORDER BY 子句就是对查询返回的值或对象排序。

如果 ORDER BY 子句包含多个元素，则元素的从左到右序列确定了从高到低的优先级。

ASC 关键字指定递增顺序，这是默认值，DESC 关键字指示递减顺序。

使用 ORDER BY 子句时，SELECT 子句必须返回对象或值的一个可排序的集合。对于不是由 SELECT 子句返回的值或对象不能排序。例如，下面的查询是合法的，因为 ORDER BY 子句使用了 SELECT 子句返回的对象：

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity, o.totalcost
```

下面的例子是不合法的，因为 ORDER BY 子句使用了并非由 SELECT 子句返回的值：

```
SELECT p.product_name
FROM CustomerOrder o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname = 'Faehmel' AND c.firstname = 'Robert'
ORDER BY o.quantity
```

10.5.8 GROUP BY 和 HAVING 子句

GROUP BY 子句允许根据一组属性对值分组。

下面的查询将按顾客的国家对顾客分组，并返回每个国家的顾客人数：

```
SELECT c.country, COUNT(c)
FROM Customer c GROUP BY c.country
```

HAVING 子句结合 GROUP BY 子句使用，进一步限制查询的返回结果。

下面的查询将按顾客状态对订单分组，并返回顾客状态以及相同状态顾客所有订单的平均总价格 totalPrice。另外，它只考虑状态为 1、2 或 3 的顾客，所以其他顾客的订单不在考虑范围内：

```
SELECT c.status, AVG(o.totalPrice)
FROM CustomerOrder o JOIN o.customer c
GROUP BY c.status HAVING c.status IN (1, 2, 3)
```

Chapter 11 第 11 章

使用 Criteria API 创建查询

Criteria API 通过创建查询定义对象（query-defining object）为实体及其持久化状态定义查询。Criteria 查询使用 Java API 编写，这些查询是类型安全的，而且是可移植的。不论底层数据库是什么，这些查询都可以执行。

本章将介绍以下内容：

- Criteria 和 Metamodel API 概述
- 使用 Metamodel API 为实体类建模
- 使用 Criteria API 和 Metamodel API 创建基本的类型安全查询

11.1 Criteria 和 Metamodel API 概述

与 JPQL 类似，Criteria API 建立在持久化实体、实体关系以及内嵌对象的抽象模式基础之上。Criteria API 作用于这个抽象模式，使开发人员可以通过调用 Java 持久化 API 实体操作来查找、修改和删除持久化实体。Metamodel API 与 Criteria API 合作，为 Criteria 查询实现持久化实体类建模。

Criteria API 和 JPQL 紧密相关，而且设计为支持查询中的类似操作。熟悉 JPQL 语法的开发人员会发现 Criteria API 中往往有等价的对象级操作。

下面是一个简单的 Criteria 查询，它会返回数据源中 Pet 实体的所有实例：

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

```
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> allPets = q.getResultList();
```

等价的 JPQL 查询为：

```
SELECT p
FROM Pet p
```

这个查询展示了创建一个 Criteria 查询的基本步骤。

1. 使用一个 EntityManager 实例来创建一个 CriteriaBuilder 对象。

2. 通过创建 CriteriaQuery 接口的一个实例来创建查询对象。这个查询对象的属性将利用查询的详细信息修改。

3. 通过在 CriteriaQuery 对象上调用 from 方法设置查询根。

4. 通过调用 CriteriaQuery 对象的 select 方法指定查询结果的类型。

5. 创建一个 TypedQuery<T> 实例并指定查询结果的类型，准备执行查询。

6. 通过在 TypedQuery<T> 对象上调用 getResultList 方法执行查询。由于这个查询返回一个实体集合，所以结果存储在一个 List 中。

每一步相关的任务将在本章中详细讨论。

要创建一个 CriteriaBuilder 实例，需要在 EntityManager 实例上调用 getCriteriaBuilder 方法：

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

使用这个 CriteriaBuilder 实例来创建一个查询对象：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

查询将返回 Pet 实体的实例。要创建一个类型安全的查询，创建 CriteriaQuery 对象时就要指定查询的类型。

调用查询对象的 from 方法来设置查询的 FROM 子句，并指定查询的根：

```
Root<Pet> pet = cq.from(Pet.class);
```

调用查询对象的 select 方法并传入查询根，以设置查询的 SELECT 子句：

```
cq.select(pet);
```

现在使用查询对象创建一个可以在数据源上执行的 TypedQuery<T> 对象。保存对查询对象的修改，创建一个准备执行的查询：

```
TypedQuery<Pet> q = em.createQuery(cq);
```

通过调用查询对象的 getResultList 方法来执行这个有类型的查询对象，因为这个查询将返回多个实体实例。下面的语句将结果存储在一个 List<Pet> 集合值对象中：

```
List<Pet> allPets = q.getResultList();
```

11.2 使用 Metamodel API 为实体类建模

使用 Metamodel API 为特定的持久化单元中的托管实体创建元模型。对于一个特定包中的各个实体类，创建元模型类时，要在末尾增加一个下划线，另外包含一些属性，分别对应于实体类的持久化字段或属性。

以下实体类 com.example.Pet 有 4 个持久化字段：id、name、color 和 owners：

```
package com.example;
...
@Entity
public class Pet {
    @Id
    protected Long id;
    protected String name;
    protected String color;
    @ManyToOne
    protected Set<Person> owners;
    ...
}
```

相应的元模型（Metamodel）类如下所示：

```
package com.example;
...
@Static Metamodel(Pet.class)
public class Pet_ {
    public static volatile SingularAttribute<Pet, Long> id;
    public static volatile SingularAttribute<Pet, String> name;
    public static volatile SingularAttribute<Pet, String> color;
    public static volatile SetAttribute<Pet, Person> owners;
}
```

Criteria 查询使用元模型类及其属性来指示托管实体类及其持久化状态和关系。

11.2.1 使用元模型类

对应于实体类的元模型类有以下类型：

```
javax.persistence.metamodel.EntityType<T>
```

注解处理器通常在开发时或运行时生成元模型类。使用 Criteria 查询的应用开发人员可以采用以下方法：

- 通过使用持久化提供者的注解处理器来生成静态元模型类
- 通过以下方式得到元模型类：
  - 在查询根对象上调用 getModel 方法
  - 得到 Metamodel 接口的一个实例，再将实体类型传递到这个实例的 entity 方法

下面的代码段展示了如何通过调用 Root<T>.getModel 得到 Pet 实体的元模型类：

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
EntityType<Pet> Pet_ = pet.getModel();
```

下面的代码段展示了首先使用 EntityManager.getMetamodel 得到一个元模型实例，再在这个元模型实例上调用 entity 来得到 Pet 实体的元模型类：

```
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
```

说明　最常见的用法是在开发时生成类型安全的静态元模型类。通过调用 Root<T>.getModel 或 EntityManager.getMetamodel 然后调用 entity 方法来动态地得到元模型类并不支持类型安全，也不允许应用在元模型类上调用持久化字段或属性名。

11.3　使用 Criteria API 和 Metamodel API 创建基本的类型安全查询

Criteria 查询的基本语义包括一个 SELECT 子句、一个 FROM 子句和一个可选的 WHERE 子句，这与 JPQL 查询类似。Criteria 查询使用 Java 对象设置这些子句，所以可以采用一种类型安全的方式创建查询。

11.3.1　创建 Criteria 查询

javax.persistence.criteria.CriteriaBuilder 接口用于构造：

- Criteria 查询
- 选择
- 表达式
- 谓词
- 排序

要得到 CriteriaBuilder 接口的一个实例，可以在 EntityManager 或 EntityManagerFactory 实例上调用 getCriteriaBuilder 方法。

下面的代码显示了如何使用 EntityManager.getCriteriaBuilder 方法得到 CriteriaBuilder 实例：

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
```

通过得到以下接口的一个实例来构造 Criteria 查询：

```
javax.persistence.criteria.CriteriaQuery
```

CriteriaQuery 对象定义一个特定的查询，可以在一个或多个实体上导航。通过调用一个 CriteriaBuilder.createQuery 方法来得到 CriteriaQuery 实例。要创建类型安全的查询，可以如下调用 CriteriaBuilder.createQuery 方法：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

CriteriaQuery 对象的类型应当设置为查询的期望结果类型。在前面的代码中，对于查找 Pet 实体实例的查询，对象的类型设置为 CriteriaQuery<Pet>。

下面的代码段为一个返回 String 的查询创建一个 CriteriaQuery 对象：

```
CriteriaQuery<String> cq = cb.createQuery(String.class);
```

11.3.2 查询根

对于一个特定的 CriteriaQuery 对象，查询的根实体（所有导航都从这里开始）称为**查询根**（query root）。这类似于 JPQL 查询中的 FROM 子句。

通过在 CriteriaQuery 实例上调用 from 方法来创建查询根。from 方法的参数可以是实体类，或者是实体的一个 EntityType<T> 实例。

以下代码将查询根设置为 Pet 实体：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

下面的代码使用一个 EntityType<T> 实例将查询根设置为 Pet 类：

```
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet_);
```

Criteria 查询可以有多个查询根。查询从多个实体导航时通常会发生这种情况。

下面的代码有两个 Root 实例：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet1 = cq.from(Pet.class);
Root<Pet> pet2 = cq.from(Pet.class);
```

11.3.3 使用联接查询关系

对于导航到相关实体类的查询，查询必须定义与相关实体的一个联接，为此可以在查询根对象或另外一个联接对象上调用一个 From.join 方法。join 方法与 JPQL 中的 JOIN 关键字类似。

联接的目标使用 EntityType<T> 类型的元模型类指定联接实体的持久化字段或属性。

join 方法返回一个 Join<X, Y> 类型的对象，其中 X 是源实体，Y 是联接的目标实体。在下面的代码段中，Pet 是源实体，Owner 是目标实体，Pet_ 是一个静态生成的元模型类：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
```

可以把多个联接串在一起导航到目标实体的相关实体，而不必为每一个联接分别创建一个 Join<X, Y> 实例：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join(Pet_.owners).join(Owner_.addresses);
```

11.3.4　Criteria 查询中的路径导航

Criteria 查询的 SELECT 和 WHERE 子句中使用的 Path 对象可以是查询根实体、联接实体或其他 Path 对象。使用 Path.get 方法可以导航到查询实体的属性。

get 方法的参数是实体元模型类的相应属性。这个属性可以是一个单值属性，由元模型类中的 @SingularAttribute 指定，或者是一个集合值属性，由 @CollectionAttribute、@SetAttribute、@ListAttribute 或 @MapAttribute 之一指定。

下面的查询返回数据库中所有宠物的名字。get 方法在查询根 pet 上调用，Pet 实体的元模型类 Pet_ 的 name 属性作为参数：

```
CriteriaQuery<String> cq = cb.createQuery(String.class);

Root<Pet> pet = cq.from(Pet.class);
cq.select(pet.get(Pet_.name));
```

11.3.5　限制 Criteria 查询结果

通过调用 CriteriaQuery.where 方法所设置的条件可以限制在 CriteriaQuery 对象上执行查询的结果。调用 where 方法类似于在 JPQL 查询中设置 WHERE 子句。

where 方法会计算 Expression 接口的实例，根据表达式的条件限制结果。要创建 Expression 实例，可以使用 Expression 和 CriteriaBuilder 接口中定义的方法。

11.3.5.1　Expression 接口方法

Expression 对象在查询的 SELECT、WHERE 或 HAVING 子句中使用。表 11-1 显示了 Expression 对象可用的条件方法。

表 11-1　Expression 接口中的条件方法

方　法	描　述
isNull	检查一个表达式是否为 null
isNotNull	检查一个表达式是否非 null
in	检查一个表达式是否在一个值列表中

下面的查询使用 Expression.isNull 方法查找 color 属性为 null 的所有宠物：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).isNull());
```

下面的查询使用 Expression.in 方法查找所有棕色和黑色的宠物：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).in("brown", "black"));
```

in 方法还可以检查一个属性是否是一个集合的成员。

11.3.5.2 CriteriaBuilder 接口中的表达式方法

CriteriaBuilder 接口还定义了另外一些创建表达式的方法。这些方法对应于 JPQL 的算术、字符串、日期、时间以及 case 操作符和函数。表 11-2 显示了 CriteriaBuilder 对象可用的条件方法。

表 11-2 CriteriaBuilder 接口中的条件方法

条件方法	描 述
equal	检查两个表达式是否相等
notEqual	检查两个表达式是否不相等
gt	检查第一个数值表达式是否大于第二个数值表达式
ge	检查第一个数值表达式是否大于或等于第二个数值表达式
lt	检查第一个数值表达式是否小于第二个数值表达式
le	检查第一个数值表达式是否小于或等于第二个数值表达式
between	检查第一个表达式的值是否介于第二个和第三个表达式之间
like	检查表达式是否与一个给定模式匹配

下面的代码使用了 CriteriaBuilder.equal 方法：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
```

下面的代码使用了 CriteriaBuilder.gt 方法：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date someDate = new Date(...);
cq.where(cb.gt(pet.get(Pet_.birthday), date));
```

以下代码使用了 CriteriaBuilder.between 方法：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date firstDate = new Date(...);
Date secondDate = new Date(...);
cq.where(cb.between(pet.get(Pet_.birthday), firstDate, secondDate));
```

以下代码使用了 CriteriaBuilder.like 方法：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.like(pet.get(Pet_.name), "*do"));
```

要指定多个条件谓词，可以使用 CriteriaBuilder 接口的复合谓词方法，如表 11-3 所示。

表 11-3　CriteriaBuilder 接口中的复合谓词方法

方　法	描　述
and	两个布尔表达式的逻辑合取
or	两个布尔表达式的逻辑析取
not	一个给定布尔表达式的逻辑非

以下代码显示了查询中如何使用复合谓词：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido")
        .and(cb.equal(pet.get(Pet_.color), "brown")));
```

11.3.6　管理 Criteria 查询结果

对于返回多个结果的查询，适当组织这些结果往往很有帮助。CriteriaQuery 接口定义了以下排序和分组方法：

- orderBy 方法根据实体的属性对查询结果排序
- groupBy 方法根据实体的属性对查询结果分组，having 方法根据一个条件来限制这些组

11.3.6.1　结果排序

要对一个查询的结果排序，可以调用 CriteriaQuery.orderBy 方法，并传入一个 Order 对象。要创建 Order 对象，可以调用 CriteriaBuilder.asc 或 CriteriaBuilder.desc 方法。asc 方法按传入的表达式参数的递增值对结果排序。desc 方法按传入的表达式参数的递减值对结果排序。下面的查询显示了如何使用 desc 方法：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get(Pet_.birthday)));
```

在这个查询中，结果将按宠物生日从高到低排序。也就是说，12 月出生的宠物会出现在 5 月出生的宠物前面。

下面的查询显示了 asc 方法的用法：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join(Pet_.owners).join(Owner_.address);
cq.select(pet);
cq.orderBy(cb.asc(address.get(Address_.postalCode)));
```

在这个查询中，结果将按宠物主人邮政编码从低到高排序。也就是说，主人邮政编码为10001的宠物会出现在主人邮政编码为91000的宠物前面。

如果有多个Order对象传入orderBy，这些对象的优先级则由它们在orderBy参数表中出现的顺序来确定。第一个Order对象最优先。

下面的代码根据多个规则对结果排序：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
cq.select(pet);
cq.orderBy(cb.asc(owner.get(Owner_.lastName)), owner.get(Owner_.firstName)));
```

这个查询的结果将根据宠物主人的名字按字母顺序排序，先按姓排序，再按名排序。

11.3.6.2 结果分组

CriteriaQuery.groupBy方法将查询结果划分为组。要设置这些组，需要向groupBy传入一个表达式：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
```

这个查询返回所有Pet实体，并按宠物的颜色对结果分组。

结合groupBy使用CriteriaQuery.having方法可以对组进行过滤。having方法取一个条件表达式作为参数，可以根据这个条件表达式限制查询结果：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
cq.having(cb.in(pet.get(Pet_.color)).value("brown").value("blonde"));
```

在这个例子中，查询按颜色对返回的Pet实体分组，这与前面的例子一样。不过，只会返回color属性设置为brown或blonde的实体组。也就是说，这个查询中不会返回灰色的宠物。

11.3.7 执行查询

为准备执行一个查询，需要用查询结果的类型创建一个TypedQuery<T>对象，将CriteriaQuery对象传递到EntityManager.createQuery。

要执行查询，可以在TypedQuery<T>对象上调用getSingleResult或getResultList。

11.3.7.1　单值查询结果

使用 TypedQuery<T>.getSingleResult 方法来执行返回单个结果的查询：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
Pet result = q.getSingleResult();
```

11.3.7.2　集合值查询结果

使用 TypedQuery<T>.getResultList 方法来执行返回一个对象集合的查询：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```

Chapter 12 第 12 章

创建和使用基于字符串的 Criteria 查询

本章介绍如何创建基于字符串的弱类型 Criteria API 查询，包括以下内容：

- 基于字符串的 Criteria API 查询
- 创建基于字符串的查询
- 执行基于字符串的查询

12.1 基于字符串的 Criteria API 查询概述

基于字符串的 Criteria API 查询（“基于字符串的查询”）是指使用字符串的 Java 查询，遍历一个数据层次体系时，这些查询使用字符串而不是强类型的元模型对象来指定实体属性。基于字符串的查询的构建与元模型查询类似，可以静态构建或动态构建，另外可以像强类型元模型查询一样提供同样的查询和操作。

强类型元模型查询是构造 Criteria API 查询的首选方法。

基于字符串的查询相对于元模型查询的主要优点是：能够在开发时构造 Criteria 查询，而不需要生成静态元模型类或者动态访问生成的元模型类。

基于字符串的查询的主要缺点是缺乏类型安全性；这个问题可能导致由于类型不匹配而出现运行时错误，如果使用强类型的元模型查询，类型不匹配的问题在开发时就能捕获到。

关于构造 criteria 查询的更多信息，参见第 11 章。

12.2 创建基于字符串的查询

要创建一个基于字符串的查询，可以作为字符串直接指定实体类的属性名，而不是指定

元模型类的属性。例如，下面这个查询要查找 name 属性值为 Fido 的所有 Pet 实体：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get("name"), "Fido"));
```

属性名指定为一个字符串。这个查询等价于以下元模型查询：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
```

说明　如果基于字符串的查询中存在类型不匹配错误，在运行时执行代码时才会被发现，这与前面的元模型查询不同，在元模型查询中，类型不匹配在编译时就能被发现。

可以采用同样的方式指定联接：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join("owners").join("addresses");
```

元模型查询中使用的所有条件表达式、方法表达式、路径导航方法和结果限制方法在基于字符串的查询中也可以使用。在这些情况下，属性使用字符串指定。例如，下面是一个基于字符串的查询，这里使用了 in 表达式：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get("color").in("brown", "black"));
```

下面是一个基于字符串的查询，会按日期以降序对结果排序：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get("birthday")));
```

12.3　执行基于字符串的查询

基于字符串的查询的执行与强类型 Criteria 查询很类似。首先，将 criteria 查询对象传入 EntityManager.createQuery 方法，以创建一个 javax.persistence.TypedQuery 对象，然后在这个查询对象上调用 getSingleResult 或 getResultList 来执行这个查询：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get("name"), "Fido"));
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```

Chapter 13 第 13 章

使用锁定控制对实体数据的并发访问

本章详细介绍如何处理对实体数据的并发访问以及 Java 持久化 API 应用开发人员可以使用的锁定策略，包括以下内容：

- 实体锁定和并发概述
- 锁模式

13.1 实体锁定和并发概述

如果数据源中的数据同时由多个应用访问，就会**并发访问**实体数据。要确保并发访问时底层数据的完整性。

在一个事务中，更新数据库表中的数据时，持久化提供者假设数据库管理系统会持有短期读锁和长期写锁来维护数据的完整性。大多数持久化提供者会延迟到事务结束才完成数据库写操作，除非应用显式地调用了一个刷新输出（也就是说，应用调用了 `EntityManager.flush` 方法，或执行查询时将刷新输出模式设置为 `AUTO`）。

默认地，持久化提供者会使用**乐观锁定**（optimistic locking），这是指在提交数据修改之前，持久化提供者会检查自读这个数据以来，没有其他事务修改或删除数据。可以通过数据库表中的一个版本列来完成这个检查，而且在实体类中有一个相应的版本属性。某一行有修改时，版本值会增加。原事务要检查这个版本属性，如果数据已经被另一个事务修改，就会抛出一个 `javax.persistence.OptimisticLockException`，原事务将回滚。应用指定乐观锁模式时，持久化提供者会验证从数据库读取实体以来这个特定实体未改变过（即使实体数据未修改）。

悲观锁定（Pessimistic locking）比乐观锁定做的工作更多。采用悲观锁定时，持久化提供者要创建一个事务，得到数据的一个长期锁，直到事务完成，这会防止其他事务修改或删除数据，直到锁使用结束。当底层数据由多个事务频繁访问和修改时，相对于乐观锁定，悲观锁定是一种更优的策略。

说明　对于不经常修改的实体，使用悲观锁会导致应用性能下降。

13.1.1　使用乐观锁定

使用 javax.persistence.Version 注解将一个持久化字段或属性标志为实体的版本属性。这个版本属性支持实体完成乐观并发控制。在一个事务中修改一个实体实例时，持久化提供者会读取和更新版本属性。应用可以读取版本属性，但是不能修改这个值。

说明　尽管一些持久化提供者也可以对一些没有版本属性的实体提供乐观锁定支持，不过使用乐观锁定时，可移植的应用都应当使用有版本属性的实体。如果应用试图锁定一个没有版本属性的实体，而持久化提供者不支持无版本属性实体的乐观锁定，就会抛出一个 PersistenceException。

@Version 注解有以下需求：

- 每个实体只能定义一个 @Version 属性。
- 对于映射到多个表的实体，@Version 属性必须在主表中。
- @Version 属性必须是以下类型：int、Integer、long、Long、short、Short 或 java.sql.Timestamp。

下面的代码段显示了如何在一个有持久化字段的实体中定义一个版本属性：

```
@Version
protected int version;
```

下面的代码段显示了如何在一个有持久化属性的实体中定义一个版本属性：

```
@Version
protected Short getVersion() { ... }
```

13.2　锁模式

通过指定使用锁模式，应用可以提升一个实体的锁定级别。可以指定锁模式来提高乐观锁定的级别或者请求使用悲观锁。

使用乐观锁模式时，会让持久化提供者检查一个事务期间读取（但不修改）的实体以及

更新的实体的版本属性。

使用悲观锁模式则要求持久化提供者为数据库数据立即请求长期读或写锁（请求读锁还是写锁对应于实体状态）。

可以通过指定 javax.persistence.LockModeType 枚举类型中定义的一个锁模式来设置实体操作的锁模式，锁模式见表 13-1。

表 13-1 并发实体访问的锁模式

锁模式	描 述
OPTIMISTIC	对于有版本属性的所有实体，得到一个乐观读锁
OPTIMISTIC_FORCE_INCREMENT	对于有版本属性的所有实体，得到一个乐观读锁，并递增版本属性值
PESSIMISTIC_READ	立即得到数据的长期读锁，避免数据被修改或删除。维护这个锁时，其他事务可以读这个数据，但是不能修改或删除数据。 请求一个读锁时，允许持久化提供者得到一个数据库写锁，但反之不行
PESSIMISTIC_WRITE	立即得到数据的长期写锁，避免数据被读取、修改或删除
PESSIMISTIC_FORCE_INCREMENT	立即得到数据的长期锁，避免数据被修改或删除，并递增有版本实体的版本属性值
READ	OPTIMISTIC 的同义词，对于新应用，更倾向于使用 LockModeType.OPTIMISTIC
WRITE	OPTIMISTIC_FORCE_INCREMENT 的同义词，对于新应用，更倾向于使用 LockModeType.OPTIMISTIC_FORCE_INCREMENT
NONE	对数据库中的数据没有锁定

13.2.1 设置锁模式

要指定锁模式，可以使用以下技术之一：

- 调用 EntityManager.lock 方法并传入一个锁模式：

```
EntityManager em = ...;
Person person = ...;
em.lock(person, LockModeType.OPTIMISTIC);
```

- 调用一个 EntityManager.find 方法，以锁模式作为参数：

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK,
    LockModeType.PESSIMISTIC_WRITE);
```

- 调用一个 EntityManager.refresh 方法，以锁模式作为参数：

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK);
...
em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

- 调用 Query.setLockMode 或 TypedQuery.setLockMode 方法，并传入锁模式作为参数：

```
Query q = em.createQuery(...);
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

- 向 @NamedQuery 注解增加一个 lockMode 元素：

```
@NamedQuery(name="lockPersonQuery",
  query="SELECT p FROM Person p WHERE p.name LIKE :name",
  lockMode=PESSIMISTIC_READ)
```

13.2.2　使用悲观锁定

有版本的实体以及没有版本属性的实体可以采用悲观方式锁定。

要采用悲观方式锁定实体，需要把锁模式设置为 PESSIMISTIC_READ、PESSIMISTIC_WRITE 或 PESSIMISTIC_FORCE_INCREMENT。

如果在数据库行上无法得到一个悲观锁，而且无法锁定数据会导致一个事务回滚，就会抛出一个 PessimisticLockException。如果无法得到一个悲观锁，但是锁定失败不会导致事务回滚，则抛出 LockTimeoutException。

用 PESSIMISTIC_FORCE_INCREMENT 以悲观方式锁定一个有版本的实体时，会使版本属性值递增，尽管实体数据并未修改。以悲观方式锁定一个有版本的实体时，持久化提供者会完成乐观锁定时进行的版本检查，如果版本检查失败，将抛出一个 OptimisticLockException。试图用 PESSIMISTIC_FORCE_INCREMENT 锁定一个无版本属性的实体是不可移植的，而且如果持久化提供者不支持无版本实体的乐观锁，还可能导致一个 PersistenceException。用 PESSIMISTIC_WRITE 锁定一个有版本实体时，如果事务成功提交，则会使版本属性值递增。

13.2.2.1　悲观锁定超时

通过使用 javax.persistence.lock.timeout 属性，可以指定持久化提供者得到数据库表的一个锁时要等待多长时间（毫秒数）。如果得到锁所花费的时间超过了这个属性的值，则会抛出一个 LockTimeoutException，不过不会将当前事务标志为回滚。如果将这个属性设置为 0，无法立即得到一个锁时，持久化提供者就会抛出一个 LockTimeoutException。

说明　可移植应用不能依赖于 javax.persistence.lock.timeout 的设置，因为锁定策略和底层数据库可能意味着这个超时值无法使用。javax.persistence.lock.timeout 的值只是一个提示，而不是一个约定。

可以通过编程设置这个属性，传入允许指定锁模式的 EntityManager 方法、Query.

setLockMode 和 TypedQuery.setLockMode 方法、@NamedQuery 注解以及 Persistence.createEntityManagerFactory 方法。还可以在 persistence.xml 部署描述文件中设置为一个属性。

如果在多处设置了 javax.persistence.lock.timeout，则会按以下顺序确定它的值：

1. 某个 EntityManager 或 Query 方法的参数。
2. @NamedQuery 注解中的设置。
3. Persistence.createEntityManagerFactory 方法的参数。
4. persistence.xml 部署描述文件中的值。

第 14 章 Chapter 14

用实体图创建获取计划

本章解释如何使用实体图为 Java 持久化 API 操作和查询创建获取计划。

实体图（Entity graph）是一个特定持久化查询或操作的模板。创建**获取计划**（fetch plan）时会用到实体图，在检索的同时分组持久化字段时也会用到实体图。应用开发人员使用获取计划将相关的持久化字段分组在一起，来提高运行时性能。

默认地，实体字段或属性采用懒惰方式获取。开发人员指定字段或属性作为一个获取计划的一部分，持久化提供者再采用积极方式获取。

例如，一个 email 应用将消息存储为 `EmailMessage` 实体，这个应用会优先获取某些字段。在邮箱视图中以及显示消息时，最常查看的是发送者、主题和日期。`EmailMessage` 实体包含相关 `EmailAttachment` 实体的一个集合。出于性能方面的原因，除非真正需要，否则不会获取附件，不过附件的文件名很重要。开发这个应用的开发人员可以建立一个获取计划，从 `EmailMessage` 和 `EmailAttachment` 以积极方式获取这些重要字段，而采用懒惰方式获取较低优先级的数据。

本章将介绍以下内容：

- 实体图基础
- 使用命名实体图
- 在查询操作中使用实体图

14.1 实体图基础

可以使用注解或部署描述文件静态地创建实体图，或者也可以使用标准接口动态创建。

可以结合 EntityManager.find 方法使用实体图，或者指定实体图作为操作或查询的一个提示，用作 JPQL 或 Criteria API 查询的一部分。

实体图包含的属性对应于 find 或查询操作中将要积极获取的那些字段。由于总会获取实体类的主键和版本字段，所以不需要将主键和版本字段显式地添加到实体图中。

14.1.1 默认实体图

默认情况下，实体中的所有字段都会采用懒惰方式获取，除非实体元数据的 fetch 属性设置为 javax.persistence.FetchType.EAGER。实体中设置为要积极获取的字段就组成了默认实体图。

例如，以下 EmailMessage 实体指定了一些会积极获取的字段：

```
@Entity
public class EmailMessage implements Serializable {
    @Id
    String messageId;
    @Basic(fetch=EAGER)
    String subject;
    String body;
    @Basic(fetch=EAGER)
    String sender;
    @OneToMany(mappedBy="message", fetch=LAZY)
    Set<EmailAttachment> attachments;
    ...
}
```

这个实体的默认实体图会包含 messageId、subject 和 sender 字段，但不包含 body 或 attachments 字段。

14.1.2 在持久化操作中使用实体图

使用实体图需创建 javax.persistence.EntityGraph 接口的一个实例，创建这个实例时，对于命名实体图，可以调用 EntityManager.getEntityGraph，或者可以调用 EntityManager.createEntityGraph 创建动态实体图。

命名实体图是由实体类上应用的 @NamedEntityGraph 注解指定的一个实体图，或者由应用部署描述文件中的 named-entity-graph 元素指定。部署描述文件中定义的命名实体图会覆盖所有基于注解的同名实体图。

所创建的实体图可以是一个**获取图**（fetch graph）或一个**加载图**（load graph）。

14.1.2.1 获取图

要指定一个获取图，执行 EntityManager.find 或查询操作时要设置 javax.persistence.fetchgraph。获取图只包括 EntityGraph 实例中显式指定的字段，而忽略默认实体图设置。

在下面的例子中，会忽略默认实体图，只有 body 字段包含在动态创建的获取图中：

```
EntityGraph<EmailMessage> eg = em.createEntityGraph(EmailMessage.class);
eg.addAttributeNodes("body");
...
Properties props = new Properties();
props.put("javax.persistence.fetchgraph", eg);
EmailMessage message = em.find(EmailMessage.class, id, props);
```

14.1.2.2　加载图

要指定一个加载图，需要在执行 EntityManager.find 或查询操作时设置 javax.persistence.loadgraph 属性。加载图包括 EntityGraph 实例中显式指定的字段以及默认实体图中的字段。

在下面的例子中，动态创建的加载图包含默认实体图中的所有字段以及 body 字段：

```
EntityGraph<EmailMessage> eg = em.createEntityGraph(EmailMessage.class);
eg.addAttributeNodes("body");
...
Properties props = new Properties();
props.put("javax.persistence.loadgraph", eg);
EmailMessage message = em.find(EmailMessage.class, id, props);
```

14.2　使用命名实体图

可以使用实体类上应用的注解或应用部署描述文件中的 named-entity-graph 元素及其子元素来创建命名实体图。持久化提供者会扫描应用中的所有命名实体图（包括注解和 XML 中定义的命名实体图）。使用注解设置的命名实体图可能被使用 named-entity-graph 设置的命名实体图覆盖。

14.2.1　对实体类应用命名实体图注解

javax.persistence.NamedEntityGraph 注解定义了一个命名实体图，将在类级应用。可以为一个类定义多个 @NamedEntityGraph 注解，为此要把它们添加在一个 javax.persistence.NamedEntityGraphs 类级注解中。

@NamedEntityGraph 注解必须在实体图的根上应用。也就是说，如果 EntityManager.find 或查询操作以 EmailMessage 类作为其根实体，这个操作中使用的命名实体图就必须在 EmailMessage 类中定义：

```
@NamedEntityGraph
@Entity
public class EmailMessage {
    @Id
    String messageId;
```

```
    String subject;
    String body;
    String sender;
}
```

在这个例子中，EmailMessage 类有一个 @NamedEntityGraph 注解，定义了一个命名实体图，默认为类名，即 EmailMessage。@NamedEntityGraph 注解中未包括任何字段作为属性节点，而且字段没有增加元数据注解来设置获取类型，所以在加载图或获取图中将以积极方式获取的唯一字段就是 messageId。

命名实体图的属性是应当包含在实体图中的实体字段。要向实体图增加字段，可以在 @NamedEntityGraph 的 attributeNodes 元素中用 javax.persistence.NamedAttributeNode 注解来指定：

```
@NamedEntityGraph(name="emailEntityGraph", attributeNodes={
    @NamedAttributeNode("subject"),
    @NamedAttributeNode("sender")
})
@Entity
public class EmailMessage { ... }
```

在这个例子中，命名实体图的名字为 emailEntityGraph，包括 subject 和 sender 字段。

通过将多个 @NamedEntityGraph 分组到一个 @NamedEntityGraphs 注解中，可以向一个类应用多个 @NamedEntityGraph 定义。

在下面的例子中，EmailMessage 类上定义了两个实体图。一个用于预览窗口，只获取消息的发送者、主题和消息体。另一个用于消息的完整视图，还包括消息附件：

```
@NamedEntityGraphs({
    @NamedEntityGraph(name="previewEmailEntityGraph", attributeNodes={
        @NamedAttributeNode("subject"),
        @NamedAttributeNode("sender"),
        @NamedAttributeNode("body")
    }),
    @NamedEntityGraph(name="fullEmailEntityGraph", attributeNodes={
        @NamedAttributeNode("sender"),
        @NamedAttributeNode("subject"),
        @NamedAttributeNode("body"),
        @NamedAttributeNode("attachments")
    })
})
@Entity
public class EmailMessage { ... }
```

14.2.2 从命名实体图获得 EntityGraph 实例

使用 EntityManager.getEntityGraph 方法，并传入命名实体图名，可以为一个

命名实体图获得 EntityGraph 实例：

```
EntityGraph<EmailMessage> eg = em.getEntityGraph("emailEntityGraph");
```

14.3　在查询操作中使用实体图

要为有类型和无类型查询指定实体图，需要在查询对象上调用 setHint 方法，并指定 javax.persistence.loadgraph 或 javax.persistence.fetchgraph 作为属性名，另外指定一个 EntityGraph 实例作为值：

```
EntityGraph<EmailMessage> eg = em.getEntityGraph("previewEmailEntityGraph");
List<EmailMessage> messages = em.createNamedQuery("findAllEmailMessages")
        .setParameter("mailbox", "inbox")
        .setHint("javax.persistence.loadgraph", eg)
        .getResultList();
```

在这个例子中，previewEmailEntityGraph 用于 findAllEmailMessages 命名查询。

对于有类型查询，可以使用同样的技术：

```
EntityGraph<EmailMessage> eg = em.getEntityGraph("previewEmailEntityGraph");

CriteriaQuery<EmailMessage> cq = cb.createQuery(EmailMessage.class);
Root<EmailMessage> message = cq.from(EmailMessage.class);
TypedQuery<EmailMessage> q = em.createQuery(cq);
q.setHint("javax.persistence.loadgraph", eg);
List<EmailMessage> messages = q.getResultList();
```

Chapter 15

第 15 章

对 Java 持久化 API 应用使用二级缓存

本章将解释如何修改二级缓存模式设置来提高使用 Java 持久化 API 的应用的性能，包括以下内容：

- 二级缓存概述
- 指定缓存模式设置来提高性能

15.1 二级缓存概述

二级缓存（second-level cache）是持久化提供者管理的一个实体数据本地存储库，用来提高应用的性能。二级缓存可以避免开销昂贵的数据库调用、将实体数据保持在应用本地，从而有助于提高性能。二级缓存对应用通常是透明的，因为它由持久化提供者管理，在应用的持久化上下文底层。也就是说，应用会通过正常的实体管理器操作读取和提交数据，而完全不知道缓存的存在。

说明　不要求持久化提供者一定支持二级缓存。可移植应用不应依赖于持久化提供者对二级缓存的支持。

持久化单元的二级缓存可以配置为以下某种二级缓存模式。下面是 Java 持久化 API 定义的缓存模式设置。

表 15-1　二级缓存的缓存模式设置

缓存模式设置	描　述
ALL	所有实体数据存储在这个持久化单元的二级缓存中
NONE	没有数据在持久化单元中缓存。持久化提供者不缓存任何数据
ENABLE_SELECTIVE	允许缓存用于用 @Cacheable 注解显式设置的实体
DISABLE_SELECTIVE	允许缓存用于除了用 @Cacheable(false) 注解显式设置的实体以外的所有其他实体
UNSPECIFIED	持久化单元的缓存行为未定义。将使用持久化提供者的默认缓存行为

在应用中使用二级缓存的一个后果是：数据库表中的底层数据可能已经改变，而缓存中的值还没有改变，这种情况称为**陈旧读**（stale read）。为了避免陈旧读，可以使用以下策略：

- 将二级缓存改为某个缓存模式设置
- 对哪些实体可以缓存加以控制（参见 15.1.1 节）
- 改变缓存的获取或存储模式（参见 15.2.1 节）

要避免陈旧读，以上哪一种策略最合适取决于具体的应用。

15.1.1　控制实体是否可以缓存

javax.persistence.Cacheable 注解用于指定使用 ENABLE_SELECTIVE 或 DISABLE_SELECTIVE 缓存模式时可以缓存的实体类及子类。子类可以增加 @Cacheable 注解并修改值，从而覆盖父类的 @Cacheable 设置。

要指定一个可以缓存的实体，可以在类级增加一个 @Cacheable 注解：

```
@Cacheable
@Entity
public class Person { ... }
```

默认地，@Cacheable 注解为 true。下面的例子是等价的：

```
@Cacheable(true)
@Entity
public class Person{ ... }
```

要指定一个不能缓存的实体，需要增加一个 @Cacheable 注解并将其设置为 false：

```
@Cacheable(false)
@Entity
public class OrderStatus { ... }
```

设置 ENABLE_SELECTIVE 缓存模式时，持久化提供者会缓存所有增加了 @Cacheable(true) 注解的实体以及该实体未被覆盖的所有子类。持久化提供者不会缓存有 @Cacheable(false) 注解或者没有 @Cacheable 注解的实体。也就是说，ENABLE_SELECTIVE 模式只缓存使用 @Cacheable 注解显式标志要缓存的实体。

设置 DISABLE_SELECTIVE 缓存模式时，持久化提供者将缓存没有 @Cacheable(false) 注解的所有实体。没有 @Cacheable 注解的实体以及有 @Cacheable(true) 注解的实体

将被缓存。也就是说，DISABLE_SELECTIVE 模式将缓存未显式设置为不允许缓存的所有实体。

如果缓存模式设置为 UNDEFINED，或者保留为未设置，增加了 @Cacheable 注解的实体的行为则未定义。如果缓存模式设置为 ALL 或 NONE，则持久化提供者将忽略 @Cacheable 注解的值。

15.2 指定缓存模式设置来提高性能

要调整一个持久化单元的缓存模式设置，可以指定某个缓存模式作为 persistence.xml 部署描述文件中 shared-cache-mode 元素的值（用**粗体**显示）：

```
<persistence-unit name="examplePU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    <shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>
</persistence-unit>
```

说明 由于对二级缓存的支持在 Java 持久化 API 规范中没有严格要求，所以如果你使用一个没有实现二级缓存的持久化提供者，则在 persistence.xml 中设置二级缓存模式没有任何作用。

或者，可以指定共享缓存模式，将 javax.persistence.sharedCache.mode 属性设置为某个共享缓存模式：

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory(
        "myExamplePU", new Properties().add(
            "javax.persistence.sharedCache.mode", "ENABLE_SELECTIVE"));
```

15.2.1 设置缓存获取和存储模式

如果通过设置共享缓存模式来支持一个持久化单元的二级缓存，则可以设置 javax.persistence.cache.retrieveMode 和 javax.persistence.cache.storeMode 属性进一步修改二级缓存的行为。通过将属性名和值传递到 EntityManager.setProperty 方法，可以在持久化上下文级设置属性；或者可以在各个 EntityManager 操作上（EntityManager.find 或 EntityManager.refresh）或在各个查询级设置这些属性。

15.2.1.1 缓存获取模式

缓存获取模式由 javax.persistence.retrieveMode 属性设置，会控制如何为

EntityManager.find 方法调用从缓存以及从查询读取数据。

可以将 retrieveMode 属性设置为由 javax.persistence.CacheRetrieveMode 枚举类型定义的某个常量，可以是 USE（默认值）或 BYPASS。

这个属性设置为 USE 时，数据从二级缓存获取（如果可用）。如果数据不在缓存中，持久化提供者将从数据库读取数据。

这个属性设置为 BYPASS 时，会绕过二级缓存，调用数据库来获取数据。

15.2.1.2　缓存存储模式

缓存存储模式由 javax.persistence.storeMode 属性设置，会控制数据在缓存中如何存储。

storeMode 属性可以设置为由 javax.persistence.CacheStoreMode 枚举类型定义的某个常量：可以是 USE（默认值）、BYPASS 或 REFRESH。

这个属性设置为 USE 时，从数据库读取数据或者向数据库提交数据时，就会创建或更新缓存数据。如果数据已经在缓存中，则设置存储模式为 USE 时，从数据库读取数据不会强制刷新。

这个属性设置为 BYPASS 时，从数据库读取或提交到数据库的数据不会插入或更新到缓存中。也就是说，缓存保持不变。

这个属性设置为 REFRESH 时，从数据库读取数据或者向数据库提交数据时，就会创建或更新缓存数据，而且从数据库读取数据时会强制对缓存中的数据进行刷新。

15.2.1.3　设置缓存获取或存储模式

要为持久化上下文设置缓存获取或存储模式，需要调用 EntityManager.setProperty 方法并提供属性名值对：

```
EntityManager em = ...;
em.setProperty("javax.persistence.cache.storeMode", "BYPASS");
```

调用 EntityManager.find 或 EntityManager.refresh 方法时要设置缓存获取或存储模式，首先创建一个 Map<String, Object> 实例，并增加一个名 / 值对，如下所示：

```
EntityManager em = ...;
Map<String, Object> props = new HashMap<String, Object>();
props.put("javax.persistence.cache.retrieveMode", "BYPASS");
String personPK = ...;
Person person = em.find(Person.class, personPK, props);
```

说明　调用 EntityManager.refresh 方法时会忽略缓存获取模式，因为 refresh 调用总会导致从数据库读取数据，而不是从缓存读取。

使用查询时要设置获取或存储模式，要调用 Query.setHint 或 TypedQuery.

setHint 方法，这取决于查询的类型：

```
EntityManager em = ...;
CriteriaQuery<Person> cq = ...;
TypedQuery<Person> q = em.createQuery(cq);
q.setHint("javax.persistence.cache.storeMode", "REFRESH");
...
```

如果在查询中设置获取或存储模式，或者调用 EntityManager.find 或 EntityManager.refresh 方法设置获取或存储模式，这会覆盖实体管理器的设置。

15.2.2 通过编程方式控制二级缓存

javax.persistence.Cache 接口定义了一些方法，可以通过编程与二级缓存交互。Cache 接口定义的方法可以完成以下工作：

- 检查一个特定实体是否有缓存数据
- 从缓存删除一个特定实体
- 从缓存删除一个实体类的所有实例（及子类实例）
- 清空所有实体数据的缓存

说明 如果已经禁用二级缓存，则调用 Cache 接口的方法没有任何作用，但 contains 除外，禁用二级缓存时这个方法总是返回 false。

15.2.2.1 检查一个实体的数据是否缓存

调用 Cache.contains 方法可以查找给定实体当前是否在二级缓存中。如果实体数据已经缓存，contains 方法会返回 true，如果数据未在缓存中，则返回 false：

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;
if (cache.contains(Person.class, personPK)) {
    // the data is cached
} else {
    // the data is NOT cached
}
```

15.2.2.2 从缓存删除一个实体

调用 Cache.evict 方法可以从二级缓存删除一个特定实体或者给定类型的所有实体。要从缓存删除某一个实体，可以调用 evict 方法，并传入这个实体类以及实体的主键：

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;
cache.evict(Person.class, personPK);
```

要删除一个特定实体类（包括子类）的所有实例，可以调用 evict 方法并指定这个实体类：

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evict(Person.class);
```

Person 实体类的所有实例将从缓存删除。如果 Person 实体有一个子类 Student，调用以上方法还会从缓存中删除 Student 的所有实例。

15.2.2.3　从缓存删除所有数据

调用 Cache.evictAll 方法可以完全清空二级缓存：

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evictAll();
```

第四部分 *Part 4*

消息传送

第四部分介绍消息传送，包括以下各章：

- 第 16 章　Java 消息服务概念
- 第 17 章　Java 消息服务示例

Chapter 16 第 16 章

Java 消息服务概念

本章将介绍 Java 消息服务（Java Message Service，JMS）API，这个 Java API 允许应用使用可靠、异步、松耦合的通信创建、发送、接收和读取消息，包括以下内容：

- JMS API 概述
- 基本 JMS API 概念
- JMS API 编程模型
- 使用高级 JMS 特性
- 在 Java EE 应用中使用 JMS API
- 关于 JMS 的更多信息

16.1 JMS API 概述

这个概述会给出消息传送概念的定义，描述 JMS API 以及它在哪里使用，并解释 Java EE 平台中 JMS API 如何工作。

16.1.1 什么是消息传送

消息传送是软件组件或应用之间的一种通信方法。消息传送系统是一个端对端工具：消息客户端可以向任何其他客户端发送消息以及从其他客户端接收消息。每个客户端连接到一个消息传送代理，这个代理提供创建、发送、接收和读取消息的工具。

采用消息传送，可以建立**松耦合**的分布式通信。一个组件向一个目的地发送一个消息，接收者可以从目的地获取这个消息。通信之所以可以做到松耦合，关键是目的地是发送者和

接收者共有的。发送者和接收者要建立通信，并不要求同时可用。实际上，发送者不需要对接收者有任何了解，接收者也不需要知道发送者。发送者和接收者只需要知道采用何种消息格式以及要使用哪个目的地。在这个方面，消息传送与紧耦合技术有所区别，如远程方法调用（Remote Method Invocation，RMI），采用 RMI 时，应用必须知道远程应用的方法。

消息传送与电子邮件（email）也有区别，电子邮件是人们之间或者软件应用与人之间的一种通信方法。消息传送则用于软件应用或软件组件之间的通信。

16.1.2 什么是 JMS API

Java 消息服务（Java Message Service，JMS）是一个允许应用创建、发送、接收和读取消息的 Java API。JMS API 定义了一组公共的接口和相关语义，允许用 Java 编写的程序与其他消息传送实现进行通信。

利用 JMS API，程序员只需学习最少的概念就可以使用消息产品，另外 JMS API 提供了足够的特性来支持复杂的消息传送应用。另外还尽可能地提高了 JMS 应用对不同 JMS 提供者的可移植性。

JMS 支持的通信不仅可以做到松耦合，还有以下特性：

- **异步**：接收客户端不必在发送客户端发送消息的同时接收消息。发送客户端可以发送消息，然后继续完成其他任务；接收客户端可以在以后某个时间接收这些消息。
- **可靠**：实现 JMS API 的消息提供者可以确保一个消息传送一次而且只传送一次。对于允许漏消息或允许接收重复消息的应用来说，可以提供较低级别的可靠性。

JMS 规范的当前版本是版本 2.0。可以从 Java 社区进程（Java Community Process）网站下载这个规范：

```
http://www.jcp.org/en/jsr/detail?id=343.
```

16.1.3 何时使用 JMS API

在以下情况下，企业应用提供商可能更愿意选择消息传送 API 而不是紧耦合的 API（如远程过程调用 RPC）：

- 提供商希望组件不要依赖于其他组件接口的有关信息，这样组件就能很容易地替换。
- 提供商希望不论是否所有组件都同时在运行，应用都能正常运行。
- 应用业务模型允许一个组件向另一个组件发送信息，然后继续完成自己的操作，而不用立即接收响应。

例如，一个汽车制造商的企业应用中，组件可以在类似下面的情况下使用 JMS API：

- 某个产品的库存量低于某个等级时，库存组件可以向工厂组件发送一个消息，使工厂生产更多汽车。
- 工厂组件可以向部件组件发送一个消息，使得工厂可以装配它需要的部件。

- 部件组件可以进一步向它们自己的库存和订单组件发送消息来更新库存，并从供应商订购新部件。
- 工厂和部件组件可以向会计组件发送消息来更新预算金额。
- 企业可以向销售部门发布更新后的产品目录。

通过使用消息传送来完成这些任务，不同组件之间可以高效地交互，而不需要占用网络或其他资源。图 16-1 展示了这个简单的例子如何工作。

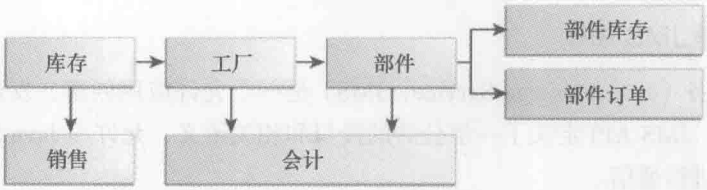


图 16-1 企业应用中的消息传送

生产制造只是企业使用 JMS API 的一个例子。零售应用、金融服务应用、健康服务应用以及很多其他的应用都可以使用消息传送。

16.1.4 Java EE 平台如何使用 JMS API

JMS API 最初引入时，它最重要的作用是允许 Java 应用访问已有的面向消息中间件（messaging-oriented middleware，MOM）系统。在那之后，很多开发商已经采用并实现了 JMS API，所以现在 JMS 产品可以为企业提供完整的消息传送功能。

JMS API 是 Java EE 平台中不可缺少的一部分，应用开发人员可以结合使用消息传送和 Java EE 组件。JMS 2.0 是 Java EE 7 的一部分。

Java EE 平台中的 JMS API 有以下特性：

- 应用客户端、企业 JavaBeans（EJB）组件和 Web 组件可以发送或同步接收 JMS 消息。应用客户端还可以设置消息监听器，在消息可用时通知客户端，向它异步传送 JMS 消息。
- 消息驱动 bean 作为一种企业 bean，允许在 EJB 容器中异步使用消息。应用服务器通常会建立消息驱动 bean 池来实现消息的并发处理。
- 消息发送和接收操作可以加入到 JTA 事务中（Java Transaction API，JTA，即 Java 事务 API），从而允许 JMS 操作和数据库访问在一个事务中完成。

JMS API 还对 Java EE 平台的其他方面有所提升：简化了企业开发，允许 Java EE 组件与能够实现消息传送的遗留系统之间进行松耦合、可靠、异步的交互。开发人员可以很容易地为有业务事件的 Java EE 应用增加新行为，为此可以增加一个新的消息驱动 bean 来处理特定的业务事件。另外，Java EE 平台对 JMS API 也有促进，对 JTA 事务提供了支持，并允许消息的并发使用。有关的更多信息，参见 Enterprise JavaBeans 规范 v3.2。

JMS 提供者可以使用 Java EE 连接器架构与应用服务器集成。你可以通过一个资源适配器访问 JMS 提供者。这个功能使得开发商可以创建能够插入到多个应用服务器的 JMS 提供者，而且允许应用服务器支持多个 JMS 提供者。有关的更多信息，参见 Java EE 连接器架构规范 v1.7.S。

16.2 基本 JMS API 概念

这一节介绍最基本的 JMS API 概念，必须掌握这些概念才能开始编写使用 JMS API 的简单应用客户端程序。

下一节将介绍 JMS API 编程模型。后面的小节还会介绍一些更高级的概念，包括编写使用消息驱动 bean 的应用时所需的一些概念。

16.2.1 JMS API 架构

JMS 应用由以下部分组成：

- **JMS 提供者**（JMS provider）是一个实现 JMS 接口的消息传送系统，提供管理和控制特性。支持全部特性的 Java EE 平台实现就包括一个 JMS 提供者。
- **JMS 客户端**（JMS client）是用 Java 编程语言编写的程序或组件，可以生成（生产）和使用（消费）消息。所有 Java EE 应用组件都可以作为一个 JMS 客户端。

Java SE 应用也可以作为 JMS 客户端；GlassFish 服务器文档中的 *Message Queue Developer's Guide for Java Clients*（`https://glassfish.java.net/docs/`）解释了如何将 Java SE 应用作为 JMS 客户端。

- **消息**（Message）是在 JMS 客户端之间传递信息的对象。
- **受管理对象**（Administered Object）是为客户端配置的 JMS 对象。有两类 JMS 受管理对象：目的地（destination）和连接工厂（connection factory），将在 16.3.1 节中介绍。管理员可以创建公共可用的对象，所有使用某个 GlassFish 服务器的应用都可以使用这些对象；或者，开发人员可以使用注解来创建特定于某个应用的对象。

图 16-2 展示了这些部分之间如何交互。利用管理工具或注解，可以将目的地和连接工厂绑定到一个 JNDI 命名空间。然后 JMS 客户端可以使用资源注入来访问命名空间中的受管理对象，再通过 JMS 提供者建立与这些对象的一个逻辑连接。

16.2.2 消息传送方式

在 JMS API 出现之前，大多数消息传送产品都支持点对点（PTP）或发布 / 订阅消息传送方式。JMS 规范为这两种方式都定义了兼容形式。JMS 提供者必须实现这两种方式，JMS API 还分别提供了这两种方式特定的接口。下面各小节将介绍这些消息传送方式。

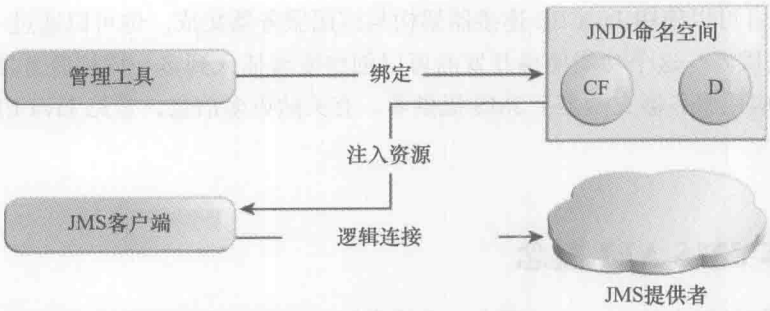


图 16-2 JMS API 架构

不过，利用 JMS API，没有必要只使用其中一种方式。它允许你使用相同的代码发送和接收消息，可以使用点对点方式，也可以使用发布 / 订阅方式。使用的目的地仍特定于某一个方式，另外应用的行为在某种程度上依赖于使用队列还是主题。不过，代码本身对这两种方式都是通用的，这就可以保证应用的灵活性和可重用性。本教程将介绍和展示如何采用这种方法编写代码，这里将使用 JMS 2.0 提供的大幅简化的 API。

16.2.2.1 点对点消息传送方式

点对点（point-to-point，PTP）产品或应用建立在消息队列、发送者和接收者的概念基础上。建立队列来维护消息，每个消息都发送到一个特定的队列，接收客户端从这些队列中抽取消息。队列会保留发送给它们的所有消息，直到消息被使用或到期。

PTP 消息传送见图 16-3，有以下特点：

- 每个消息只有一个消费者（即使用方）。
- 接收者可以获取消息，而不论客户端发送消息时接收者是否正在运行。

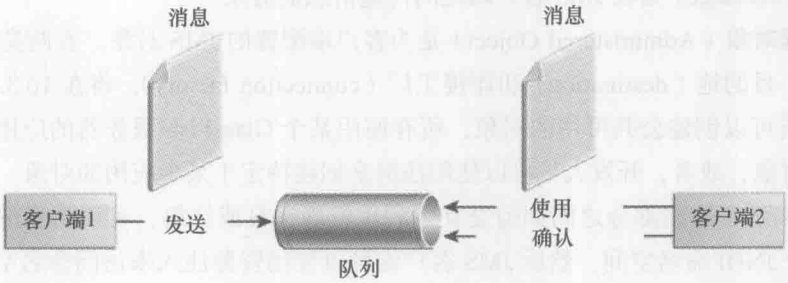


图 16-3 点对点消息传送

如果你发送的每一个消息都必须由一个消费者成功地处理，就可以使用 PTP 消息传送。

16.2.2.2 发布 / 订阅消息传送方式

在一个**发布 / 订阅**（publish/subscribe，pub/sub）产品或应用中，客户端将消息发送到一

个**主题**（topic），它的功能有些类似于公告板。发布者和订阅者可以动态地向主题发布或订阅。会有多个发布者向主题发送消息，系统负责将到达主题的消息发布给主题的多个订阅者。主题只是在向订阅者发布消息时才保留消息。

采用 pub/sub 消息传送方式时，要区分订阅主题的消费者（订阅者）和所创建的消息订阅（subscription），这很重要。消费者是应用中的一个 JMS 对象，而订阅是 JMS 提供者中的一个实体。一般地，一个主题可以有多个消费者，但是一个订阅只有一个订阅者。不过，有可能创建共享的订阅。详细信息参见 16.3.6.5 节。关于发布 / 订阅消息传送的语义，更多详细信息参见 16.2.6.3 节。

发布 / 订阅消息传送有以下特点：

- 每个消息可以有多个消费者。
- 对于订阅一个主题的客户端，只有这个客户端创建一个订阅之后才能使用所发送的消息，消费者必须保持活动状态才能使用消息。

JMS API 在一定程度上放松了这个需求，允许应用创建**持久订阅**（durable subscription），利用持久订阅，消费者不活动时也可以接收发送的消息。持久订阅提供了队列的灵活性和可靠性，而且仍然允许客户端向多个接收方发送消息。关于持久订阅的更多信息，参见 16.3.6.4 节。

如果每个消息可以由任意多个消费者（或者没有任何消费者）处理，就可以使用发布 / 订阅消息传送方式。图 16-4 展示了发布 / 订阅消息传送。

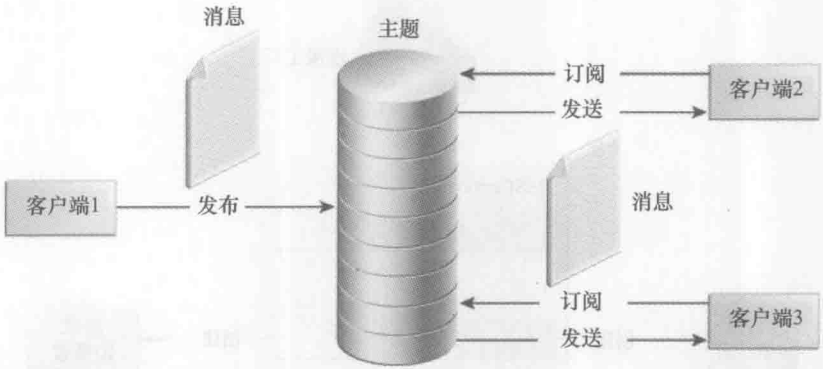


图 16-4　发布 / 订阅消息传送

16.2.3　消息使用

消息产品本质上是异步的：在消息的生成和使用之间并没有时间依赖关系。不过，JMS 规范使用这个术语有一个更精确的含义。可以采用以下两种方式使用消息：

- **同步**：消费者通过调用 `receive` 方法，显式地从目的地获取消息。`receive` 方法会阻塞，直到消息到达，或者如果消息未在指定的时限内到达则会超时。

- **异步**：应用客户端或 Java SE 客户端可以向消费者注册一个**消息监听器**（message listener）。消息监听器类似于事件监听器。一旦有消息到达目的地，JMS 提供者就通过调用这个监听器的 `onMessage` 方法传送消息，这个方法会处理消息的内容。在一个 Java EE 应用中，消息驱动 bean 会作为消息监听器（它也有一个 `onMessage` 方法），不过客户端不需要向消费者注册这个监听器。

16.3 JMS API 编程模型

JMS 应用的基本构件包括：

- 受管理对象：连接工厂和目的地
- 连接
- 会话
- `JMSContext` 对象，它将一个连接和一个会话结合在一个对象中
- 消息生产者（即生成方）
- 消息消费者（即使用方）
- 消息

图 16-5 显示了所有这些对象如何集成在一个 JMS 客户端应用中。

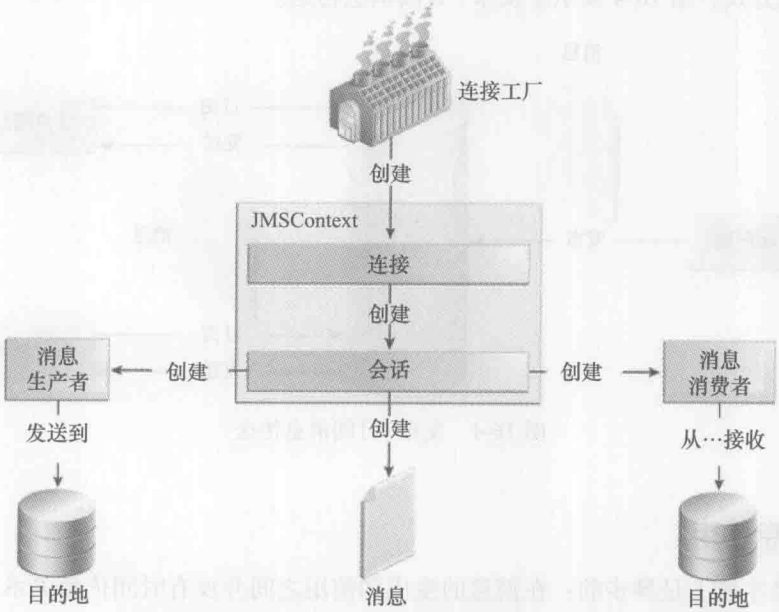


图 16-5 JMS API 编程模型

JMS 还提供了队列浏览器，队列浏览器对象允许应用浏览队列中的消息。

这一节将简要地介绍所有这些对象，并提供一些示例命令和代码段，展示如何创建和使用这些对象。最后一小节将简要介绍 JMS API 异常处理。

第 17 章从 17.2 节开始将提供一些例子，展示如何在应用中结合使用所有这些对象。有关的更多详细信息，参见 JMS API 文档（Java EE API 文档的一部分）。

16.3.1 JMS 受管理对象

JMS 应用中有两个部分（目的地和连接工厂）通常采用管理方式而不是编程方式来维护。对于 JMS API 的不同实现，这些对象的底层技术可能有很大差别。因此，与其他管理任务一样，这些对象的管理也会随提供者的不同而不同。

JMS 客户端通过一些可移植的接口访问受管理对象，所以客户端应用可以只做少量修改甚至不做修改就能在 JMS API 的多个实现上运行。正常情况下，管理员会在一个 JNDI 命名空间中配置受管理对象，然后 JMS 客户端通过使用资源注入来访问这些对象。

利用 GlassFish 服务器，可以使用 `asadmin create-jms-resource` 命令或管理控制台以连接器资源的形式创建 JMS 受管理对象。还可以在一个名为 `glassfish-resources.xml` 的文件中指定这些资源，这个文件可以与应用一同打包。

NetBeans IDE 提供了一个向导工具，允许为 GlassFish 服务器创建 JMS 资源。更多详细信息参见 17.2.2 节。

Java EE 平台规范允许开发人员使用注解或部署描述文件元素创建受管理对象。以这种方式创建的对象特定于创建该对象的应用。更多信息参见 16.5.1 节。部署描述文件中的定义会覆盖注解指定的定义。

16.3.1.1 JMS 连接工厂

连接工厂（connection factory）是客户端用来创建连接的对象，客户端可以用它来创建与一个提供者的连接。连接工厂封装了由管理员定义的一组连接配置参数。每个连接工厂是 `ConnectionFactory`、`QueueConnectionFactory` 或 `TopicConnectionFactory` 接口的一个实例。要了解如何创建连接工厂，可参见 17.2.2. 节。

在 JMS 客户端程序最前面，通常要将一个连接工厂资源注入到一个 `ConnectionFactory` 对象。Java EE 服务器必须提供一个逻辑 JNDI 名为 `java:comp/DefaultJMSConnectionFactory` 的 JMS 连接工厂。实际 JNDI 名由具体实现确定。

例如，下面的代码段会查找默认 JMS 连接工厂，并赋给一个 `ConnectionFactory` 对象：

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
```

16.3.1.2 JMS 目的地

客户端可以用**目的地**（destination）对象指定所生成消息的目标以及所使用消息的来源。在 PTP 消息传送方式中，目的地称为队列。在发布 / 订阅消息传送方式中，目的地称为主

题。JMS 应用可以使用多个队列或主题（或者二者都使用）。要了解如何创建目的地资源，参见 17.2.2 节。

要使用 GlassFish 服务器创建一个目的地，可以创建一个 JMS 目的地资源，并为目的地指定一个 JNDI 名。

在 JMS 的 GlassFish 服务器实现中，每个目的地资源指示一个物理目的地。可以显式地创建一个物理目的地，不过如果没有创建物理目的地，应用服务器便会在需要时创建，并在你删除目的地资源时将它删除。

除了将连接工厂资源注入客户端程序外，通常还要注入一个目的地资源。与连接工厂不同，目的地特定于 PTP 或发布 / 订阅消息传送方式。要创建一个允许对主题和队列使用相同代码的应用，则要将目的地赋给一个 Destination 对象。

下面的代码指定了两个资源：一个队列和一个主题。资源名映射到 JNDI 命名空间中创建的目的地资源：

```
@Resource(lookup = "jms/MyQueue")
private static Queue queue;

@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

在 Java EE 应用中，JMS 受管理对象通常放在 jms 命名子上下文中。

利用通用接口，可以混合使用或匹配连接工厂和目的地。也就是说，除了使用 ConnectionFactory 接口，还可以注入一个 QueueConnectionFactory 资源并与一个 Topic 结合使用，而且可以注入一个 TopicConnectionFactory 资源并与一个 Queue 结合使用。应用的行为取决于所使用的目的地类型，而与使用的连接工厂类型无关。

16.3.2 连接

连接（connection）封装了与一个 JMS 提供者的虚拟连接。例如，连接可以表示客户端和提供者服务后台程序之间的一个打开的 TCP/IP socket。可以使用连接来创建一个或多个会话。

说明 在 Java EE 平台中，只有应用客户端能够为一个连接创建多个会话。在 Web 和企业 bean 组件中，一个连接最多只能创建一个会话。

通常通过创建 JMSContext 对象来创建连接。更多详细信息参见 16.3.4 节。

16.3.3 会话

会话（session）是生成和使用消息的一个单线程上下文。

通常会通过创建一个 JMSContext 对象来创建会话（以及连接）。更多详细信息参见

16.3.4 节。可以使用会话来创建消息生产者、消息消费者、消息、队列浏览器和临时目的地。

会话将串行化消息监听器的执行；有关的详细信息参见 16.3.6.1 节。

会话提供了一个事务上下文，利用它可以将一组发送和接收操作分组到一个原子工作单元。更多详细信息参见 16.4.4 节。

16.3.4 JMSContext 对象

JMSContext 对象将一个连接和一个会话结合在一个对象中。也就是说，它既为 JMS 提供者提供一个活动的连接，又提供了一个单线程上下文来发送和接收消息。

可以使用 JMSContext 创建以下对象：

- 消息生产者
- 消息消费者
- 消息
- 队列浏览器
- 临时队列和主题（参见 16.4.3 节）

可以在 try-with-resources 块中创建 JMSContext。

要创建 JMSContext，需要在连接工厂上调用 createContext 方法：

```
JMSContext context = connectionFactory.createContext();
```

如果从应用客户端或 Java SE 客户端调用（但是没有提供参数），或者从 Java EE Web 或 EJB 容器调用，而当前没有活动的 JTA 事务，则 createContext 方法会创建一个非事务性会话，其确认模式为 JMSContext.AUTO_ACKNOWLEDGE。从 Web 或 EJB 容器调用时，如果无活动 JTA 事务而且未提供参数，则 createContext 方法会创建一个事务性会话。关于在 Java EE 应用中使用 JMS 事务的有关信息，参见 16.5 节。

从应用客户端或 Java SE 客户端调用 createContext 方法时，还可以提供参数 JMSContext.SESSION_TRANSACTED 来创建一个事务性会话：

```
JMSContext context =
        connectionFactory.createContext(JMSContext.SESSION_TRANSACTED);
```

会话使用本地事务；详细信息参见 16.4.4 节。或者，可以指定一个非默认的确认模式；有关的更多信息参见 16.4.1 节。

使用 JMSContext 时，通常一旦创建一个消费者，消息传送就开始了。更多信息参见 16.3.6 节。

如果在一个 try-with-resources 块中创建 JMSContext，则不需要显式地关闭它。try 块结束时它就会关闭。要确保你的应用中所有 JMS 活动都在 try-with-resources 块中完成。如果没有使用 try-with-resources 块，则必须对 JMSContext 调用 close 方法，保证应用完成工作时关闭这个连接。

16.3.5 JMS 消息生产者

消息生产者（message producer）是由 JMSContext 或会话创建的一个对象，用来向一个目的地发送消息。由 JMSContext 创建的消息生产者实现了 JMSProducer 接口。可以使用以下方法来创建：

```
try (JMSContext context = connectionFactory.createContext();) {
    JMSProducer producer = context.createProducer();
    ...
```

不过，JMSProducer 是一个轻量级对象，不会使用太过庞大的资源。出于这个原因，不需要把 JMSProducer 保存在变量中；每次发送一个消息时可以创建一个新的 JMSProducer 对象。向一个特定目的地发送消息时要使用 send 方法。例如：

```
context.createProducer().send(dest, message);
```

在发送消息前，可以先创建这个消息并保存在一个变量中（如上所示），或者也可以在 send 调用中直接创建消息。更多信息参见 16.3.7 节。

16.3.6 JMS 消息消费者

消息消费者（message consumer）是由 JMSContext 或会话创建的一个对象，用来接收发送到一个目的地的消息。由 JMSContext 创建的消息消费者实现了 JMSConsumer 接口。要创建消息消费者，最简单的方法是使用 JMSContext.createConsumer 方法：

```
try (JMSContext context = connectionFactory.createContext();) {
    JMSConsumer consumer = context.createConsumer(dest);
    ...
```

利用消息消费者，JMS 客户端可以向一个 JMS 提供者注册它对一个目的地感兴趣。这个 JMS 提供者会管理从一个目的地向该目的地已注册消费者的消息传送。

使用 JMSContext 创建一个消息消费者时，一旦创建了消费者，消息传送就会开始。可以在创建 JMSContext 时调用 setAutoStart(false) 来禁用这种行为，然后调用 start 方法显式地启动消息传送。如果希望临时停止消息传送而不关闭连接，则可以调用 stop 方法；要重新启动消息传送，可以调用 start。

利用 receive 方法来同步地使用消息。可以在创建消费者之后的任何时间使用这个方法。

如果没有指定参数，或者指定了一个参数 0，那么这个方法会无限阻塞，直到一个消息到达：

```
Message m = consumer.receive();
Message m = consumer.receive(0);
```

对于一个简单的客户端，这可能没有什么影响。但是如果消息有可能不会到达，则要使用一个有超时设置的同步接收方法：调用 receive 方法时提供一个大于 0 的超时参数。推

荐的超时值是 1 秒：

```
Message m = consumer.receive(1000); // time out after a second
```

要从一个应用客户端或 Java SE 客户端支持异步消息传送，可以使用消息监听器，见下一节的介绍。

可以使用 JMSContext.createDurableConsumer 方法来创建一个持久主题订阅。只有使用主题时这个方法才有效。有关的详细信息，参见 16.3.6.4 节。对于主题，还可以创建共享消费者；参见 16.3.6.5 节。

16.3.6.1 JMS 消息监听器

消息监听器对象相当于消息的一个异步事件处理器。这个对象实现了 MessageListener 接口，其中包含一个方法 onMessage。在 onMessage 方法中，定义了消息到达时要采取的动作。

对于一个应用客户端或 Java SE 客户端，可以使用 setMessageListener 方法向一个特定的消息消费者注册消息监听器。例如，如果定义一个名为 Listener 的类且这个类实现了 MessageListener 接口，则可以如下注册消息监听器：

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

消息传送开始时，只要有消息传送，JMS 提供者就会自动调用消息监听器的 onMessage 方法。onMessage 方法取一个类型为 Message 的参数，具体的方法实现可能会根据需要把它强制转换为另外一个消息子类型（参见 16.3.7.3 节）。

在 Java EE Web 或 EJB 容器中，会使用消息驱动 bean 来完成异步消息传送。消息驱动 bean 也实现了 MessageListener 接口，并包含一个 onMessage 方法。有关的详细信息，参见 16.5.4 节。

onMessage 方法应当处理所有异常。如果抛出一个 RuntimeException，则会认为是一个编程错误。

17.2.6 节提供了关于如何使用消息监听器的一个简单例子。第 17 章中给出了更多关于消息监听器和消息驱动 bean 的例子。

16.3.6.2 JMS 消息选择器

如果你的消息应用需要过滤接收到的消息，则可以使用一个 JMS 消息选择器，它允许一个目的地的消息消费者指定它感兴趣的消息。消息选择器将过滤消息的工作指派给 JMS 提供者而不是交给应用。17.7 节给出了一个使用消息选择器的应用的例子。

消息选择器是包含一个表达式的 String。这个表达式的语法建立在 SQL92 条件表达式语法的一个子集基础上。下面这个例子中的消息选择器会选择 NewsType 属性设置为值 'Sports' 或 'Opinion' 的所有消息：

```
NewsType = 'Sports' OR NewsType = 'Opinion'
```

createConsumer 和 createDurableConsumer 方法以及创建共享消费者的方法允许在创建消息消费者时指定一个消息选择器作为参数，这样一来，消息消费者只接收首部和属性与选择器匹配的消息（参见 16.3.7.1 节）。消息选择器不能根据消息体的内容来选择消息。

16.3.6.3 使用主题消息

使用主题消息的语义要比使用队列消息的语义更复杂。

应用使用主题消息时，要在该主题上创建一个订阅，并为这个订阅创建一个消费者。订阅可以是持久的或非持久的，另外可以共享或非共享。

可以认为订阅是 JMS 提供者本身的一个实体，而消费者（即订阅者）是应用中的一个 JMS 对象。

创建订阅后，订阅会接收发送到这个主题的各个消息的一个副本（除非指定了消息选择器）。如果指定了一个消息选择器，则只有属性与消息选择器匹配的那些消息才会增加到订阅。

非共享订阅仅限于一个消费者。在这种情况下，订阅中的所有消息都会传送到这个消费者。共享订阅允许有多个消费者。如果是共享订阅，则订阅中的各个消息分别只传送到某一个消费者。JMS 没有定义同一个订阅的多个消费者之间如何分布消息。

订阅可以是持久或非持久的。

只有当订阅有一个活动的消费者时，非持久订阅才有意义。这意味着，只有当存在一个消费者而且这个消费者未关闭时，发送到主题的消息才会增加到这个订阅。

非持久订阅可以共享或非共享：

- 非共享非持久订阅没有名字，只能有一个消费者对象与它关联。创建消费者对象时它会自动创建。它不会持久存储，消费者对象关闭时，就会将它自动删除。

如果将一个主题指定为目的地，则 JMSContext.createConsumer 方法会在一个非共享非持久订阅上创建消费者。

- 共享非持久订阅由订阅名和一个可选的客户端标识符来标识，可以有多个消费者对象使用这个订阅上的消息。创建第一个消费者对象时，它会自动创建。它不会持久存储，最后一个消费者对象关闭时它就会自动删除。更多相关信息参见 16.3.6.5 节。

以更高的开销为代价，订阅也可以是持久的。持久订阅会持久存储，即使已经没有消费者对象使用这个订阅的消息，它也会继续累积消息，直到将其显式删除。更多详细信息参见 16.3.6.4 节。

16.3.6.4 创建持久订阅

为了确保一个发布 / 订阅应用接收到发送的所有消息，要为主题上的消费者使用持久订阅。

与非持久订阅类似，持久订阅也可以共享或非共享。

- 非共享持久订阅由订阅名和客户端标识符（必须设置）来标识，只能有一个消费者对象与它关联。
- 共享持久订阅由订阅名和一个可选的客户端标识符来标识，可以有多个消费者对象使用它的消息。

如果存在一个持久订阅，但是当前并没有未关闭的消费者对象与它关联，则称这个持久订阅是不活动的。

可以使用 JMSContext.createDurableConsumer 方法在一个非共享持久订阅上创建消费者。非共享持久订阅一次只能有一个活动的消费者。

消费者指定一个唯一的标识（由 JMS 提供者保留）来标识它使用哪一个持久订阅上的消息。以后有相同标识的消费者对象就会将订阅恢复到上一个消费者留下的那个状态。如果一个持久订阅没有活动的消费者，则 JMS 提供者会保留订阅的消息，直到订阅接收消息或者这些消息到期。

建立一个非共享持久订阅的唯一标识时，要设置以下内容：

- 连接的客户端 ID
- 订阅的主题和订阅名

可以使用命令行或者管理控制台采用管理方式为特定于一个客户端的连接工厂设置客户端 ID。（在应用客户端或 Java SE 客户端中，可以调用 JMSContext.setClientID 来设置。）

使用这个连接工厂创建 JMSContext 之后，调用 createDurableConsumer 方法时要提供两个参数：主题和一个指定订阅名的字符串：

```
String subName = "MySub";
JMSConsumer consumer = context.createDurableConsumer(myTopic, subName);
```

创建消费者之后，订阅会变为活动状态。之后可以关闭消费者：

```
consumer.close();
```

JMS 提供者会存储发送到队列的消息，同样地，JMS 提供者也会存储发送到主题的消息。如果程序或另一个应用使用相同的连接工厂及其客户端 ID、相同的主题以及相同的订阅名调用 createDurableConsumer，则会重新激活这个订阅，JMS 提供者将传送订阅在非活动状态时未接收的消息。

要删除一个持久订阅，首先要关闭消费者，然后调用 unsubscribe 方法并提供订阅名作为参数：

```
consumer.close();
context.unsubscribe(subName);
```

unsubscribe 方法会删除提供者为订阅维护的状态。

图 16-6 和图 16-7 显示了非持久和持久订阅之间的差别。对于一个正常的非持久订阅，消费者和订阅在同一点开始和结束，实际上，二者是相同的。消费者关闭时，订阅也会结

束。在这里，create 表示一个 JMSContext.createConsumer 调用（并提供 Topic 参数），close 表示一个 JMSConsumer.close 调用。在第一个 close 与第二个 create 之间发送到主题的所有消息不会增加到任意一个订阅中。在图 16-6 中，消费者会接收消息 M1、M2、M5 和 M6，不过不会接收消息 M3 和 M4。

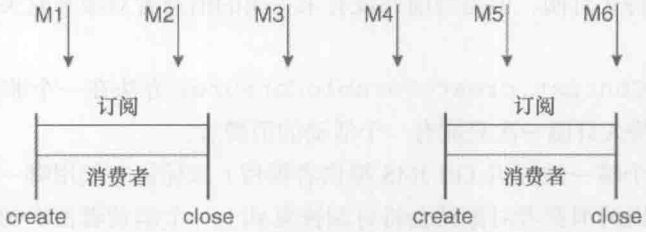


图 16-6 非持久订阅和消费者

对于一个持久订阅，可以关闭和重新创建消费者，不过订阅会继续存在，并保存消息，直到应用调用 unsubscribe 方法。在图 16-7 中，create 表示一个 JMSContext.createDurableConsumer 调用，close 表示一个 JMSConsumer.close 调用，unsubscribe 表示一个 JMSContext.unsubscribe 调用。第一个消费者关闭之后发送的消息会在（同一个持久订阅上的）第二个消费者创建时接收，所以即使消息 M2、M4 和 M5 到达时没有消费者，这些消息也不会丢失。

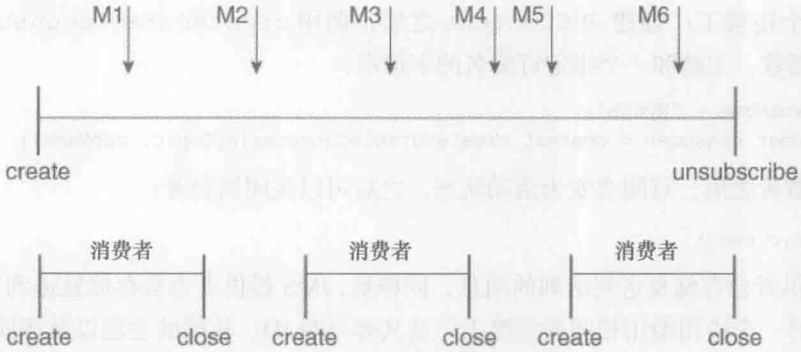


图 16-7 持久订阅上的消费者

共享持久订阅允许使用多个消费者来接收一个持久订阅的消息。如果使用共享持久订阅，则所使用的连接工厂不需要一个客户端标识符。要创建一个共享持久订阅，可以调用 JMSContext.createSharedDurableConsumer 方法并指定主题和订阅名：

```
JMSConsumer consumer =
        context.createSharedDurableConsumer(topic, "MakeItLast");
```

关于使用持久订阅的 Java EE 应用的例子，参见以下各节：17.2.9 节、17.3.1 节、17.4.2 节、17.7 节。

16.3.6.5 创建共享订阅

由 createConsumer 或 createDurableConsumer 方法创建的主题订阅只能有一个消费者（尽管主题本身可以有多个消费者）。根据定义，使用同一个主题的多个客户端可以有这个主题的多个订阅，所有客户端会接收发送到这个主题的所有消息（除非用消息选择器过滤了消息）。

不过，可以使用 createSharedConsumer 方法并指定一个目的地和订阅名，为一个主题创建非持久共享订阅：

```
consumer = context.createSharedConsumer(topicName, "SubName");
```

对于一个共享订阅，消息会分布于使用相同主题和订阅名的多个客户端。发送到主题的各个消息将增加到每一个订阅（但可能会受消息选择器的限制），不过增加到一个订阅的各个消息只会传送到这个订阅的某一个消费者，所以只会由某一个客户端接收。如果希望订阅的多个消费者之间共享消息，而不是仅由订阅的一个消费者接收全部消息，共享订阅就很有用。这个特性可以改善 Java EE 应用客户端应用和 Java SE 应用的可扩展性。（消息驱动 bean 会把一个主题的消息处理工作分摊到多个线程。）

17.4.1 节给出了一个使用共享非持久订阅的简单例子。

还可以使用 JMSContext.createSharedDurableConsumer 方法创建共享持久订阅。有关的详细信息，参见 16.3.6.4 节。

16.3.7 JMS 消息

JMS 应用的最终目的是生成和使用消息，然后由其他软件应用使用。JMS 消息有一个基本格式，很简单，但是相当灵活，允许创建与异构平台上非 JMS 应用所用格式匹配的消息。

JMS 消息可以有 3 个部分：一个首部、一些属性和一个体。只有首部是必要的。下面各节将介绍这些部分。

关于消息首部、属性和体的完整文档，参见 Message 接口的 API 文档。16.3.7.3 节给出了可能的消息类型列表。

16.3.7.1 消息首部

JMS 消息首部包含一些预定义字段，客户端和提供者会使用其中包含的值来标识和路由转发消息。表 16-1 列出并描述了 JMS 消息首部字段，并指示如何设置这些字段的值。例如，每个消息都有一个唯一的标识符，在首部字段 JMSMessageID 中表示。另一个首部字段 MSDestination 的值表示消息要发送到哪个队列或主题。另外还包括时间戳和优先级等字段。

每个首部字段都有一个关联的设置方法和获取方法，参见 Message 接口文档的描述。有些首部字段要由客户端来设置，不过很多首部字段都会由 send 方法自动设置，send 方法中的设置会覆盖客户端设置的值。

表 16-1 JMS 消息首部字段值如何设置

首部字段	描 述	由…设置
JMSDestination	消息发送到的目的地	JMS 提供者 send 方法
JMSDeliveryMode	发送消息时指定的传送模式（参见 16.4.2.1 节）	JMS 提供者 send 方法
JMSDeliveryTime	消息发送的时间以及发送消息时指定的传送延迟（参见 16.4.2.4 节）	JMS 提供者 send 方法
JMSExpiration	消息的到期时间（参见 16.4.2.3 节）	JMS 提供者 send 方法
JMSPriority	消息的优先级（参见 16.4.2.2 节）	JMS 提供者 send 方法
JMSMessageID	这个值唯一标识提供者发送的各个消息	JMS 提供者 send 方法
JMSTimestamp	消息由提供者发送的时间	JMS 提供者 send 方法
JMSCorrelationID	将一个消息链接到另一个消息的值；通常会使用 JMSMessageID 值关联两个消息	客户端应用
JMSReplyTo	发送消息应答的目的地	客户端应用
JMSType	客户端应用提供的类型标识符	客户端应用
JMSRedelivered	消息是否重新传送	JMS 提供者优先传送

16.3.7.2 消息属性

如果除了首部字段提供的值以外还需要其他值，可以创建和设置消息的属性。利用属性可以提供与其他消息传送系统的兼容性，或者使用属性来创建消息选择器（参见 16.3.6.2 节）。17.7 节给出了一个设置属性用作消息选择器的例子。

JMS API 提供了一些预定义的属性名，以 JMSX 开头。其中只有一个属性 JMSXDeliveryCount（指定消息已经传送的次数）要求 JMX 提供者必须实现；其余的都是可选的。应用中可以使用这些预定义属性或者用户定义的属性，也可以不使用。

16.3.7.3 消息体

JMS API 定义了 6 种不同类型的消息。每个消息类型对应一个不同的消息体。基于这些消息类型，允许发送和接收多种不同形式的数据。表 16-2 提供了这些消息类型的描述。

表 16-2 JMS 消息类型

消息类型	体包含
TextMessage	java.lang.String 对象（例如，一个 XML 文件的内容）
MapMessage	一组名 - 值对，名为 String 对象，值为 Java 编程语言中的基本类型。可以由枚举器顺序地访问各个实体，或者按名随机访问。实体的顺序未定义
BytesMessage	未解释的字节流。这个消息类型用于对消息体编码，从而与一个现有的消息格式匹配
StreamMessage	Java 编程语言中基本类型值的流，可以顺序填充和读取
ObjectMessage	Java 编程语言中的 Serializable 对象
Message	无内容。消息只由首部字段和属性组成。不需要消息体时，这个消息类型很有用

JMS API 提供了一些方法来创建各种不同类型的消息以及填充其内容。例如，要创建和发送一个 TextMessage，可以使用以下语句：

```
TextMessage message = context.createTextMessage();
message.setText(msg_text);      // msg_text is a String
context.createProducer().send(message);
```

在消费者一方，消息作为一个通用 Message 对象到达。可以将这个对象强制转换为适当的消息类型，并使用更特定的方法访问消息体和抽取消息内容（如果需要，也可以抽取其首部和属性）。例如，可以使用 BytesMessage 的面向流的读取方法。必须强制转换为适当的消息类型才能获取 StreamMessage 的消息体。

也可以不把消息强制转换为某个消息类型，而是在 Message 上调用 getBody 方法，并指定消息类型作为参数。例如，可以作为一个 String 来获取 TextMessage。下面的代码段使用了 getBody 方法：

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
    String message = m.getBody(String.class);
    System.out.println("Reading message: " + message);
} else {
    // Handle error or process another message type
}
```

JMS API 为创建和接收 TextMessage、BytesMessage、MapMessage 或 ObjectMessage 提供了快捷方式。例如，不用把字符串包装在一个 TextMessage 中；实际上，可以直接发送和接收字符串。例如，可以如下发送一个字符串：

```
String message = "This is a message";
context.createProducer().send(dest, message);
```

可以使用 receiveBody 方法接收这个消息：

```
String message = receiver.receiveBody(String.class);
```

可以使用 receiveBody 方法接收除 StreamMessage 和 Message 以外其他类型的消息，只要可以将消息体赋至这个特定的类型。

如果希望发送一个消息作为对应用发出的一个信号，则空 Message 可能很有用。第 17 章中的一些例子在发送一系列文本消息之后就会发送一个空消息。例如：

```
context.createProducer().send(dest, context.createMessage());
```

消费者代码会把非文本消息解释为一个信号，表示现在所有消息都已经接收。

第 17 章中的例子使用了 TextMessage、MapMessage 和 Message 类型的消息。

16.3.8 JMS 队列浏览器

发送到队列的消息会保留在队列中，直到这个队列的消息消费者使用了这些消息。JMS API 提供了一个 QueueBrowser 对象，允许浏览队列中的消息，并显示各个消息的首部值。可以使用 JMSContext.createBrowser 方法创建一个 QueueBrowser 对象。例如：

```
QueueBrowser browser = context.createBrowser(queue);
```

17.2.7 节中给出了一个使用 QueueBrowser 对象的例子。

创建 QueueBrowser 时，createBrowser 方法允许指定一个消息选择器作为第二个参数。有关消息选择器的更多信息，参见 16.3.6.2 节。

JMS API 没有提供浏览主题的机制。对于主题中的消息，通常消息一出现就会从主题消失：如果没有消息消费者使用消息，JMS 提供者会将其删除。对于持久订阅来说，尽管在没有活动的消息消费者时允许在主题上保留消息，但 JMS 没有定义相应的功能来检查这些消息。

16.3.9 JMS 异常处理

JMS API 中所有受查异常的根类是 JMSException。JMS API 中所有非受查异常的根子句是 JMSRuntimeException。

通过捕获 JMSException 和 JMSRuntimeException，这就提供了一种通用方法来处理与 JMS API 相关的所有异常。

JMSException 和 JMSRuntimeException 类包括以下子类，这些在 API 文档中有相应的介绍：

- IllegalStateException, IllegalStateRuntimeException
- InvalidClientIDException, InvalidClientIDRuntimeException
- InvalidDestinationException, InvalidDestinationRuntimeException
- InvalidSelectorException, InvalidSelectorRuntimeException
- JMSSecurityException, JMSSecurityRuntimeException
- MessageEOFException
- MessageFormatException, MessageFormatRuntimeException
- MessageNotReadableException
- MessageNotWriteableException, MessageNotWriteableRuntimeException
- ResourceAllocationException, ResourceAllocationRuntimeException
- TransactionInProgressException, TransactionInProgressRuntimeException
- TransactionRolledBackException, TransactionRolledBackRuntimeException

本教程中的所有例子都会在适当时捕获和处理 JMSException 或 JMSRuntime-Exception。

16.4 使用高级 JMS 特性

这一节解释如何使用 JMS API 的特性来得到应用所需的可靠性和性能。很多人之所以在应用中使用 JMS，只是因为他们无法容忍消息丢失或重复，他们要求每个消息接收一次而且仅一次。JMS API 提供了这个功能。

生成消息最可靠的方法是发送一个 PERSISTENT 消息，而且要在一个事务中发送消息。

JMS 消息默认为 PERSISTENT 消息；即使 JMS 提供者失败，PERSISTENT 消息也不会丢失。有关的详细信息，参见 16.4.2.1 节。

事务允许在一个原子操作中发送或接收多个消息。在 Java EE 平台中，还允许在一个原子事务中将消息发送和接收与数据库读写相结合。**事务**（transaction）是一个工作单元，可以将一系列操作（如消息发送和接收）分组到这样一个工作单元中，使得这些操作要么全部成功，要么全部失败。有关的详细信息，参见 16.4.4 节。

使用消息最可靠的方法是在事务中使用消息，可以使用一个队列的消息，或者从一个主题的持久订阅使用消息。有关的详细信息，参见以下小节：16.3.6.4 节、16.4.3 节、16.4.4 节。

还有一些特性主要用于支持应用提高性能。例如，可以设置消息在一定时间后到期（参见 16.4.2.3 节），使消费者不用接收不必要的过期信息。另外可以异步地发送消息，参见 16.4.5 节。

还可以对消息确认指定不同级别的控制；参见 16.4.1 节。

其他特性提供了另外一些有用的功能，这些功能与可靠性无关。例如，可以创建临时目的地，只在创建该目的地的连接期间有效。更多信息参见 16.4.3 节。

下面各小节将介绍如何在应用客户端或 Java SE 客户端应用这些特性。在 Java EE Web 或 EJB 容器中，有些特性的工作有所不同，如果存在这种情况，下面会指出相应的差别，另外还会在 16.5 节中详细解释。

16.4.1 控制消息确认

在 JMS 消息得到确认之前，并不认为它已经成功使用。要成功使用消息，通常分为 3 个阶段：

1. 客户端接收消息。
2. 客户端处理消息。
3. 消息得到确认。确认可能由 JMS 提供者发出，或者由客户端发出，这取决于会话确认模式。

在本地事务性会话中（参见 16.4.4 节），会话提交时消息就得到确认。如果一个事务回滚，则已使用的所有消息都会重新传送。

（Java EE Web 或 EJB 容器中）在 JTA 事务中，事务提交时消息会得到确认。

在非事务性会话中，消息何时确认以及如何确认取决于一个值，这个值可以作为 createContext 方法的一个参数来指定。可能的参数值如下：

- JMSContext.AUTO_ACKNOWLEDGE：这是应用客户端和 Java SE 客户端的默认设置。客户端成功地从一个 receive 调用返回时或者用来处理消息所调用的 MessageListener 成功返回时，JMSContext 会自动确认客户端接收到一个消息。

前面介绍过，消息使用是一个包括 3 个阶段的过程，不过如果一个 JMSContext 配置

为使用自动确认，则在这样一个 JMSContext 中，同步接收有所例外，对于同步接收，接收和确认一步完成，后面才是对消息的处理。

- JMSContext.CLIENT_ACKNOWLEDGE：客户端通过调用消息的 acknowledge 方法来确认一个消息。采用这种模式，确认在会话级发生：确认使用了一个消息，也就自动确认了会话所使用的所有消息都已经接收。例如，如果一个消息消费者使用了 10 个消息，然后对发送的第 5 个消息做出确认，那么所有消息都得到确认。

说明 在 Java EE 平台中，JMSContext.CLIENT_ACKNOWLEDGE 设置只能在应用客户端中使用，而不能用在 Web 组件或企业 bean 中。

- JMSContext.DUPS_OK_ACKNOWLEDGE：这个选项指示 JMSContext 采用懒惰方式确认消息的传送。如果 JMS 提供者失败，这很可能导致传送一些重复的消息，所以只有可以容忍重复消息的消费者才能使用这个模式。（如果 JMS 提供者传送一个消息，就必须将 JMSRedelivered 消息首部的值设置为 true。）这个选项可以尽量减少会话为避免重复所做的工作，从而降低会话开销。

如果已经从队列接收了消息，但是在 JMSContext 关闭时还没有确认，JMS 提供者就会保留这些消息，当消费者下一次访问这个队列时再重新传送这些消息。如果应用关闭了一个 JMSContext，而它已经使用了一个持久订阅的消息，则提供者也会保留未确认的消息。（参见 16.3.6.4 节。）从非持久订阅接收的未确认消息将在 JMSContext 关闭时删除。

如果使用一个队列或持久订阅，则可以用 JMSContext.recover 方法停止非事务性 JMSContext，再由第一个未确认的消息重启。这样一来，JMSContext 的已传送消息序列会重置到最后一个得到确认的消息之后的位置。如果有消息到期或者有更高优先级的消息到达，那么它现在传送的消息可能与原来传送的消息不同。对于一个非持久订阅的消费者，调用 JMSContext.recover 方法时，提供者可能会删除未确认的消息。

17.2.9 节中的示例程序展示了两种方法，可以确保消息处理完成后才会确认消息。

16.4.2 为发送消息指定选项

发送消息时可以设置很多选项。利用这些选项可以完成以下任务：

- 指定消息持久化，这意味着即使提供者失败，消息也不会丢失（16.4.2.1 节）。
- 设置消息的优先级，这会影响传送消息的顺序（16.4.2.2 节）。
- 为消息指定一个到期时间，保证如果消息过期就不再传送（16.4.2.3 节）。
- 为消息指定一个传送延迟，使消息在经过指定的一个时间之后才会传送（16.4.2.4 节）。

利用方法串链，可以在创建一个生产者并调用 send 方法时指定以上多个选项；参见 16.4.2.5 节。

16.4.2.1 指定消息持久化

JMS API 支持两种传送模式，可以指定如果 JMS 提供者失败消息是否丢失。这两个传送模式是 DeliveryMode 接口的字段。

- 默认传送模式为 PERSISTENT，指示 JMS 提供者要特别注意确保消息在传输过程中即使 JMS 提供者失败也不会丢失。采用这种传送模式发送消息，消息在发送时会记录到一个稳定的存储位置。
- NON_PERSISTENT 传送模式不要求 JMS 提供者存储消息，也不用保证提供者失败时消息不丢失。

要指定传送模式，可以使用 JMSProducer 接口的 setDeliveryMode 方法，为这个生产者发送的所有消息设置传送模式。

创建一个生产者时，可以使用方法串链来设置传送模式并发送消息。下面的调用创建了一个采用 NON_PERSISTENT 传送模式的生产者，并用这个生产者发送一个消息：

```
context.createProducer()
        .setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(dest, msg);
```

如果没有指定传送模式，则默认模式为 PERSISTENT。使用 NON_PERSISTENT 传送模式可以提高性能，减少存储开销，不过只有当应用能够容忍丢失消息时才能使用这个模式。

16.4.2.2 设置消息优先级

可以使用消息优先级指示 JMS 提供者先发送紧急的消息。使用 JMSProducer 接口的 setPriority 方法来为这个生产者所发送的所有消息设置优先级。

创建一个生产者时，可以使用方法串链设置优先级并发送消息。例如，下面的调用为一个生产者设置优先级 7，然后发送一个消息：

```
context.createProducer().setPriority(7).send(dest, msg);
```

10 级优先级为从 0（最低）到 9（最高）。如果没有指定优先级，则默认优先级为 4。JMS 提供者会尝试先传送高优先级消息，然后再传送低优先级消息，不过并不一定严格按优先级顺序传送消息。

16.4.2.3 允许消息到期

默认地，消息永远不会到期。不过，如果一个消息在指定的一段时间后过期，则可能会希望设置一个到期时间。使用 JMSProducer 接口的 setTimeToLive 方法来为这个生产者所发送的所有消息设置一个默认的到期时间。

例如，一个消息可能包含快速改变的数据，如股价，这个消息会在几分钟后过时，所以可以配置消息在这个时间之后到期。

创建一个生产者时，可以使用方法串链设置生存时间并发送消息。例如，以下调用为一个生产者设置生存时间为 5 分钟，然后发送一个消息：

```
context.createProducer().setTimeToLive(300000).send(dest, msg);
```

如果指定的 timeToLive 值为 0，消息就永远不会过期。

发送消息时，会将指定的 timeToLive 增加到当前时间来得到到期时间。在指定的到期时间之前未能传送的所有消息将被撤销。撤销过期的消息可以节省存储空间和计算资源。

16.4.2.4 指定传送延迟

可以指定一个消息发送之后并在 JMS 提供者传送这个消息之前所要经过的时间。可以使用 JMSProducer 接口的 setDeliveryDelay 方法为这个生产者所发送的所有消息设置一个传送延迟。

创建一个生产者时，可以使用方法串链设置传送延迟，并发送消息。例如，以下调用为一个生产者设置传送延迟为 3 秒，然后发送一个消息：

```
context.createProducer().setDeliveryDelay(3000).send(dest, msg);
```

16.4.2.5 使用 JMSProducer 方法串链

JMSProducer 接口的设置方法会返回 JMSProducer 对象，所以可以使用方法串链来创建一个生产者、设置多个属性，然后发送一个消息。例如，以下串链的方法调用会创建一个生产者，然后为消息设置一个用户自定义的属性，设置到期时间、传送模式和优先级，再将一个消息发送到一个队列：

```
context.createProducer()
        .setProperty("MyProperty", "MyValue")
        .setTimeToLive(10000)
        .setDeliveryMode(NON_PERSISTENT)
        .setPriority(2)
        .send(queue, body);
```

还可以调用 JMSProducer 方法在消息上设置属性，然后用一个单独的 send 方法调用发送这个消息。也可以直接在消息上设置消息属性。

16.4.3 创建临时目的地

正常情况下，一般会用管理方式创建 JMS 目的地（队列和主题），而不是通过编程来创建。JMS 提供者包含有一个工具，用来创建和删除目的地，通常这用于创建长时间保持的目的地。

JMS API 还允许创建临时目的地（TemporaryQueue 和 TemporaryTopic 对象），只是在创建这些目的地的连接期间保持。可以使用 JMSContext.createTemporaryQueue 和 JMSContext.createTemporaryTopic 方法动态地创建这些临时目的地，如下例所示：

```
TemporaryTopic replyTopic = context.createTemporaryTopic();
```

对于一个临时目的地，只有创建这个目的地的同一个连接所创建的消息消费者才可以使用这个目的地的消息。任何消息生产者都可以向临时目的地发送消息。如果关闭了创建一个临时目的地的连接，这个目的地也会关闭，它的内容会丢失。

可以使用临时目的地来实现一个简单的请求 / 应答机制。如果创建了一个临时目的地，在发送消息时指定这个临时目的地作为 JMSReplyTo 消息首部字段，则消息的消费者可以使用 JMSReplyTo 字段的值作为发送应答的目的地。消费者还可以将应答消息的 JMSCorrelationID 首部字段设置为请求的 JMSMessageID 首部字段值来引用原来的请求。例如，onMessage 方法可以创建一个 JMSContext，从而可以为它接收的消息发送应答。可以使用以下代码：

```
replyMsg = context.createTextMessage("Consumer processed message: "
        + msg.getText());
replyMsg.setJMSCorrelationID(msg.getJMSMessageID());
context.createProducer().send((Topic) msg.getJMSReplyTo(), replyMsg);
```

17.8 节给出了一个有关的例子。

16.4.4 使用 JMS 本地事务

事务（transaction）将一系列操作分组到一个原子工作单元中。如果其中任意一个操作失败，整个事务就会回滚，这些操作可能会从头重新执行。如果所有操作都成功，事务就可以提交。

在一个应用客户端或 Java SE 客户端中，可以使用本地事务对消息发送和接收分组。使用 JMSContext.commit 方法来提交事务。一个事务中可以发送多个消息，在事务提交之前，这些消息不会真正增加到队列或主题。如果在一个事务中接收多个消息，则事务提交之前它们不会得到确认。

可以使用 JMSContext.rollback 方法回滚一个事务。事务回滚意味着生成的所有消息都被撤销，所有已经使用的消息会恢复并重新传送，除非它们已经到期（参见 16.4.2.3 节）。

事务性会话总是包含在事务中。要创建一个事务性会话，可以调用 createContext 方法，如下所示：

```
JMSContext context =
        connectionFactory.createContext(JMSContext.SESSION_TRANSACTED);
```

一旦调用 commit 或 rollback 方法，便有一个事务结束，另一个事务开始。关闭一个事务性会话会回滚正在进行的事务，包括所有未完成的发送和接收。

对于在 Java EE Web 或 EJB 容器中运行的应用，不能使用本地事务。实际上，这些应用要使用 JTA 事务，见 16.5 节的介绍。

可以把多个发送和接收结合到一个 JMS 本地事务中，前提是它们都使用相同的 JMSContext 完成。

如果使用请求 / 应答机制，即发送一个消息，然后接收该消息的一个应答，就不要使用单个事务。如果试图使用单个事务，程序就会挂起，因为在事务提交之前发送并不会发生。下面的代码段可以说明这个问题：

```
// Don't do this!
outMsg.setJMSReplyTo(replyQueue);
context.createProducer().send(outQueue, outMsg);
consumer = context.createConsumer(replyQueue);
inMsg = consumer.receive();
context.commit();
```

由于事务中发送的消息实际上在事务提交之前并不会真正发送，所以事务中包含的所有接收操作都不能依赖于这个消息已经发送，即不能把已经发送这个消息作为其他接收操作的前提。

不能把一个消息的生成和使用同时包含在同一个事务中。原因在于，事务在客户端和JMS提供者之间发生，这会干预消息的生成和使用。图16-8展示了这种交互。

图16-8 使用JMS本地事务

客户端1向一个或多个目的地发送一个或多个消息，这可以构成一个事务，因为它使用一个JMSContext建立了与JMS提供者的一组交互。类似地，客户端2从一个或多个目的地接收一个或多个消息也构成一个事务，这里同样也使用了一个JMSContext。不过，由于这两个客户端没有直接交互，而且使用了两个不同的JMSContext对象，所以它们之间不会发生任何事务。

也可以这样来理解，事务是客户端和JMS提供者之间的一个契约，定义了一个消息是否发送到一个目的地，或者是否从目的地接收一个消息。这不是发送客户端和接收客户端之间的契约。

这也是消息传送与同步处理之间的根本差别。JMS并不是将消息的发送者和接收者紧耦合，而只是将消息发送者与目的地耦合，另外将目的地与消息接收者耦合。因此，尽管发送和接收分别与JMS提供者有一个紧耦合，但它们相互之间没有耦合。

创建一个JMSContext时，可以使用createContext方法的JMSContext.SESSION_TRANSACTED参数指定是否是事务性JMSContext。例如：

```
try (JMSContext context = connectionFactory.createContext(
        JMSContext.SESSION_TRANSACTED);) {
    ...
```

本地事务的commit和rollback方法与JMSContext底层的会话关联。如果使用同一个会话来完成操作，则可以将多个队列或主题（或者队列和主题的某种组合）上的操作结合在一个事务中。例如，可以使用同一个JMSContext接收一个队列的消息，并向相同事务中的一个主题发送消息。

17.3.2节中的例子展示了如何使用JMS本地事务。

16.4.5 异步发送消息

正常情况下，发送一个持久化消息时，send 方法会阻塞，直到 JMS 提供者确认这个消息已经成功发送。异步发送机制则允许应用发送一个消息，在等待确认发送是否完成的同时，可以继续完成自己的工作。

这个特性目前只在应用客户端或 Java SE 客户端中可用。

异步发送一个消息需要提供一个回调对象。可以用 onCompletion 方法指定一个 CompletionListener。例如，下面的代码会实例化一个名为 SendListener 的 CompletionListener。然后调用 setAsync 方法指定这个生产者的发送操作是异步的，并且要使用指定的监听器：

```
CompletionListener listener = new SendListener();
context.createProducer().setAsync(listener).send(dest, message);
```

CompletionListener 类必须实现两个方法：onCompletion 和 onException。如果发送成功，则调用 onCompletion 方法，如果发送失败则调用 onException 方法。下面给出这些方法的一个简单的实现：

```
@Override
public void onCompletion(Message message) {
    System.out.println("onCompletion method: Send has completed.");
}

@Override
public void onException(Message message, Exception e) {
    System.out.println("onException method: send failed: " + e.toString());
    System.out.println("Unsent message is: \n" + message);
}
```

16.5 在 Java EE 应用中使用 JMS API

这一节介绍在企业 bean 应用或 Web 应用中使用 JMS API 与在应用客户端中使用有什么不同。

Java EE 平台规范中的一般规则适用于 EJB 或 Web 容器中使用 JMS API 的所有 Java EE 组件：Web 和 EJB 容器中的应用组件不能在一个连接上创建多个活动的（未关闭的）Session 对象。不过，多个 JMSContext 对象是允许的，因为 JMSContext 对象结合了一个连接和一个会话。

这个规则不适用应用客户端。应用客户端容器支持为每个连接创建多个会话。

16.5.1 为 Java EE 应用创建资源

可以使用注解为 Java EE 企业 bean 或 Web 组件创建应用特定的连接工厂和目的地。以这种方式创建的资源只对创建这些资源的应用可见。

还可以使用部署描述文件元素来创建这些资源。部署描述文件中指定的元素会覆盖注解中指定的元素。参见本书第1卷5.1节，了解有关部署描述文件的基本信息。必须使用部署描述文件为应用客户端创建应用特定的资源。

要创建一个目的地，可以在类上使用一个@JMSDestinationDefinition注解，如下所示：

```
@JMSDestinationDefinition(
    name = "java:app/jms/myappTopic",
    interfaceName = "javax.jms.Topic",
    destinationName = "MyPhysicalAppTopic"
  )
```

Name、interfaceName和destinationName元素是必要的元素。还可以指定一个description元素，这是可选的。要创建多个目的地，需要将它们包含在一个@JMSDestinationDefinitions注解中，用逗号分隔。

要创建一个连接工厂，可以在类上使用一个@JMSConnectionFactoryDefinition注解，如下所示：

```
@JMSConnectionFactoryDefinition(
    name="java:app/jms/MyConnectionFactory"
)
```

name元素是必要的。还可以指定其他一些元素（这些元素可选）。例如，如果想使用连接工厂建立持久订阅，可以指定clientId，或者还可以指定description。如果没有指定interfaceName元素，则默认接口为javax.jms.ConnectionFactory。要创建多个连接工厂，可以把它们包围在一个@JMSConnectionFactoryDefinitions注解中，用逗号分隔。

对于一个给定的应用，这个注解只需要指定一次，可以在任意组件中指定。

说明 如果应用中包含一个或多个消息驱动bean，则可能希望把注解放在某个消息驱动bean中。如果在一个发送组件（如一个应用客户端）上增加这个注解，则需要指定mappedName元素来查找主题，而不是使用激活配置规范的destinationLookup属性。

向一个组件注入这个资源时，要使用定义注解（@JMSConnectionFactory-Definition）中name元素的值作为@Resource注解中lookup元素的值：

```
@Resource(lookup = "java:app/jms/myappTopic")
private Topic topic;
```

可以使用以下可移植的JNDI命名空间。具体使用哪一个命名空间取决于你的应用如何打包。

- java:global：使资源对所有已部署的应用都可用。
- java:app：使资源对一个应用中所有模块中的所有组件都可用。
- java:module：使资源对一个给定模块中的所有组件（例如，一个 EJB 模块中的所有企业 bean）都可用。
- java:comp：使资源只对一个组件可用（Web 应用除外。在 Web 应用中，java:comp 等价于 java:module）。

关于这些注解的详细信息参见 API 文档。17.5 节、17.7 节和 17.8 节中的例子都使用了 @JMSDestinationDefinition 注解。其他 JMS 例子没有使用这些注解。只包含应用客户端的示例并不部署在应用服务器上，所以必须使用以管理方式创建的资源相互通信，这些资源会放在各个应用之外。

16.5.2 在企业 bean 或 Web 组件中使用资源注入

Java EE 应用中，可以使用资源注入来注入受管理对象和 JMSContext 对象。

16.5.2.1 注入 ConnectionFactory、Queue 或 Topic

通常情况下，会使用 @Resource 注解向 Java EE 应用注入 ConnectionFactory、Queue 或 Topic。这些对象必须在部署应用之前通过管理方式创建。你可能希望使用默认连接工厂，其 JNDI 名为 java:comp/DefaultJMSConnectionFactory。

在一个应用客户端组件中使用资源注入时，通常要将这个 JMS 资源声明为静态：

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(lookup = "jms/MyQueue")
private static Queue queue;
```

不过，在会话 bean、消息驱动 bean 或 Web 组件中使用这个注解时，不要将资源声明为静态：

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private ConnectionFactory connectionFactory;

@Resource(lookup = "jms/MyTopic")
private Topic topic;
```

如果在这些组件中将资源声明为静态，则会导致运行时错误。

16.5.2.2 注入一个 JMSContext 对象

要在一个企业 bean 或 Web 组件中访问 JMSContext 对象，不用注入 ConnectionFactory 资源再创建 JMSContext，可以使用 @Inject 和 @JMSConnectionFactory 注解直接注入一个 JMSContext。要使用默认的连接工厂，可以使用类似下面的代码：

```
@Inject
private JMSContext context1;
```

要使用你自己的连接工厂，可以使用类似下面的代码：

```
@Inject
@JMSConnectionFactory("jms/MyConnectionFactory")
private JMSContext context2;
```

16.5.3 使用 Java EE 组件生成和同步接收消息

生成消息或同步接收消息的应用可以使用一个 Java EE Web 或 EJB 组件来完成这些操作，如托管 bean、servlet 或会话 bean。17.7 节中的例子使用了一个无状态会话 bean 向一个主题发送消息。17.5 节中的例子使用托管 bean 来生成和使用消息。

如果同步接收没有指定超时时间，将占用服务器资源，因此这种机制通常不是 Web 或 EJB 组件的最佳应用设计。实际上，可以使用一个指定了超时值的同步接收，或者使用一个消息驱动 bean 来异步地接收消息。有关同步接收的详细信息，参见 16.3.6 节。

在 Java EE 组件中使用 JMS API 时，很多方面与在应用客户端中使用 JMS API 很类似。主要区别在于资源管理和事务。

16.5.3.1 在 Web 和 EJB 组件中管理 JMS 资源

JMS 资源包括一个连接和一个会话，通常结合在一个 `JMSContext` 对象中。一般地，不再使用一个 JMS 资源时，一定要将其释放，这很重要。下面给出一些很有用的实践做法：

- 如果只想在一个业务方法的生命期期间维护一个 JMS 资源，可以使用一个 `try-with-resources` 语句来创建 `JMSContext`，使它在 `try` 块结束时自动关闭。
- 要在一个事务或请求期间维护一个 JMS 资源，可以如 16.5.2.2 节所述注入 `JMSContext`。这样一来，不再使用资源时也会将它释放。
- 如果想在一个企业 bean 实例的生命期期间维护一个 JMS 资源，可以使用 `@PostConstruct` 回调方法来创建资源，并使用 `@PreDestroy` 回调方法关闭资源。不过，通常没有必要这样做，因为应用服务器通常会维护一个连接池。如果使用有状态会话 bean，而且希望维护缓存状态中的 JMS 资源，就必须在 `@PrePassivate` 回调方法中关闭这个资源，并将它的值设置为 `null`，而且必须在 `@PostActivate` 回调方法中再次创建这个资源。

16.5.3.2 在会话 bean 中管理事务

可以不使用本地事务，而是使用 JTA 事务。容器托管事务或 bean 托管事务都可以使用。正常情况下，对于完成发送或接收的 bean 方法，可以使用容器托管事务，这样就会让 EJB 容器处理事务界定。由于容器托管事务是默认的，所以不要求显式指定。

也可以使用 bean 托管事务和 `javax.transaction.UserTransaction` 接口的事务界定方法，不过只有在应用有特殊需求时，而且你在使用事务方面是行家，才应当这么做。

一般情况下，容器托管事务就能提供最高效、最正确的行为。本教程不打算提供 bean 托管事务的例子。

16.5.4 使用消息驱动 bean 异步接收消息

3.3 节和 16.1.4 节介绍了 Java EE 平台支持一种特殊的企业 bean，即消息驱动 bean，它允许 Java EE 应用异步地处理 JMS 消息。其他 Java EE Web 和 EJB 组件只允许发送消息并同步接收，而不能异步接收。

消息驱动 bean 是一个消息监听器，可以从一个队列或主题将消息发送到这个监听器。消息可以由任意 Java EE 组件（应用客户端、另一个企业 bean 或一个 Web 组件）发送，也可以从一个没有使用 Java EE 技术的应用或系统发送。

消息驱动 bean 类有以下需求：

- 如果没有使用部署描述文件，则必须有 `@MessageDriven` 注解。
- 这个类必须定义为 `public`，而不能定义为 `abstract` 或 `final`。
- 必须包含一个无参数的公共构造函数。

建议（但不要求）消息驱动 bean 类为它支持的消息类型实现消息监听器接口。支持 JMS API 的 bean 实现 `javax.jms.MessageListener` 接口，这说明它必须提供一个 `onMessage` 方法，方法签名如下：

```
void onMessage(Message inMessage)
```

消息到达目标 bean 时，会由 bean 的容器调用 `onMessage` 方法。这个方法包含处理消息的业务逻辑。将由消息驱动 bean 负责解析消息，并完成必要的业务逻辑。

消息驱动 bean 与应用客户端的消息监听器有以下区别：

- 在应用客户端中，必须创建一个 `JMSContext`，然后创建 `JMSConsumer`，再调用 `setMessageListener` 来激活监听器。对于消息驱动 bean，只需要定义类并为类增加注解，EJB 容器会为你创建监听器。
- bean 类使用 `@MessageDriven` 注解，这通常包含一个 `activationConfig` 元素，其中包含 `@ActivationConfigProperty` 注解来指定 bean 或连接工厂使用的属性。这些属性可能包括连接工厂、目的地类型、持久订阅、消息选择器或确认模式。第 17 章中的一些例子设置了这些属性。也可以在部署描述文件中设置这些属性。
- 应用客户端容器只有一个 `MessageListener` 实例，一次只在一个线程上调用。不过，消息驱动 bean 可以包括多个实例，由容器配置，它们可能由多个线程并发调用（不过每个实例一次只能由一个线程调用）。因此消息驱动 bean 可以比消息监听器更快地处理消息。
- 除非使用 bean 托管事务，否则不需要指定消息确认模式。消息会在调用 `onMessage` 方法的事务中使用。

表 16-3 列出了 JMS 规范定义的激活配置属性。

表 16-3 消息驱动 bean 的 @ActivationConfigProperty 设置

属性名	描 述
acknowledgeMode	确认模式，仅用于 bean 托管事务；默认为 Auto-acknowledge（也允许 Dups-ok-acknowledge）
destinationLookup	队列或主题的查找名，bean 将从这个队列或主题接收消息
destinationType	可以是 javax.jms.Queue 或 javax.jms.Topic
subscriptionDurability	对于持久订阅，设置值为 Durable；有关的更多信息参见 16.3.6.4 节
clientId	对于持久订阅，指定连接的客户端 ID（可选）
subscriptionName	对于持久订阅，指定订阅名
messageSelector	过滤消息的字符串；有关的详细信息参见 16.3.6.2 节
connectionFactoryLookup	用来连接 JMS 提供者的连接工厂查找名，bean 将从这个 JMS 提供者接收消息

例如，下面是 17.6 节中使用的消息驱动 bean：

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "jms/MyQueue"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "javax.jms.Queue")
})
public class SimpleMessageBean implements MessageListener {

    @Resource
    private MessageDrivenContext mdc;
    static final Logger logger = Logger.getLogger("SimpleMessageBean");

    public SimpleMessageBean() {
    }

    @Override
    public void onMessage(Message inMessage) {

        try {
            if (inMessage instanceof TextMessage) {
                logger.log(Level.INFO,
                        "MESSAGE BEAN: Message received: {0}",
                        inMessage.getBody(String.class));
            } else {
                logger.log(Level.WARNING,
                        "Message of wrong type: {0}",
                        inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            logger.log(Level.SEVERE,
                    "SimpleMessageBean.onMessage: JMSException: {0}",
```

```
                e.toString());
            mdc.setRollbackOnly();
        }
    }
}
```

如果 JMS 使用一个资源适配器与应用服务器集成，则 JMS 资源适配器会为 EJB 容器处理这些任务。

bean 类通常会注入一个 MessageDrivenContext 资源，它还提供了另外一些完成事务管理的方法（例如，setRollbackOnly）：

```
@Resource
private MessageDrivenContext mdc;
```

消息驱动 bean 没有本地或远程接口。实际上，它只有一个 bean 类。

消息驱动 bean 在某些方面与无状态会话 bean 很类似：消息驱动 bean 实例生存期相当短，而且不会为特定客户端保留状态。消息驱动 bean 实例的实例变量可能包含一些状态，这些状态会在处理不同客户端消息期间持续保留：例如，一个打开的数据库连接，或一个企业 bean 对象的对象引用。

类似于无状态会话 bean，消息驱动 bean 可以有多个可互换的实例在同时运行。容器可以将这些实例入池，从而允许并发地处理消息流。在不影响消息处理并发性的前提下，容器会尝试按时间顺序传送消息，不过对于具体以什么顺序向消息驱动 bean 类的实例传送消息则没有保证。如果消息顺序对于你的应用很重要，则可能需要将应用服务器配置为只使用消息驱动 bean 的一个实例。

关于消息驱动 bean 生命周期的详细信息，参见 3.7.4 节。

16.5.5 管理 JTA 事务

Java EE 应用客户端和 Java SE 客户端使用 JMS 本地事务（见 16.4.4 节中的介绍），允许将发送和接收分组在一个特定的 JMS 会话中。在 Web 或 EJB 容器中运行的 Java EE 应用通常使用 JTA 事务来确保对外部资源访问的完整性。JTA 事务和 JMS 本地事务之间的关键区别在于 JTA 事务由应用服务器的事务管理器来控制。JTA 事务可以是**分布式的**，这意味着在同一个事务中可以包含多个资源，如 JMS 提供者和数据库。

例如，分布式事务允许多个应用对同一个数据库完成原子更新，也允许一个应用对多个数据库完成原子更新。

在一个使用 JMS API 的 Java EE 应用中，可以使用事务将消息发送或接收与数据库更新以及其他资源管理器操作相结合。可以在一个事务中从多个应用组件访问资源。例如，servlet 可以开始一个事务，访问多个数据库，调用一个发送 JMS 消息的企业 bean，再调用另一个企业 bean（这个企业 bean 使用连接器架构修改一个 EIS 系统），最后提交这个事务。不过，应用不能在同一个事务中既发送一个 JMS 消息又接收对它的应答。

EJB 和 Web 容器中的 JTA 事务可以是以下两种类型：

- **容器托管事务**：容器会控制事务的完整性，而不需要调用 commit 或 rollback.。容器托管事务比 bean 托管事务更易于使用。可以为企业 bean 方法指定适当的事务属性。

使用 Required 事务属性（默认）可以确保一个方法总是某个事务的一部分。如果调用这个方法时事务正在进行中，这个方法就会成为那个事务的一部分；如果没有正在进行的事务，那么在调用这个方法之前会开始一个新事务，这个事务会在这个方法返回时提交。更多信息参见 22.3.1 节。

- **bean 托管事务**：可以与 javax.transaction.UserTransaction 接口结合使用，这个接口提供了自己的 commit 和 rollback 方法，可以用来界定事务边界。建议只有对事务编程很有经验的人才可以使用 bean 托管事务。

对于消息驱动 bean，可以使用容器托管事务，也可以使用 bean 托管事务。为了确保所有消息都在一个事务的上下文中接收和处理，应当使用容器托管事务，并对 onMessage 方法使用 Required 事务属性（默认）。

使用容器托管事务时，可以调用以下 MessageDrivenContext 方法。

- setRollbackOnly：使用这个方法来完成错误处理。如果出现一个异常，则 setRollbackOnly 会相应地标志当前事务，使得这个事务只能回滚。
- getRollbackOnly：使用这个方法检查当前事务是否标志为回滚。

如果使用 bean 托管事务，则向 onMessage 方法传送消息发生在 JTA 事务上下文之外。在 onMessage 方法中调用 UserTransaction.begin 方法时，事务才开始，调用 UserTransaction.commit 或 UserTransaction.rollback 时，事务结束。所有 Connection.createSession 方法调用都必须在事务中发生。

使用 bean 托管事务允许通过使用多个事务来处理消息，或者可以让消息处理的某些部分在一个事务上下文之外发生。不过，如果使用容器托管事务，则消息由 MDB 接收，并由同一个事务中的 onMessage 方法处理。利用 bean 托管事务则无法做到这一点。

对于 Web 或 EJB 容器，在一个 JTA 事务中创建一个 JMSContext 时，容器会忽略你指定的所有参数，因为容器会处理所有事务性属性。在 Web 或 EJB 容器中创建一个 JMSContext 时，如果没有 JTA 事务，则传入 createContext 方法的值（如果有）应当是 JMSContext.AUTO_ACKNOWLEDGE 或 JMSContext.DUPS_OK_ACKNOWLEDGE。

使用容器托管事务时，通常对企业 bean 的业务方法使用 Required 事务属性（默认）。创建一个使用容器托管事务的消息驱动 bean 时不用指定激活配置属性 acknowledgeMode。容器提交事务时会自动确认消息。

如果一个消息驱动 bean 使用 bean 托管事务，则消息的接收不能作为 bean 托管事务的一部分。可以将激活配置属性 acknowledgeMode 设置为 Auto-acknowledge 或 Dups-ok-acknowledge，来指定你希望如何确认消息驱动 bean 接收的消息。

如果 onMessage 方法抛出一个 RuntimeException，则容器不会确认处理消息。在这种情况下，JMS 提供者将来会重新传送未确认的消息。

16.6 关于 JMS 的更多信息

有关 JMS 的更多信息，参见：

- Java 消息服务网站：

  http://www.oracle.com/technetwork/java/index-jsp-142945.html

- Java 消息服务规范 2.0 版：

  http://jcp.org/en/jsr/detail?id=343

Chapter 17

第 17 章

Java 消息服务示例

本章会给出一些例子，展示如何在各种不同类型的 Java EE 应用中使用 JMS API，包括以下内容：

- JMS 示例概述
- 编写简单的 JMS 应用
- 编写更高级的 JMS 应用
- 编写高性能、可扩展的 JMS 应用
- 使用一个简单的 Web 应用发送和接收消息
- 使用消息驱动 bean 异步接收消息
- 从会话 bean 向 MDB 发送消息
- 使用实体联接两个 MDB 的消息
- 使用 NetBeans IDE 创建 JMS 资源

这些例子都放在 *tut-install*/`examples/jms/` 目录中。

构建和运行各个示例的步骤如下：

1. 使用 NetBeans IDE 或 Maven 编译、打包以及（有些情况下）部署示例。

2. 使用 NetBeans IDE、Maven 或 `appclient` 命令运行应用客户端，或者使用浏览器运行 Web 应用示例。

部署或运行示例之前，需要为它们创建资源。有些示例有一个 `glassfish-resources.xml` 文件，用来为该示例以及其他示例创建资源。可以使用 `asadmin` 命令来创建资源。

要使用 `asadmin` 和 `appclient` 命令，需要在命令路径中加入 GlassFish 服务器的 `bin` 目录，参见 2.1.1.1 节。

17.1 JMS 示例概述

下表列出了本章中使用的示例，描述了各个示例做什么，并给出了详细介绍示例的具体章节链接。各个示例的目录都是 *tut-install*/examples/jms/ 目录下的相对目录。

表 17-1 展示如何使用 Java EE 应用客户端的 JMS 示例

示例目录	描 述
simple/producer	使用一个应用客户端发送消息；参见 17.2.4 节
simple/synchconsumer	使用一个应用客户端同步接收消息；参见 17.2.5 节
simple/asynchconsumer	使用一个应用客户端异步接收消息；参见 17.2.6 节
simple/messagebrowser	使用一个应用客户端用 QueueBrowser 浏览一个队列；参见 17.2.7 节
simple/clientackconsumer	使用一个应用客户端同步确认接收的消息；参见 17.2.9 节
durablesubscriptionexample	使用一个应用客户端在主题上创建一个持久订阅；参见 17.3.1 节
transactedexample	使用一个应用客户端在本地事务中发送和接收消息（也使用请求 - 应答消息传送）；参见 17.3.2 节
shared/sharedconsumer	使用一个应用客户端创建共享非持久主题订阅；参见 17.4.1 节
shared/shareddurableconsumer	使用一个应用客户端创建共享持久主题订阅；参见 17.4.2 节

表 17-2 展示如何使用 Java EE Web 和 EJB 组件的 JMS 示例

示例目录	描 述
websimplemessage	使用托管 bean 发送消息，并同步接收消息；参见 17.5 节
simplemessage	使用一个应用客户端发送消息，并使用一个消息驱动 bean 异步接收消息；参见 17.6 节
clientsessionmdb	使用一个会话 bean 发送消息，并使用一个消息驱动 bean 接收消息；参见 17.7 节
clientmdbentity	使用一个应用客户端、两个消息驱动 bean 和 JPA 持久化来创建一个简单的人力资源（HR）应用；参见 17.8 节

17.2 编写简单的 JMS 应用

这一节介绍如何创建、打包和运行简单的 JMS 客户端（打包为应用客户端）。这些客户端应用展示了 JMS 应用必须完成的基本任务：

- 创建一个 JMSContext
- 创建消息生产者和消费者
- 发送和接收消息

每个示例使用两个客户端：一个发送消息，另一个接收消息。可以在两个终端窗口中运行这些客户端。

编写在企业 bean 应用中运行的 JMS 客户端时，与应用客户端类似，可以按相同的顺序使用一些同样的方法。不过，也有一些重要的差别。16.5 节介绍了这些差别，本章还会提供一些例子来具体说明。

这一节的例子位于 *tut-install*/examples/jms/simple/ 目录，分别放在以下子目录中：

```
producer/
synchconsumer/
asynchconsumer/
messagebrowser/
clientackconsumer/
```

运行示例前，需要启动 GlassFish 服务器，并创建受管理对象。

17.2.1 启动 JMS 提供者

使用 GlassFish 服务器时，JMS 提供者就是 GlassFish 服务器。按 2.2 节的介绍启动这个服务器。

17.2.2 创建 JMS 受管理对象

这个例子使用以下 JMS 受管理对象：

- 一个连接工厂
- 两个目的地资源：一个主题和一个队列

运行应用前，可以使用 asadmin add-resources 命令创建必要的 JMS 资源，指定文件 glassfish-resources.xml 作为参数。可以在工程中使用 NetBeans IDE 创建这个文件，不过也可以手动地自行创建。所需资源的相应文件位于 jms/simple/producer/src/main/setup/ 目录。

这个 JMS 示例使用了一个连接工厂，其逻辑 JNDI 查找名为 java:comp/DefaultJMSConnectionFactory，这是 GlassFish 服务器中预配置的一个连接工厂。

还可以使用 asadmin create-jms-resource 命令创建资源，使用 asadmin list-jms-resources 命令显示资源名，以及使用 asadmin delete-jms-resource 命令删除资源。

17.2.2.1 为简单示例创建资源

一个 Maven 工程中的 glassfish-resources.xml 文件可以为这些简单示例创建所需的全部资源。

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 在一个命令窗口中，进入 Producer 示例。

```
cd tut-install/jms/simple/producer
```

3. 使用 asadmin add-resources 命令创建资源：

```
asadmin add-resources src/main/setup/glassfish-resources.xml
```

4. 验证资源是否已经创建：

```
asadmin list-jms-resources
```

这个命令会列出 glassfish-resources.xml 文件中指定的两个目的地和连接工厂(平台默认连接工厂):

```
jms/MyQueue
jms/MyTopic
jms/__defaultConnectionFactory
Command list-jms-resources executed successfully.
```

在 GlassFish 服务器中，Java EE java:comp/DefaultJMSConnectionFactory 资源映射到一个名为 jms/__defaultConnectionFactory 的连接工厂。

17.2.3 构建所有简单示例

要使用 GlassFish 服务器运行这些简单示例，需要把各个示例打包在一个应用客户端 JAR 文件中。应用客户端 JAR 文件需要一个清单文件，清单文件与相应的 .class 文件位于各个示例的 src/main/java/META-INF/ 目录中。

每个示例的 pom.xml 文件指定了一个插件，用来创建应用客户端 JAR 文件。可以使用 NetBeans IDE 或 Maven 来构建这些示例。

17.2.3.1 使用 NetBeans IDE 构建所有简单示例

1. 从 File（文件）菜单选择 Open Project（打开工程）。

2. 在 Open Project 对话框中，导航至：

```
tut-install/examples/jms
```

3. 展开 jms 节点并选择 simple 文件夹。

4. 单击 Open Project，打开所有简单示例。

5. 在 Projects（工程）标签页中，右键单击 simple 工程并选择 Build（构建）来构建所有示例。

这个命令将应用客户端 JAR 文件放在示例的 target 目录。

17.2.3.2 使用 Maven 构建所有简单示例

1. 在一个终端窗口中，进入 simple 目录：

```
cd tut-install/jms/simple
```

2. 输入以下命令来构建所有工程：

```
mvn install
```

这个命令将应用客户端 JAR 文件放在示例的 target 目录。

17.2.4 发送消息

这一节介绍如何使用一个客户端发送消息。在所有这些例子中，将由 Producer.java 客户端发送消息。

示例的一般步骤如下：

1. 为示例使用的受管理对象注入资源。

2. 接受并验证命令行参数。可以使用这个示例向一个队列或主题发送任意多个消息，所以运行这个程序时，要在命令行上指定目的地类型以及消息数目。

3. 创建一个 JMSContext，然后以字符串形式发送指定数目的文本消息，参见 16.3.7.3 节的介绍。

4. 发送最后一个 Message 类型的消息，指示消费者不再有更多消息。

5. 捕获可能的异常。

17.2.4.1 Producer.java 客户端

发送客户端 Producer.java 完成以下步骤：

1. 为连接工厂、队列和主题注入资源：

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(lookup = "jms/MyQueue")
private static Queue queue;
@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

2. 获取和验证指定目的地类型和消息个数的命令行参数：

```
final int NUM_MSGS;
String destType = args[0];
System.out.println("Destination type is " + destType);
if ( ! ( destType.equals("queue") || destType.equals("topic") ) ) {
    System.err.println("Argument must be \"queue\" or " + "\"topic\"");
    System.exit(1);
}
if (args.length == 2){
    NUM_MSGS = (new Integer(args[1])).intValue();
} else {
    NUM_MSGS = 1;
}
```

3. 根据指定的目的地类型，将队列或主题赋至一个目的地对象：

```
Destination dest = null;
try {
    if (destType.equals("queue")) {
        dest = (Destination) queue;
    } else {
        dest = (Destination) topic;
    }
} catch (Exception e) {
    System.err.println("Error setting destination: " + e.toString());
    System.exit(1);
}
```

4. 在一个 try-with-resources 块中创建一个 JMSContext：

```
try (JMSContext context = connectionFactory.createContext();) {
```

5. 将消息数设置为 0，然后创建一个 JMSProducer，向目的地发送一个或多个消息，并增加消息数。字符串形式的消息使用 TextMessage 消息类型：

```
int count = 0;
for (int i = 0; i < NUM_MSGS; i++) {
    String message = "This is message " + (i + 1)
            + " from producer";
    // Comment out the following line to send many messages
    System.out.println("Sending message: " + message);
    context.createProducer().send(dest, message);
    count += 1;
}
System.out.println("Text messages sent: " + count);
```

6. 发送一个空控制消息，指示消息流结束：

```
context.createProducer().send(dest, context.createMessage());
```

发送一个未指定类型的空消息是一种便利方法，应用可以采用这种方法告诉消费者最后一个消息已经到达。

7. 捕获并处理可能的异常。try-with-resources 块结束时，JMSContext 会自动关闭：

```
} catch (Exception e) {
    System.err.println("Exception occurred: " + e.toString());
    System.exit(1);
}
System.exit(0);
```

17.2.4.2 运行 Producer 客户端

可以使用 appclient 命令运行客户端。Producer 客户端取一个或两个命令行参数：需要指定一个目的地类型，另外还可以有一个消息数（可选）。如果没有指定消息数，客户端就会发送一个消息。

这里将使用客户端向一个队列发送 3 个消息。

1. 确保已启动 GlassFish 服务器（参见 2.2 节），而且已经创建资源并构建了简单 JMS 示例（参见 17.2.2 节和 17.2.3 节）。

2. 在一个终端窗口中，进入 producer 目录：

```
cd producer
```

3. 运行 Producer 程序，向队列发送 3 个消息：

```
appclient -client target/producer.jar queue 3
```

这个程序的输出如下（还包括一些额外的输出）：

```
Destination type is queue
Sending message: This is message 1 from producer
```

```
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```

消息现在已经在队列中，等待接收。

说明 运行一个应用客户端时，命令可能需要很长时间才能完成。

17.2.5 同步接收消息

这一节介绍接收客户端，它使用 receive 方法同步地使用消息。然后本节将解释如何使用 GlassFish 服务器运行这些客户端。

17.2.5.1 SynchConsumer.java 客户端

接收客户端 SynchConsumer.java 完成以下步骤：

1. 为连接工厂、队列和主题注入资源。

2. 根据指定的目的地类型，将队列或主题赋至一个目的地对象。

3. 在一个 try-with-resources 块中，创建一个 JMSContext。

4. 创建一个 JMSConsumer，开始消息传送：

```
consumer = context.createConsumer(dest);
```

5. 接收发送到目的地的消息，直到接收到消息流末尾（end-of-message-stream）控制消息：

```
int count = 0;
while (true) {
    Message m = consumer.receive(1000);
    if (m != null) {
        if (m instanceof TextMessage) {
            System.out.println(
                    "Reading message: " + m.getBody(String.class));
            count += 1;
        } else {
            break;
        }
    }
}
System.out.println("Messages received: " + count);
```

由于控制消息不是一个 TextMessage，因此接收客户端会终止 while 循环，在这个控制消息到达后将停止接收消息。

6. 捕获并处理可能的异常。try-with-resources 块结束时会使 JMSContext 自动关闭。

SynchConsumer 客户端使用一个无限 while 循环接收消息，调用 receive 时指定了一个超时参数。

17.2.5.2 运行 SynchConsumer 和 Producer 客户端

可以使用 appclient 命令运行这些客户端。SynchConsumer 客户端取一个命令行参数，即目的地类型。

这些步骤展示了如何使用一个队列和主题同步接收和发送消息。这些步骤假设你已经运行 Producer 客户端，而且已经有 3 个消息在队列中等待。

1. 在运行 Producer 的同一个终端窗口中，进入 synchconsumer 目录：

```
cd ../synchconsumer
```

2. 运行 SynchConsumer 客户端，指定队列：

```
appclient -client target/synchconsumer.jar queue
```

这个客户端的输出如下（还包括一些额外的输出）：

```
Destination type is queue
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Messages received: 3
```

3. 现在试着以相反的顺序运行这些客户端。先运行 SynchConsumer 客户端：

```
appclient -client target/synchconsumer.jar queue
```

这个客户端会显示目的地类型，然后等待消息。

4. 打开一个新的终端窗口，并运行 Producer 客户端：

```
cd tut-install/jms/simple/producer
appclient -client target/producer.jar queue 3
```

发送消息之后，SynchConsumer 客户端接收这些消息，并退出。

5. 现在使用一个主题而不是队列来运行 Producer 客户端：

```
appclient -client target/producer.jar topic 3
```

这个客户端的输出如下（还包括一些额外的输出）：

```
Destination type is topic
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```

6. 再在另一个终端窗口中使用主题运行 SynchConsumer 客户端：

```
appclient -client target/synchconsumer.jar topic
```

不过，结果有所不同。因为你在使用一个主题上的订阅，所以在主题上创建订阅之前所发送的消息不会增加到这个订阅，也不会传送给消费者。（有关的详细内容，参见 16.2.2.2 节和 16.3.6.3 节。）客户端不会接收消息，而是等待消息到达。

7. 让 SynchConsumer 客户端继续运行，并运行 Producer 客户端：

```
appclient -client target/producer.jar topic 3
```

现在 SynchConsumer 客户端接收到消息：

```
Destination type is topic
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Messages received: 3
```

由于这些消息是消费者启动之后发送的，所以客户端可以接收这些消息。

17.2.6 使用消息监听器完成异步消息传送

这一节介绍一个示例中的接收客户端，这个示例使用一个消息监听器来完成异步消息传送。然后会解释如何使用 GlassFish 服务器编译和运行这些客户端。

说明 在 Java EE 平台中，消息监听器只能在应用客户端中使用，如本例所示。要在一个 Web 或企业 bean 应用中支持异步消息传送，需要使用消息驱动 bean，见本章后面的例子。

17.2.6.1 编写 AsynchConsumer.java 和 TextListener.java 客户端

发送客户端是 Producer.java，17.2.4 节和 17.2.5 节中也同样使用了这个发送客户端。

正常情况下，异步消费者会无限运行下去。这个客户端会一直运行，直到用户键入字符 q 或 Q 来停止客户端。

客户端 AsynchConsumer.java 完成以下步骤：

1. 为连接工厂、队列和主题注入资源。
2. 根据指定的目的地类型，将队列或主题赋至一个目的地对象。
3. 在一个 try-with-resources 块中，创建一个 JMSContext。
4. 创建一个 JMSConsumer。
5. 创建一个 TextListener 类实例，并注册为 JMSConsumer 的消息监听器：

```
listener = new TextListener();
consumer.setMessageListener(listener);
```

6. 监听发送到目的地的消息，用户键入字符 q 或 Q 时停止（为此使用了一个 java.io.InputStreamReader）。
7. 捕获并处理可能的异常。try-with-resources 块结束时会使 JMSContext 自动关闭，这会停止向消息监听器传送消息。

消息监听器 TextListener.java 完成以下步骤：

1. 消息到达时，自动调用 onMessage 方法。

2. 如果消息是一个 TextMessage，则 onMessage 方法将其内容显示为一个字符串值。如果这个消息不是一个文本消息，则做出报告：

```
public void onMessage(Message m) {
    try {
        if (m instanceof TextMessage) {
            System.out.println(
                    "Reading message: " + m.getBody(String.class));
        } else {
            System.out.println("Message is not a TextMessage");
        }
    } catch (JMSException | JMSRuntimeException e) {
        System.err.println("JMSException in onMessage(): "
                + e.toString());
    }
}
```

对于这个例子，要使用 17.2.2.1 节中创建的连接工厂和目的地。

这里的步骤假定你已经使用 NetBeans IDE 或 Maven 完成了所有示例的构建和打包。

17.2.6.2 运行 AsynchConsumer 和 Producer 客户端

运行这个示例需要两个终端窗口，这与 17.2.5 节中一样。

1. 在运行 SynchConsumer 客户端的终端窗口中，进入 asynchconsumer 示例目录：

```
cd tut-install/jms/simple/asynchconsumer
```

2. 运行 AsynchConsumer 客户端，指定 topic 目的地类型：

```
appclient -client target/asynchconsumer.jar topic
```

客户端会显示以下输出行（还包括一些额外的输出），然后等待消息到达：

```
Destination type is topic
To end program, enter Q or q, then <return>
```

3. 在之前运行 Producer 客户端的终端窗口中，再次运行这个客户端，发送 3 个消息：

```
appclient -client target/producer.jar topic 3
```

客户端的输出如下（还包括一些额外的输出）：

```
Destination type is topic
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```

在另一个窗口中，AsynchConsumer 客户端显示以下内容（还包括一些额外的输出）：

```
Destination type is topic
To end program, enter Q or q, then <return>
Reading message: This is message 1 from producer
```

```
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Message is not a TextMessage
```

之所以会显示最后一行，是因为这个客户端接收到 Producer 客户端发送的非文本控制消息。

4. 输入 Q 或 q，并按下回车键停止 AsynchConsumer 客户端。

5. 现在使用一个队列来运行这些客户端。

在这种情况下，与同步示例一样，可以先运行 Producer 客户端，因为发送方和接收方之间没有时间依赖：

```
appclient -client target/producer.jar queue 3
```

这个客户端的输出如下所示：

```
Destination type is queue
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```

6. 在另一个窗口中，运行 AsynchConsumer 客户端：

```
appclient -client target/asynchconsumer.jar queue
```

这个客户端的输出如下（还包括一些额外的输出）：

```
Destination type is queue
To end program, enter Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Message is not a TextMessage
```

7. 输入 Q 或 q，按下回车键停止这个客户端。

17.2.7 浏览队列中的消息

这一节介绍一个创建 QueueBrowser 对象来检查队列消息的例子，如 16.3.8 节中所述。然后会解释如何使用 GlassFish 服务器编译、打包和运行这个示例。

17.2.7.1 MessageBrowser.java 客户端

要为一个队列创建 QueueBrowser，需要调用 JMSContext.createBrowser 方法，并提供这个队列作为参数。队列中的消息将作为一个 Enumeration 对象得到。然后迭代处理这个 Enumeration 对象，显示各个消息的内容。

MessageBrowser.java 客户端完成以下步骤：

1. 为连接工厂和队列注入资源。

2. 在一个 try-with-resources 块中，创建一个 JMSContext。

3. 创建一个 QueueBrowser。

```
QueueBrowser browser = context.createBrowser(queue);
```

4. 获取包含消息的 Enumeration：

```
Enumeration msgs = browser.getEnumeration();
```

5. 验证 Enumeration 确实包含消息，然后显示消息的内容：

```
if ( !msgs.hasMoreElements() ) {
    System.out.println("No messages in queue");
} else {
    while (msgs.hasMoreElements()) {
        Message tempMsg = (Message)msgs.nextElement();
        System.out.println("Message: " + tempMsg);
    }
}
```

6. 捕获并处理可能的异常。try-with-resources 块结束时会使 JMSContext 自动关闭。

将消息内容转储到标准输出时，会采用一种格式获取消息体和属性，这种格式取决于 toString 方法的具体实现。在 GlassFish 服务器中，消息格式如下所示：

```
Text:   This is message 3 from producer
Class:                  com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID():   ID:ID:8-10.152.23.26(bf:27:4:e:e7:ec)-55645-1363100335526
getJMSTimestamp():      1129061034355
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             {JMSXDeliveryCount=0}
```

也可以不采用这种方式显示消息内容，而是调用 Message 接口的获取方法来获取你想看到的消息部分。

对于这个例子，要使用为 17.2.5 节创建的连接工厂和队列。这里假设已经完成所有示例的构建和打包。

17.2.7.2 运行 QueueBrowser 客户端

要使用 appclient 命令运行 MessageBrowser 示例，需要完成以下步骤。

还需要 Producer 示例向队列发送消息，另外在检查消息之后，还需要一个消费者客户端使用这些消息。

要运行这些客户端，需要两个终端窗口。

1. 在一个终端窗口中，进入 producer 目录：

```
cd tut-install/examples/jms/simple/producer
```

2. 运行 Producer 客户端，向队列发送一个消息以及一个非文本控制消息：

```
appclient -client target/producer.jar queue
```

客户端的输出如下（还包括一些额外的输出）：

```
Destination type is queue
Sending message: This is message 1 from producer
Text messages sent: 1
```

3. 在另一个终端窗口中，进入 messagebrowser 目录：

```
cd tut-install/jms/simple/messagebrowser
```

4. 使用以下命令运行 MessageBrowser 客户端：

```
appclient -client target/messagebrowser.jar
```

客户端的输出如下（还包括一些额外的输出）：

```
Message:
Text:   This is message 1 from producer
Class:                  com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID():  ID:9-10.152.23.26(bf:27:4:e:e7:ec)-55645-1363100335526
getJMSTimestamp():      1363100335526
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             {JMSXDeliveryCount=0}

Message:
Class:                  com.sun.messaging.jmq.jmsclient.MessageImpl
getJMSMessageID(): ID:10-10.152.23.26(bf:27:4:e:e7:ec)-55645-1363100335526
getJMSTimestamp():      1363100335526
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             {JMSXDeliveryCount=0}
```

第一个消息是 TextMessage，第二个是非文本控制消息。

5. 进入 synchconsumer 目录。

6. 运行 SynchConsumer 客户端来使用这些消息：

```
appclient -client target/synchconsumer.jar queue
```

客户端的输出如下（还包括一些额外的输出）：

```
Destination type is queue
Reading message: This is message 1 from producer
Messages received: 1
```

17.2.8 在同一个目的地运行多个消费者

为了进一步说明点对点和发布 / 订阅消息传送是如何工作的，可以使用 Producer 和 SynchConsumer 示例来发送消息，然后由两个同时运行的客户端使用。

1. 打开 3 个命令窗口。在一个窗口中，进入 producer 目录。在另外两个窗口中，进入 synchconsumer 目录。

2. 在每一个 synchconsumer 窗口中，开始运行客户端，从一个队列接收消息：

```
appclient -client target/synchconsumer.jar queue
```

等待，直到看到两个窗口中出现 Destination type is queue（目的地类型是队列）消息。

3. 在 producer 窗口中，运行客户端，向队列发送 20 个消息：

```
appclient -client target/producer.jar queue 20
```

4. 查看 synchconsumer 窗口中的输出。采用点对点消息传送时，每个消息只能有一个消费者。因此，各个客户端分别接收一些消息。其中一个客户端接收到非文本控制消息，报告接收的消息数，并退出。

5. 在没有接收到非文本控制消息的客户端窗口中，输入 Control-C 来退出程序。

6. 接下来，使用一个主题来运行 synchconsumer 客户端。在各个窗口中，运行以下命令：

```
appclient -client target/synchconsumer.jar topic
```

等待，直到看到两个窗口中出现 Destination type is topic（目的地类型是主题）消息。

7. 在 producer 窗口中，运行客户端，向主题发送 20 个消息：

```
appclient -client target/producer.jar topic 20
```

8. 再次查看 synchconsumer 窗口中的输出。采用发布 / 订阅消息传送时，会向主题的订阅发送所有消息的一个副本。因此，两个客户端分别都会接收所有 20 个文本消息以及非文本控制消息。

17.2.9 确认消息

JMS 为消费者客户端提供了两种方法来确保应用完成消息处理前不会确认消息：

- 在配置为使用 CLIENT_ACKNOWLEDGE 设置的 JMSContext 中，使用一个同步消费者。

- 在配置为使用默认 AUTO_ACKNOWLEDGE 设置的 JMSContext 中，为异步消息传送使用一个消息监听器。

说明 在 Java EE 平台中，CLIENT_ACKNOWLEDGE 会话只能在应用客户端中使用，如这个例子所示。

Clientackconsumer 示例展示了第一种方法，其中会有一个同步消费者使用客户端确认。17.2.6 节中介绍的 Asynchconsumer 示例展示了第二种方法。

关于消息确认的更多信息，参见 16.4.1 节。

表 17-3 描述了消费者类型和确认类型的 4 种可能的组合。

表 17-3 同步和异步消费者的消息确认

消费者类型	确认类型	行 为
同步	客户端	消息处理完成后客户端确认消息
异步	客户端	消息处理完成后客户端确认消息
同步	自动	receive 调用后立即确认；如果后续的某个处理步骤失败，消息不能重新传送
异步	自动	onMessage 方法返回时，自动确认消息

这个例子存放在 *tut-install*/examples/jms/simple/clientackconsumer/ 目录。

示例客户端 ClientAckConsumer.java 创建一个 JMSContext，指定使用客户端确认：

```
try (JMSContext context =
      connectionFactory.createContext(JMSContext.CLIENT_ACKNOWLEDGE);) {
   ...
```

与 SynchConsumer.java 类似，这个客户端使用了一个几乎完全相同的 while 循环，唯一的区别是处理各个消息之后，它会调用 JMSContext 的 acknowledge 方法：

```
context.acknowledge();
```

这个例子使用了以下对象：

- 为 17.2.5 节创建的 jms/MyQueue 资源
- java:comp/DefaultJMSConnectionFactory，这是 GlassFish 服务器预配置的平台默认连接工厂

17.2.9.1 运行 ClientAckConsumer 客户端

1. 在一个终端窗口中，进入以下目录：

tut-install/examples/jms/simple/producer/

2. 运行 Producer 客户端，向队列发送一些消息：

```
appclient -client target/producer.jar queue 3
```

3. 在另一个终端窗口中，进入以下目录：

tut-install/examples/jms/simple/clientackconsumer/

4. 要运行客户端，使用以下命令：

```
appclient -client target/clientackconsumer.jar
```

客户端的输出如下（还包括一些额外的输出）：

```
Created client-acknowledge JMSContext
Reading message: This is message 1 from producer
Acknowledging TextMessage
Reading message: This is message 2 from producer
Acknowledging TextMessage
Reading message: This is message 3 from producer
Acknowledging TextMessage
Acknowledging non-text control message
```

处理消息之后，客户端会显式地确认各个消息，就像配置为使用 AUTO_ACKNOWLEDGE 的 JMSContext 一样，MessageListener 处理一个异步接收的消息并成功返回后，JMSContext 会自动确认消息。

17.3 编写更高级的 JMS 应用

下面的例子显示了如何使用 JMS API 的一些更高级的特性：持久订阅和事务。

17.3.1 使用持久订阅

Durablesubscriptionexample 示例将介绍非共享持久订阅如何工作。这里展示了在没有活动的消费者时，持久订阅仍继续存在并积累消息。

这个例子包括两个模块：一个创建持久订阅和使用消息的 durableconsumer 应用以及一个 unsubscriber 应用，这个应用允许你在运行完 durableconsumer 应用之后从持久订阅取消订阅。

有关持久订阅的更多信息，参见 16.3.6.4 节。

主客户端 DurableConsumer.java 存放在 *tut-install*/examples/jms/durablesubscriptionexample/durableconsumer/ 目录下。

这个例子使用了一个连接工厂 jms/DurableConnectionFactory（有一个客户端 ID）。

DurableConsumer 客户端使用这个连接工厂创建一个 JMSContext。然后停止 JMSContext，调用 createDurableConsumer 并指定一个订阅名，从而在主题上创建一个持久订阅和一个消费者，然后注册一个消息监听器，之后再次启动 JMSContext。订阅不存在时才会创建订阅，所以这个例子可以反复运行：

```
try (JMSContext context = durableConnectionFactory.createContext();) {
    context.stop();
    consumer = context.createDurableConsumer(topic, "MakeItLast");
```

```
    listener = new TextListener();
    consumer.setMessageListener(listener);
    context.start();
    ...
```

要向主题发送消息，需要运行 Producer 客户端。

unsubscriber 示例包含一个非常简单的 Unsubscriber 客户端，它会在同一个连接工厂上创建一个 JMSContext，然后调用 unsubscribe 方法并指定订阅名：

```
try (JMSContext context = durableConnectionFactory.createContext();) {
    System.out.println("Unsubscribing from durable subscription");
    context.unsubscribe("MakeItLast");
} ...
```

17.3.1.1 为持久订阅示例创建资源

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在一个命令窗口中，进入 durableconsumer 示例。

```
cd tut-install/jms/durablesubscriptionexample/durableconsumer/
```

3. 使用 asadmin add-resources 命令创建资源：

```
asadmin add-resources src/main/setup/glassfish-resources.xml
```

命令输出会报告创建了一个连接器连接池和一个连接器资源。

4. 验证资源是否已经创建：

```
asadmin list-jms-resources
```

除了为简单示例创建的资源，这个命令还会列出这个新的连接工厂：

```
jms/MyQueue
jms/MyTopic
jms/__defaultConnectionFactory
jms/DurableConnectionFactory
Command list-jms-resources executed successfully.
```

17.3.1.2 运行持久订阅示例

1. 在一个终端窗口中，进入以下目录：

```
tut-install/examples/jms/durablesubscriptionexample/
```

2. 构建 durableconsumer 和 unsubscriber 示例：

```
mvn install
```

3. 进入 durableconsumer 目录：

```
cd durableconsumer
```

4. 要运行这个客户端，输入以下命令：

```
appclient -client target/durableconsumer.jar
```

这个客户端会创建持久消费者，然后等待消息：

```
Creating consumer for topic
Starting consumer
To end program, enter Q or q, then <return>
```

5. 在另一个终端窗口中，运行 Producer 客户端，向主题发送一些消息：

```
cd tut-install/examples/jms/simple/producer
appclient -client target/producer.jar topic 3
```

6. DurableConsumer 客户端接收消息之后，输入 q 或 Q 退出程序。在这里，这个客户端表现得就像其他异步消费者一样。

7. DurableConsumer 客户端还未运行时，使用 Producer 客户端发送更多消息：

```
appclient -client target/producer.jar topic 2
```

如果持久订阅不存在，这些消息会丢失，因为主题上当前没有消费者在运行。不过，持久订阅仍是活动的，它会保留这些消息。

8. 再次运行 DurableConsumer 客户端。它会立即接收它不活动时发送的那些消息：

```
Creating consumer for topic
Starting consumer
To end program, enter Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Message is not a TextMessage
```

9. 输入 q 或 Q 退出程序。

17.3.1.3 运行 unsubscriber 示例

DurableConsumer 客户端运行完成后，运行 unsubscriber 示例从这个持久订阅取消订阅。

1. 在一个终端窗口中，进入以下目录：

```
tut-install/examples/jms/durablesubscriptionexample/unsubscriber/
```

2. 要运行 Unsubscriber 客户端，输入以下命令：

```
appclient -client target/unsubscriber.jar
```

客户端会报告它从这个持久订阅取消了订阅。

17.3.2 使用本地事务

Transactedexample 示例将展示在一个 JMS 客户端应用中如何使用本地事务。这里还会展示 16.4.3 节中介绍的请求 / 应答消息传送模式如何使用，不过会使用永久目的地而不是临时目的地。这个例子包括 3 个模块：genericsupplier、retailer 和 vendor，这些模块都放在 *tut-install*/examples/jms/transactedexample/ 目录下。源代码在各个模块的 src/main/java/javaeetutorial 树中。Genericsupplier 和 retailer 模

块各包含一个类，分别是genericsupplier/GenericSupplier.java和retailer/Retailer.java。Vendor模块更复杂一些，包含4个类：vendor/Vendor.java、vendor/VendorMessageListener.java、vendor/Order.java和vendor/SampleUtilities.java。

这个例子展示了如何在一个事务中使用队列和主题，以及如何向消息监听器的构造函数传递一个JMSContext。这个例子是一个相当简化的电子商务应用，其中完成了以下动作。

1. 零售商（retailer/src/main/java/javaeetutorial/retailer/Retailer.java）向分销商订单队列发送一个MapMessage，订购一定数量的计算机，并等待分销商的应答：

```
outMessage = context.createMapMessage();
outMessage.setString("Item", "Computer(s)");
outMessage.setInt("Quantity", quantity);
outMessage.setJMSReplyTo(retailerConfirmQueue);
context.createProducer().send(vendorOrderQueue, outMessage);
System.out.println("Retailer: ordered " + quantity + " computer(s)");
orderConfirmReceiver = context.createConsumer(retailerConfirmQueue);
```

2. 分销商（vendor/src/main/java/javaeetutorial/retailer/Vendor.java）在一个事务中接收这个零售商的订单消息，向供应商订单主题发送一个订单消息。这个JMS事务使用了一个会话，所以可以把从队列接收和向主题发送的操作结合在一起。下面是使用同一个会话为一个队列创建消费者的代码：

```
vendorOrderReceiver = session.createConsumer(vendorOrderQueue);
```

下面的代码接收到来的消息，发出一个消息，并提交JMSContext。为了使代码序列简单，这里省略了具体的消息处理：

```
inMessage = vendorOrderReceiver.receive();
// Process the incoming message and format the outgoing
// message
...
context.createProducer().send(supplierOrderTopic, orderMessage);
...
context.commit();
```

为简单起见，这里只有两个供应商，一个供应CPU，另一个供应硬盘。

3. 每个供应商（genericsupplier/src/main/java/javaeetutorial/retailer/GenericSupplier.java）从订单主题接收订单，检查库存，然后将订购的商品发送到订单消息JMSReplyTo字段中指定的队列。如果这个商品没有足够的库存，那么供应商会发出现有的全部库存。从主题同步接收和发送到队列的操作在同一个JMS事务中进行：

```
receiver = context.createConsumer(SupplierOrderTopic);
...
inMessage = receiver.receive();
if (inMessage instanceof MapMessage) {
```

```
    orderMessage = (MapMessage) inMessage;
} ...
// Process message
outMessage = context.createMapMessage();
// Add content to message
context.createProducer().send(
        (Queue) orderMessage.getJMSReplyTo(),
        outMessage);
// Display message contents
context.commit();
```

4. 分销商从其确认队列接收到供应商的应答，并更新订单的状态。消息由一个异步消息监听器 VendorMessageListener 处理；这一步显示了如何结合使用 JMS 事务和消息监听器：

```
MapMessage component = (MapMessage) message;
...
int orderNumber = component.getInt("VendorOrderNumber");
Order order = Order.getOrder(orderNumber).processSubOrder(component);
context.commit();
```

5. 对于一个给定的订单，所有未完成的应答都已经处理时，分销商消息监听器会发出一个消息，通知零售商是否能履行这个订单：

```
Queue replyQueue = (Queue) order.order.getJMSReplyTo();
MapMessage retailerConfirmMessage = context.createMapMessage();
// Format the message
context.createProducer().send(replyQueue, retailerConfirmMessage);
context.commit();
```

6. 零售商从分销商接收消息：

```
inMessage = (MapMessage) orderConfirmReceiver.receive();
```

然后零售商再下一个订单，这一次计算机数量是第一个订单中的两倍，所以这些步骤会执行两次。

图 17-1 展示了这些步骤。

所有消息都使用 MapMessage 消息类型。除了分销商处理供应商应答外，所有其他消息接收都使用同步接收。这里会异步处理供应商应答，以展示消息监听器中如何使用事务。

Vendor 客户端会以随机的间隔抛出一个异常来模拟数据库问题，并导致回滚。

除 Retailer 外，所有客户端都使用事务性上下文。

这个例子使用了 3 个队列，分别名为 jms/AQueue、jms/BQueue 和 jms/CQueue，另外使用了一个名为 jms/OTopic 的主题。

17.3.2.1 为 transactedexample 示例创建资源

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

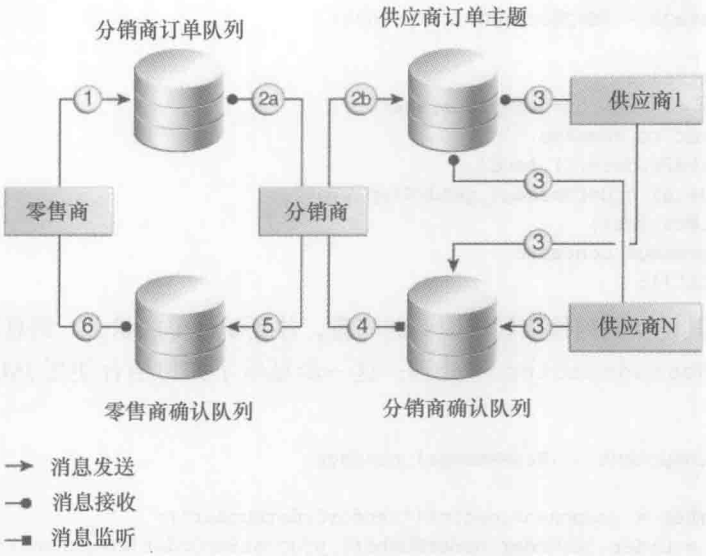


图 17-1 事务：JMS 客户端示例

2. 在一个命令窗口中，进入 genericsupplier 示例：

```
cd tut-install/jms/transactedexample/genericsupplier
```

3. 使用 asadmin add-resources 命令创建资源：

```
asadmin add-resources src/main/setup/glassfish-resources.xml
```

4. 验证资源是否已经创建：

```
asadmin list-jms-resources
```

除了为简单示例以及持久订阅示例创建的资源，这个命令还会列出 4 个新的目的地：

```
jms/MyQueue
jms/MyTopic
jms/AQueue
jms/BQueue
jms/CQueue
jms/OTopic
jms/__defaultConnectionFactory
jms/DurableConnectionFactory
Command list-jms-resources executed successfully.
```

17.3.2.2 运行 transactedexample 客户端

需要 4 个终端窗口来运行这些客户端。一定要按正确的顺序启动这些客户端。

1. 在一个终端窗口中，进入以下目录：

```
tut-install/examples/jms/transactedexample/
```

2. 要构建和打包所有模块，输入以下命令：

```
mvn install
```

3. 进入 enericsupplier 目录：

```
cd genericsupplier
```

4. 使用以下命令启动 CPU 供应商客户端：

```
appclient -client target\genericsupplier.jar CPU
```

在初始一些输出之后，客户端会报告以下输出：

```
Starting CPU supplier
```

5. 在第二个终端窗口中，进入 genericsupplier 目录：

```
cd tut-install/examples/jms/transactedexample/genericsupplier
```

6. 使用以下命令启动硬盘供应商客户端：

```
appclient -client target\genericsupplier.jar HD
```

在初始一些输出之后，客户端报告以下输出：

```
Starting Hard Drive supplier
```

7. 在第三个终端窗口中，进入 vendor 目录：

```
cd tut-install/examples/jms/transactedexample/vendor
```

8. 使用以下命令启动 Vendor 客户端：

```
appclient -client target\vendor.jar
```

在初始一些输出之后，客户端报告以下输出：

```
Starting vendor
```

9. 在另一个终端窗口中，进入 retailer 目录：

```
cd tut-install/examples/jms/transactedexample/retailer
```

10. 使用下面的命令运行 Retailer 客户端，参数指定了要订购的计算机数量：

```
appclient -client target/retailer.jar 4
```

在初始一些输出之后，Retailer 客户端报告类似下面的输出。在这里第一个订单已经履行，不过第二个订单没有完成：

```
Retailer: Quantity to be ordered is 4
Retailer: Ordered 4 computer(s)
Retailer: Order filled
Retailer: Placing another order
Retailer: Ordered 8 computer(s)
Retailer: Order not filled
```

Vendor 客户端会报告类似下面的输出，指出在这里可以发出第一个订单订购的所有计算机，但是无法发出第二个订单所要的全部计算机：

```
Vendor: Retailer ordered 4 Computer(s)
Vendor: Ordered 4 CPU(s) and hard drive(s)
  Vendor: Committed transaction 1
Vendor: Completed processing for order 1
Vendor: Sent 4 computer(s)
  Vendor: committed transaction 2
Vendor: Retailer ordered 8 Computer(s)
Vendor: Ordered 8 CPU(s) and hard drive(s)
  Vendor: Committed transaction 1
Vendor: Completed processing for order 2
Vendor: Unable to send 8 computer(s)
  Vendor: Committed transaction 2
```

CPU 供应商报告下面的输出。在这里，它能发出两个订单都需要的全部 CPU：

```
CPU Supplier: Vendor ordered 4 CPU(s)
CPU Supplier: Sent 4 CPU(s)
  CPU Supplier: Committed transaction
CPU Supplier: Vendor ordered 8 CPU(s)
CPU Supplier: Sent 8 CPU(s)
  CPU Supplier: Committed transaction
```

硬盘供应商报告类似下面的输出。在这里，它无法完全供应第二个订单的硬盘：

```
Hard Drive Supplier: Vendor ordered 4 Hard Drive(s)
Hard Drive Supplier: Sent 4 Hard Drive(s)
  Hard Drive Supplier: Committed transaction
Hard Drive Supplier: Vendor ordered 8 Hard Drive(s)
Hard Drive Supplier: Sent 1 Hard Drive(s)
  Hard Drive Supplier: Committed transaction
```

11. 根据你的需要，将第 4 步到第 10 步重复多次。有时分销商可能会报告一个导致回滚的异常：

```
Vendor: JMSException occurred: javax.jms.JMSException: Simulated
database concurrent access exception
  Vendor: Rolled back transaction 1
```

12. 客户端运行完成后，可以使用以下命令删除目的地资源：

```
asadmin delete-jms-resource jms/AQueue
asadmin delete-jms-resource jms/BQueue
asadmin delete-jms-resource jms/CQueue
asadmin delete-jms-resource jms/OTopic
```

17.4 编写高性能和可扩展的 JMS 应用

本节介绍如何使用 JMS API 编写能健壮地处理大量消息的应用。这些示例会使用持久和非持久共享消费者。

17.4.1 使用共享非持久订阅

本节将介绍接收客户端，这里给出了一个例子，展示如何使用共享消费者将发送到一个

主题的消息分布于多个不同的消费者。然后这一节将解释如何使用 GlassFish 服务器编译和运行这些客户端。

你可能想将这个例子与 17.2.8 节的结果做个比较，那一节中使用了一个非共享消费者。在那个例子中，消息分布到一个队列的多个消费者，但主题上的各个消费者会接收到所有消息，因为主题上的各个消费者分别使用一个单独的主题订阅。

不过，在这个例子中，消息会分布到主题上的多个消费者，因为所有消费者都共享同一个订阅。增加到主题订阅上的各个消息只能由一个消费者接收，这与增加到队列的消息只能由一个消费者接收很类似。

一个主题可以有多个订阅。发送到主题的各个消息将增加到各个主题订阅。不过，如果某个订阅上有多个消费者，则增加到订阅的各个消息只会传送到其中一个消费者。

17.4.1.1 编写共享消费者示例的客户端

发送客户端是 `Producer.java`，也就是前面的例子中使用的客户端。

接收客户端是 `SharedConsumer.java`。这与 `AsynchConsumer.java` 非常类似，只不过它会使用一个主题。这个客户端会完成以下步骤：

1. 为连接工厂和主题注入资源。
2. 在一个 `try-with-resources` 块中，创建一个 `JMSContext`。
3. 在一个共享非持久订阅上创建一个消费者，指定一个订阅名：

```
consumer = context.createSharedConsumer(topic, "SubName");
```

4. 创建 `TextListener` 类的一个实例，将它注册为共享消费者的消息监听器。
5. 监听发布到目的地的消息，用户键入字符 `q` 或 `Q` 时停止。
6. 捕获并处理可能的异常。`try-with-resources` 块结束时会使 `JMSContext` 自动关闭。

`TextListener.java` 类与 `asynchconsumer` 示例中的监听器相同。对于这个例子，将使用 17.2.2.1 节中创建的默认连接工厂和主题。

17.4.1.2 运行 SharedConsumer 和 Producer 客户端

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 打开 3 个命令窗口。在第一个窗口中，进入 `simple/producer/` 目录：

```
cd tut-install/examples/jms/simple/producer
```

3. 在第二个和第三个命令窗口中，进入 `shared/sharedconsumer/` 目录：

```
cd tut-install/examples/jms/shared/sharedconsumer/
```

4. 在一个 `sharedconsumer` 窗口中构建这个示例：

```
mvn install
```

5. 在两个 `sharedconsumer` 窗口中分别开始运行客户端。不需要指定 `topic` 参数：

```
appclient -client target/sharedconsumer.jar
```

等待，直到看到两个窗口出现以下输出：

```
Waiting for messages on topic
To end program, enter Q or q, then <return>
```

6. 在 producer 窗口中运行生产者客户端，并指定主题和消息数：

```
appclient -client target/producer.jar topic 20
```

各个消费者客户端会分别接收部分消息。只有其中一个客户端会接收到指示消息流结束的非文本消息。

7. 输入 Q 或 q，并按回车键停止各客户端，查看接收到的文本消息数报告。

17.4.2 使用共享持久订阅

shareddurableconsumer 客户端显示如何使用共享持久订阅。这里展示了共享持久订阅同时具备持久订阅的优点（客户端不活动时订阅仍保持活动）和共享消费者的优点（消息负载可以分摊到多个客户端）。

相对于 DurableConsumer.java 客户端，这个例子与 sharedconsumer 示例更相似。它使用两个类，即 SharedDurableConsumer.java 和 TextListener.java，这两个类都在 *tut-install*/examples/jms/shared/shareddurableconsumer/ 目录下。

这个客户端使用 java:comp/DefaultJMSConnectionFactory，这是一个没有客户端标识符的连接工厂，也是共享持久订阅的推荐做法。它使用了 createSharedDurableConsumer 方法并指定一个订阅名来建立订阅：

```
consumer = context.createSharedDurableConsumer(topic, "MakeItLast");
```

可以结合 Producer.java 客户端运行这个示例。

17.4.2.1 运行 SharedDurableConsumer 和 Producer 客户端

1. 确保已启动 Glass Fish 服务器（参见 2.2 节）。

2. 在一个终端窗口中，进入以下目录：

tut-install/examples/jms/shared/shareddurableconsumer/

3. 输入以下命令编译和打包客户端：

```
mvn install
```

4. 运行第一个客户端来建立持久订阅：

```
appclient -client target/shareddurableconsumer.jar
```

5. 客户端显示以下输出并暂停：

```
Waiting for messages on topic
To end program, enter Q or q, then <return>
```

6. 在 shareddurableconsumer 窗口中，输入 Q 或 q 退出程序。订阅保持活动，不

过客户端没有运行。

7. 打开另一个终端窗口，进入 producer 示例目录：

```
cd tut-install/examples/jms/simple/producer
```

8. 运行 producer 示例，向主题发送一些消息：

```
appclient -client target/producer.jar topic 6
```

9. 生产者发送消息之后，打开第三个终端窗口，进入 shareddurableconsumer 目录。

10. 在第一个和第三个终端窗口中运行客户端程序。先启动的客户端将接收到无活动订阅者时发送的所有消息：

```
appclient -client target/shareddurableconsumer.jar
```

11. 两个 shareddurableconsumer 客户端都在运行时，进入 producer 窗口，向主题发送大量消息：

```
appclient -client target/producer.jar topic 25
```

现在消息将由两个消费者客户端共享。如果继续向主题发送多组消息，则每个客户端将分别接收其中一部分消息。如果退出其中一个客户端，再发送更多消息，则另一个客户端将接收所有消息。

17.5 使用一个简单 Web 应用发送和接收消息

Web 应用可以使用 JMS API 发送和接收消息，如 16.5.3 节所述。这一节介绍一个非常简单的 Web 应用组件，该应用使用了 JMS API。

本节假设你已经熟悉 JSF 技术的基础知识，参见本书第 1 卷第三部分。

示例 websimplemessage 在 *tut-install*/jms/examples/ 目录下。它使用了发送和接收 Facelets 页面以及相应的后备 bean。用户在发送页面的文本域中输入一个消息并单击一个按钮时，这个页面的后备 bean 将这个消息发送到一个队列，并将其在页面上显示。用户进入接收页面，单击另一个按钮时，该页面的后备 bean 会同步接收这个消息并将其显示。

17.5.1 websimplemessage Facelets 页面

这个示例的 Facelets 页面如下：

- sender.xhtml，提供一个有标签的 h:InputText 标记，用户将在这个文本域中输入消息，另外还有两个命令按钮。用户单击 Send Message（发送消息）按钮时，会调用 senderBean.sendMessage 方法将消息发送到队列，并显示其内容。用户单击 Go to Receive Page（进入接收页面）按钮时，会显示 receiver.xhtml 页面。

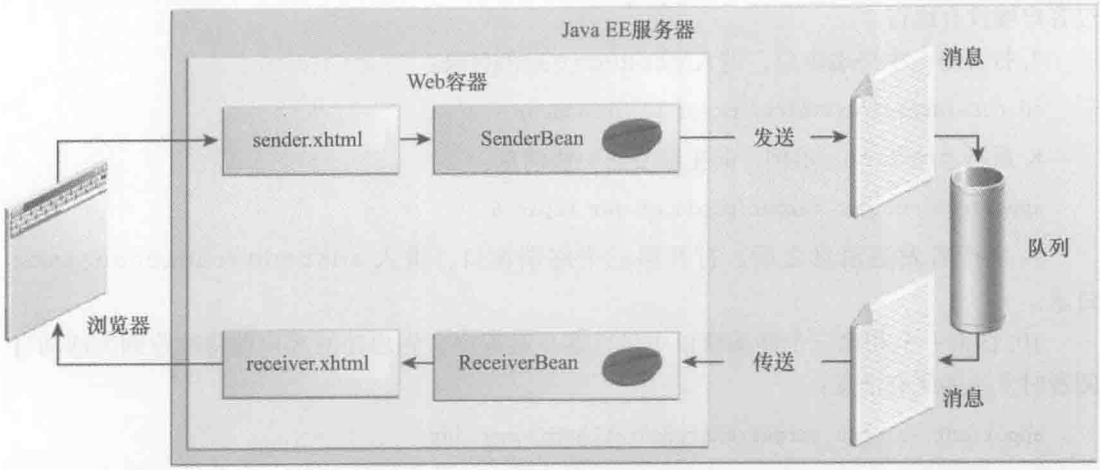


图 17-2 websimplemessage 应用

- receiver.xhtml，也提供了两个命令按钮。用户单击 Receive Message（接收消息）按钮时，会调用 receiverBean.getMessage 方法从队列获取消息，并显示其内容。用户单击 Send Another Message（发送另一个消息）按钮时，sender.xhtml 页面会再次出现。

17.5.2 websimplemessage 托管 bean

这个例子的两个托管 bean 如下：

- SenderBean.java 是一个 CDI 托管 bean，有一个属性 messageText 和一个业务方法 sendMessage。这个类增加了 @JMSDestinationDefinition 注解来创建一个组件私有的队列：

```
@JMSDestinationDefinition(
        name = "java:comp/jms/webappQueue",
        interfaceName = "javax.jms.Queue",
        destinationName = "PhysicalWebappQueue")
@Named
@RequestScoped
public class SenderBean {
```

sendMessage 方法注入一个 JMSContext（使用默认连接工厂）和队列，创建一个生产者，发送用户在 Facelets 页面上键入的消息，并创建一个 FacesMessage 在 Facelets 页面上显示：

```
@Inject
private JMSContext context;
@Resource(lookup = "java:comp/jms/webappQueue")
private Queue queue;
private String messageText;
```

```
...
public void sendMessage() {
    try {
        String text = "Message from producer: " + messageText;
        context.createProducer().send(queue, text);
        FacesMessage facesMessage =
                new FacesMessage("Sent message: " + text);
        FacesContext.getCurrentInstance()
                .addMessage(null, facesMessage);
    } catch (Throwable t) {
        logger.log(Level.SEVERE,
                "SenderBean.sendMessage: Exception: {0}",
                t.toString());
    }
}
```

- ReceiverBean.java，这是一个 CDI 托管 bean，有一个业务方法 getMessage。这个方法注入一个 JMSContext（使用默认连接工厂）以及 SenderBean 中定义的队列，创建一个消费者，接收消息，并创建一个 FacesMessage 在 Facelets 页面上显示：

```
@Inject
private JMSContext context;
@Resource(lookup = "java:comp/jms/webappQueue")
private Queue queue;
...
public void getMessage() {
    try {
        JMSConsumer receiver = context.createConsumer(queue);
        String text = receiver.receiveBody(String.class);

        if (text != null) {
            FacesMessage facesMessage =
                    new FacesMessage("Reading message: " + text);
            FacesContext.getCurrentInstance().addMessage(null,
                    facesMessage);
        } else {
            FacesMessage facesMessage =
                    new FacesMessage("No message received after 1 second");
            FacesContext.getCurrentInstance().addMessage(null,
                    facesMessage);
        }
    } catch (Throwable t) {
        logger.log(Level.SEVERE,
                "ReceiverBean.getMessage: Exception: {0}",
                t.toString());
    }
}
```

17.5.3 运行 websimplemessage 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 websimplemessage 应用。

17.5.3.1 为 websimplemessage 示例创建资源

这个例子使用了注解定义的一个队列和预配置的默认连接工厂 `java:comp/DefaultJMSConnectionFactory`。

17.5.3.2 使用 NetBeans IDE 打包和部署 websimplemessage

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/jms
```

4. 选择 `websimplemessage` 文件夹。
5. 单击 Open Project。
6. 在 Projects(工程）标签页，右键单击 `websimplemessage` 工程并选择 Build(构建)。

这个命令会构建并部署这个工程。

17.5.3.3 使用 Maven 打包和部署 websimplemessage

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 在一个终端窗口中导航至：

```
tut-install/examples/jms/websimplemessage/
```

3. 要编译源文件、打包并部署这个应用，可以使用以下命令：

```
mvn install
```

17.5.3.4 运行 websimplemessage 示例

1. 在 Web 浏览器中输入以下 URL：

```
http://localhost:8080/websimplemessage
```

2. 在文本域中输入一个消息，单击 Send Message（发送消息）。

例如，如果输入 "Hello, Duke"，就会有以下消息出现在按钮下方：

```
Sent message: Message from producer: Hello, Duke
```

3. 单击 Go to Receive Page（进入接收页面）。
4. 单击 Receive Message（接收消息）。

按钮下面会显示以下内容：

```
Reading message: Message from producer: Hello, Duke
```

5. 单击 Send Another Message（发送另一个消息），返回到发送页面。
6. 应用运行完成后，使用 NetBeans IDE 的 Services（服务）标签页或 `mvn cargo:undeploy` 命令取消部署。

17.6 使用消息驱动 bean 异步接收消息

如果要编写一个在 Java EE 应用客户端容器或者在 Java SE 平台上运行的应用，希望它异步地接收消息，则需要定义一个实现 MessageListener 接口的类，创建一个 JMSConsumer，并调用方法 setMessageListener。

如果要编写一个在 Java EE Web 或 EJB 容器中运行的应用，希望它异步接收消息，也需要定义一个实现 MessageListener 接口的类。不过，不用创建 JMSConsumer 并调用方法 setMessageListener，而必须将消息监听器类配置为一个消息驱动 bean。应用服务器会负责其余的工作。

消息驱动 bean 可以实现任意的消息类型。不过，大多数情况下，消息驱动 bean 都会实现 Java 消息服务（Java Message Service，JMS）技术。

这一节介绍一个简单的消息驱动 bean 示例。在学习下面的内容之前，应当先读一读 3.3 节以及 16.5.4 节中介绍的基本概念。

17.6.1 simplemessage 示例概述

simplemessage 应用有以下组件：

- SimpleMessageClient：这是一个应用客户端，向一个队列发送多个消息。
- SimpleMessageBean：这是一个消息驱动 bean，异步地处理发送到队列的消息。

图 17-3 展示了这个应用的结构。应用客户端向队列发送消息（这个队列使用管理控制台以管理方式创建）。JMS 提供者（在这里就是 GlassFish 服务器）将消息传送到消息驱动 bean 实例，然后再处理这些消息。

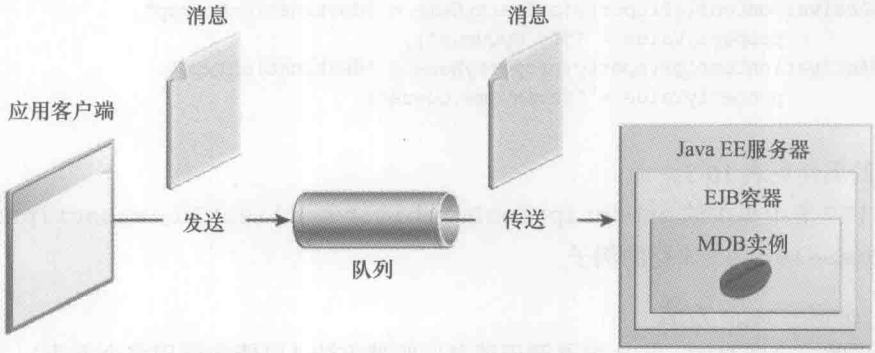


图 17-3 simplemessage 应用

这个应用的源代码放在 *tut-install*/examples/jms/simplemessage/ 目录中。

17.6.2 simplemessage 应用客户端

SimpleMessageClient 向 SimpleMessageBean 监听的队列发送消息。客户端首

先注入连接工厂和队列资源：

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(lookup = "jms/MyQueue")
private static Queue queue;
```

接下来，客户端在一个 try-with-resources 块中创建 JMSContext：

```
String text;
final int NUM_MSGS = 3;

try (JMSContext context = connectionFactory.createContext();) {
```

最后，客户端向队列发送多个文本消息：

```
for (int i = 0; i < NUM_MSGS; i++) {
    text = "This is message " + (i + 1);
    System.out.println("Sending message: " + text);
    context.createProducer().send(queue, text);
}
```

17.6.3 simplemessage 消息驱动 bean 类

16.5.4 节中介绍了一个消息驱动 bean 类，SimpleMessageBean 类的代码展示了这个消息驱动 bean 类的需求。

SimpleMessageBean 类的前几行代码使用 @MessageDriven 注解的 activationConfig 属性来指定配置属性：

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "jms/MyQueue"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "javax.jms.Queue")
})
```

可用的属性见表 16-3。

参见 17.7 节中提供的 subscriptionDurability、clientId、subscriptionName 和 messageSelector 属性的例子。

17.6.3.1 onMessage 方法

队列接收一个消息时，EJB 容器调用消息监听器方法（可能会调用多个方法）。对于一个使用 JMS 的 bean，则要调用 MessageListener 接口的 onMessage 方法。

在 SimpleMessageBean 类中，onMessage 方法将到来的消息强制转换为一个 TextMessage，并显示消息文本：

```
public void onMessage(Message inMessage) {

    try {
```

```
        if (inMessage instanceof TextMessage) {
            logger.log(Level.INFO,
                    "MESSAGE BEAN: Message received: {0}",
                    inMessage.getBody(String.class));
        } else {
            logger.log(Level.WARNING,
                    "Message of wrong type: {0}",
                    inMessage.getClass().getName());
        }
    } catch (JMSException e) {
        logger.log(Level.SEVERE,
                "SimpleMessageBean.onMessage: JMSException: {0}",
                e.toString());
        mdc.setRollbackOnly();
    }
}
```

17.6.4 运行 simplemessage 示例

可以使用 NetBeans IDE 或 Maven 构建、部署和运行 simplemessage 示例。

17.6.4.1 为 simplemessage 示例创建资源

这个例子使用了一个名为 jms/MyQueue 的队列以及预配置的默认连接工厂 java:comp/DefaultJMSConnectionFactory。

如果已经运行过 17.2 节中给出的简单 JMS 示例，并且还没有删除那些资源，那么你已经有了这个队列。否则，按照 17.2.2.1 节中的说明创建这个队列。

关于创建 JMS 资源的更多信息，参见 17.2.2 节。

17.6.4.2 使用 NetBeans IDE 运行 simplemessage 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 从 File（文件）菜单选择 Open Project（打开工程）。

3. 在 Open Project 对话框中，导航至：

tut-install/examples/jms/simplemessage

4. 选择 simplemessage 文件夹。

5. 确保 Open Required Projects（打开必要工程）复选框选中，然后单击 Open Project（打开工程）。

6. 在 Projects（工程）标签页中，右键单击 simplemessage 工程并选择 Build（构建）。（如果 NetBeans IDE 建议运行启动构建，单击选择框确认。）

这个命令会打包应用客户端和消息驱动 bean，然后在 simplemessage-ear/target/ 目录中创建一个名为 simplemessage.ear 的文件。接下来会部署 simplemessage-ear 模块，获取客户端桩，并运行这个应用客户端。

输出窗口中的输出如下（前面是应用客户端容器输出）：

```
Sending message: This is message 1
Sending message: This is message 2
Sending message: This is message 3
To see if the bean received the messages,
 check <install_dir>/domains/domain1/logs/server.log.
```

在服务器日志文件中，会出现类似下面的日志行：

```
MESSAGE BEAN: Message received: This is message 1
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

接收到的消息可能并不按其发送的顺序出现。

7. 应用结束运行后，使用 Services（服务）标签页取消部署。

17.6.4.3 使用 Maven 运行 simplemessage 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在一个终端窗口中导航至：

```
tut-install/examples/jms/simplemessage/
```

3. 要编译源文件并打包应用，可以使用以下命令：

```
mvn install
```

这个目标将打包应用客户端和消息驱动 bean，然后在 simplemessage-ear/target/ 目录中创建一个名为 simplemessage.ear 的文件。接下来会部署 simplemessage-ear 模块，获取客户端桩，并运行这个应用客户端。

终端窗口中的输出如下（前面是应用客户端容器输出）：

```
Sending message: This is message 1
Sending message: This is message 2
Sending message: This is message 3
To see if the bean received the messages,
 check <install_dir>/domains/domain1/logs/server.log.
```

在服务器日志文件中，可以看到类似下面的日志行：

```
MESSAGE BEAN: Message received: This is message 1
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

接收到的消息可能并不按其发送的顺序出现。

7. 应用运行完成后，使用 mvn cargo:undeploy 命令取消部署。

17.7 从会话 bean 向 MDB 发送消息

这一节解释如何编写、编译、打包、部署和运行一个结合会话 bean 使用 JMS API 的应用。这个应用包含以下组件：

- 调用一个会话 bean 的应用客户端
- 向一个主题发布多个消息的会话 bean
- 使用一个持久主题订阅和一个消息选择器接收和处理消息的消息驱动 bean

这一节的有关源文件位于 *tut-install*/examples/jms/clientsessionmdb/ 目录。本节中的路径名都相对于这个目录。

17.7.1 为 clientsessionmdb 示例编写应用组件

这个应用展示了如何从一个企业 bean（在这里是一个会话 bean）而不是从应用客户端发送消息，如 17.6 节中的例子所示。图 17-4 展示了这个应用的结构。从企业 bean 发送消息与从托管 bean 发送消息非常类似，参见 17.5 节。

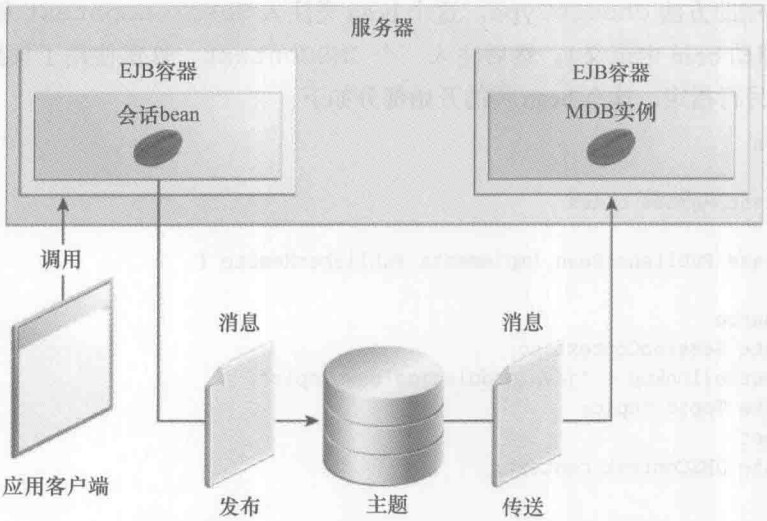


图 17-4 An 企业 bean 应用：客户端到会话 bean 到消息驱动 bean

这个例子中的 Publisher 企业 bean 是类似通讯社新闻提要的企业应用，将新闻划分为 6 类。消息驱动 bean 可以表示一个新闻编辑部，这里有不同类型新闻的编辑部，例如，体育编辑部可以为有关体育的所有新闻建立一个订阅。

这个示例中的应用客户端注入 Publisher 企业 bean 的远程主接口，然后调用这个 bean 的业务方法。这个企业 bean 将创建 18 个文本消息。对于每个消息，会设置一个 String 属性值，这个值从 6 个值中随机选取（6 个值分别表示 6 类新闻），然后将这个消息发布到一个主题。消息驱动 bean 对这个属性使用一个消息选择器，来限制要传送哪些已发布的消息。

17.7.1.1 编写应用客户端：MyAppClient.java

应用客户端 MyAppClient.java 位于 clientsessionmdb-app-client 目录，它并不完成任何 JMS API 操作，所以比 17.6 节中的客户端简单。这个客户端使用依赖注入来得

到 Publisher 企业 bean 的业务接口：

```
@EJB(name="PublisherRemote")
private static PublisherRemote publisher;
```

然后将这个 bean 的业务方法调用两次。

17.7.1.2 编写 Publisher 会话 bean

Publisher bean 是一个无状态会话 bean，有一个业务方法。Publisher bean 使用一个远程接口而不是本地接口，因为要从应用客户端访问这个 bean。

远程接口 PublisherRemote.java 在 clientsessionmdb-ejb 下，声明了一个业务方法 publishNews。

bean 类 PublisherBean.java 也在 clientsessionmdb-ejb 下，实现了 publishNews 方法以及它的辅助方法 chooseType。这个 bean 类注入 SessionContext 和 Topic 资源（主题在消息驱动 bean 中定义）。然后注入一个 JMSContext，这里使用了预配置的默认连接工厂，除非另行指定。这个 bean 类的开始部分如下：

```
@Stateless
@Remote({
    PublisherRemote.class
})
public class PublisherBean implements PublisherRemote {

    @Resource
    private SessionContext sc;
    @Resource(lookup = "java:module/jms/newsTopic")
    private Topic topic;
    @Inject
    private JMSContext context;
    ...
```

业务方法 publishNews 创建一个 JMSProducer，并发布消息。

17.7.1.3 编写消息驱动 bean: MessageBean.java

消息驱动 bean 类 MessageBean.java 位于 clientsessionmdb-ejb 下，与 17.6 节中的消息驱动 bean 类几乎完全相同。不过，@MessageDriven 注解有所不同，因为这里没有使用队列，这个 bean 使用了一个主题、一个持久订阅和一个消息选择器。这个 bean 为应用定义了一个主题；该定义使用了 java:module 作用域，因为会话 bean 和消息驱动 bean 在同一个模块中。由于目的地在消息驱动 bean 中定义，所以 @MessageDriven 注解使用了 destinationLookup 激活配置属性（有关的更多信息参见 16.5.1 节）。这个注解还设置了激活配置属性 messageSelector、subscriptionDurability、clientId 和 subscriptionName，如下所示：

```
@JMSDestinationDefinition(
        name = "java:module/jms/newsTopic",
```

```
        interfaceName = "javax.jms.Topic",
        destinationName = "PhysicalNewsTopic")
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "java:module/jms/newsTopic"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "messageSelector",
            propertyValue = "NewsType = 'Sports' OR NewsType = 'Opinion'"),
    @ActivationConfigProperty(propertyName = "subscriptionDurability",
            propertyValue = "Durable"),
    @ActivationConfigProperty(propertyName = "clientId",
            propertyValue = "MyID"),
    @ActivationConfigProperty(propertyName = "subscriptionName",
            propertyValue = "MySub")
})
```

主题就是 PublisherBean 中定义的主题。这里的消息选择器表示体育以及评论编辑部，这是为了展示消息选择器的语法。

JMS 资源适配器使用这些属性为消息驱动 bean 创建一个连接工厂，允许 bean 使用持久订阅。

17.7.2 运行 clientsessionmdb 示例

可以使用 NetBeans IDE 或 Maven 构建、部署和运行 simplemessage 示例。

这个例子使用注解定义的一个主题和预配置的默认连接工厂 java:comp/DefaultJMSConnectionFactory，所以不用为它创建资源。

17.7.2.1 使用 NetBeans IDE 运行 clientsessionmdb

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

tut-install/examples/jms/clientsessionmdb

4. 选择 clientsessionmdb 文件夹。
5. 确保 Open Required Projects（打开必要工程）复选框选中，然后单击 Open Project。
6. 在 Projects（工程）标签页中，右键单击 clientsessionmdb 工程并选择 Build（构建）。（如果 NetBeans IDE 建议运行启动构建，单击选择框确认。）

这个命令会创建以下组件：

- 应用客户端 JAR 文件，包含客户端类文件和会话 bean 的远程接口，另外还包括一个清单文件，这个清单文件指定了主类，并将 EJB JAR 文件放在它的类路径中。
- EJB JAR 文件，包含会话 bean 和消息驱动 bean。
- 应用 EAR 文件，包含两个 JAR 文件。

clientsessionmdb.ear 文件创建在 clientsessionmdb-ear/target/ 目录中。

然后这个命令部署 EAR 文件，获取客户端桩，并运行这个客户端。

客户端显示以下输出行：

```
To view the bean output,
 check <install_dir>/domains/domain1/logs/server.log.
```

这个企业 bean 的输出会出现在服务器日志文件中。Publisher 会话 bean 发送两组 18 个消息（编号从 0 到 17）。由于有消息选择器，因此消息驱动 bean 只接收 NewsType 属性设置为 Sports 或 Opinion 的消息。

7. 应用运行结束后，使用 Services（服务）标签页取消部署应用。

17.7.2.2 使用 Maven 运行 clientsessionmdb

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在终端窗口中，进入以下目录：

```
tut-install/examples/jms/clientsessionmdb/
```

3. 要编译源文件，并打包、部署和运行这个应用，输入以下命令：

```
mvn install
```

这个命令创建以下组件：

- 应用客户端 JAR 文件，包含客户端类文件和会话 bean 的远程接口，另外还包括一个清单文件，这个清单文件指定了主类，并将 EJB JAR 文件放在它的类路径中。
- EJB JAR 文件，包含会话 bean 和消息驱动 bean。
- 应用 EAR 文件，包含两个 JAR 文件。

clientsessionmdb.ear 文件创建在 clientsessionmdb-ear/target/ 目录中。

然后这个命令部署 EAR 文件，获取客户端桩，并运行这个客户端。

客户端显示以下输出行：

```
To view the bean output,
 check <install_dir>/domains/domain1/logs/server.log.
```

这个企业 bean 的输出会出现在服务器日志文件中。Publisher 会话 bean 发送两组 18 个消息（编号从 0 到 17）。由于有消息选择器，因此消息驱动 bean 只接收 NewsType 属性设置为 Sports 或 Opinion 的消息。

4. 应用运行结束后，取消部署应用。

```
mvn cargo:undeploy
```

17.8 使用实体联接两个 MDB 的消息

这一节解释如何编写、编译、打包、部署和运行一个结合使用实体和 JMS API 的应用。

这个应用使用了以下组件：

- 一个发送和接收消息的应用客户端
- 两个消息驱动 bean
- 一个实体类

本节的源代码在 *tut-install*/examples/jms/clientmdbentity/ 目录中。这里的路径名都是相对于这个目录的相对位置。

17.8.1 clientmdbentity 示例应用概述

这个应用采用一种简化方式模拟了一个公司的人力资源（HR）部门在处理新员工招聘时的工作流。该应用还展示了如何使用 Java EE 平台完成很多 JMS 应用都需要完成的一个任务。

消息客户端通常必须等待来自多个信息源的消息。然后使用所有这些消息中的信息装配一个消息，再将这个消息发送到另一个目的地。这个设计模式通常称为**联接消息**（joining message），这个概念并不只是特定于 JMS。这样一个任务必须是事务性的，所有接收以及发送都作为一个事务。如果没有成功地接收所有消息，则事务会回滚。17.3.2 节给出了一个展示这个任务的应用客户端示例。

消息驱动 bean 在事务中一次只能处理一个消息。为了提供联接消息的能力，应用可以让消息驱动 bean 将中间信息存储在一个 Java 持久化 API 实体中。然后这个实体确定是否接收到所有消息，如果接收到所有消息，这个实体会报告某个消息驱动 bean，再由这个消息驱动 bean 创建消息，并把创建的消息发送到另一个目的地。完成任务后，可以将中间实体删除。

这个应用的基本步骤如下：

1. HR 部门的应用客户端为每个新员工生成一个员工 ID，然后发布一个消息（M1），其中包含这个新员工的名字、员工 ID 和职位。这个消息会发布到一个主题，因为消息需要由两个消息驱动 bean 使用。然后客户端再创建一个临时队列 ReplyQueue，它有一个消息监听器在等待这个消息的应答。（有关的更多信息参见 16.4.3 节。）

2. 两个消息驱动 bean 分别处理各个消息：一个是 OfficeMDB bean，负责指定这个新员工的办公室编号，另一个是 EquipmentMDB bean，为这个新员工指定设备。第一个处理消息的 bean 会创建并持久存储一个名为 SetupOffice 的实体，然后调用这个实体的一个业务方法存储它生成的信息。第二个 bean 找到已有的这个实体，并调用另一个业务方法来增加它的信息。.

3. 办公室和设备都已经指定时，实体的业务方法会向调用这个方法的消息驱动 bean 返回一个 true 值。然后消息驱动 bean 向应答队列发送一个消息（M2），这个消息描述了所指定的这些值（办公室和设备）。然后它会删除这个实体。应用客户端的消息监听器获取信息。

图 17-5 展示了这个应用的结构。当然，实际的 HR 应用还会有更多组件；可能还有其他 bean 来建立薪酬和福利记录、安排培训等。

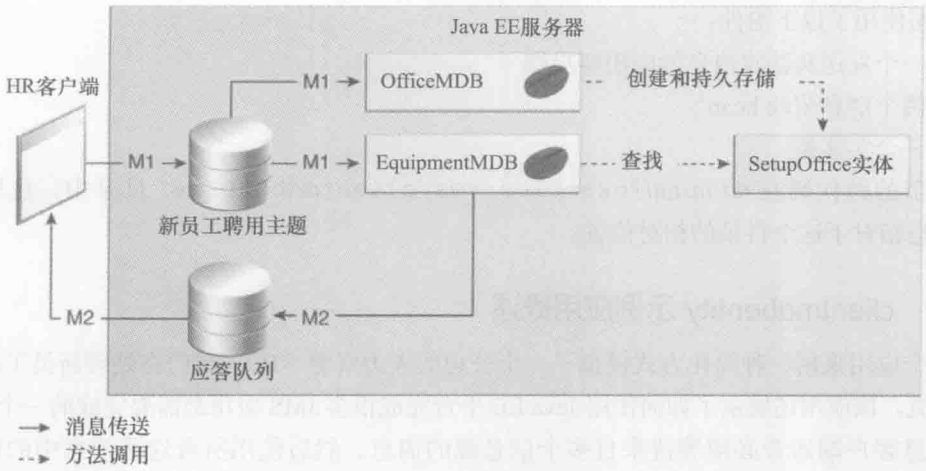


图 17-5 企业 bean 应用：客户端到消息驱动 bean 再到实体

图 17-5 假设 OfficeMDB 是第一个使用客户端消息的消息驱动 bean。OfficeMDB 创建并持久存储 SetupOffice 实体，然后存储办公室信息。接下来，EquipmentMDB 查找到这个实体，存储设备信息，确定实体已经完成工作。然后 EquipmentMDB 向应答队列发送消息，并删除这个实体。

17.8.2 为 clientmdbentity 示例编写应用组件

编写这个应用的组件包括编写应用客户端、消息驱动 bean 和实体类。

17.8.2.1 编写应用客户端：HumanResourceClient.java

应用客户端 HumanResourceClient.java 在 clientmdbentity-app-client 下，完成以下步骤：

1. 为应用定义一个主题，这里使用 java:app 命名空间，因为这个主题在应用客户端和 EJB 模块中都要使用。

2. 注入 ConnectionFactory 和 Topic 资源。

3. 创建一个 TemporaryQueue，根据发布的新员工聘用事件接收处理通知。

4. 为 TemporaryQueue 创建一个 JMSConsumer，设置 JMSConsumer 的消息监听器，并启动连接。

5. 创建一个 MapMessage。

6. 用随机生成的名字、职位和 ID 号（依序）创建 5 个新员工，并发布 5 个包含相应信息的消息。

消息监听器 HRListener 等待消息，这些消息中包含为每个员工指定的办公室和设备。一个消息到达时，消息监听器显示接收到的信息，并确定是否 5 个消息都已经到达。如果 5 个消息都已经到达，消息监听器就通知 main 方法，然后退出。

17.8.2.2 编写 clientmdbentity 示例的消息驱动 bean

这个例子使用了两个消息驱动 beans，都放在 clientmdbentity-ejb 下：

- EquipmentMDB.java
- OfficeMDB.java

这些 bean 完成以下步骤：

1. 注入一个 MessageDrivenContext 资源、一个 EntityManager 和一个 JMSContext。

2. onMessage 方法获取消息中的信息。EquipmentMDB 的 onMessage 方法根据新员工的职位选择设备；OfficeMDB 的 onMessage 方法随机生成一个办公室编号。

3. 为模拟真实世界中的处理拖延情况，这里稍做延迟，然后 onMessage 方法调用一个辅助方法 compose。

4. compose 方法完成以下步骤：

   a. 创建并持久存储 SetupOffice 实体，或者根据主键查找这个实体。

   b. 使用这个实体将设备或办公室信息存储在数据库中，为此要调用 doEquipmentList 或 doOfficeNumber 业务方法。

   c. 如果业务方法返回 true，这说明所有信息都已经保存，则从消息获取应答目的地，创建一个 JMSProducer，并发送一个应答消息，其中包含实体中存储的信息。

   d. 删除实体。

17.8.2.3 编写 clientmdbentity 示例的实体类

SetupOffice.java 类也放在 clientmdbentity-ejb 下，这是一个实体类。实体和消息驱动 bean 都打包在一个 EJB JAR 文件中。实体类声明如下：

```
@Entity
public class SetupOffice implements Serializable {
```

这个类包含一个无参数构造函数，另外还有一个构造函数有两个参数（员工 ID 和员工名字）。这个实体类还包含员工 ID、名字、办公室编号和设备列表的获取方法和设置方法。员工 ID 的获取方法增加了 @Id 注解，指示这个字段是主键：

```
@Id
public String getEmployeeId() {
    return id;
}
```

这个类还实现了两个业务方法 doEquipmentList 和 doOfficeNumber，以及它们的辅助方法 checkIfSetupComplete。

消息驱动 bean 调用业务方法和获取方法。

这个实体的 persistence.xml 文件指定了最基本的设置：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
```

```
            xmlns="http://xmlns.jcp.org/xml/ns/persistence"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
              http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="clientmdbentity-ejbPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    <properties>
      <property name="eclipselink.ddl-generation"
                value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

17.8.3 运行 clientmdbentity 示例

可以使用 NetBeans IDE 或 Maven 来构建、部署和运行 clientmdbentity 示例。

由于这个示例定义了自己的应用私有的主题，并使用预配置的默认连接工厂 java:comp/DefaultJMSConnectionFactory 和预配置的默认 JDBC 资源 java:comp/Default-DataSource，所以不需要为它创建资源。

17.8.3.1 使用 NetBeans IDE 运行 clientmdbentity

1. 确保已启动 GlassFish 服务器（参见 2.2 节）以及数据库服务器（参见 2.4 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

   tut-install/examples/jms/clientmdbentity

4. 选择 clientmdbentity 文件夹。
5. 选中 Open Required Projects（打开必要工程）复选框，然后单击 Open Project。
6. 在 Projects(工程）标签页中，右键单击 clientmdbentity 工程并选择 Build(构建)。

这个命令创建以下组件：

- 一个应用客户端 JAR 文件，其中包含客户端类和监听器类文件，另外还包含一个指定主类的清单文件。
- 一个 EJB JAR 文件，其中包含消息驱动 bean 和实体类以及 persistence.xml 文件。
- 一个应用 EAR 文件，其中包含两个 JAR 文件以及一个 application.xml 文件。

clientmdbentity.ear 文件创建在 clientmdbentity-ear/target/ 目录中。

然后这个命令部署 EAR 文件，获取客户端桩，并运行应用客户端。

17.8.3.2 使用 Maven 运行 clientmdbentity

1. 确保已启动 GlassFish 服务器（参见 2.2 节）以及数据库服务器（参见 2.4 节）。
2. 在终端窗口中，进入以下目录：

```
tut-install/examples/jms/clientmdbentity/
```

3. 要编译源文件，并打包、部署和运行应用，输入以下命令：

```
mvn install
```

这个命令会创建以下组件：

- 一个应用客户端 JAR 文件，其中包含客户端类和监听器类文件，另外还包含一个指定主类的清单文件。
- 一个 EJB JAR 文件，其中包含消息驱动 bean 和实体类以及 persistence.xml 文件。
- 一个应用 EAR 文件，其中包含两个 JAR 文件以及一个 application.xml 文件。

然后这个命令部署 EAR 文件，获取客户端桩，并运行应用客户端。

17.8.3.3 查看应用输出

NetBeans IDE 输出窗口或终端窗口中的输出如下（前面是应用客户端容器输出和 Maven 输出）：

```
SENDER: Setting hire ID to 50, name Bill Tudor, position Programmer
SENDER: Setting hire ID to 51, name Carol Jones, position Senior Programmer
SENDER: Setting hire ID to 52, name Mark Wilson, position Manager
SENDER: Setting hire ID to 53, name Polly Wren, position Senior Programmer
SENDER: Setting hire ID to 54, name Joe Lawrence, position Director
Waiting for 5 message(s)
New hire event processed:
  Employee ID: 52
  Name: Mark Wilson
  Equipment: Tablet
  Office number: 294
Waiting for 4 message(s)
New hire event processed:
  Employee ID: 53
  Name: Polly Wren
  Equipment: Laptop
  Office number: 186
Waiting for 3 message(s)
New hire event processed:
  Employee ID: 54
  Name: Joe Lawrence
  Equipment: Mobile Phone
  Office number: 135
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 50
  Name: Bill Tudor
  Equipment: Desktop System
  Office number: 200
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 51
```

```
Name: Carol Jones
Equipment: Laptop
Office number: 262
```

消息驱动 bean 和实体类的输出会出现在服务器日志中。

对于每个员工，应用首先创建实体，然后找到这个实体。服务器日志中可能会看到运行时错误，而且可能出现事务回滚。如果两个消息驱动 bean 同时发现实体还不存在，于是它们都试图创建实体，此时就会出现错误。第一个尝试会成功，不过，第二个尝试就会失败，因为这个实体已经存在。回滚之后，第二个消息驱动 bean 再次尝试，并能成功地找到这个实体。尽管会有这些错误，不过容器托管事务允许应用正确运行，而不需要特殊编程。

运行结束之后，可以使用 Services（服务）标签页或执行 `mvn cargo:undeploy` 命令取消部署应用。

17.9 使用 NetBeans IDE 创建 JMS 资源

编写你自己的 JMS 应用时，需要为应用创建资源。这一节解释如何使用 NetBeans IDE 创建 `src/main/setup/glassfish-resources.xml` 文件，该文件与本章其他例子中使用的资源文件类似。这里还将解释如何使用 NetBeans IDE 删除资源。

还可以使用管理控制台或 `asadmin create-jms-resource`、`asadmin list-jms-resources` 和 `asadmin delete-jms-resources` 命令创建、列出和删除资源。有关的信息，参考 GlassFish 服务器文档，或者输入 `asadmin help` *command-name*。

17.9.1 使用 NetBeans IDE 创建 JMS 资源

按照以下步骤使用 NetBeans IDE 在 GlassFish 服务器中创建一个 JMS 资源。要为需要的每一个资源重复这些步骤。

1. 右键单击想要创建资源的工程，并选择 New（新建），然后选择 Other（其他）。

2. 在 New File（新建文件）向导中，在 Categories（类别）下选择 GlassFish。

3. 在 File Types（文件类型）下选择 JMS Resource（JMS 资源）。

4. 在 General Attributes（一般属性）- JMS Resource（JMS 资源）页面上，在 JNDI Name（JNDI 名）域中输入资源名。

按约定，JMS 资源名以 `jms/` 开头。

5. 选择资源类型的选项。

正常情况下，这可以是 `javax.jms.Queue`、`javax.jms.Topic` 或 `javax.jms.ConnectionFactory`。

6. 单击 Next（下一步）。

7. 在队列或主题的 JMS Properties（JMS 属性）页面上，在 Name（名）属性的 Value（值）

域中输入一个实际队列名。

这个域是必填域，可以输入任意的值。

连接工厂没有必要的属性。不过有些情况下可能需要指定一个属性。

8. 单击 Finish（完成）。

这会在 Maven 工程中创建一个名为 `glassfish-resources.xml` 的文件，放在 `src/main/setup/` 目录中。在 Projects（工程）标签页中，可以在 Other Sources（其他来源）节点下找到这个文件。需要运行 `asadmin add-resources` 命令在 GlassFish 服务器中创建资源。

17.9.2 使用 NetBeans IDE 删除 JMS 资源

1. 在 Services（服务）标签页中，展开 Servers（服务器）节点，然后展开 GlassFish Server（GlassFish 服务器）节点。

2. 展开 Resources（资源）节点，然后展开 Connector Resources（连接器资源）节点。

3. 展开 Admin Object Resources（管理对象资源）节点。

4. 右键单击要删除的目的地，并选择 Unregister（取消注册）。

5. 展开 Connector Connection Pools（连接器连接池）节点。

6. 右键单击对应所删除连接工厂的连接池，并选择 Unregister（取消注册）。

删除一个连接器连接池时，相关的连接器资源也会删除。这个动作会删除连接工厂。

第五部分 Part 5

安　全

第五部分介绍安全概念和示例，包括以下各章：

- 第 18 章　Java EE 平台安全介绍
- 第 19 章　Web 应用安全入门
- 第 20 章　企业应用安全入门
- 第 21 章　Java EE 安全：高级主题

Chapter 18 第 18 章

Java EE 平台安全介绍

本章介绍一些基本的安全概念和安全机制。有关这些概念和机制的更多信息可以参见 Java EE 7 规范中涉及安全的章节。

这一部分另外几章将讨论 Web 层和企业层应用中的安全需求。

- 第 19 章解释如何为 Web 组件增加安全性，如 servlet。
- 第 20 章解释如何为 Java EE 组件增加安全性，如企业 bean 和应用客户端。

本章将介绍以下内容：

- Java EE 安全概述
- 安全机制
- 保护容器安全
- 保护 GlassFish 服务器安全
- 使用安全域、用户、组和角色
- 使用 SSL 建立安全连接
- 有关安全性的更多信息

18.1 Java EE 安全概述

如果企业包含可能由多个用户访问的敏感资源，或者包含的资源可能访问未受保护的开放网络（如 Internet），这些企业就需要进行保护。

企业层和 Web 层应用由部署在多个容器中的组件组成。这些组件结合在一起来构建一个多层企业应用。组件的安全由其容器提供。容器提供两类安全：声明式安全和编程式安全。

- **声明式安全**（Declarative security）使用部署描述文件或注解来描述应用组件的安全需求。

部署描述文件是应用外部的一个 XML 文件，可以描述应用的安全结构，包括安全角色、访问控制和认证需求。有关部署描述文件的更多信息，参见 18.3.2 节。

注解也称为元数据，可以在类文件中指定有关安全的信息。部署应用时，可以使用这个信息，也可能被应用部署描述文件覆盖。通过使用注解，可以免于在 XML 描述文件中编写声明信息。实际上，只需要在代码上增加注解，就会自动生成所需的信息。在本书中，会尽可能使用注解来保护应用的安全。有关注解的更多信息，参见 18.3.1 节。

- **编程式安全**（Programmatic security）内嵌在应用中，用来完成安全决策。如果声明式安全单独不足以描述一个应用的安全模型，编程式安全就很有用。有关编程式安全的更多信息，参见 18.3.3 节。

18.1.1　简单的应用安全演示

通过分析一个包含 Web 客户端、用户界面和企业 bean 业务逻辑的简单应用中发生了什么，可以更好地了解 Java EE 环境的安全行为。

在下面的例子中（这个例子取自 Java EE 7 规范），Web 客户端以 Web 服务器作为它的认证代理，从客户端收集用户认证数据，并用这些认证数据建立一个认证会话。

18.1.1.1　第 1 步：初始请求

在这个例子的第一步中，Web 客户端请求主应用 URL。这个动作如图 18-1 所示。

由于客户端还没有向应用环境完成认证，因此负责传送应用的 Web 部分的服务器（后面简称为 Web 服务器）会检测到这一点，并为这个资源调用适当的认证机制。有关这些机制的更多信息可以参见 18.2 节。

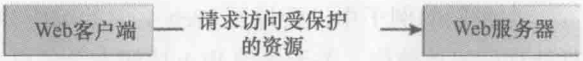


图 18-1　初始请求

18.1.1.2　第 2 步：初始认证

Web 服务器返回一个表单，Web 客户端使用这个表单从用户收集认证数据，如用户名和密码。Web 客户端将这些认证数据转发给 Web 服务器，并由 Web 服务器验证其合法性，如图 18-2 所示。验证机制可能是服务器的本地验证机制，也可能会利用底层的安全服务。在验证的基础上，Web 服务器为用户设置一个凭据。

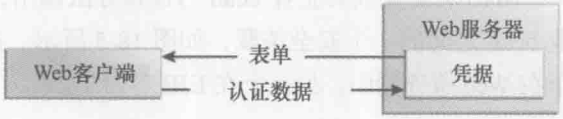


图 18-2　初始认证

18.1.1.3　第 3 步：URL 授权

这个凭据将用来确定用户是否得到授权来访问它请求的受限资源。Web 服务器会参考与 Web 资源相关的安全策略，确定允许访问这个资源的安全角色。安全策略由注解或由部署描述文件得到。Web 容器再对各个角色检查用户的凭据，确定是否能够将用户映射到这个角色。

图 18-3 显示了这个过程。

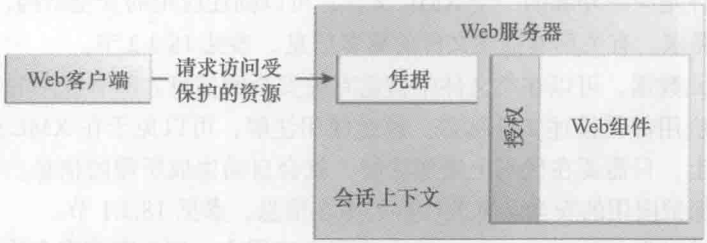


图 18-3 URL 授权

如果 Web 服务器能够将用户映射到一个角色，则会得到一个 is authorized（已授权）结果。如果 Web 服务器无法将用户映射到任何合法的角色，则得到 not authorized（未授权）结果。

18.1.1.4 第 4 步：完成初始请求

如果用户得到授权，则 Web 服务器返回原 URL 请求的结果，如图 18-4 所示。

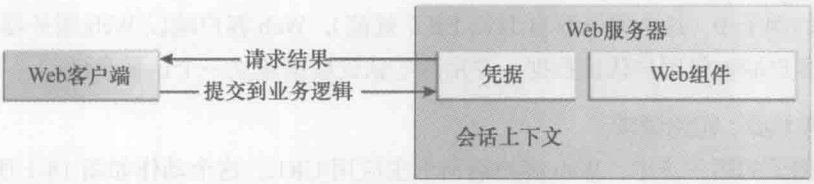


图 18-4 完成初始请求

在我们的例子中，会返回 Web 页面的响应 URL，允许用户提交需要由应用业务逻辑组件处理的表单数据。关于保护 Web 应用安全的更多信息参见第 19 章。

18.1.1.5 第 5 步：调用企业 bean 业务方法

Web 页面完成对企业 bean 的远程方法调用，为此要使用用户的凭据建立 Web 页面与企业 bean 之间的一个安全关联，如图 18-5 所示。这个关联实现为两个相关的安全上下文：一个在 Web 服务器中，另一个在 EJB 容器中。

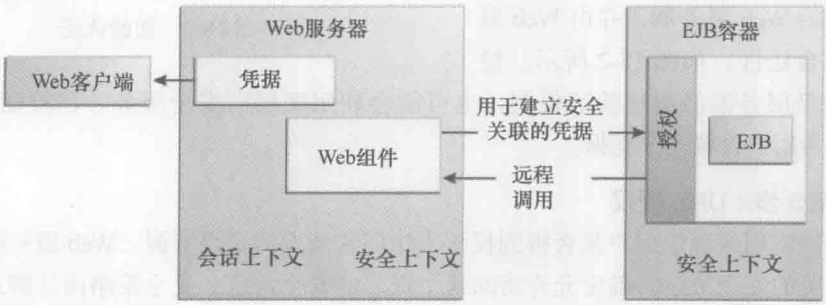


图 18-5 调用企业 bean 业务方法

EJB 容器负责执行对企业 bean 方法的访问控制。容器参考与企业 bean 关联的安全策略，确定哪些安全角色允许访问这个方法。安全策略由注解或由部署描述文件得到。对于每个角色，EJB 容器通过使用与调用关联的安全上下文来确定是否能够将调用者映射到该角色。

如果容器能够将调用者的凭据映射到一个角色，则会得到一个 is authorized（已授权）结果。如果容器无法将调用者映射到任何合法角色，则得到 not authorized（未授权）结果。“未授权”结果导致容器抛出一个异常，并传回调用 Web 页面。

如果调用得到授权，容器将控制分派到企业 bean 方法。这个 bean 的调用执行结果将返回到 Web 页面，并最终由 Web 服务器和 Web 客户端返回用户。

18.1.2　安全机制特性

合理实现的安全机制可以提供以下功能：

- 防止未授权访问应用功能和业务或个人数据（认证）
- 保证系统用户对他们完成的操作负责（非抵赖）
- 保护系统不会遭遇服务中断或受到其他破坏而影响服务质量

理想情况下，合理实现的安全机制还有以下特性：

- 易于管理
- 对系统用户透明
- 可以跨应用和企业边界互操作

18.1.3　应用安全特点

Java EE 应用由组件组成，可能包含受保护和不受保护的资源。通常情况下，需要保护资源来确保只有授权用户能够访问这些资源。授权可以为受保护资源提供受控的访问。授权基于身份鉴别和认证。**身份鉴别**（Identification）是系统识别一个实体的过程，**认证**（authentication）是验证计算机系统中一个用户、设备或其他实体身份的过程，通常是允许访问系统资源的前提条件。

实体访问未受保护资源时，并不需要授权和认证。不经认证访问一个资源称为未认证或匿名访问。

如果能够充分实现，那么应用安全特点有助于尽量减少企业面对的安全威胁，包括：

- **认证**：通过认证，通信实体（如客户端和服务器）可以向对方证明它们是已得到访问授权的特定身份。这可以确保用户确实是其声称的那个用户。
- **授权**（Authorization）或**访问控制**（access control）：通过授权或访问控制，与资源的交互仅限于某些用户或程序集合，以保证完整性、机密性或可用性约束。这可以确保用户确实得到了完成操作或访问数据的授权。
- **数据完整性**（Data integrity）：可以用来证明信息未被第三方修改，第三方是指信息源以外的某个实体。例如，一个接收方接收通过一个开放网络发送的数据时，必须能够

检测并丢弃发送后被修改的消息。这样可以确保只有授权的用户才能修改数据。

- **机密性**（Confidentiality）或**数据私密性**（data privacy）：可以用来确保只有得到访问授权的用户才能得到这个信息。这样可以确保只有授权用户才可以查看敏感数据。
- **非抵赖性**（Non-repudiation）：可以用来证明一个已完成某个动作的用户不能抵赖自己的行为。这样可以确保能证明事务确实发生。
- **服务质量**（Quality of Service）：基于多种技术可以对选定的网络业务流提供更好的服务。
- **审计**（Auditing）：可以用来捕获安全相关事件的抗攻击记录，从而能够评估安全策略和机制的有效性。为支持这一点，系统会维护事务和安全信息的一个记录。

18.2 安全机制

在确定为应用提供的安全层和安全类型时，应当考虑应用的特点。下面各小节将讨论可以用来保护 Java EE 应用安全的常用机制的特点。各种机制可以单独使用，也可以根据实现的特定需要与其他机制一同来提供保护层。

18.2.1 Java SE 安全机制

Java SE 为多种安全特性和机制提供了支持。

- **Java 认证和授权服务**（Java Authentication and Authorization Service，JAAS）是一组 API，支持服务对用户完成认证和访问控制。JAAS 提供了一个可插式、可扩展的框架，可实现编程式用户认证和授权。JAAS 是一个核心 Java SE API，这也是多种 Java EE 安全机制的底层技术。
- **Java 通用安全服务**（Java Generic Security Services，Java GSS-API）是一个基于 token 的 API，用于在通信应用之间安全地交换消息。GSS-API 允许应用程序员在多种底层安全机制（包括 Kerberos）之上采用统一的方式访问安全服务。
- **Java 加密扩展**（Java Cryptography Extension，JCE）为加密、密钥生成和密钥协议以及消息认证码（Message Authentication Code，MAC）算法提供了一个框架和多个实现。加密支持包括对称加密、非对称加密、块加密和流加密。块加密处理字节组；流加密则一次处理一个字节。这个软件还支持安全流和密封对象。
- **Java 安全套接字扩展**（Java Secure Sockets Extension，JSSE）为 Java 版本的安全套接字层（Secure Sockets Layer，SSL）和传输层安全（Transport Layer Security，TLS）协议提供了一个框架和一个实现，包括数据加密、服务器认证、消息完整性和客户端认证（可选）等功能，以支持安全的 Internet 通信。
- **简单认证和安全层**（Simple Authentication and Security Layer，SASL）是一个 Internet 标准（RFC 2222），指定了一个认证协议，并在客户端和服务器应用之间建立一个安全层（可选）。SASL 定义了如何交换认证数据，但是本身没有指定这个数据的内容。

SASL 是一个框架，涵盖多个特定的认证机制，这些认证机制指定了具体的认证数据内容和语义。

Java SE 还提供了一组工具来管理密钥库、证书和策略文件；生成和验证 JAR 签名；获得、列举和管理 Kerberos 票据。

有关 Java SE 安全的更多信息，请访问：`http://docs.oracle.com/javase/7/docs/technotes/guides/security/`。

18.2.2 Java EE 安全机制

Java EE 安全服务由组件容器提供，可以使用声明式或编程式技术来实现（参见 18.3 节）。Java EE 安全服务提供了一个健壮而且易于配置的安全机制，可以完成用户认证以及授权访问多个不同层上的应用功能和相关数据。Java EE 安全服务与操作系统的安全机制有所区别。

18.2.2.1 应用层安全

在 Java EE 中，组件容器负责提供应用层安全，根据应用的需要为特定应用类型提供安全服务。在应用层，应用防火墙可以用来增强应用保护，保护通信流和所有相关的应用资源不受攻击。

Java EE 安全很容易实现和配置，可以对应用功能和数据提供细粒度的访问控制。不过，应用层安全有一个固有的性质，安全属性不能转移到其他环境中运行的应用，只有处在这个应用环境时才能保护数据。在一个传统的企业应用上下文中，这可能不算是个问题，不过对于一个 Web 服务应用，由于数据通常会跨多个中间节点传递，因此需要使用 Java EE 安全机制以及传输层安全和消息层安全来提供一个完整的安全解决方案。

使用应用层安全的优点包括：

- 特别针对应用的具体需要提供安全性。
- 根据应用特定的设置，提供细粒度安全性。

使用应用层安全的缺点包括：

- 应用可能依赖一些无法在不同应用类型之间转移的安全属性。
- 由于支持多个协议，使得这种安全性很脆弱。
- 数据成为应用的脆弱点。

有关在应用层提供安全性的更多信息，参见 18.3 节。

18.2.2.2 传输层安全

客户端和提供者之间使用一些传输机制通过网络传输信息，这些传输机制会提供传输层安全；因此，传输层安全依赖于使用安全套接字层（SSL）的安全 HTTP 传输（secure HTTP transport，HTTPS）。传输层安全是一个点对点安全机制，可以用于提供认证、消息完整性和机密性。在一个 SSL 保护的会话中运行时，应用协议传输或接收第一个数据字节之前，服务

器和客户端可以相互认证，并协商一个加密算法以及加密密钥。从数据离开客户端开始直到它到达其目的地，或者反过来，从离开服务器开始直到到达客户端，都会启用安全，甚至可以跨中间节点。不过还存在一个问题，一旦到达目的地，数据就不再受保护。对此的一种解决方案是在发送前对消息加密。

传输层安全分多个阶段完成，包括：

- 客户端和服务器协商一个合适的算法。
- 使用公钥加密和基于证书的认证交换一个密钥。
- 交换信息时使用对称加密。

运行使用 SSL 的 HTTPS 时必须有数字证书。除非已经安装了一个数字证书，否则大多数 Web 服务器的 HTTPS 服务不会运行。已经为 GlassFish 服务器创建了数字证书。

使用传输层安全的优点包括：

- 这是一种相对简单、容易理解的标准技术。
- 既应用于消息体，也可以应用于消息附件。

使用传输层安全的缺点包括：

- 它与传输层协议紧耦合。
- 这是一种全有或全无的安全方法。这说明，该安全机制不关心消息内容，所以不能像消息层安全那样有选择地对消息的某些部分应用安全。
- 保护是临时的。消息只有在正在传输时才会得到保护。一旦端点接收到消息，就会自动取消保护。
- 这不是一种端到端解决方案，只是点对点解决方案。

有关传输层安全的更多信息，参见 18.6 节。

18.2.2.3 消息层安全

在消息层安全中，安全信息包含在 SOAP 消息和 / 或 SOAP 消息附件中，使得安全信息可以随消息或附件传输。例如，发送者可以对消息的一部分签名，还可以为某个特定的接收者加密。从初始发送者发出时，消息在到达其目标接收者之前可能会经过多个中间节点。在这种情况下，加密部分对所有中间节点仍是不透明的，只能由目标接收者解密。出于这个原因，消息层安全有时也称为一种端到端安全。

使用消息层安全的优点包括：

- 消息传输过程中以及消息到达其目的地之后都能提供安全保护。
- 可以对消息的不同部分有选择地应用安全，如果使用了 XML Web 服务安全，还可以对附件应用安全。
- 可以对多跳传输中的中间节点应用消息安全。
- 消息安全独立于应用环境或传输协议。

使用消息层安全的缺点在于它相当复杂，而且增加了一些处理开销。

GlassFish 服务器使用 Metro 支持消息安全，这是一个 Web 服务栈，使用 Web 服务安全（Web Services Security，WSS）来保护消息安全。由于这个消息安全特定于 Metro，不是 Java EE 平台的一部分，所以本书不讨论如何使用 WSS 保护消息安全。更多信息参见《*Metro User's Guide*》(`https://metro.java.net/guide/`)。

18.3　保护容器安全

在 Java EE 中，组件容器负责提供应用安全。容器提供了两类安全：声明式安全和编程式安全。

18.3.1　使用注解指定安全信息

注解支持声明式编程，所以涵盖了声明式和编程式安全概念。用户可以在类文件中通过使用注解来指定关于安全的信息。部署应用时，GlassFish 服务器会使用这个信息。不过，并不是所有安全信息都可以通过使用注解来指定。有些信息必须在应用部署描述文件中指定。

20.2.1 节中将介绍可以在企业 bean 类文件中用来指定安全信息的注解。第 19 章会介绍如何使用注解来尽可能地保护 Web 应用安全。必要时还会介绍部署描述文件。

关于注解的更多信息，参见 18.7 节。

18.3.2　使用部署描述文件提供声明式安全

声明式安全是指通过使用部署描述文件来描述一个应用组件的安全需求。由于部署描述文件信息是声明式的，所以可以改变这些信息而无需修改源代码。运行时，Java EE 服务器读取部署描述文件，并相应地设置应用、模块或组件。如果注解中没有提供安全信息，或者如果要使用非默认设置，则部署描述文件必须为各个组件提供一些结构信息。

本书这一部分不会介绍如何创建部署描述文件；这里只介绍部署描述文件中与安全有关的元素。NetBeans IDE 提供了一些工具来创建和修改部署描述文件。

对于不同类型的组件，其部署描述文件会使用不同的格式或模式。本书讨论的部署描述文件的安全元素包括：

- Web 组件可以使用一个 Web 应用部署描述文件 `web.xml`。

Web 组件部署描述文件的模式在 Java Servlet 3.1 规范（JSR 340）的第 14 章提供，可以从以下地址下载：`http://jcp.org/en/jsr/detail?id=340`。

- 企业 JavaBeans 组件可以使用一个 EJB 部署描述文件 `META-INF/ejb-jar.xml`，包含在 EJB JAR 文件中。

企业 bean 部署描述文件的模式在 EJB 3.2 核心契约和需求规范（JSR 345）的第 14 章提供，可以从以下地址下载：`http://jcp.org/en/jsr/detail?id=345`。

18.3.3 使用编程式安全

编程式安全内嵌在应用中，用来完成安全决策。如果声明式安全单独不足以描述一个应用的安全模型，编程式安全就很有用。编程式安全 API 由 `EJBContext` 接口和 `HttpServletRequest` 接口的方法组成。利用这些方法，组件可以根据调用者或远程用户的安全角色完成业务逻辑决策。

编程式安全将在以下小节更详细地讨论：

- 19.3 节
- 20.2.2 节

18.4 保护 GlassFish 服务器安全

本书介绍了如何在 GlassFish 服务器上部署应用，它基于 Java EE 安全模型提供了高安全、可互操作的分布式组件计算。GlassFish 服务器支持 Java EE 7 安全模型。可以配置 GlassFish 服务器完成以下工作。

- 增加、删除或修改授权用户。有关这个主题的更多信息，参见 18.5 节。
- 配置安全 HTTP 和 Internet Inter-Orb Protocol（IIOP）监听器。
- 配置安全 Java 管理扩展（Java Management Extensions，JMX）连接器。
- 增加、删除或修改已有或定制安全域。
- 使用 Java 容器授权契约（Java Authorization Contract for Containers，JACC）为可插式授权提供者定义一个接口。JACC 定义了 GlassFish 服务器与授权策略模型之间的安全契约。这些契约指定了如何安装、配置以及在访问决策中使用授权提供者。
- 使用可插式审计模块。
- 定制认证机制。Java EE 7 兼容 Web 容器的所有实现都要求支持 JSR 196 的 Servlet Profile，它提供了一种定制认证机制的途径，可以定制 Web 容器为应用提供的认证机制。
- 设置和改变应用的策略权限。

18.5 使用安全域、用户、组和角色

通常需要保护资源来确保只有授权用户可以访问这些资源。18.1.3 节中对认证、身份鉴别和授权做了介绍。

这一节讨论如何建立用户，以便正确地完成身份鉴别，从而允许访问受保护的资源，或者如果未授权访问受保护的资源，则要拒绝其访问。要认证一个用户，需要完成以下基本步骤：

1. 应用开发人员编写代码，提示用户输入一个用户名和密码。19.2.2 节中将讨论多种不同的认证方法。

2. 应用开发人员使用一个元数据注解或部署描述文件来指定如何为已部署的应用建立安全。这个步骤将在 18.5.3 节中讨论。

3. 服务器管理员在 GlassFish 服务器中建立授权用户和组。这将在 18.5.2 节中讨论。

4. 应用部署人员将应用的安全角色映射到 GlassFish 服务器中定义的用户、组和主体。这个主题将在 18.5.4 节中讨论。

18.5.1　什么是安全域、用户、组和角色

安全域（realm）是为 Web 或应用服务器定义的一个安全策略域。安全域包含一个用户集合，这些用户可能分配到一个组，也可能没有分配到组。如何在 GlassFish 服务器中管理用户将在 18.5.2 节中讨论。

应用在允许访问一个受保护的资源之前，通常会提示用户输入一个用户名和密码。输入用户名和密码后，这个信息会传递到服务器，服务器将认证用户，并发送这个受保护的资源，也可能无法认证用户，在这种情况下，则拒绝访问受保护的资源。这种类型的用户认证将在 19.2.3 节中讨论。

在一些应用中，会为授权用户分配某些角色。在这种情况下，分配到应用中用户的角色必须映射到应用服务器上定义的一个主体或组，如图 18-6 所示。有关角色如何映射到用户和组的更多信息，参见 18.5.3 节。

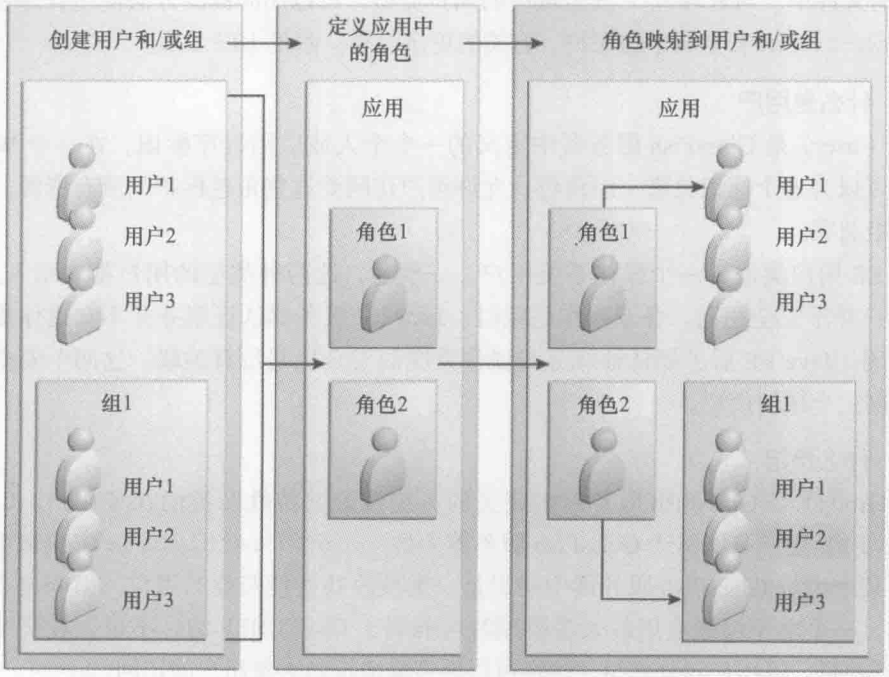


图 18-6　角色映射到用户和组

下面各小节会更详细地介绍安全域、用户、组和角色。

18.5.1.1 什么是安全域

服务器上的受保护资源可以划分到一组保护空间，每个保护空间都有自己的认证机制和/或包含用户和组集合的授权数据库。安全域（realm）是用户和组的一个完整的数据库，这些用户和组标识为一个或多个应用的合法用户，由同一个认证策略控制。

Java EE 服务器认证服务可以控制多个安全域中的用户。File、admin-realm 和 certificate 安全域是为 GlassFish 服务器预配置的安全域。

在 file 安全域中，服务器将用户凭据本地存储在一个名为 keyfile 的文件中。可以使用管理控制台管理 file 安全域中的用户。使用 file 安全域时，服务器认证服务通过检查 file 安全域验证用户身份。除了使用 HTTPS 和证书的 Web 浏览器客户端以外，所有其他客户端的认证都使用 file 安全域。

在 certificate 安全域中，服务器将用户凭据存储在一个证书数据库中。使用 certificate 安全域时，服务器结合 HTTPS 使用证书来认证 Web 客户端。要验证 certificate 安全域中一个用户的身份，认证服务需要验证一个 X.509 证书。对于如何创建这种类型的证书，参见 21.1 节，其中逐步骤地介绍了如何创建证书。X.509 证书中公共名字段用作主体名。

admin-realm 也是一个 file 安全域，将管理员用户凭据本地存储在一个名为 admin-keyfile 的文件中。与管理 file 安全域中的用户类似，可以用同样的方法使用管理控制台来管理 admin-realm 安全域中的用户。有关的更多信息，参见 18.5.2 节。

18.5.1.2 什么是用户

用户（user）是 GlassFish 服务器中定义的一个个人或应用程序标识。在一个 Web 应用中，用户可以为这个标识关联一组角色，允许用户访问受这些角色保护的所有资源。用户可以与一个组关联。

Java EE 用户类似于一个操作系统用户。一般地，这两种类型的用户都表示人。不过，这两类用户并不完全相同。登录操作系统时，Java EE 服务器认证服务并不知道你提供的用户名和密码。Java EE 服务器认证服务与操作系统的安全机制没有关联。这两个安全服务管理属于不同安全域的用户。

18.5.1.3 什么是组

组（Group）是 GlassFish 服务器中定义的一组按公共特性归类的认证用户。file 安全域中的 Java EE 用户可以属于 GlassFish 服务器中的一个组。（certificate 安全域中的用户不能属于某个组。）GlassFish 服务器中的组是一类按公共特性归类的用户，如职位或顾客简况。例如，一个电子商务应用的大多数顾客可能属于 CUSTOMER 组，不过大客户可能属于 PREFERRED 组。将用户划分到组，这样可以更容易地控制大量用户的访问。

GlassFish 服务器中的组与角色有不同的作用域。组是为整个 GlassFish 服务器指定的，

而角色只与 GlassFish 服务器中一个特定的应用关联。

18.5.1.4　什么是角色

角色（role）是一个抽象名，表示允许访问应用中一组特定的资源。可以把角色比作能打开一把锁的钥匙。很多人都可能配了这把钥匙。锁并不关心你是谁，它只关心你有没有合适的钥匙。

18.5.1.5　其他术语

描述 Java EE 平台的安全需求时还用到了以下术语：

- **主体**（principal）是一个实体，可以由企业中部署的一个安全服务的认证协议来认证。主体通过使用主体名来标识，使用认证数据来认证。
- **安全策略域**（security policy domain），也称为**安全域**（security domain 或 realm），这是一个作用域，在这个作用域上定义了一个公共的安全策略，由安全服务的安全管理员来保证执行。
- **安全属性**（Security attribute）是与每个主体关联的一组属性。安全属性有很多用途：例如，访问受保护资源和用户审计。可以由一个认证协议将安全属性与一个主体关联。
- **凭据**（credential）是包含或引用一些安全属性的对象，为 Java EE 服务认证一个主体时将使用这些安全属性。主体在认证时获得一个凭据，或者如果另一个主体允许使用其凭据，也可以从该主体获得凭据。

18.5.2　在 GlassFish 服务器中管理用户和组

运行教程示例之前，按照以下步骤管理用户。

18.5.2.1　为 GlassFish 服务器增加用户

1. 如果还没有启动 GlassFish 服务器，现在启动。

关于启动 GlassFish 服务器的信息参见 2.2 节。

2. 如果还没有启动管理控制台，现在启动。

要启动管理控制台，打开一个 Web 浏览器，并指定 URL `http://localhost:4848/`。

如果安装时改变了默认管理端口，要输入正确的端口号取代 `4848`。

3. 在导航树中，展开 Configurations（配置）节点，然后展开 server-config（服务器配置）节点。

4. 展开 Security（安全）节点。

5. 展开 Realms（安全域）节点。

6. 选择要增加用户的安全域。

- 如果希望所增加的用户访问 `file` 安全域中运行的应用，可以选择 `file` 安全域。

对于这里的安全应用示例，要选择 `file` 安全域。

- 如果希望所增加的用户作为 GlassFish 服务器的系统管理员，则选择 `admin-realm`。

使用管理控制台无法将用户增加到 certificate 安全域，只能在 certificate 安全域中增加证书。有关向 certificate 安全域增加（导入）证书的更多信息，参见 21.1.2 节。

7. 在编辑安全域页面，单击 Manage Users（管理用户）。

8. 在文件用户或管理员用户页面，单击 New（新建），向这个安全域增加一个新用户。

9. 在新建文件安全域用户页面，在 User ID（用户 ID）、Group List（组列表）、New Password（新密码）和 Confirm New Password（确认新密码）域中输入值。

对于 Admin 安全域，Group List（组列表）域是只读的，组名为 asadmin。向 Admin 安全域增加一个用户后，重启 GlassFish 服务器和管理控制台。

关于这个属性的更多信息，参见 18.5 节。

对于这里的安全应用示例，可以用你喜欢的任何名字和密码指定一个用户，不过要确保将用户分配到组 TutorialUser。用户名和密码是区分大小写的。把用户名和密码记录下来，以便在本书后面的例子中使用。

10. 单击 OK 将用户增加到这个安全域，或者单击 Cancel 退出而不保存。

18.5.3 建立安全角色

设计一个企业 bean 或 Web 组件时，通常要考虑哪些用户要访问这个组件。例如，如果有一个人力资源部门的 Web 应用，对于 DEPT_ADMIN 角色的用户和 DIRECTOR 角色的用户可能会有不同的请求 URL。DEPT_ADMIN 角色允许查看员工数据，而 DIRECTOR 角色允许修改员工数据，包括工资数据。这些安全角色分别是一个抽象的用户逻辑分组，由装配应用的人定义。部署应用时，部署人员将把这些角色映射到操作环境中的安全身份，如图 18-6 所示。

对于 Java EE 组件，要使用 @DeclareRoles 和 @RolesAllowed 元数据注解定义安全角色。

下面的应用示例中，DEPT-ADMIN 角色得到授权可以访问查看员工工资数据的方法，DIRECTOR 得到授权可以访问改变员工工资数据的方法。

下面的代码显示了如何为企业 bean 增加注解：

```
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
...
@DeclareRoles({"DEPT-ADMIN", "DIRECTOR"})
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;

    @RolesAllowed("DEPT-ADMIN")
    public void reviewEmployeeInfo(EmplInfo info) {
      oldInfo = ... read from database;

      // ...
    }
```

```
    @RolesAllowed("DIRECTOR")
    public void updateEmployeeInfo(EmplInfo info) {

        newInfo = ... update database;
        // ...
    }
    ...
}
```

对于 servlet，可以在 @ServletSecurity 注解中使用 @HttpConstraint 注解来指定哪些角色可以访问这个 servlet。例如，可以如下为一个 servlet 增加注解：

```
@WebServlet(name = "PayrollServlet", urlPatterns = {"/payroll"})
@ServletSecurity(
@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
    rolesAllowed = {"DEPT-ADMIN", "DIRECTOR"}))
public class GreetingServlet extends HttpServlet {
```

这些注解将在 19.4.2.1 节和 20.2.1 节中更详细地讨论。

如果用户提供了登录信息，而且应用已经声明了哪些角色得到授权可以访问应用中受保护的部分，下一步就是将安全角色映射到一个用户或主体的名。

18.5.4　角色映射到用户和组

开发一个 Java EE 应用时，你不需要知道将要运行应用的安全域中定义了哪些类型的用户。在 Java EE 平台中，安全架构提供了一个机制，可以将应用中定义的角色映射到运行时环境安全域中定义的用户或组。

应用中使用的角色名通常与 GlassFish 服务器中定义的组名相同。在这些情况下，通过使用管理控制台，可以启用 GlassFish 服务器中默认的主体 – 角色映射。19.4.1 节中解释了如何启用默认映射。本书中所有安全示例都使用默认的主体 – 角色映射。启用这个设置后，如果 GlassFish 服务器中定义的组名与应用中定义的角色名匹配，就没有必要再使用运行时环境部署描述文件来提供映射。只要组名和角色名匹配，应用服务器就会隐式地完成映射。

如果应用中使用的角色名与服务器中定义的组名不相同，则可以使用运行时环境部署描述文件来指定映射。下面的例子展示了如何在 glassfish-web.xml 文件中指定这个映射，这也是 Web 应用使用的部署描述文件：

```
<glassfish-web-app>
    ...
    <security-role-mapping>
        <role-name>Mascot</role-name>
        <principal-name>Duke</principal-name>
    </security-role-mapping>

    <security-role-mapping>
        <role-name>Admin</role-name>
        <group-name>Director</group-name>
```

```
    </security-role-mapping>
    ...
</glassfish-web-app>
```

一个角色可以映射到特定的主体、特定的组或者二者的组合。主体或组名必须是当前默认安全域（或者 login-config 元素中指定的安全域）中合法的主体或组。在这个例子中，应用中使用的 Mascot 角色映射到一个名为 Duke 的主体，这是应用服务器上存在的一个主体。通过将一个角色映射到一个特定的主体，当赋予了这个角色的人可能变化时，这会很有用。对于这个应用，只需要修改运行时环境部署描述文件，而不需要在整个应用中搜索和替换对这个主体的引用。

另外，在这个例子中，Admin 角色映射到组名为 Director 的用户组。这很有用，因为授权访问主管级管理数据的人员组只能在 GlassFish 服务器中维护。应用开发人员不需要知道这些人是谁，只需要定义允许访问这个信息的人员组。

role-name 属性必须匹配相应部署描述文件的 security-role 元素中的 role-name，或者要与 @DeclareRoles 注解中定义的角色名匹配。

18.6 使用 SSL 建立安全连接

安全套接字层（Secure Sockets Layer，SSL）技术是在传输层实现的安全技术（关于传输层安全的更多信息，参见 18.2.2.2 节）。SSL 允许 Web 浏览器和 Web 服务器在一个安全连接上通信。在这个安全连接中，数据发送前会加密，接收时会在处理之前解密。浏览器和服务器在发送任何数据之前会对业务流加密。

SSL 解决了以下重要的安全问题。

- **认证**（Authentication）：首次尝试在一个安全连接上与一个 Web 服务器通信时，服务器会以服务器证书的形式（也称为公钥证书）为 Web 浏览器提供一组凭据。这些证书的作用是验证网站确实是所声称的真实网站。有些情况下，服务器可能会请求一个证书，来证明客户端确实是所声称的真实客户端；这种机制称为客户端认证（client authentication）。
- **机密性**（Confidentiality）：数据在客户端和服务器之间通过网络传递时，第三方可以查看和拦截这个数据。可以对 SSL 响应加密，使得数据无法由第三方解密，以保证数据的机密性。
- **完整性**（Integrity）：数据在客户端和服务器之间通过网络传递时，第三方可以查看和拦截这个数据。SSL 有助于保证数据在传输过程中不会被第三方修改。

SSL 协议设计为在保证安全的同时尽可能高效。不过，从性能的角度讲，加密和解密是计算量相当大的过程。没有必要让整个 Web 应用都基于 SSL 运行，通常会由开发人员来确定哪些页面需要一个安全连接，而哪些不需要。可能需要安全连接的页面包括登录页面、个

人信息页面、购物车结账页面或信用卡信息传输页面。应用中任何页面都可以在地址前加 `https:` 前缀而不是 `http:`，这样就会通过一个安全套接字来请求这个页面。需要安全连接的页面要检查与页面请求关联的协议类型，如果没有指定 `https:`，则要采取适当的动作。

如果在一个安全连接上使用基于名的虚拟主机，这可能会有问题。这是 SSL 协议本身的一个设计局限。**SSL 握手**（SSL handshake）必须在访问 HTTP 请求之前完成，SSL 握手即客户端浏览器接收服务器证书。因此，在认证之前无法确定包含虚拟主机名的请求信息，也不可能为一个 IP 地址分配多个证书。如果一个 IP 地址上的所有虚拟主机都基于同一个证书完成认证，那么增加多个虚拟主机不会影响服务器上正常的 SSL 操作。不过要当心，大多数客户端浏览器将把服务器的域名与证书中所列的域名（如果有）进行比较，这主要适用于由证书授权机构（certificate authority，CA）签署的官方证书。如果域名不匹配，这些浏览器就会向客户端显示一个警告。一般地，生产环境中 SSL 通常只使用基于地址的虚拟主机。

18.6.1 验证和配置 SSL 支持

一般来讲，服务器要启用 SSL，必须处理以下问题：

- 服务器部署描述文件中必须有一个对应 SSL 连接器的 `Connector` 元素。
- 必须有合法的密钥库和证书文件。
- 必须在服务器部署描述文件中指定密钥库文件及其密码的位置。

GlassFish 服务器中已经启用了 SSL HTTPS 连接器。

为进行测试，并验证已正确安装 SSL 支持，可以用一个 URL 加载默认介绍页面，这个 URL 连接到服务器部署描述文件中定义的端口：

```
https://localhost:8181/
```

这个 URL 中的 `https` 指示浏览器应当使用 SSL 协议。这里的 `localhost` 假设目前属于开发过程，你在本地主机上运行示例。这个例子中的 `8181` 是指定的安全端口，将在这个端口上创建 SSL 连接器。如果使用一个不同的服务器或端口，则需要相应地改变这个值。

第一次加载这个应用时，会出现新站点证书或安全警告对话框。单击 **Next**（下一步）完成这一系列对话框，到达最后一个对话框时单击 **Finish**（完成）。证书只在第一次出现。接收这些证书后，以后访问这个网站就会认为你仍信任其内容。

18.7 有关安全的更多信息

关于 Java EE 应用安全的更多信息，参见：

- Java EE 7 规范：

  `http://jcp.org/en/jsr/detail?id=342`

- 企业 JavaBeans 3.2 规范：

http://jcp.org/en/jsr/detail?id=345

- 实现企业 Web 服务 1.3 规范：

  http://jcp.org/en/jsr/detail?id=109

- Java SE 安全信息：

  http://docs.oracle.com/javase/7/docs/technotes/guides/security/

- Java Servlet 3.1 规范：

  http://jcp.org/en/jsr/detail?id=340

- Java 容器授权契约 1.5 规范：

  http://jcp.org/en/jsr/detail?id=115

第 19 章 Chapter 19

Web 应用安全入门

18.3 节中概括介绍了实现 Java EE Web 应用安全的几种方法。本章将更详细地讨论这一内容，并给出几个例子来研究与 Web 组件相关的这些安全服务。

可以使用一个 Web 浏览器通过网络访问 Web 应用，如 Internet 或一个公司的内部网。1.3 节中讨论过，Java EE 平台使用一个分布式多层应用模型，Web 应用在 Web 层运行。

Web 应用包含可能由多个用户访问的资源。这些资源通常要通过不受保护的开放网络传递，如 Internet。在这样一个环境中，很多 Web 应用都需要某种类型的安全性。

对于业务层和 EIS 层中的应用及其客户端，第 20 章将讨论如何保护其安全。

本章将介绍以下内容：

- Web 应用安全概述
- 保护 Web 应用安全
- Web 应用使用编程式安全
- 示例：保护 Web 应用安全

19.1 Web 应用安全概述

在 Java EE 平台中，Web 组件为 Web 服务器提供动态扩展功能。Web 组件可以是 Java servlet 或 JSF 页面。

在 Web 容器中安装或部署应用时，可以对 Web 应用安全的某些方面进行配置。可以用注解和 / 或部署描述文件为部署人员传递有关安全性和应用的其他方面的信息。通过在注解中或部署描述文件中指定这个信息，可以帮助部署人员为 Web 应用建立适当的安全策略。部

署描述文件中显式指定的值会覆盖注解中指定的值。

Java EE Web 应用安全可以采用以下方式实现：

- **声明式安全**，使用元数据注解或应用的部署描述文件实现声明式安全。更多信息参见 18.1 节。

Web 应用的声明式安全在 19.2 节中介绍。

- **编程式安全**，内嵌在应用中，当声明式安全单独不足以描述一个应用的安全模型时，可以采用编程方式完成安全决策。在一个应用中，如果特定工作流中要求在某些条件下登录，而不是所有情况下都需要登录，这种情况下声明式安全就无法满足要求。更多信息参见 18.1 节。

Servlet 3.1 提供了 `HttpServletRequest` 接口的 `authenticate`、`login` 和 `logout` 方法。由于 Servlet 规范增加了 `authenticate`、`login` 和 `logout` 方法，因此对于 Web 应用，应用部署描述文件不再是必需的，不过仍然可以用来进一步指定基本默认值以外的安全需求。

编程式安全在 19.3 节中讨论。

- **消息安全**，消息安全用于 Web 服务，会在 SOAP 消息首部中结合一些安全特性，如数字签名和加密，消息安全作用于应用层，可以确保端到端安全。消息安全不是 Java EE 7 的一部分，这里提到消息安全只是为了建立一个全面的认识。

这一章的很多内容都建立在本书前面的内容基础上。特别是本章假设你已经熟悉以下各章的内容：

- 本书第 1 卷第 6 章
- 本书第 1 卷第 7 章
- 本书第 1 卷第 17 章
- 本书第 18 章

19.2 保护 Web 应用安全

Web 应用由应用开发人员创建，开发人员将应用提供、销售或转移到应用部署人员，由部署人员将应用安装到一个运行时环境。应用开发人员通过使用注解或部署描述文件来说明如何为所部署的应用建立安全性。这个信息传递到部署人员，部署人员再使用这个信息为安全角色定义方法权限、建立用户认证，并建立适当的传输机制。如果应用开发人员没有定义安全需求，部署人员就必须独立地确定安全需求。

并非所有 Web 应用都可以使用注解来指定 Web 应用安全的一些必要元素。这一章将解释如何尽可能使用注解来保护 Web 应用安全，另外还会解释无法使用注解时如何使用部署描述文件。

19.2.1 指定安全约束

安全约束（security constraint）通过使用 URL 映射来定义一个资源集合的访问权限。

如果你的 Web 应用使用了一个 servlet，则可以使用注解来描述安全约束信息。具体而言，可以在 `@ServletSecurity` 注解中使用 `@HttpConstraint` 和 `@HttpMethodConstraint`（可选）注解来指定安全约束。

不过，如果 Web 应用没有使用 servlet，就必须在部署描述文件中指定一个 `security-constraint` 元素。认证机制无法使用注解来描述，所以如果使用非 `BASIC`（默认）的其他认证方法，就需要有一个部署描述文件。

`security-constraint` 可以有以下子元素：

- **Web 资源集合**（`web-resource-collection`）：这是 URL 模式和 HTTP 操作的一个列表，描述了一组要保护的资源。这里的 URL 模式是 URL 的一部分，即想要约束的主机名和端口后面的部分，而 HTTP 操作是与所约束 URL 模式匹配的文件中的方法。Web 资源集合将在 19.2.1.1 节中介绍。
- **授权约束**（`auth-constraint`）：指定是否使用认证以及哪些角色得到授权可以完成受约束的请求。关于授权约束的更多信息，参见 19.2.1.2 节。
- **用户数据约束**（`user-data-constraint`）：指定数据在客户端和服务器之间传输时如何进行保护。用户数据约束将在 19.2.1.3 节中讨论。

19.2.1.1 指定 Web 资源集合

Web 资源集合包括以下子元素：

- `web-resource-name` 是用于这个资源的名字。这个子元素可选。
- `url-pattern` 用于列出要保护的请求 URI。很多应用同时包含受保护和不受保护的资源。要对一个资源提供非受限访问，就不要为这个特定的请求 URI 配置安全约束。

请求 URI 是 URL 中主机名和端口后面的部分。例如，假设有一个电子商务网站，其中的产品目录允许所有人访问和浏览，但购物车部分仅供顾客访问。可以为这个 Web 应用建立相应的路径，使得模式 `/cart/*` 受保护，而其他资源不受保护。假设这个应用安装在上下文路径 `/myapp` 中，则有：

  - `http://localhost:8080/myapp/index.xhtml` 不受保护。
  - `http://localhost:8080/myapp/cart/index.xhtml` 受保护。

用户第一次访问 `cart/` 子目录中的一个资源时，会提示用户登录。

- `http-method` 或 `http-method-omission` 用来指定哪些方法应受到保护，而哪些方法不用保护。在以下情况下，HTTP 方法会受 `web-resource-collection` 保护：
  - 如果集合中没有指定 HTTP 方法（这说明所有方法都受保护）。
  - 如果集合特意在一个 `http-method` 子元素中指定了 HTTP 方法。
  - 如果集合包含一个或多个 `http-method-omission` 元素，其中都没有指定 HTTP 方法。

19.2.1.2 指定授权约束

授权约束（auth-constraint）包含 role-name 元素。可以根据需要使用多个 role-name 元素。

授权约束会建立一个认证需求，对于这个安全约束声明的 URL 模式和 HTTP 方法，授权约束会指定哪些角色得到授权允许访问这些 URL 模式和 HTTP 方法。如果没有授权约束，则容器必须接受请求，而不要求用户认证。如果有一个授权约束，但是其中没有指定角色，那么容器在任何情况下都不允许访问受约束的请求。这里指定的各个角色名必须对应于为 Web 应用定义的某个 security-role 元素的角色名，或者必须是特殊的保留角色名 *，这表示 Web 应用中的所有角色。角色名区分大小写。为应用定义的角色必须映射到服务器上定义的用户和组，除非使用默认的主体 - 角色映射。

关于安全角色的更多信息，参见 19.2.4 节。关于映射安全角色的更多信息，参见 18.5.4 节。

对于 servlet，@HttpConstraint 和 @HttpMethodConstraint 注解接受一个 rolesAllowed 元素，可以指定授权角色。

19.2.1.3 指定安全连接

用户数据约束（部署描述文件中的 user-data-constraint）包含 transport-guarantee 子元素。对于安全约束中指定的受约束 URL 模式和 HTTP 方法，可以用用户数据约束来要求使用一个受保护的传输层连接，如 HTTPS。传输保证可以选择 CONFIDENTIAL、INTEGRAL 或 NONE。如果指定 CONFIDENTIAL 或 INTEGRAL 作为安全约束，则通常表示要使用 SSL，而且这个安全约束会应用于与 Web 资源集合中 URL 模式匹配的所有请求，而不只是登录对话框。

所需的保护强度由传输保证的值来定义，如下：

- 应用要求传输数据时避免其他实体观察传输的内容，这种情况下要指定 CONFIDENTIAL。
- 应用要求数据在客户端和服务器之间发送时应保证数据在传输时不会被修改，这种情况下要指定 INTEGRAL。
- 指定 NONE 表示容器必须接受任何连接上的受约束请求，包括不受保护的连接。

说明 在实际中，Java EE 服务器会对 CONFIDENTIAL 和 INTEGRAL 传输保证值做相同的处理。

结合基本用户认证和基于表单的用户认证，用户数据约束使用很方便。登录认证方法设置为 BASIC 或 FORM 时，密码不受保护，这说明，在一个非保护会话上，客户端和服务器之间发送的密码可能会被第三方查看和拦截。结合用户认证机制使用用户数据约束可以缓解这个问题。如何配置用户认证机制将在 19.2.3 节中介绍。

为了保证数据在一个安全连接上传输，要确保为服务器配置了 SSL 支持。已经为 GlassFish 服务器配置了 SSL 支持。

说明 一个会话切换到 SSL 时，这个会话的其余部分不能再接受任何非 SSL 请求。例如，一个购物网站在结账页面之前可能并不使用 SSL，然后可能切换为使用 SSL 接受你的信用卡号。切换到 SSL 后，就要停止为这个会话监听非 SSL 请求。之所以这么做，原因在于在之前的通信中会话 ID 本身未加密。如果只是你自己挑选商品，情况还不是太糟糕，不过，信用卡信息存储到会话中之后，你肯定不希望其他人使用这个信息冒用你的信用卡购买商品。使用过滤器很容易实现这一点。

19.2.1.4 为资源指定安全约束

可以在应用中为资源创建安全约束。例如，允许 PARTNER 角色的用户完全访问 URL 模式 /acme/wholesale/* 的所有资源，并允许 CLIENT 角色的用户完全访问 URL 模式 /acme/retail/* 的所有资源。如果你想保护某些 HTTP 方法，而其他 HTTP 方法不受保护，就推荐采用这种方法来保护资源。下面的部署描述文件示例展示了这个功能：

```
<!-- SECURITY CONSTRAINT #1 -->
<security-constraint>
    <web-resource-collection>
        <web-resource-name>wholesale</web-resource-name>
        <url-pattern>/acme/wholesale/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>PARTNER</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>

<!-- SECURITY CONSTRAINT #2 -->
<security-constraint>
    <web-resource-collection>
        <web-resource-name>retail</web-resource-name>
        <url-pattern>/acme/retail/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>CLIENT</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

19.2.2 指定认证机制

用户认证机制可以指定：

- 用户访问 Web 内容的方式
- 对于基本认证，会指定认证用户的安全域
- 对于基于表单的认证，还会指定其他一些属性

指定一个认证机制时，用户首先必须认证，然后才会允许访问受一个安全约束限制的资源。可以有多个安全约束应用到多个资源，不过对于一个应用中的所有受约束资源，会应用相同的认证方法。

认证一个用户之前，必须有一个用户数据库，其中包含用户名、密码和 Web 或应用服务器上配置的角色。有关建立用户数据库的更多信息，参见 18.5.2 节。

Java EE 平台支持以下认证机制：

- 基本认证（basic authentication）
- 基于表单的认证（form-based authentication）
- 摘要认证（digest authentication）
- 客户端认证（client authentication）
- 相互认证（mutual authentication）

基本认证、基于表单的认证和摘要认证将在这一节讨论。客户端认证和相互认证将在第 21 章中讨论。

HTTP 基本认证和基于表单的认证并不是非常安全的认证机制。基本认证在互联网上将用户名和密码作为 Base64 编码文本发送。基于表单的认证将这个数据作为明文发送。在这两种情况下，并不对目标服务器完成认证。因此，这些认证方式可能导致用户数据暴露和受到攻击。如果有人拦截数据传输，则用户名和密码信息很容易被解码。

不过，如果结合基本认证或基于表单的认证使用一个安全传输机制（如 SSL）或者网络层安全（如 Internet Protocol Security，IPsec）协议或虚拟专用网（virtual private network，VPN）策略，上面的一些问题就能得到缓解。要指定一个安全传输机制，可以使用 19.2.1.3 节中介绍的元素。

19.2.2.1 HTTP 基本认证

指定 **HTTP 基本认证**时，要求服务器从 Web 客户端请求一个用户名和密码，通过与指定或默认安全域中授权用户的数据库比较，来验证用户名和密码是否合法。

未指定任何认证机制时，默认为基本认证。

使用基本认证时，会完成以下动作。

1. 客户端请求访问一个受保护的资源。

2. Web 服务器返回一个对话框，请求用户名和密码。

3. 客户端向服务器提交用户名和密码。

4. 服务器在指定安全域认证用户，如果成功，则返回所请求的资源。

图 19-1 显示了指定 HTTP 基本认证时会发生什么。

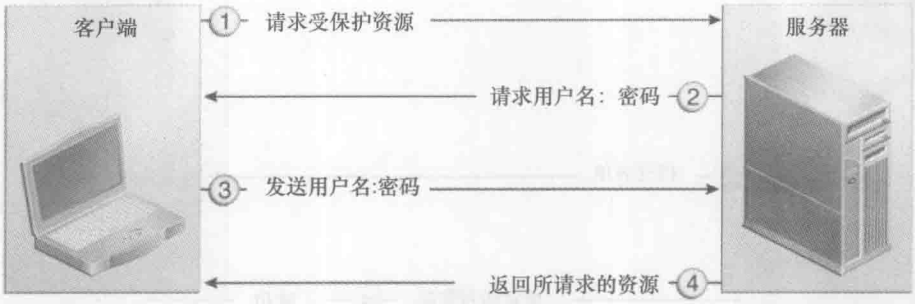


图 19-1　HTTP 基本认证

19.2.2.2　基于表单的认证

基于表单的认证允许开发人员通过定制 HTTP 浏览器提供给最终用户的登录屏幕和错误页面，来控制登录认证屏幕的外观。声明基于表单的认证时，会发生以下动作：

1. 客户端请求访问一个受保护的资源。

2. 如果客户端未得到认证，则服务器将客户端重定向到登录页面。

3. 客户端向服务器提交登录表单。

4. 服务器尝试认证用户。

- 如果认证成功，则检查认证用户的主体，确保其角色得到授权可以访问这个资源。如果用户得到授权，则服务器使用所存储的 URL 路径将客户端重定向到这个资源。
- 如果认证失败，则客户端会转移或重定向到一个错误页面。

图 19-2 显示了指定基于表单的认证时会发生什么。

19.4.3 节会介绍一个示例应用，其中就使用了基于表单的认证。

创建一个基于表单的登录时，一定要使用 cookie 或 SSL 会话信息维护会话。

要保证认证正常进行，登录表单的动作必须是 `j_security_check`。之所以有这个限制，是为了保证不论请求哪个资源，登录表单总能工作，另外可以不要求服务器指定表单的动作字段。下面的代码段显示了 HTML 页面中应当如何编写表单：

```
<form method="POST" action="j_security_check">
    <input type="text" name="j_username">
    <input type="password" name="j_password">
</form>
```

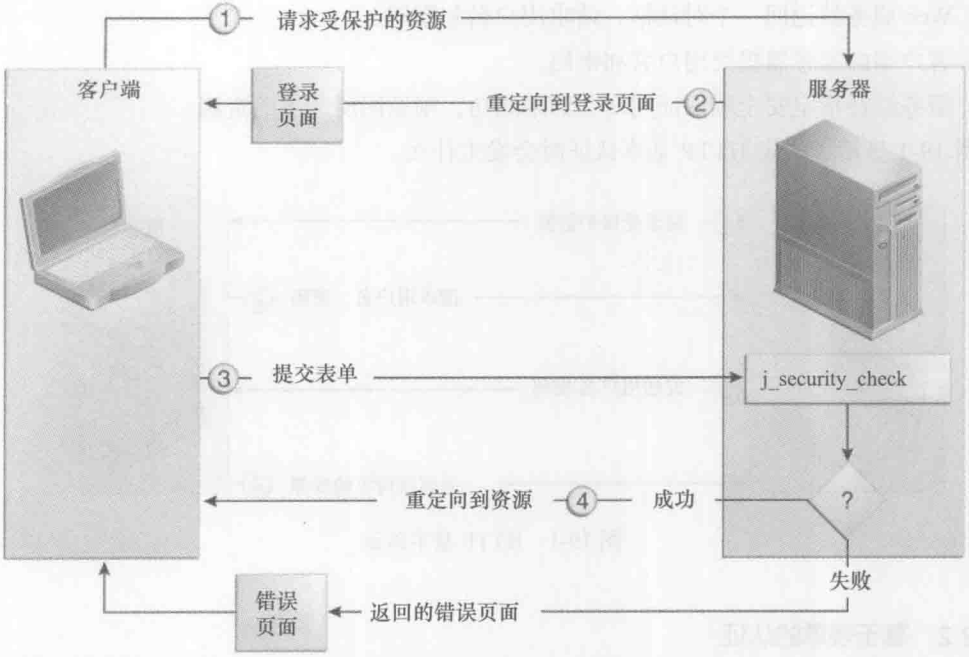


图 19-2 基于表单的认证

19.2.2.3 摘要认证

类似于基本认证，**摘要认证**也是根据用户名和一个密码来认证用户。不过，与基本认证不同，摘要认证不会在网络上发送用户密码。实际上，客户端会发送密码和其他数据的一个单向加密散列。尽管密码不在网络上发送，但摘要认证要求认证容器可以得到相应的明文密码，从而能通过计算期望的摘要来验证接收到的认证码。

19.2.3 在部署描述文件中指定认证机制

可以使用 `login-config` 元素指定认证机制。它可以包含以下子元素：

- `auth-method` 子元素为 Web 应用配置认证机制。元素内容必须是 `NONE`、`BASIC`、`DIGEST`、`FORM` 或 `CLIENT-CERT`。
- `realm-name` 子元素指示为 Web 应用选择基本认证机制时使用的安全域名。
- `form-login-config` 子元素指定使用基于表单的登录时所要使用的登录和错误页面。

说明 要指定基于表单的认证，另一种方法是使用 `HttpServletRequest` 的 `authenticate`、`login` 和 `logout` 方法，参见 19.3.1 节。

尝试访问一个受 `security-constraint` 元素约束的 Web 资源时，Web 容器会激活为

这个资源配置的认证机制。认证机制指定了如何提示用户登录。如果有 login-config 元素而且 auth-method 元素包含一个非 NONE 的值，则用户必须经过认证才能访问这个资源。如果没有指定认证机制，就不需要认证用户。

下面的例子显示了如何在部署描述文件中声明基于表单的认证：

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>file</realm-name>
    <form-login-config>
        <form-login-page>/login.xhtml</form-login-page>
        <form-error-page>/error.xhtml</form-error-page>
    </form-login-config>
</login-config>
```

要相对于部署描述文件的位置来指定登录和错误页面的位置。登录和错误页面的例子见 19.4.3.1 节。

下面的例子显示了如何在部署描述文件中声明摘要认证：

```
<login-config>
    <auth-method>DIGEST</auth-method>
</login-config>
```

19.2.4 声明安全角色

通过使用部署描述文件的 security-role 元素，可以声明 Web 应用中使用的安全角色名。使用这个元素列出应用中引用的所有安全角色。

下面的代码段取自一个部署描述文件，这里使用 security-role 元素声明了将在应用中使用的角色，并使用 auth-constraint 元素指定哪些角色得到授权允许访问受保护资源：

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Protected Area</web-resource-name>
        <url-pattern>/security/protected/*</url-pattern>
        <http-method>PUT</http-method>
        <http-method>DELETE</http-method>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>

 <!-- Security roles used by this web application -->
<security-role>
    <role-name>manager</role-name>
</security-role>
```

```
<security-role>
    <role-name>employee</role-name>
</security-role>
```

在这个例子中，security-role 元素列出了应用中使用的所有安全角色：manager 和 employee。这就允许部署人员将应用中定义的所有角色映射到 GlassFish 服务器中定义的用户和组。

auth-constraint 元素指定角色 manager 可以在 url-pattern 元素指定的目录中（/security/protected/*）访问 HTTP 方法 PUT、DELETE、GET 和 POST。

这种情况下不能使用 @ServletSecurity 注解，因为其约束会应用到 @WebServlet 注解指定的所有 URL 模式。

19.3 Web 应用使用编程式安全

对于非常强调安全的应用，如果声明式安全本身不足以描述应用的安全模型，就会使用编程式安全。

19.3.1 通过编程方式认证用户

利用 tpServletRequest 接口的以下方法，可以采用编程方式为 Web 应用完成用户认证。

- authenticate 允许应用从一个非约束请求上下文要求容器认证请求调用者。这会显示一个登录对话框，收集用户名和密码来完成认证。
- login 允许应用收集用户名和密码信息，这是在应用部署描述文件中指定基于表单的认证的一种替代方法。
- logout 允许应用重置一个请求的调用者身份。

下面的代码示例显示了如何使用 login 和 logout 方法（这些方法用粗体显示）：

```
package test;

import java.io.IOException;
import java.io.PrintWriter;
import java.math.BigDecimal;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="TutorialServlet", urlPatterns={"/TutorialServlet"})
public class TutorialServlet extends HttpServlet {
```

```
    @EJB
    private ConverterBean converterBean;

    /**
     * Processes requests for both HTTP <code>GET</code>
     *     and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet TutorialServlet</title>");
            out.println("</head>");
            out.println("<body>");
            request.login("TutorialUser", "TutorialUser");
            BigDecimal result =
                converterBean.dollarToYen(new BigDecimal("1.0"));
            out.println("<h1>Servlet TutorialServlet result of dollarToYen= "
                + result + "</h1>");
            out.println("</body>");
            out.println("</html>");
        } catch (Exception e) {
            throw new ServletException(e);
        } finally {
            request.logout();
            out.close();
        }
    }
}
```

下面的示例代码展示了如何使用 authenticate 方法（这个方法用粗体显示）：

```
package com.example.test;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class TestServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
```

```
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            request.authenticate(response);
            out.println("Authenticate Successful");
        } finally {
            out.close();
        }
    }
```

19.3.2 通过编程方式检查调用者身份

一般地，要由容器采用一种对 Web 组件透明的方式实现安全管理。这一节中介绍的安全 API 只在一些不太常见的情况下使用，即 Web 组件方法需要访问安全上下文信息。

Servlet 3.1 指定了以下方法，允许访问有关组件调用者的安全信息：

- getRemoteUser 确定用户名，客户端要用这个用户名来完成认证。getRemoteUser 方法会返回容器为请求关联的远程用户（调用者）名。如果没有用户得到认证，这个方法返回 null。
- isUserInRole 确定一个远程用户是否是一个特定的安全角色。如果没有用户得到认证，这个方法返回 false。该方法需要一个 String 类型的用户角色名（role-name）参数。

要在部署描述文件中声明 security-role-ref 元素，并有一个 role-name 子元素包含将传入 isUserInRole 方法的角色名。如何使用安全角色引用将在 19.3.4 节中讨论。

- getUserPrincipal 确定当前用户的主体名，并返回一个 java.security.Principal 对象。如果没有用户得到认证，这个方法返回 null。在 getUserPrincipal 返回的 Principal 上调用 getName 方法将返回远程用户名。

你的应用可以根据使用这些 API 得到的信息完成业务逻辑决策。

19.3.3 编程式安全的示例代码

下面的代码展示了如何使用编程式安全实现编程式登录。这个 servlet 完成以下工作：

1. 显示当前用户的信息。
2. 提示用户登录。
3. 再次打印信息，来展示 login 方法的作用。
4. 注销用户。
5. 再次打印信息，来展示 logout 方法的作用。

```
package enterprise.programmatic_login;

import java.io.*;
import java.net.*;
```

```
import javax.annotation.security.DeclareRoles;
import javax.servlet.*;
import javax.servlet.http.*;

@DeclareRoles("javaee7user")
public class LoginServlet extends HttpServlet {

    /**
     * Processes requests for both HTTP GET and POST methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request,
                HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            String userName = request.getParameter("txtUserName");
            String password = request.getParameter("txtPassword");

            out.println("Before Login" + "<br><br>");
            out.println("IsUserInRole?.."
                        + request.isUserInRole("javaee7user")+"<br>");
            out.println("getRemoteUser?.."
                        + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                        + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.."
                        + request.getAuthType()+"<br><br>");

            try {
                request.login(userName, password);
            } catch(ServletException ex) {
                out.println("Login Failed with a ServletException.."
                    + ex.getMessage());
                return;
            }
            out.println("After Login..."+"<br><br>");
            out.println("IsUserInRole?.."
                        + request.isUserInRole("javaee7user")+"<br>");
            out.println("getRemoteUser?.."
                        + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                        + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.."
                        + request.getAuthType()+"<br><br>");

            request.logout();
            out.println("After Logout..."+"<br><br>");
```

```
            out.println("IsUserInRole?.."
                        + request.isUserInRole("javaee7user")+"<br>");
            out.println("getRemoteUser?.."
                        + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                        + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.." + request.getAuthType()+"<br>");
        } finally {
            out.close();
        }
    }
    ...
}
```

19.3.4 声明和链接角色引用

安全角色引用（security role reference）是一个映射，即从一个Web组件使用isUserInRole(String role)调用的角色名与应用中定义的一个安全角色名之间的映射。如果部署描述文件中没有声明security-role-ref元素，而是调用了isUserInRole方法，则默认地，容器会根据为Web应用定义的安全角色列表来检查所提供的角色名。一般情况下，可以自由地改变应用中的角色名而不需要重新编译完成调用的servlet，不过如果使用这个默认的isUserInRole方法，而不是使用security-role-ref元素，则可能会对此带来一些限制。

应用使用HttpServletRequest.isUserInRole(String role)时，要使用security-role-ref元素。传入isUserInRole方法的值是一个String，表示用户的角色名。role-name元素的值必须是一个String，用作HttpServletRequest.isUserInRole(String role)的参数。role-link必须包含security-role元素中定义的一个安全角色名。容器在确定这个调用的返回值时会使用security-role-ref到security-role的映射。

例如，要将安全角色引用cust映射到角色名为bankCustomer的安全角色，元素如下所示：

```
<servlet>
...
    <security-role-ref>
        <role-name>cust</role-name>
        <role-link>bankCustomer</role-link>
    </security-role-ref>
...
</servlet>
```

如果由一个安全角色为bankCustomer的用户调用这个servlet方法，则isUserInRole("cust")返回true。

security-role-ref 元素中的 role-link 元素必须与同一个 web.xml 部署描述文件中的 security-role 元素定义的一个 role-name 匹配，如下所示：

```
<security-role>
    <role-name>bankCustomer</role-name>
</security-role>
```

对于安全角色引用（包括引用定义的角色名），其作用域为部署描述文件包含 security-role-ref 部署描述文件元素的组件。

19.4 示例：保护 Web 应用安全

正确地运行示例应用之前，需要完成一些基本设置工作。这个例子使用了注解、编程式安全和 / 或声明式安全，来展示如何为现有的 Web 应用增加安全性。

以下章节（和地址）提供了保护不同类型应用安全的一些例子：

- 20.3.1 节
- 20.3.2 节
- GlassFish 示例：https://glassfish-samples.java.net/

19.4.1 设置系统来运行安全示例

要设置系统来运行安全示例，需要配置一个用户数据库，应用可以使用这个用户数据库认证用户。在继续后面的内容之前，先完成以下步骤：

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 向 GlassFish 服务器增加一个授权用户。对于本章和第 20 章中的例子，要为 GlassFish 服务器的 file 安全域增加一个用户，并将这个用户指定到 TutorialUser 组。

   a. 从管理控制台展开 Configurations（配置）节点，然后展开 server-config（服务器配置）节点。

   b. 展开 Security（安全）节点。

   c. 展开 Realms（安全域）节点。

   d. 选择 File（文件）节点。

   e. 在编辑安全域页面上，单击 Manage Users（管理用户）。

   f. 在文件用户页面上，单击 New（新建）。

   g. 在 User ID（用户 ID）域中输入一个用户 ID。

   h. 在 Group List（组列表）域中输入 TutorialUser。

   i. 在 New Password（新密码）和 Confirm New Password（确认新密码）域中输入一个密码。

   j. 单击 OK。

对于你创建的用户，要把它的用户名和密码写下来，以便测试示例应用。认证时用户名和密码都是区分大小写的，所以要准确地写下用户名和密码。这个内容在 18.5.2 节中做过更详细地讨论。

3. 在 GlassFish 服务器中建立默认的主体 – 角色映射。

   a. 从管理控制台展开 Configurations，然后展开 server-config 节点。

   b. 选择 Security 节点。

   c. 选择 Default Principal to Role Mapping Enabled（启用默认主体 – 角色映射）复选框。

   d. 单击 Save（保存）。

19.4.2 hello2-basicauth 示例：对 servlet 使用基本认证

这个例子解释了如何对一个 servlet 使用基本认证。利用 servlet 的基本认证，Web 浏览器会显示一个不可定制的标准登录对话框。用户提交名字和密码时，服务器确定这个用户名和密码是否是授权用户的用户名和密码，如果用户得到授权，允许查看所请求的 Web 资源，则将 Web 资源发送给这个用户。

一般地，要为一个不安全的 servlet 增加基本认证（如本书第 1 卷第 6 章中介绍的 servlet），必须完成以下步骤。实际上，对于本书中的示例应用，很多步骤已经为你完成，这里列出这些步骤只是为了说明创建一个类似的应用需要做哪些工作。这个应用位于 *tut-install/*`examples/security/hello2-basicauth/` 目录。

1. 完成 19.4.1 节中的步骤。
2. 为本书第 1 卷第 6 章中介绍的 servlet 示例 `hello2` 创建一个 Web 模块。
3. 为这个 servlet 增加适当的安全注解。19.4.2.1 节中将介绍安全注解。
4. 按照 19.4.2.2 节或 19.4.2.3 节中介绍的步骤构建、打包和部署这个 Web 应用。
5. 按照 19.4.2.4 节中介绍的步骤运行这个 Web 应用。

19.4.2.1 使用注解指定基本认证安全

GlassFish 服务器使用的默认认证机制是基本认证。利用基本认证，GlassFish 服务器会生成一个标准登录对话框，为一个受保护的资源收集用户名和密码数据。一旦用户得到认证，则允许他访问这个受保护的资源。

可以使用 `@ServletSecurity` 注解为一个 servlet 指定安全性。这个注解允许在 HTTP 方法上指定特定约束，另外还可以指定更一般的约束，应用于未指定特定约束的所有 HTTP 方法。在 `@ServletSecurity` 注解中，可以指定以下注解：

- `@HttpMethodConstraint` 注解，应用于一个特定的 HTTP 方法。
- 更一般的 `@HttpConstraint` 注解，应用于所有无 `@HttpMethodConstraint` 注解的 HTTP 方法。

`@ServletSecurity` 注解中的 `@HttpMethodConstraint` 和 `@HttpConstraint`

注解可以指定以下元素：

- transportGuarantee 元素，指定接收请求的连接必须满足的数据保护需求（也就是说，是否需要 SSL/TLS）。这个元素的合法值为 NONE 和 CONFIDENTIAL。
- rolesAllowed 元素，指定授权的角色名。

对于 hello2-basicauth 应用，GreetingServlet 有以下注解：

```
@WebServlet(name = "GreetingServlet", urlPatterns = {"/greeting"})
@ServletSecurity(
@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
    rolesAllowed = {"TutorialUser"}))
```

这些注解指定请求 URI/greeting 只能由得到授权允许访问这个 URL 的用户来访问（已经验证为角色 TutorialUser）。数据会在受保护的传输层上发送，以避免用户名和密码数据在传输过程中被第三方读取。

如果使用 @ServletSecurity 注解，则不需要在部署描述文件中指定安全设置。对于非默认认证机制，则无法使用 @ServletSecurity 注解来指定安全设置，在这种情况下就可以使用部署描述文件来指定。

19.4.2.2 使用 NetBeans IDE 构建、打包和部署 hello2-basicauth 示列

1. 完成 19.4.1 节中的步骤。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

   tut-install/examples/security

4. 选择 hello2-basicauth 文件夹。
5. 单击 Open Project。
6. 在 Projects(工程）标签页中，右键单击 hello2-basicauth 工程并选择 Build(构建)。这个命令会构建示例应用并将其部署到 GlassFish 服务器实例。

19.4.2.3 使用 Maven 构建、打包和部署 hello2-basicauth 示例

1. 完成 19.4.1 节中的步骤。
2. 在一个终端窗口中导航至：

   tut-install/examples/security/hello2-basicauth/

3. 输入以下命令：

```
mvn install
```

这个命令会构建应用并将其打包到一个 WAR 文件 hello2-basicauth.war，该文件位于 target 目录，然后部署这个 WAR 文件。

19.4.2.4 运行 hello2-basicauth 示例

1. 在 Web 浏览器中输入以下 URL：

```
https://localhost:8181/hello2-basicauth/greeting
```

可能会提示你接受服务器的安全证书。如果是这样，则按提示接受安全证书。如果浏览器警告证书非法，原因是证书是自签名的，则为应用增加一个安全例外情况。

出现一个要求认证对话框。取决于你使用的浏览器，这个对话框的外观可能不同。

2. 输入一个用户名和密码组合，这对应于 GlassFish 服务器 file 安全域中创建的一个用户（而且已经指定到组 TutorialUser）；然后单击 OK。

基本认证对于用户名和密码都是区分大小写的，所以要按 GlassFish 服务器中的定义准确地输入用户名和密码。

如果满足以下所有条件，则服务器会返回所请求的资源：

- 对于你输入的用户名，已经为 GlassFish 服务器定义了相应的用户。
- 对于你输入的用户名，相应用户的密码与你输入的密码一致。
- 你输入的用户名和密码组合指定到 GlassFish 服务器中的组 TutorialUser。
- 应用中定义的 TutorialUser 角色映射到 GlassFish 服务器中定义的组 TutorialUser。

3. 在输入域中输入一个名字，并单击 Submit（提交）。

由于已经得到授权，所以这一步输入的名字不会有任何限制。现在你可以不加限制地访问应用了。

应用会对你输入的名字响应一个 "Hello"。

19.4.3 hello1-formauth 示例：对 JSF 应用使用基于表单的认证

这个例子解释如何对一个 JSF 应用使用基于表单的认证。利用基于表单的认证，可以定制提供给 Web 客户端完成用户名和密码认证的登录屏幕和错误页面。用户提交其名字和密码时，服务器会确定用户名和密码是否是一个授权用户的用户名和密码，如果得到授权，则发送所请求的 Web 资源。

这个示例（hello1-formauth）将为本书第 1 卷 6.3 节中介绍的基本 JSF 应用增加安全性。

一般地，为一个不安全的 JSF 应用增加基于表单的认证时，与 19.4.2 节类似，也要完成类似的步骤。主要区别在于，如 19.4.3.2 节中所述，这里必须使用一个部署描述文件来指定使用基于表单的认证。另外，必须创建一个登录页面和一个登录错误页面，参见 19.4.3.1 节。

这个应用放在 *tut-install*/examples/security/hello1-formauth/ 目录中。

19.4.3.1 创建登录表单和错误页面

使用基于表单的登录机制时，必须指定一个页面，其中包含收集用户名和密码的表单，另外还要指定登录认证失败时显示的一个错误页面。这一节将介绍这个例子中使用的登录表单和错误页面。19.4.3.2 节展示了如何在部署描述文件中指定这些页面。

登录页面可以是 HTML 页面或 servlet，它必须返回一个包含表单的 HTML 页面，这个页面要符合特定的命名约束（关于这些需求的更多信息参见 Java Servlet 3.1 规范）。为此，在登录页面的 <form></form> 标记之间增加一些元素来收集用户名和密码信息。登录页面的 HTML 页面或 servlet 的内容可以写为：

```
<form method="post" action="j_security_check">
    <input type="text" name="j_username">
    <input type="password" name= "j_password">
</form>
```

这个例子中使用的登录页面的完整代码在 *tut-install*/examples/security/hello1-formauth/src/main/webapp/login.html 中。下面给出这个页面的代码：

```
<html lang="en">
    <head>
        <title>Login Form</title>
    </head>
    <body>
        <h2>Hello, please log in:</h2>
        <form method="post" action="j_security_check">
            <table role="presentation">
                <tr>
                    <td>Please type your user name: </td>
                    <td><input type="text" name="j_username"
                               size="20"/></td>
                </tr>
                <tr>
                    <td>Please type your password: </td>
                    <td><input type="password" name="j_password"
                               size="20"/></td>
                </tr>
            </table>
            <p></p>
            <input type="submit" value="Submit"/>
             
            <input type="reset" value="Reset"/>
        </form>
    </body>
</html>
```

如果用户输入一个用户名和密码组合，但是相应用户没有得到授权，即不允许访问受保护的 URI，就会显示登录错误页面。在这个例子中，登录错误页面位于 *tut-install*/examples/security/hello1-formauth/src/main/webapp/error.html。对于这个例子，登录错误页面会解释为什么会接收到这个错误页面，并提供一个链接，允许用户再次尝试。下面给出这个页面的代码：

```
<html lang="en">
    <head>
        <title>Login Error</title>
    </head>
    <body>
        <h2>Invalid user name or password.</h2>

        <p>Please enter a user name or password that is authorized to access
           this application. For this application, this means a user that
```

```
            has been created in the <code>file</code> realm and has been
            assigned to the <em>group</em> of <code>TutorialUser</code>.</p>
        <p><a href="login.html">Return to login page</a></p>
    </body>
</html>
```

19.4.3.2 为基于表单的认证示例指定安全性

这个例子为一个基于 servlet 的 Web 应用增加基于表单的安全性，这是一个非常简单的 Web 应用。为了指定基于表单的认证，而不是 JSF 示例中增加的基本认证，这里必须使用部署描述文件。

下面的示例代码显示了增加到部署描述文件的安全元素，这个部署描述文件位于 *tut-install*/examples/security/hello1-formauth/src/main/webapp/WEB-INF/web.xml 中：

```
<security-constraint>
    <display-name>Constraint1</display-name>
    <web-resource-collection>
        <web-resource-name>wrcoll</web-resource-name>
        <description/>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description/>
        <role-name>TutorialUser</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>file</realm-name>
    <form-login-config>
        <form-login-page>/login.xhtml</form-login-page>
        <form-error-page>/error.xhtml</form-error-page>
    </form-login-config>
</login-config>

<security-role>
    <description/>
    <role-name>TutorialUser</role-name>
</security-role>
```

19.4.3.3 使用 NetBeans IDE 构建、打包和部署 hello1-formauth 示例

1. 完成 19.4.1 节中的步骤。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

tut-install/examples/security

4. 选择 hello1-formauth 文件夹。

5. 单击 Open Project。

6. 在 Projects(工程) 标签页中，右键单击 hello1-formauth 工程并选择 Build(构建)。

这个命令会构建示例应用并将其部署到 GlassFish 服务器实例，然后在一个浏览器中打开应用。

19.4.3.4　使用 Maven 构建、打包和部署 hello1-formauth 示例

1. 完成 19.4.1 节中的步骤。

2. 在一个终端窗口中导航至：

```
tut-install/examples/security/hello1-formauth/
```

3. 在终端窗口或命令行提示窗口输入以下命令：

```
mvn install
```

这个命令会构建应用并将其打包到一个 WAR 文件 hello1-formauth.war，这个文件位于 target 目录，然后将这个 WAR 文件部署到 GlassFish 服务器。

19.4.3.5　运行 hello1-formauth 示例

要运行 hello1-formauth 的 Web 客户端，需要完成以下步骤：

1. 打开 Web 浏览器访问以下 URL：

```
http://localhost:8080/hello1-formauth/
```

2. 在登录表单中，输入一个用户名和密码组合，这对应于 GlassFish 服务器 file 安全域中创建的一个用户，而且该用户已经指定到组 TutorialUser。

基于表单的认证对于用户名和密码都是区分大小写的，所以要按 GlassFish 服务器中的定义准确地输入用户名和密码。

3. 单击 Submit (提交)。

如果输入 My_Name 作为用户名，My_Pwd 作为密码，若满足以下所有条件，则服务器会返回所请求的资源：

- 已经为 GlassFish 服务器定义了用户名为 My_Name 的用户。
- GlassFish 服务器中为用户名为 My_Name 的用户定义了密码为 My_Pwd。
- 密码为 My_Pwd 的用户 My_Name 已经指定到 GlassFish 服务器上的组 TutorialUser。
- 应用中定义的角色 TutorialUser 映射到为 GlassFish 服务器定义的组 TutorialUser。

这些条件都满足时，服务器会通过这个用户的认证，将打开应用。

4. 输入你的名字并单击 Submit (提交)。

由于已经得到授权，所以这一步输入的名字不会有任何限制。现在可以不加限制地访问应用了。

应用会对你输入的名字响应一个 "Hello"。

后续步骤

要完成其他测试和查看生成的登录错误页面，可以关闭再重新打开你的浏览器，输入应用 URL，然后输入一个未授权的用户名和密码。

Chapter 20 第 20 章

企业应用安全入门

本章介绍如何管理企业应用的安全，包括以下内容：

- 企业应用的基本安全任务
- 保护企业 bean 安全
- 示例：保护企业 bean 安全

20.1 企业应用的基本安全任务

系统管理员、应用开发人员、bean 提供者以及部署人员负责管理企业应用的安全。基本安全任务包括：

- 建立用户数据库，并将用户指定到适当的组
- 建立身份传播
- 设置 GlassFish 服务器属性，使应用能正确运行，如设置默认的主体 – 角色映射。
- 为企业应用的类和方法增加注解，提供有关信息指出哪些方法的访问需要受限。

本章和前一章介绍安全示例的小节解释了如何完成这些任务。

20.2 保护企业 bean 安全

企业 bean 是实现 EJB 技术的 Java EE 组件。企业 bean 在 EJB 容器中运行，这是 GlassFish 服务器中的一个运行时环境。尽管对应用开发人员是透明的，不过 EJB 容器提供了大量系统级服务，如对企业 bean 提供了事务和安全服务，这也是事务性 Java EE 应用的核心。

可以采用以下方法保护企业 bean 方法的安全：

- **声明式安全**（首选）：使用部署描述文件或注解描述一个应用组件的安全需求。有些情况下，为了实现方法保护和认证，只需要在企业 bean 类的业务方法中增加一个注解来指定方法权限。这一节就来讨论保护企业 bean 安全的这种简单而高效的方法。

保护企业 bean 安全的这种简化方法还存在一些局限性，所以有些情况下，你可能还想使用部署描述文件来指定安全信息。要保证这个简单的解决方案正常工作，必须在服务器上配置一个认证机制。GlassFish 服务器的默认认证方法是基本认证。

本书将解释如何用指定方法权限的注解装饰企业应用的业务方法，从而调用授权用户的用户名 / 密码认证。

为了更加简化部署人员的任务，应用开发人员可以定义一些安全角色。安全角色是一个权限分组，给定类型的应用用户必须有这些权限才能成功地使用应用。例如，在一个工资应用中，一些用户希望查看他们自己的工资信息（employee），另外一些用户需要查看其他人的工资信息（manager），还有一些用户需要能够修改其他人的工资信息（payrollDept）。应用开发人员要确定应用可能有哪些类型的用户，另外哪些用户可以访问哪些方法。应用开发人员再用注解装饰企业 bean 的类和方法，指定哪些类型的用户得到授权允许访问这些方法。使用注解指定授权用户的有关内容在 20.2.1.1 节中介绍。

使用某个注解定义方法权限时，部署系统会自动要求完成用户名 / 密码认证。在这种认证中，会提示用户输入一个用户名和密码，并与已知用户数据库进行比较。如果找到这个用户，而且密码匹配，再将为这个用户分配的角色与授权访问这个方法的角色进行比较。如果用户通过认证，而且其角色得到授权允许访问这个方法，则向用户返回数据。

使用声明式安全的有关内容在 20.2.1 节中介绍。

- **编程式安全**：对于一个企业 bean，可以在业务方法中嵌入代码，通过编程访问调用者的身份，并使用这个信息完成安全决策。如果声明式安全本身不足以描述一个应用的安全模型，编程式安全就很有用。

一般地，要由容器采用一种对企业 bean 业务方法透明的方式完成安全管理。这一章介绍的编程式安全 API 只在一些不太常见的情况下使用，即企业 bean 业务方法需要访问安全上下文信息，例如希望根据当天时间或特定角色的特殊条件检查来指定访问授权，这种情况下就可以使用编程式安全。

编程式安全在 20.2.2 节中讨论。

本章介绍的一些内容假设你已经读过第 3 章、第 4 章和第 18 章。

本节讨论如何保护一个 Java EE 应用的安全，这里将一个或多个模块（如 EJB JAR 文件）打包到一个 EAR 文件中，这是存放应用的归档文件。将在 Java 类文件中使用安全注解来指定授权用户和认证机制（基本认证或用户名 / 密码认证）。

企业 bean 通常要提供 Web 应用的业务逻辑。在这些情况下，将企业 bean 打包在 Web 应用的 WAR 模块中可以简化部署和应用组织。企业 bean 可以打包在一个 WAR 模块中（如

Java 类文件），也可以打包在 WAR 模块中的一个 JAR 文件中。如果由 servlet 或 JSF 页面处理 Web 前端，而且应用作为一个 Java 类文件打包到一个 WAR 模块中，那么可以在应用的 web.xml 文件中处理应用的安全。如果需要，WAR 文件中的 EJB 可以有自己的部署描述文件 ejb-jar.xml。第 19 章讨论了使用 web.xml 保护 Web 应用安全的有关内容。

以下各小节讨论了用来保护企业 bean 资源的声明式安全机制和编程式安全机制。受保护的资源包括从应用客户端、Web 组件或其他企业 bean 调用的企业 bean 方法。

关于这个主题的更多信息，参见 Enterprise JavaBeans 3.2 规范。这个文档可以从 http://jcp.org/en/jsr/detail?id=345 下载。规范中第 12 章介绍了企业 bean 的安全管理。

20.2.1 使用声明式安全保护企业 bean 安全

利用声明式安全，应用开发人员可以指定哪些用户得到授权允许访问企业 bean 的哪些方法，另外可以指定使用基本认证或用户名 / 密码认证来完成用户认证。很多情况下，开发企业应用的人并不是负责部署这个应用的人。应用开发人员使用声明式安全来定义方法权限和认证机制，向部署人员传递 EJB JAR 中包含的企业 bean 的一个安全视图。安全视图传递到部署人员时，部署人员使用这个信息为安全角色定义方法权限。如果没有定义一个安全视图，部署人员就必须自行确定各个业务方法做什么，从而确定哪些用户得到授权来调用各个方法。

安全视图由一组安全角色组成，安全角色是权限的一个语义分组，给定类型的应用用户必须有这些权限才能成功地访问应用。安全角色是指一些逻辑角色，表示一类用户。可以为每个安全角色定义方法权限。方法权限就是调用企业 bean 业务接口、主接口、组件接口和 / 或 Web 服务端点中一组指定方法的权限。定义方法权限后，将使用用户名 / 密码认证来验证用户的身份。

要记住重要的一点，安全角色用来定义应用的逻辑安全视图。不要把安全角色与用户组、用户、主体和 GlassFish 服务器中的其他概念混淆。另外还需要完成一个步骤，要将应用中定义的角色映射到用户、组和主体，这些是 GlassFish 服务器 file 安全域中用户数据库的组件。这些步骤在 18.5.4 节中做过简要说明。

下面各小节将介绍应用开发人员如何使用声明式安全保护一个应用的安全，或者创建一个安全视图传递给部署人员。

20.2.1.1 通过声明安全角色指定授权用户

这一节讨论如何使用注解为一个 bean 类的方法指定方法权限。关于这些注解的更多信息，参见 Java 平台的公共注解规范（http://jcp.org/en/jsr/detail?id=250）。

可以在类或类的业务方法上指定方法权限，也可以同时在类和类的业务方法上指定方法权限。在 bean 类方法上指定的方法权限会覆盖整个 bean 类上指定的方法权限。可以用下面的注解指定方法权限。

- @DeclareRoles：指定应用将使用的所有角色，包括 @RolesAllowed 注解中未指定的角色。应用使用的安全角色集合是 @DeclareRoles 和 @RolesAllowed 注解

中定义的全部安全角色。

@DeclareRoles 注解在 bean 类上指定，对于增加了这个注解的类，@DeclareRoles 注解会声明可由该类中方法测试的角色（例如，通过调用 isCallerInRole）。声明用作 isCallerInRole(String roleName) 方法的参数的角色名时，所声明的角色名必须与参数值相同。

下面的示例代码展示了如何使用 @DeclareRoles 注解：

```
@DeclareRoles("BusinessAdmin")
public class Calculator {
    ...
}
```

下面的例子展示了声明多个角色的语法：

```
@DeclareRoles({"Administrator", "Manager", "Employee"})
```

- @RolesAllowed("*list-of-roles*")：指定允许访问应用方法的安全角色。这个注解可以在类上指定，也可以在一个或多个方法上指定。在类级指定时，这个注解应用于类中的所有方法。在一个方法上指定时，这个注解只应用于这个方法，而且会覆盖类级上指定的值。

  要指定任何角色都不允许访问一个应用中的方法，可以使用 @DenyAll 注解。要指定任何角色的用户都得到授权允许访问应用，则使用 @PermitAll 注解。

  结合 @DeclareRoles 注解时，应用会组合使用这两个注解指定的安全角色。

  下面的示例代码展示了如何使用 @RolesAllowed 注解：

```
@DeclareRoles({"Administrator", "Manager", "Employee"})
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        ...
    }
}
```

- @PermitAll：指定所有安全角色都可以执行指定的一个方法（或多个方法）。使用这个注解时，不再根据数据库检查用户来确保他得到授权来访问应用。

  这个注解可以在类上指定，也可以在一个或多个方法上指定。在类上指定这个注解时，意味着要应用于这个类的所有方法。在方法级指定时，则说明注解只应用于这个方法。

  下面的示例代码展示了如何使用 @PermitAll 注解：

```
import javax.annotation.security.*;
@RolesAllowed("RestrictedUsers")
public class Calculator {

    @RolesAllowed("Administrator")
```

```
    public void setNewRate(int rate) {
        //...
    }
    @PermitAll
    public long convertCurrency(long amount) {
        //...
    }
}
```

- @DenyAll：指定任何安全角色都不允许执行指定的一个方法（或多个方法）。这说明，不允许在 Java EE 容器中执行这些方法。

  下面的代码示例显示了如何使用 @DenyAll 注解：

```
import javax.annotation.security.*;
@RolesAllowed("Users")
public class Calculator {
    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        //...
    }
    @DenyAll
    public long convertCurrency(long amount) {
        //...
    }
}
```

下面的代码段展示了如何对 isCallerInRole 方法使用 @DeclareRoles 注解。在这个例子中，@DeclareRoles 注解声明了一个角色，企业 bean PayrollBean 使用这个角色来完成安全检查，使用 isCallerInRole("payroll") 验证调用者得到授权，允许修改工资数据：

```
@DeclareRoles("payroll")
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;

    public void updateEmployeeInfo(EmplInfo info) {

        oldInfo = ... read from database;

        // The salary field can be changed only by callers
        // who have the security role "payroll"
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (info.salary != oldInfo.salary && !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }
        ...
    }
    ...
}
```

下面的示例代码展示了如何使用 @RolesAllowed 注解：

```
@RolesAllowed("admin")
public class SomeClass {
    public void aMethod () { ... }
    public void bMethod () { ... }
    ...
}

@Stateless public class MyBean extends SomeClass implements A  {

    @RolesAllowed("HR")
    public void aMethod () { ... }

    public void cMethod () { ... }
    ...
}
```

在这个例子中，假设 aMethod、bMethod 和 cMethod 是业务接口 A 的方法，方法 aMethod 和 bMethod 的方法权限值分别是 @RolesAllowed("HR") 和 @RolesAllowed ("admin")。方法 cMethod 的方法权限未指定。

需要说明，子类本身并不继承注解。实际上，注解只应用于由子类继承的超类方法。

20.2.1.2 指定认证机制和安全连接

指定方法权限时，将由 GlassFish 服务器调用基本用户名 / 密码认证。

要使用一种不同类型的认证，或者需要一个使用 SSL 的安全连接，可以在应用部署描述文件中指定该信息。

20.2.2 通过编程方式保护企业 bean 安全

编程式安全是指在业务方法中内嵌代码，通过编程方式访问调用者的身份，并用这个信息在方法中完成安全决策。

20.2.2.1 访问企业 bean 调用者的安全上下文

一般地，要由容器采用一种对企业 bean 业务方法透明的方式完成安全管理。这一节介绍的安全 API 只在一些不太常见的情况下使用，即企业 bean 业务方法需要访问安全上下文信息，如希望限制某个特定时间的访问，这种情况下就可以使用编程式安全。

javax.ejb.EJBContext 接口提供了两个方法，允许 bean 提供者访问企业 bean 调用者的安全信息。

- getCallerPrincipal：允许企业 bean 方法得到当前调用者的主体名。例如，企业 bean 方法可以使用这个主体名作为数据库中信息的键。

下面的代码示例展示了如何使用 getCallerPrincipal 方法：

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    @Resource SessionContext ctx;
    @PersistenceContext EntityManager em;

    public void changePhoneNumber(...) {
        ...
        // obtain the caller principal
        callerPrincipal = ctx.getCallerPrincipal();

        // obtain the caller principal's name
        callerKey = callerPrincipal.getName();

        // use callerKey as primary key to find EmployeeRecord
        EmployeeRecord myEmployeeRecord =
            em.find(EmployeeRecord.class, callerKey);

        // update phone number
        myEmployeeRecord.setPhoneNumber(...);

        ...
    }
}
```

在这个例子中，企业 bean 得到当前调用者的主体名，并用它作为主键来查找一个 EmployeeRecord 实体。这个例子假设应用已经适当部署，当前调用者主体包含有主键（例如，员工号），可以用于标识员工。

- isCallerInRole：允许开发人员编写无法使用方法权限来简单定义的安全检查。这种检查可能会基于角色对请求施加限制，或者可能会依赖于数据库中存储的某些信息。

企业 bean 代码可以使用 isCallerInRole 方法检查是否已经为当前调用者分配一个给定的安全角色。安全角色由 bean 提供者或应用装配人员定义，并由部署人员分配到操作环境中已有的主体或主体组。

下面的代码示例展示了如何使用 isCallerInRole 方法：

```
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;

    public void updateEmployeeInfo(EmplInfo info) {

        oldInfo = ... read from database;

        // The salary field can be changed only by callers
        // who have the security role "payroll"
        if (info.salary != oldInfo.salary &&
            !ctx.isCallerInRole("payroll")) {
                throw new SecurityException(...);
        }
```

```
        ...
    }
    ...
}
```

有些情况下要使用编程式安全以这种方式动态地控制对一个方法的访问，如除了某个特定的时间外，希望拒绝其他时间内的访问。20.3.2 节中介绍了一个使用 getCallerPrincipal 和 isCallerInRole 方法的示例应用。

20.2.3　传播安全身份（Run-As）

可以指定是否使用调用者的安全身份来执行企业 bean 的指定方法，或者是否使用一个特定的 run-as 身份。图 20-1 展示了这一概念。

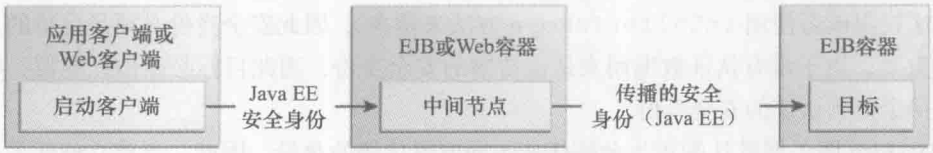


图 20-1　安全身份传播

在这个示意图中，应用客户端调用一个 EJB 容器中的企业 bean 方法。这个企业 bean 方法再调用另一个容器中的企业 bean 方法。第一个调用期间的安全身份是调用者的身份。第二个调用期间的安全身份则有以下可能：

- 默认地，中间组件的调用者身份会传播到目标企业 bean。如果目标容器信任中间容器，就会使用这个技术。
- 将一个特定身份传播到目标企业 bean。目标容器期望使用一个特定身份访问时，会使用这个技术。

将一个身份传播到目标企业 bean 时，要为这个 bean 配置一个 run-as 身份，见 20.2.3.1 节中的介绍。为企业 bean 建立 run-as 身份不会影响它的调用者身份（检查是否允许访问企业 bean 方法时，所检查的就是调用者身份）。run-as 身份则是建立企业 bean 做出调用时所使用的身份。

run-as 身份会应用到整个企业 bean，包括企业 bean 业务接口、本地和远程接口、组件接口和 Web 服务端点接口的所有方法、消息驱动 bean 的消息监听器方法、企业 bean 的超时方法，以及可能会调用的 bean 的内部方法。

20.2.3.1　配置组件的传播安全身份

可以使用 @RunAs 注解配置一个企业 bean 的 run-as 安全身份，即传播的安全身份，它定义了应用在一个 Java EE 容器中执行时的角色。这个注解可以在类上指定，允许开发人员以一个特定的角色执行一个应用。这个角色必须映射到容器安全域中的用户 / 组信息。@RunAs 注解指定一个安全角色名作为其参数。

下面的代码展示了如何使用 @RunAs 注解：

```
@RunAs("Admin")
public class Calculator {
    //...
}
```

如果给定角色与多个用户主体关联，则必须把 run-as 角色名映射到 GlassFish 服务器中定义的一个给定主体。

20.2.3.2 容器之间的信任

设计一个企业 bean 时，如果用原调用者身份或一个指定的身份来调用一个目标 bean，则这个目标 bean 只能接收到传播的身份。目标 bean 不会接收到任何认证数据。

目标容器无法认证传播的安全身份。不过，由于安全身份会在授权检查中使用（例如，检查方法权限或者使用 isCallerInRole 方法来检查），因此安全身份必须是可信的，这一点至关重要。由于没有认证数据用来认证传播的安全身份，因此目标必须信任容器，相信它传播了一个得到认证的安全身份。

GlassFish 服务器默认配置为会信任从不同容器传播的身份。因此，要建立信任关系，并不需要任何特殊的步骤。

20.2.4 部署安全企业 bean

部署人员负责确保已装配的应用部署到目标操作环境后是安全的。如果使用安全注解和 / 或部署描述文件为部署人员提供了一个安全视图，那么这个安全视图会映射到目标操作环境中由安全域所使用的机制和策略，在这里，目标操作环境就是 GlassFish 服务器。如果没有提供安全视图，则部署人员必须为企业 bean 应用建立适当的安全策略。

部署信息特定于一个 Web 或应用服务器。

20.3 示例：保护企业 bean 安全

下面的例子展示了如何使用声明式安全和编程式安全保护企业 bean 的安全。

20.3.1 cart-secure 示例：使用声明式安全保护企业 bean 安全

这一节讨论如何配置企业 bean 完成基本用户名 / 密码认证。请求以这种方式约束的 bean 时，服务器会从客户端请求一个用户名和密码，并与 GlassFish 服务器中的一个授权用户数据库比较，来验证这个用户名和密码是合法的。

如果你对认证的有关内容还不太了解，那么参见 19.2.2 节。

这个例子以一个不安全的企业 bean 应用 cart 作为起点来介绍安全。这个应用在 *tut-install*/examples/ejb/cart/ 目录中，见 5.1 节的讨论。

如果已有一个应用，其中包含有企业 bean，要向这个应用增加用户名 / 密码认证，一般必须完成以下步骤。对于本书中的示例应用，很多步骤已经为你完成了，这里列出这些步骤只是为了说明创建一个类似应用时需要做哪些工作。

1. 创建一个类似 5.1 节中的应用。这里就以这个示例为起点，展示如何向这个应用增加客户端的基本认证。这一节讨论的示例应用位于 *tut-install*/examples/security/cart-secure/。

2. 完成 19.4.1 节中的步骤，适当地配置系统来运行教程应用。

3. 修改企业 bean CartBean.java 的源代码，指定授权哪些角色访问哪些受保护的方法。这一步将在 20.3.1.1 节中讨论。

4. 构建、打包并部署企业 bean；然后根据 20.3.1.2 节或 20.3.1.3 节中的步骤构建和运行客户端应用。

20.3.1.1 增加 bean 注解

修改原 cart 应用的源代码，如以下代码段所示（修改部分用粗体显示）。最后得到的文件位于 *tut-install*/examples/security/cart-secure/cart-secure-ejb/src/main/java/javaeetutorial/cart/ejb/CartBean.java。

代码段如下所示：

```
package javaeetutorial.cartsecure.ejb;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javaeetutorial.cart.util.BookException;
import javaeetutorial.cart.util.IdVerifier;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;

@Stateful
@DeclareRoles("TutorialUser")
public class CartBean implements Cart, Serializable {
    List<String> contents;
    String customerId;
    String customerName;

    @Override
    public void initialize(String person) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }
```

```
        customerId = "0";
        contents = new ArrayList<>();
    }

    @Override
    public void initialize(String person, String id) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }
        IdVerifier idChecker = new IdVerifier();

        if (idChecker.validate(id)) {
            customerId = id;
        } else {
            throw new BookException("Invalid id: " + id);
        }

        contents = new ArrayList<>();
    }

    @Override
    @RolesAllowed("TutorialUser")
    public void addBook(String title) {
        contents.add(title);
    }
    @Override
    @RolesAllowed("TutorialUser")
    public void removeBook(String title) throws BookException {
        boolean result = contents.remove(title);

        if (result == false) {
            throw new BookException("\"" + title + "\" not in cart.");
        }
    }

    @Override
    @RolesAllowed("TutorialUser")
    public List<String> getContents() {
        return contents;
    }

    @Override
    @Remove()
    @RolesAllowed("TutorialUser")
    public void remove() {
        contents = null;
    }
}
```

这里对想要限制访问的方法指定了 @RolesAllowed 注解。在这个例子中，只有 TutorialUser 角色的用户可以为购物车增加或删除图书，以及列出购物车的内容。@RolesAllowed 注解隐式地声明了将在应用中引用的一个角色；因此，不需要 @DeclareRoles 注解。由于有 @RolesAllowed 注解，因此这也隐式地声明了用户访问这些方法时需要进行认证。如果部署描述文件中没有指定认证方法，认证类型将是用户名 / 密码认证。

20.3.1.2 使用 NetBeans IDE 运行 cart-secure 示例

1. 完成 19.4.1 节中的步骤。
2. 从 File（文件）菜单选择中 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/security
```

4. 选择 cart-secure 文件夹。
5. 选择 Open Required Projects（打开必要工程）复选框。
6. 单击 Open Project。
7. 在 Projects（工程）标签页中，右键单击 cart-secure 工程并选择 Build（构建）。

这一步会构建应用并打包到 cart-secure.ear，这个文件位于 cart-secure-ear/target/ 目录，然后将这个 EAR 文件部署到 GlassFish 服务器实例，获取客户端桩，并运行客户端。

8. 在 Login for user:（用户登录：）对话框中，输入 GlassFish 服务器中创建的一个 file 安全域用户的用户名和密码，并分配到组 TutorialUser，然后单击 OK。

如果输入的用户名和密码通过了认证，应用客户端的输出将出现在 Output（输出）标签页中：

```
...
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
Java Result: 1
...
```

如果用户名和密码未得到认证，这个对话框会再次出现，直到你输入正确的值为止。

20.3.1.3 使用 Maven 运行 cart-secure 示例

1. 完成 19.4.1 节中的步骤。
2. 在一个终端窗口中导航至：

```
tut-install/examples/security/cart-secure/
```

3. 要构建这个应用，把它打包到 cart-secure-ear/target 子目录的一个 EAR 文件

中，完成部署，运行这个应用，在终端窗口或命令行提示窗口输入以下命令：

```
mvn install
```

4. 在 Login for user: 对话框中，输入 GlassFish 服务器中创建的一个 file 安全域用户的用户名和密码，并分配到组 TutorialUser，然后单击 OK。

如果输入的用户名和密码通过了认证，应用客户端的输出将出现在 Output（输出）标签页中：

```
...
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
Java Result: 1
...
```

如果用户名和密码未得到认证，这个对话框会再次出现，直到你输入正确的值为止。

20.3.2 converter-secure 示例：使用编程式安全保护企业 bean 安全

这个例子展示如何对一个企业 bean 使用 getCallerPrincipal 和 isCallerInRole 方法。这个例子以一个非常简单的 EJB 应用 converter 作为起点，并修改 ConverterBean 的方法，使得只有当请求者的角色是 TutorialUser 才会显示货币转换结果。

可以在 *tut-install*/examples/security/converter-secure 目录中找到这个例子。这里以不安全的企业 bean 应用 converter 为基础，converter 应用见第 4 章，位于 *tut-install*/examples/ejb/converter/ 目录。本节将基于这个例子，通过增加必要的元素，使用 getCallerPrincipal 和 isCallerInRole 方法来保护应用安全，这些方法在 20.2.2.1 节中做过更详细的讨论。

一般地，要对一个企业 bean 使用 getCallerPrincipal 和 isCallerInRole 方法，必须完成以下步骤。对于本书的示例应用，很多步骤已经为你完成了，这里列出这些步骤只是为了说明希望创建一个类似的应用时需要做哪些工作。

1. 创建一个简单的企业 bean 应用。

2. 在 GlassFish 服务器 file 安全域中建立一个用户，指定到组 TutorialUser，并建立默认的主体 – 角色映射。为此，要完成 19.4.1 节中的步骤。

3. 修改 bean，增加 getCallerPrincipal 和 isCallerInRole 方法。

4. 如果应用包含一个 Web 客户端（servlet），则为这个 servlet 指定安全性，参见 19.4.2.1 节中的介绍。

5. 构建、打包、部署和运行这个应用。

20.3.2.1 修改 ConverterBean

修改原 ConverterBean 类的源代码，增加 if..else 子句来检查调用者的角色是否

是 TutorialUser。如果用户有适当的角色，就会计算和显示货币转换结果。如果用户的角色不正确，则不会完成计算，应用会显示结果为0。代码示例见 *tut-install*/examples/security/converter-secure/converter-secure-ejb/src/main/java/javaeetutorial/converter/ejb/ConverterBean.java。

代码段（修改部分用粗体显示）如下所示：

```
package javaeetutorial.convertersecure.ejb;

import java.math.BigDecimal;
import java.security.Principal;
import javax.ejb.Stateless;
import javax.annotation.Resource;
import javax.ejb.SessionContext;
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
@Stateless()
@DeclareRoles("TutorialUser")
public class ConverterBean{

    @Resource SessionContext ctx;
    private final BigDecimal yenRate = new BigDecimal("104.34");
    private final BigDecimal euroRate = new BigDecimal("0.007");

    @RolesAllowed("TutorialUser")
     public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = new BigDecimal("0.0");
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (ctx.isCallerInRole("TutorialUser")) {
            result = dollars.multiply(yenRate);
            return result.setScale(2, BigDecimal.ROUND_UP);
        } else {
            return result.setScale(2, BigDecimal.ROUND_UP);
        }
    }

    @RolesAllowed("TutorialUser")
    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = new BigDecimal("0.0");
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (ctx.isCallerInRole("TutorialUser")) {
            result = yen.multiply(euroRate);
            return result.setScale(2, BigDecimal.ROUND_UP);
        } else {
            return result.setScale(2, BigDecimal.ROUND_UP);
        }
    }
}
```

20.3.2.2 修改 ConverterServlet

使用以下注解为 converterWeb 客户端 ConverterServlet 指定安全性：

```
@WebServlet(urlPatterns = {"/"})
@ServletSecurity(
@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
    rolesAllowed = {"TutorialUser"}))
```

20.3.2.3 使用 NetBeans IDE 运行 converter-secure 示例

1. 完成 19.4.1 节中的步骤。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

tut-install/examples/security

4. 选择 converter-secure 文件夹。
5. 单击 Open Project。
6. 在 Projects(工程）标签页中，右键单击 converter-secure 工程并选择 Build(构建)。

这个命令会构建示例应用并将其部署到 GlassFish 服务器实例。

20.3.2.4 使用 Maven 运行 converter-secure 示例

1. 完成 19.4.1 节中的步骤。
2. 在一个终端窗口中导航至：

tut-install/examples/security/converter-secure/

3. 输入以下命令：

```
mvn install
```

这个命令会构建应用并打包到一个 WAR 文件 converter-secure.war，这个文件位于 target 目录，部署这个 WAR 文件。

20.3.2.5 运行 converter-secure 示例

1. 打开 Web 浏览器访问以下 URL：

http://localhost:8080/converter-secure

会出现一个要求认证（Authentication Required）对话框。

2. 输入一个用户名和密码组合，对应于 GlassFish 服务器 file 安全域中创建的一个用户（而且已指定到组 TutorialUser），然后单击 OK。
3. 在输入域中输入 100，然后单击 Submit（提交）。

会出现第二个页面，显示转换后的值。

第 21 章 Chapter 21

Java EE 安全：高级主题

本章介绍保护 Java EE 应用的一些高级主题，包括以下内容：

- 使用数字证书
- 认证机制
- 使用 JDBC 安全域完成用户认证
- 保护 HTTP 资源安全
- 保护应用客户端安全
- 保护企业信息系统应用安全
- 使用部署描述文件配置安全
- 关于高级安全主题的更多信息

21.1 使用数字证书

已经为 GlassFish 服务器生成了数字证书，可以在目录 *domain-dir*/config/ 中找到。这些数字证书是自签名的，可以在开发环境中使用；这些数字证书并不用于生产环境。在生产环境中，需要生成你自己的证书，并由证书授权机构（Certificate Authority，CA）签名。

要使用安全套接字层（Secure Sockets Layer，SSL），对于每一个外部接口或 IP 地址，应用或 Web 服务器必须有一个相关的证书来建立安全连接。这个设计的前提是服务器应当提供某种合理的保证，可以保证证书的所有者确实是你认为的那个人，特别是在要接收敏感信息时。可以认为证书是一个互联网地址的“数字驾照”。证书会指出网站与哪个公司关联，并提供有关这个网站所有者或管理员的一些基本联系信息。

数字证书由其所有者加密签名，他人很难仿造。对于涉及电子商务或其他商业交易的网站，如果身份认证很重要，可以从一些知名 CA 购买证书，如 VeriSign 或 Thawte。如果你的服务器证书是自签名的，则必须把它安装在 GlassFish 服务器密钥库文件中（keystore.jks）。如果你的客户端证书是自签名的，则应当把它安装在 GlassFish 服务器信任库文件中（cacerts.jks）。

有时，认证可能并不重要。例如，一个管理员可能只是想确保由服务器传输和接收的数据是私有的，不会被在连接上窃听的其他人窃取。在这些情况下，就不用获取 CA 证书，可以把这个时间和花费节省下来，只需要使用一个自签名的证书。

SSL 使用**公钥加密**（public-key cryptography），这要基于密钥对加密。**密钥对**（Key pairs）包含一个公钥和一个私钥。用一个密钥加密的数据只能由密钥对中另一个密钥解密。这个特性是在事务中建立信任和保密性的基础。例如，使用 SSL 时，服务器会计算一个值，并使用它的私钥来加密。加密后的值称为一个**数字签名**（digital signature）。客户端使用服务器的公钥对这个加密值解密，将这个值与它自己计算的值进行比较。如果两个值一致，则客户端可以信任这个签名是真实的，因为只有使用私钥才能生成这样一个签名。

HTTPS 要结合数字证书来认证 Web 客户端。除非安装了数字证书，否则大多数 Web 服务器的 HTTPS 服务都不会运行。可以使用 21.1.1 节中介绍的过程建立一个数字证书，你的应用或 Web 服务器可以使用这个证书来支持 SSL。

建立数字证书的一个工具是 keytool，这是 JDK 附带的一个密钥和证书管理工具。这个工具允许用户管理他们自己的公钥 / 私钥对以及用于自认证的相关证书，通过使用数字证书，用户可以向其他用户或服务（数据完整性和认证服务）完成自认证。这个工具还允许用户以证书形式缓存其通信方的公钥。

要更好地了解 keytool 和公钥加密，可参见 21.8 节，其中提供了 keytool 文档的一个链接。

21.1.1 创建服务器证书

已经为 GlassFish 服务器创建了一个服务器证书，可以在 *domain-dir*/config/ 目录中找到。这个服务器证书位于 keystore.jks。cacerts.jks 文件包含所有信任证书（包括客户端证书）。

如果必要，可以使用 keytool 生成证书。keytool 工具将密钥和证书存储在一个称为**密钥库**（keystore）的文件中，这是用来鉴别客户端或服务器证书的存储库。一般地，密钥库文件包含一个客户端或一个服务器的身份。密钥库通过使用一个密码来保护私钥。

如果指定密钥库文件名时没有指定目录，则会在运行 keytool 命令的目录中创建密钥库。这可以是应用所在的目录，也可能是多个应用通用的一个目录。

创建服务器证书的一般步骤如下：

1. 创建密钥库。

2. 从密钥库导出证书。

3. 签署证书。

4. 将证书导入到一个**信任库**（truststore）：这是用来验证证书的一个证书存储库。信任库通常包含多个证书。

下一节将介绍如何使用 keytool 工具完成这些步骤。

21.1.1.1 使用 keytool 创建服务器证书

运行 keytool 在默认开发密钥库文件 keystore.jks 中生成一个新的密钥对。这个例子使用别名 server-alias 来生成一个新的公钥 / 私钥对，并将公钥包装在 keystore.jks 的一个自签名证书中。这个密钥对使用一个 RSA 算法生成，默认密码为 changeit。关于创建和管理密钥库文件的更多信息和示例，请参见 keytool 文档。

说明 RSA 是 RSA 数据安全公司开发的一个公钥加密技术。

在要创建密钥对的目录中运行 keytool，步骤如下。

1. 生成服务器证书。

输入 keytool 命令（整个命令都在同一行上）：

```
java-home/bin/keytool -genkey -alias server-alias -keyalg RSA
-keypass changeit -storepass changeit -keystore keystore.jks
```

按下回车键时，keytool 会提示你输入服务器名、单位、组织机构、位置、州和国家代码。

必须输入服务器名来响应 keytool 的第一个提示，这里会要求输入名和姓。进行测试时，可以输入 localhost。

2. 将 keystore.jks 中生成的服务器证书导入到文件 server.cer 中。

输入 keytool 命令（整个命令都在同一行上）：

```
java-home/bin/keytool -export -alias server-alias -storepass changeit
-file server.cer -keystore keystore.jks
```

3. 如果希望一个 CA 签署这个证书，可以参考 keytool 文档中的例子。

4. 要将这个服务器证书增加到信任库文件 cacerts.jks，需从创建密钥库和服务器证书的目录运行 keytool。

使用以下参数：

```
java-home/bin/keytool -import -v -trustcacerts -alias server-alias
-file server.cer -keystore cacerts.jks -keypass changeit
-storepass changeit
```

这将显示证书上的信息，如下所示：

```
Owner: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
Issuer: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA,
 C=US
```

```
Serial number: 3e932169
Valid from: Mon Nov 26 18:15:47 EST 2012 until: Sun Feb 24 18:15:47 EST
 2013
Certificate fingerprints:
         MD5: 52:9F:49:68:ED:78:6F:39:87:F3:98:B3:6A:6B:0F:90
         SHA1: EE:2E:2A:A6:9E:03:9A:3A:1C:17:4A:28:5E:97:20:78:3F:
         SHA256: 80:05:EC:7E:50:50:5D:AA:A3:53:F1:11:9B:19:EB:0D:20:67:
          C1:12:AF:42:EC:CD:66:8C:BD:99:AD:D9:76:95
         Signature algorithm name: SHA256withRSA
         Version: 3
         ...
Trust this certificate? [no]:
```

5. 输入 yes，然后按回车键。

会出现以下信息：

```
Certificate was added to keystore
[Storing cacerts.jks]
```

21.1.2 向证书安全域增加用户

在 certificate 安全域中，会在 GlassFish 服务器安全上下文中建立用户身份，并为之填充从已认证客户端证书得到的用户数据。参见 21.1 节，其中详细介绍了创建这种证书的步骤。

21.1.3 为 GlassFish 服务器使用一个不同的服务器证书

按照 21.1.1 节中的步骤创建你自己的服务器证书，由一个 CA 签署，并把这个证书导入到 keystore.jks。

创建证书时要遵循以下规则：

- 创建服务器证书时，keytool 会提示输入你的名和姓。作为响应，必须输入你的服务器名。测试时可以输入 localhost。
- 如果想替换现有的 keystore.jks，必须把你的密钥库密码改为默认密码（changeit），或者将默认密码改为你的密钥库密码。

21.1.3.1 指定一个不同的服务器证书

要指定 GlassFish 服务器使用这个新密钥库来完成认证和授权决策，必须为 GlassFish 服务器设置 JVM 选项，使它能识别这个新密钥库。要使用其他人提供的一个不同的密钥库来完成开发，可以采用以下步骤：

1. 如果还没有启动 GlassFish 服务器，首先要启动 GlassFish 服务器。有关如何启动 GlassFish 服务器的信息参见 2.2 节。

2. 在一个 Web 浏览器中打开 GlassFish 服务器管理控制台：http://localhost:4848.

3. 展开 Configurations（配置），然后展开 server-config（服务器配置），再单击 JVM Settings（JVM 设置）。

4. 单击 JVM Options（JVM 选项）标签页。

5. 修改以下 JVM 选项，指向新密钥库的位置和名。当前设置如下所示：

```
-Djavax.net.ssl.keyStore=${com.sun.aas.instanceRoot}/config/keystore.jks
-Djavax.net.ssl.trustStore=${com.sun.aas.instanceRoot}/config/cacerts.jks
```

6. 如果改变了密钥库密码的默认值，还需要增加密码选项：

```
-Djavax.net.ssl.keyStorePassword=your-new-password
```

7. 单击 Save（保存），然后重启 GlassFish 服务器。

21.2 认证机制

这一节讨论客户端认证和相互认证机制。

21.2.1 客户端认证

利用**客户端认证**（client authentication），Web 服务器通过使用客户端的公钥证书来认证客户端。与基本认证或基于表单的认证相比，客户端认证是一种更安全的认证方法。它使用了基于 SSL 的 HTTP（HTTPS），也就是说，服务器会使用客户端的公钥证书来认证客户端。SSL 技术为 TCP/IP 连接提供了数据加密、服务器认证、消息完整性和可选的客户端认证。可以把公钥证书认为是一个数字护照。证书由一个信任机构即证书授权机构（CA）签发，为持有者提供身份鉴别。

使用客户端认证之前，要确保客户端有一个合法的公钥证书。有关创建和使用公钥证书的更多信息，参见 21.1 节。

下面显示了如何在部署描述文件中声明客户端认证：

```
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

21.2.2 相互认证

使用**相互认证**（mutual authentication）时，服务器和客户端会相互完成认证。相互认证包括两种类型：

- 基于证书的认证（参见图 21-1）
- 基于用户名 / 密码的认证（参见图 21-2）

使用基于证书的相互认证时，会完成以下动作：

1. 客户端请求访问一个受保护的资源。
2. Web 服务器向客户端提供服务器证书。
3. 客户端验证服务器的证书。

4. 如果成功，客户端向服务器发送客户端证书。

5. 服务器验证客户端的证书。

6. 如果成功，服务器允许客户端访问所请求的受保护资源。

图 21-1 显示了基于证书的相互认证中发生的动作。

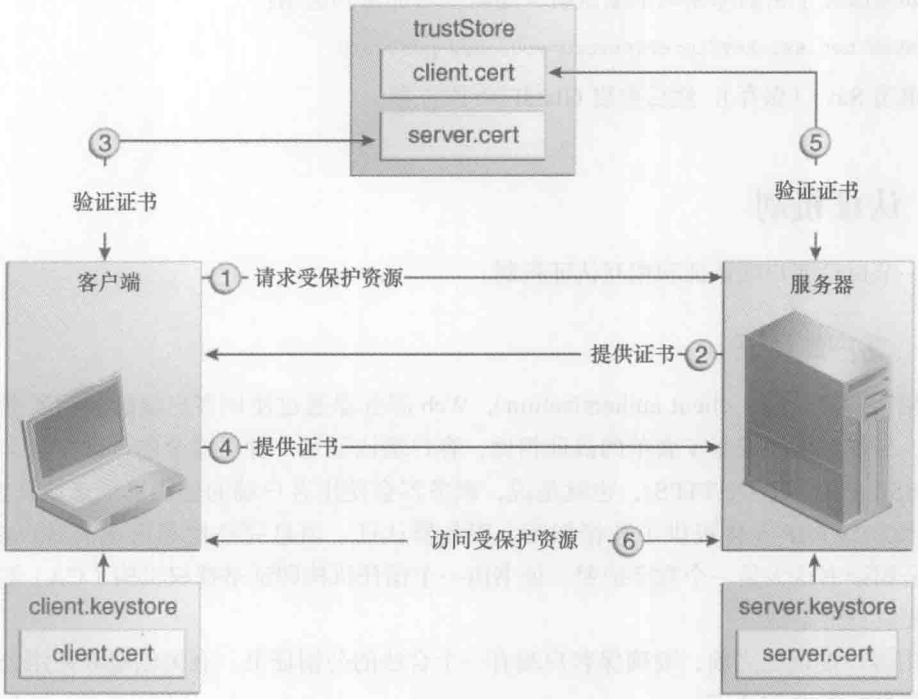


图 21-1 基于证书的相互认证

在基于用户名 / 密码的相互认证中，会完成以下动作。

1. 客户端请求访问一个受保护的资源。

2. Web 服务器向客户端提供服务器证书。

3. 客户端验证服务器的证书。

4. 如果成功，客户端向服务器发送它的用户名和密码。

5. 服务器验证客户端的凭据。

6. 如果验证成功，服务器允许客户端访问所请求的受保护资源。

图 21-2 显示了基于用户名 / 密码的相互认证中发生的动作。

21.2.2.1 支持基于 SSL 的相互认证

这一节讨论如何建立客户端认证。同时支持服务器端和客户端认证称为相互认证或双向认证。在客户端认证中，要求客户端提交你接受的一个证书授权机构签发的证书。

至少有两种方法来支持基于 SSL 的相互认证。

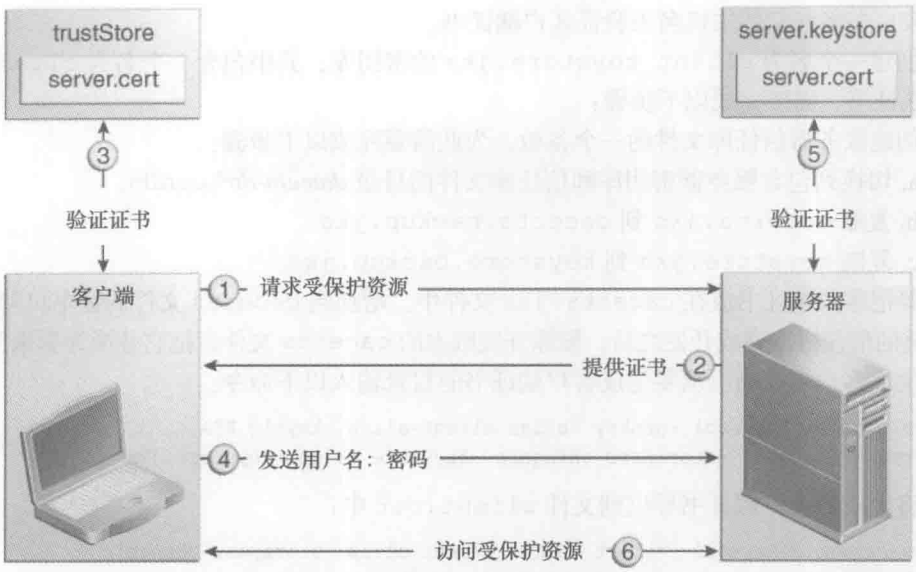


图 21-2 基于用户名 / 密码的相互认证

- 首选方法是在应用部署描述文件 web.xml 中将认证方法设置为 CLIENT-CERT。通过修改给定应用的部署描述文件来完成相互认证。采用这种方式，只会对安全约束控制的一个特定资源启用客户端认证，而且只有当应用要求客户端认证时才完成检查。
- 另外还有一种不太常用的方法，在接收一个连接之前，如果希望 SSL 栈从客户端得到一个合法的证书链，可以将 certificate 安全域中的 clientAuth 属性设置为 true。如果值为 false（默认值），则不要求一个证书链，除非客户端请求一个受安全约束（使用 CLIENT-CERT 认证）保护的资源。将 clientAuth 属性设置为 true 来启用客户端认证时，经过指定 SSL 端口的所有请求都要求完成客户端认证。如果打开 clientAuth，则会一直打开客户端认证，这会严重影响性能。

如果同时采用这两种方式启用客户端认证，则会完成两次客户端认证。

21.2.2.2 创建客户端证书完成相互认证

如果有一个由可信证书授权机构（CA，如 Verisign）签发的证书，而且 GlassFish 服务器 cacerts.jks 文件已经包含该 CA 验证的一个证书，就不再需要这一步。只有当它是你的自签名证书时，才需要把这个证书安装在 GlassFish 服务器证书文件中。

按照以下步骤，从要创建客户端证书的目录运行 keytool。按下回车键时，keytool 会提示你输入服务器名、单位、组织机构、位置、州和国家代码。

必须输入服务器名来响应 keytool 的第一个提示，这里会要求输入名和姓。测试时可以输入 localhost。如果这个例子是为了验证相互认证，而且你收到一个运行时错误，指出 HTTPS 主机名错误，则需要重新创建客户端证书，确保主机名为运行示例时所要使用的同一个主机名。例如，如果你的主机名是 duke，则输入 duke 作为证书 CN，或者在提示输入名和姓时输入这个主机名。访问应用时，输入一个指向同一个位置的 URL（例如，https://duke:8181/mutualauth/hello）。这很有必要，因为在 SSL 握手期间，服务

器会比较证书名和它的主机名来验证客户端证书。

要创建一个名为 client_keystore.jks 的密钥库，其中包含一个名为 client.cer 的客户端证书，需要完成以下步骤：

1. 创建服务器信任库文件的一个备份，为此需要完成以下步骤：

   a. 切换到包含服务器密钥库和信任库文件的目录 *domain-dir*\config。

   b. 复制 cacerts.jks 到 cacerts.backup.jks。

   c. 复制 keystore.jks 到 keystore.backup.jks。

不要把客户端证书放在 cacerts.jks 文件中。增加到 cacerts 文件的证书可能成为所有证书链的信任根。完成开发之后，删除开发版本的 cacerts 文件，把它替换为原来的版本。

2. 生成客户端证书。从要生成客户端证书的目录输入以下命令：

```
java-home\bin\keytool -genkey -alias client-alias -keyalg RSA
-keypass changeit -storepass changeit -keystore client_keystore.jks
```

3. 将生成的客户端证书导出到文件 client.cer 中：

```
java-home\bin\keytool -export -alias client-alias -storepass changeit
-file client.cer -keystore client_keystore.jks
```

4. 将这个证书增加到信任库文件 *domain-dir*/config/cacerts.jks。从创建密钥库和客户端证书的目录运行 keytool。使用以下参数：

```
java-home\bin\keytool -import -v -trustcacerts -alias client-alias
-file client.cer -keystore domain-dir/config/cacerts.jks
-keypass changeit -storepass changeit
```

Keytool 工具会返回类似下面的消息：

```
Owner: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
Issuer: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA,
 C=US
Serial number: 3e39e66a
Valid from: Tue Nov 27 12:22:47 EST 2012 until: Mon Feb 25 12:22:47 EST
 2013
Certificate fingerprints:
    MD5: 5A:B0:4C:88:4E:F8:EF:E9:E5:8B:53:BD:D0:AA:8E:5A
    SHA1:90:00:36:5B:E0:A7:A2:BD:67:DB:EA:37:B9:61:3E:26:B3:89:46:32
    Signature algorithm name: SHA1withRSA
    Version: 3
Trust this certificate? [no]: yes
Certificate was added to keystore
[Storing cacerts.jks]
```

5. 重启 GlassFish 服务器。

21.3 使用 JDBC 安全域完成用户认证

认证安全域有时称为**安全策略域**（*security policy domain*）或**安全域**（*security domain*），

在这个作用域上，应用服务器会定义并执行一个通用安全策略。安全域包含一个用户集合，这些用户可能会分配到一个组（也可能不分配）。GlassFish服务器已经预配置了文件、证书和管理安全域。管理员还可以建立LDAP、JDBC、摘要或定制安全域。

应用可以在其部署描述文件中指定要使用哪个安全域。如果应用没有指定安全域，GlassFish服务器会使用它的默认安全域，即file安全域。如果应用指定要使用JDBC安全域完成用户认证，GlassFish服务器将从一个数据库获取用户凭据。应用服务器会使用这个数据库信息以及配置文件中启用的JDBC安全域选项。

数据库提供了一种很方便的方法，可以在运行时增加、编辑或删除用户，而且允许用户自行创建自己的账户而无需提供帮助。使用数据库还有一个额外的好处：可以安全地存储额外的用户信息。可以把安全域认为是用户名和密码的一个数据库，基于与各个合法用户关联的角色列表，可以用这个数据库验证一个或一组Web应用的合法用户。使用JDBC安全域时，会授权特定角色的所有用户可以访问特定的Web应用资源，而不用列出一个相关的用户列表。可以有多个角色与一个用户名关联。

第29章和第30章提供了两个教程案例研究，其中都使用了JDBC安全域来完成用户认证。

21.3.1 配置JDBC认证安全域

GlassFish服务器允许管理员在JDBC安全域中指定用户的凭据（用户名和密码），而不是在连接池中指定。这样可以避免其他应用通过浏览数据库表来查看用户凭据。默认地，JDBC安全域不支持将密码作为明文存储。在正常情况下，密码都不应作为明文存储。

1. 创建数据库表（将在这些数据库表中存储安全域的用户凭据）。

2. 将用户凭据增加到所创建的数据库表。

3. 为数据库创建一个JDBC连接池。

可以使用管理控制台或命令行来创建一个连接池。

4. 为数据库创建一个JDBC资源。

可以使用管理控制台或命令行来创建一个JDBC资源。

5. 创建一个安全域。

这一步需要将资源与安全域关联，为用于认证的用户和组定义表和列，并定义摘要算法（将使用这个摘要算法在数据库中存储密码）。

可以使用管理控制台或命令行来创建安全域。

6. 修改应用的部署描述文件，指定JDBC安全域。

- 对于EAR文件中的企业应用，修改glassfish-application.xml文件。
- 对于WAR文件中的Web应用，修改web.xml文件。
- 对于EJB JAR文件中的企业bean，修改glassfish-ejb-jar.xml文件。

例如，对于一个假想的应用，web.xml文件可以指定jdbcRealm安全域，如下所示：

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>jdbcRealm</realm-name>
    <form-login-config>
        <form-login-page>/login.xhtml</form-login-page>
        <form-error-page>/login.xhtml</form-error-page>
    </form-login-config>
</login-config>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Secure Pages</web-resource-name>
        <description/>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>ADMINS</role-name>
    </auth-constraint>
</security-constraint>
```

/admin 下的所有 Web 页面都指定了基于表单的登录。只允许 ADMINS 角色的用户访问这些页面。

7. 为安全域中的用户或用户组分配安全角色。

为组或用户分配安全角色时，要为应用服务器的部署描述文件增加一个 security-role-mapping 元素，在这里应用服务器的部署描述文件是 glassfish-web.xml：

```
<security-role-mapping>
    <role-name>USERS</role-name>
    <group-name>USERS</group-name>
</security-role-mapping>
<security-role-mapping>
    <role-name>ADMINS</role-name>
    <group-name>ADMINS</group-name>
</security-role-mapping>
```

对于 GlassFish 服务器，由于用户创建过程中就会将用户分配到相应的组，所以这比为单个用户映射安全角色更为高效。

21.4 保护 HTTP 资源安全

如果请求 URI 与多个受约束 URL 模式匹配，那么对于这个请求，会应用与最匹配的 URL 模式相关联的约束。要确定与请求 URI 最匹配的 URL 模式，可以使用 Java Servlet 3.1 规范第 12 章中定义的 servlet 匹配规则。如果请求 URI 与受约束 URL 模式不匹配，则不会对这个请求应用任何保护需求。请求的 HTTP 方法对于为请求选择最匹配的 URL 模式没有影响。

如果一个约束定义中列出了 HTTP 方法，则这个约束定义的保护只应用于所列的方法。

约束定义中没有列出 HTTP 方法时，这个约束定义的保护将应用到所有 HTTP 方法，包括 HTTP 扩展方法。

对相同的 URL 模式和 HTTP 方法组合应用有不同的保护需求的约束时，这些保护需求的组合规则在 Java Servlet 3.1 规范的 13.8.1 节中定义。

可以按照以下原则适当地保护一个 Web 应用的安全。

- 不要在约束定义中列出 HTTP 方法。这是最简单的保护方法，可以避免 HTTP 方法失去保护。例如：

```
<!-- SECURITY CONSTRAINT #1 -->
<security-constraint>
    <display-name>Do not enumerate Http Methods</display-name>
    <web-resource-collection>
        <url-pattern>/company/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>sales</role-name>
    </auth-constraint>
</security-constraint>
```

如果在约束中列出了一些方法，则由于实际上 HTTP 方法（包括扩展方法）几乎是无限的，所以所有未列出的方法都将不受保护。一般情况下不要使用这样的约束，除非你确信这确实是你想要定义的保护机制。下面的例子显示了一个约束，其中只列出了 GET 方法，因此对于所有其他可能的 HTTP 方法没有定义任何保护：

```
<!-- SECURITY CONSTRAINT #2 -->
<security-constraint>
    <display-name>
        Protect GET only, leave all other methods unprotected
    </display-name>
    <web-resource-collection>
        <url-pattern>/company/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>sales</role-name>
    </auth-constraint>
</security-constraint>
```

- 如果需要对特定的 HTTP 方法应用特定类型的保护，应确保定义约束时要涵盖相应 URL 模式允许访问的每一个方法（有约束或无约束）。如果有些方法不允许访问，还必须创建一个约束，拒绝相应模式访问这些方法；下一个原则中的安全约束 #5 给出了这样一个例子。

例如，如果允许访问 GET 和 POST，其中 POST 要求认证，GET 没有任何约束，可以定义以下约束：

```
<!-- SECURITY CONSTRAINT #3 -->
<security-constraint>
    <display-name>Allow unprotected GET</display-name>
    <web-resource-collection>
        <url-pattern>/company/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
</security-constraint>

<!-- SECURITY CONSTRAINT #4 -->
<security-constraint>
    <display-name>Require authentication for POST</display-name>
    <web-resource-collection>
        <url-pattern>/company/*</url-pattern>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>sales</role-name>
    </auth-constraint>
</security-constraint>
```

- 如果除了允许访问的方法外想要拒绝所有其他 HTTP 方法，最简单的方法就是使用 `http-method-omission` 元素从安全约束排除那些方法，另外还要定义一个不指定任何角色的 `auth-constraint`。除了 `http-method-omission` 元素中指定的那些方法外，所有其他方法都会应用这个安全约束，另外只有与约束中的模式匹配的资源才会应用这个约束。

例如，下面的约束适用于与模式 `/company/*` 匹配的资源，除了 GET 和 POST 外，将拒绝所有其他方法：

```
<!-- SECURITY CONSTRAINT #5 -->
<security-constraint>
    <display-name>Deny all HTTP methods except GET and POST</display-name>
    <web-resource-collection>
        <url-pattern>/company/*</url-pattern>
        <http-method-omission>GET</http-method-omission>
        <http-method-omission>POST</http-method-omission>
    </web-resource-collection>
    <auth-constraint/>
</security-constraint>
```

如果希望将这个排除机制扩展到应用中未约束的部分，还要包括 URL 模式 /（前斜线）：

```
<!-- SECURITY CONSTRAINT #6 -->
<security-constraint>
    <display-name>Deny all HTTP methods except GET and POST</display-name>
    <web-resource-collection>
        <url-pattern>/company/*</url-pattern>
        <url-pattern>/</url-pattern>
        <http-method-omission>GET</http-method-omission>
```

```
        <http-method-omission>POST</http-method-omission>
    </web-resource-collection>
    <auth-constraint/>
</security-constraint>
```

- 对于你的Web应用，如果不希望访问任何资源（除非显式地定义了一个约束允许访问某个资源），则可以定义一个auth-constraint，不指定任何角色，并把它与URL模式/关联。URL模式/是最弱的匹配模式。不要在这个约束中列出任何HTTP方法：

```
<!-- SECURITY CONSTRAINT #7 -->
<security-constraint>
    <display-name>
        Switch from Constraint to Permission model
        (where everything is denied by default)
    </display-name>
    <web-resource-collection>
        <url-pattern>/</url-pattern>
    </web-resource-collection>
    <auth-constraint/>
</security-constraint>
```

21.5 保护应用客户端安全

应用客户端的Java EE认证需求与其他Java EE组件相同，另外可以使用面向其他Java EE应用组件的认证技术。访问未受保护的Web资源时不需要任何认证。

访问受保护Web资源时，可以使用通常的认证方法：HTTP基本认证、HTTP基于表单的认证或SSL客户端认证。19.2.3节介绍了如何指定HTTP基本认证和HTTP基于表单的认证。21.2.1节介绍了如何指定SSL客户端认证。

访问受保护企业bean时需要完成认证。企业bean的认证机制在20.2节中讨论。

应用客户端利用应用客户端容器提供的一个认证服务来完成用户的认证。这个容器的服务可以与平台的认证系统集成，从而提供一种登录功能。容器可以在应用启动时或访问一个受保护资源时完成用户认证。

应用客户端可以提供一个类来收集认证数据，这个类称为**登录模块**（login module）。如果提供这样一个类，它必须实现javax.security.auth.callback.CallbackHandler接口，而且必须在部署描述文件中指定这个类名。应用的回调处理器必须完全支持javax.security.auth.callback包中指定的Callback对象。

21.5.1 使用登录模块

应用客户端可以使用Java认证和授权服务（Java Authentication and Authorization Service，JAAS）来创建登录模块完成认证。基于JAAS的应用实现了javax.security.auth.

callback.CallbackHandler 接口，可以与用户交互来输入特定的认证数据，如用户名或密码，或者显示错误和警告消息。

应用实现了 CallbackHandler 接口，并将这个回调处理器传入登录上下文，登录上下文直接将它转发到底层登录模块。登录模块使用回调处理器一方面从用户收集输入（如密码或智能卡 PIN），另一方面为用户提供信息（如状态信息）。由于应用指定了回调处理器，因此底层登录模块可以保持独立，而不论应用采用何种方式与用户交互。

例如，对于一个 GUI 应用，回调处理器实现可能会显示一个窗口请求用户输入，对于一个命令行工具，回调处理器实现可能只是提示用户直接从命令行输入。

登录模块将一个合适的回调数组传递到回调处理器的 handle 方法（如对应用户名的 NameCallback，对应密码的 PasswordCallback）；回调处理器完成所请求的用户交互，并在回调中设置适当的值。例如，要处理 NameCallback，CallbackHandler 可能提示输入一个名，从用户获取值，并调用 NameCallback 的 setName 方法存储这个用户名。

在登录模块中使用 JAAS 完成认证的更多信息可以参见 21.8 节中所列的文档。

21.5.2 使用编程式登录

编程式登录允许客户端代码提供用户凭据。如果使用一个 EJB 客户端，则可以使用 com.sun.appserv.security.ProgrammaticLogin 类，它提供了方便的 login 和 logout 方法。编程式登录只适用于服务器。

21.6 保护企业信息系统应用安全

在企业信息系统（Enterprise Information Systems，EIS）应用中，组件会请求一个 EIS 资源的连接。作为这个连接的一部分，EIS 可能需要请求者登录才能访问这个资源。设计 EIS 登录时，应用组件提供者有两个选择：

- **容器托管登录**（Container-managed sign-on）：应用组件让容器负责配置和管理 EIS 登录。容器会确定用户名和密码来建立与一个 EIS 实例的连接。有关的更多信息，参见 21.6.1 节。
- **组件托管登录**（Component-managed sign-on）：应用组件代码中包含完成 EIS 登录过程的代码，以此来管理 EIS 登录。有关的更多信息，参见 21.6.2 节。

还可以为资源适配器配置安全。更多信息参见 21.6.3 节。

21.6.1 容器托管登录

在容器托管登录中，应用组件不必向 getConnection() 方法传递任何登录安全信息。安全信息由容器提供，如下例所示（方法调用用粗体显示）：

```
// Business method in an application component
Context initctx = new InitialContext();
// Perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory)initctx.lookup(
    "java:comp/env/eis/MainframeCxFactory");
// Invoke factory to obtain a connection. The security
// information is not passed in the getConnection method
javax.resource.cci.Connection cx = cxf.getConnection();
...
```

21.6.2 组件托管登录

在组件托管登录中，应用组件负责向getConnection方法传递资源所需的登录安全信息。例如，安全信息可能是一个用户名和密码，如下所示（方法调用用粗体显示）：

```
// Method in an application component
Context initctx = new InitialContext();

// Perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory)initctx.lookup(
    "java:comp/env/eis/MainframeCxFactory");

// Get a new ConnectionSpec
com.myeis.ConnectionSpecImpl properties = //..

// Invoke factory to obtain a connection
properties.setUserName("...");
properties.setPassword("...");
javax.resource.cci.Connection cx =
    cxf.getConnection(properties);
...
```

21.6.3 配置资源适配器安全

资源适配器是一个系统级软件组件，通常实现与一个外部资源管理器的网络连接。资源适配器可以扩展Java EE平台的功能，实现某个Java EE标准服务API，如JDBC驱动程序，或者为一个外部应用系统的连接器定义和实现一个资源适配器。资源适配器还可以提供完全本地的服务，可能与本地资源交互。资源适配器通过Java EE服务提供者接口（Java EE SPI）与Java EE平台交互。使用Java EE SPI关联到Java EE平台的资源适配器可以处理所有Java EE产品。

要为一个资源适配器配置安全设置，需要编辑资源适配器描述文件ra.xml。下面给出ra.xml文件的一部分，这里为一个资源适配器配置了安全属性：

```
<authentication-mechanism>
    <authentication-mechanism-type>
        BasicPassword
```

```
    </authentication-mechanism-type>
    <credential-interface>
        javax.resource.spi.security.PasswordCredential
    </credential-interface>
</authentication-mechanism>
<reauthentication-support>false</reauthentication-support>
```

查看 *as-install*/lib/schemas/connector_1_7.xsd，可以了解配置资源适配器安全选项的更多内容。资源适配器部署描述文件中可以配置以下元素：

- **认证机制**（Authentication mechanisms）：使用 authentication-mechanism 元素指定资源适配器支持的认证机制。这个支持针对的是资源适配器，而不是底层 EIS 实例。目前支持两类机制：
  - BasicPassword，支持以下接口：

    javax.resource.spi.security.PasswordCredential
  - Kerbv5，支持以下接口：

    javax.resource.spi.security.GenericCredential

  GlassFish 服务器目前不支持 Kerbv5 机制。
- **重认证支持**（Reauthentication support）：使用 reauthentication-support 元素指定资源适配器实现是否支持对已有 Managed-Connection 实例的重认证。可以为 true 或 false。
- **安全权限**（Security permissions）：使用 security-permission 元素指定资源适配器代码所需的安全权限。对安全权限的支持是可选的，当前版本的 GlassFish 服务器还不支持这个特性。不过，可以手动地更新 server.policy 文件，为资源适配器增加相关的权限。

部署描述文件中所列的安全权限与连接器规范中指定的默认权限所要求的安全权限有所不同。

关于安全权限规范的实现的更多信息参见 21.8 节中所列的安全策略文件文档。

除了在 ra.xml 文件中指定资源适配器外，还可以为连接器连接池创建一个安全映射，将应用主体或用户组映射到一个后端 EIS 主体。如果会使用一个或多个 EIS 后端主体来执行应用中不同主体或用户组启动的（EIS）操作，通常就会使用安全映射。

21.6.4 将应用主体映射到 EIS 主体

使用 GlassFish 服务器时，可以使用安全映射将应用的调用者身份（主体或用户组）映射到基于事务的容器托管后端中一个合适的 EIS 主体。一个应用主体对一个 EIS 发起请求时，GlassFish 服务器首先检查一个完全匹配的主体，为此要使用为连接器连接池定义的安全映射来确定所映射的后端 EIS 主体。如果没有完全匹配，则 GlassFish 服务器使用通配符规范（如果有）来确定所映射的后端 EIS 主体。应用用户需要执行一个 EIS 操作时，如果要求作为

EIS 中的一个特定身份来执行这个 EIS 操作，就要使用安全映射。

要使用安全映射，可以使用管理控制台。按照以下步骤从管理控制台进入安全映射页面。

1. 在导航树中，展开 Resources（资源）节点。
2. 展开 Connectors（连接器）节点。
3. 选择 Connector Connection Pools（连接器连接池）节点。
4. 在连接器连接池页面上，单击要创建安全映射的连接池。
5. 单击 Security Maps（安全映射）标签页。
6. 单击 New（新建）为这个连接池创建一个新的安全映射。
7. 输入用来引用安全映射的映射名以及其他必要的信息。

可以单击 Help（帮助）得到各个选项的更多信息。

21.7 使用部署描述文件配置安全

Java EE 7 平台中推荐使用注解来配置安全。如果想要在部署时覆盖安全设置，可以使用 `web.xml` 部署描述文件中的安全元素。这一节将介绍如何使用部署描述文件指定基本认证，以及如何覆盖默认的主体 - 角色映射。

21.7.1 在部署描述文件中指定基本认证安全

部署描述文件中可以设置元素为示例增加基本认证，这个元素告诉服务器或浏览器完成以下任务：

- 发出一个标准登录对话框来收集用户名和密码数据。
- 验证用户是否得到授权来访问这个应用。
- 如果得到授权，为用户显示 servlet。

下面的示例显示了一个部署描述文件中的安全元素，可以在 *tut-install*/`examples/security/hello2_basicauth/` 目录下的基本认证示例中使用这个部署描述文件：

```
<security-constraint>
    <display-name>SecurityConstraint</display-name>
    <web-resource-collection>
        <web-resource-name>WRCollection</web-resource-name>
        <url-pattern>/greeting</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>TutorialUser</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
<login-config>
```

```
    <auth-method>BASIC</auth-method>
    <realm-name>file</realm-name>
</login-config>
<security-role>
    <role-name>TutorialUser</role-name>
</security-role>
```

这个部署描述文件指定：要访问请求 URI /greeting，要求用户必须已输入用户名和密码，而且已经验证为角色 TutorialUser，所以得到授权允许访问这个 URL。用户名和密码数据将通过一个受保护的传输通道来发送，以保证它们在传输过程中不会被读取。

21.7.2 在部署描述文件中指定非默认主体 - 角色映射

要把应用或模块允许的一个角色名映射到服务器上定义的主体（用户）和组，可以在运行时部署描述文件（glassfish-application.xml，glassfish-web.xml 或 glassfish-ejb-jar.xml）中使用 security-role-mapping 元素。这个元素可以声明应用中使用的一个安全角色与为 GlassFish 服务器安全域定义的一个或多个组或主体之间的一个映射。下面给出一个 glassfish-web.xml 文件的例子：

```
<glassfish-web-app>
    <security-role-mapping>
        <role-name>DIRECTOR</role-name>
        <principal-name>schwartz</principal-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>DEPT-ADMIN</role-name>
        <group-name>dept-admins</group-name>
    </security-role-mapping>
</glassfish-web-app>
```

这个角色名可以映射到一个特定的主体（用户）或组，或者映射到主体和组的一个组合。所引用的主体或组名必须是 GlassFish 服务器当前默认安全域中合法的主体或组。这个例子中的 role-name 必须与相应的 web.xml 文件的 security-role 元素中的 role-name 完全匹配，或者与 @DeclareRoles 和 / 或 @RolesAllowed 注解中定义的角色名完全匹配。

21.8 关于高级安全主题的更多信息

关于这一章介绍的安全主题，更多信息可以参见：

- keytool 命令的相关文档：

  http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html

- Java 认证和授权服务（JAAS）参考指南：

  http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html

- Java 认证和授权服务（JAAS）：LoginModule 开发人员指南：

  http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASLMDevGuide.html

- 关于安全策略文件语法的文档：

  http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html#FileSyntax

第六部分 *Part 6*

Java EE 支持技术

第六部分研究 Java EE 平台的多种支持技术，包括以下各章：

- 第 22 章　事务
- 第 23 章　资源适配器和契约
- 第 24 章　资源适配器示例
- 第 25 章　使用 Java EE 拦截器
- 第 26 章　批处理
- 第 27 章　Java EE 的并发工具

Chapter 22 第 22 章

事　务

一般的企业应用都会在一个或多个数据库中访问和存储信息。由于这些信息对于业务操作非常重要，所以必须准确、即时而且可靠。如果允许多个程序同时更新相同的信息，或者如果一个系统处理交易时失败而导致相关数据部分更新，这就会破坏数据完整性。事务可以避免这些情况，从而确保数据完整性。事务会控制多个程序对数据的并发访问。如果出现系统失败，事务可以确保在系统恢复之后数据仍处于一致的状态。

本章将介绍以下内容：

- Java EE 应用中的事务
- 什么是事务
- 容器托管事务
- bean 托管事务
- 事务超时
- 更新多个数据库
- Web 组件中的事务
- 关于事务的更多信息

22.1　Java EE 应用中的事务

在 Java EE 应用中，事务是必须全部成功完成的一系列动作，否则每个动作所做的所有改变都必须回退。事务要么提交要么回滚。

Java 事务 API（Java Transaction API，JTA）允许应用采用一种独立于特定实现的方式访

问事务。JTA 指定了事务管理器与参与分布式事务系统的各个应用之间的标准 Java 接口，参与事务的应用包括：事务性应用、Java EE 服务器以及控制资源访问的管理器，这些管理器会控制受事务影响的共享资源访问。

JTA 定义了 UserTransaction 接口，应用使用这个接口开始、提交或回滚事务。应用组件使用 java:comp/UserTransaction 名或者请求注入一个 UserTransaction 对象从而通过 JNDI 查找得到一个 UserTransaction 对象。应用服务器使用 JTA 定义的一些接口与事务管理器通信，事务管理器则使用 JTA 定义的接口与资源管理器交互。

JTA 1.2 规范可以从以下地址得到：http://jcp.org/en/jsr/detail?id=907。

22.2 什么是事务

要模拟一个商业交易，程序可能需要完成多个步骤。例如，一个金融程序可能要使用以下伪代码中所列的步骤从一个支票账户向一个存款账户转账：

```
begin transaction
    debit checking account
    credit savings account
    update history log
commit transaction
```

以上 3 个步骤必须都完成，或者都不完成。否则，就会破坏数据完整性。由于一个事务中的步骤要作为一个整体，所以**事务**（transaction）通常定义为一个不可分割的工作单元。

事务可以有两种结果：提交或回滚。事务提交时，将保存事务中语句所做的数据修改。如果事务中的一个语句失败，这个事务就会回滚，并取消事务中所有语句的影响。例如，在上面的伪代码中，如果在 credit 步骤中磁盘崩溃，事务就会回滚，并取消 debit 语句所做的数据修改。尽管这个事务失败，但是数据完整性不受影响，因为账户仍保持平衡。

在前面的伪代码中，begin 和 commit 语句标志了事务的边界。设计一个企业 bean 时，可以通过指定容器托管或 bean 托管事务来确定如何设置事务边界。

22.3 容器托管事务

在一个采用**容器托管事务界定**（container-managed transaction demarcation）的企业 bean 中，由 EJB 容器设置事务的边界。可以对任意类型的企业 bean 使用容器托管事务：包括会话 bean 或消息驱动 bean。容器托管事务可以简化开发，因为企业 bean 代码不用显式地标志事务的边界。代码不包括开始和结束事务的语句。默认地，如果没有指定事务界定，企业 bean 就会使用容器托管事务界定。

一般地，容器会在企业 bean 方法开始之前开始一个事务，并在这个方法退出之前提交事务。每个方法可以与一个事务关联。一个方法中不允许有嵌套事务或多个事务。

容器托管事务不要求所有方法都与事务关联。开发一个 bean 时，可以设置事务属性来指定 bean 的哪些方法与事务关联。

使用容器托管事务界定的企业 bean 不能使用任何影响容器事务界定边界的事务管理方法。这些方法包括 `java.sql.Connection` 的 `commit`、`setAutoCommit` 和 `rollback` 方法或 `javax.jms.Session` 的 `commit` 和 `rollback` 方法。如果需要控制事务界定，必须使用应用托管事务界定。

另外，使用容器托管事务界定的企业 bean 不能使用 `javax.transaction.UserTransaction` 接口。

22.3.1 事务属性

事务属性可以控制事务的作用域。图 22-1 展示了控制这个作用域的重要性。在图中，`method-A` 开始一个事务，然后调用 `Bean-2` 的 `method-B`。`method-B` 执行时，它是在由 `method-A` 启动的事务作用域中运行还是在一个新的事务中执行取决于 `method-B` 的事务属性。

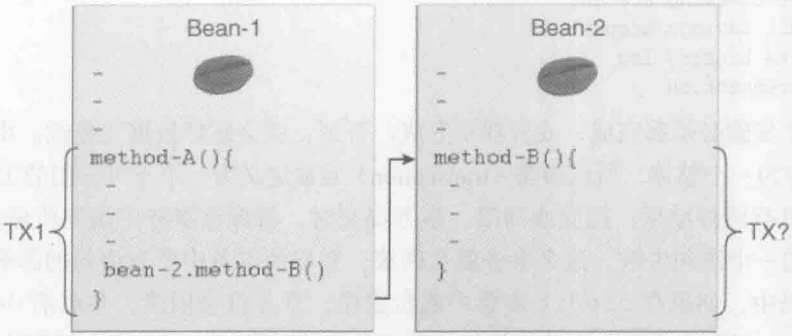


图 22-1 事务作用域

事务属性可以取以下的值：

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

22.3.1.1 Required 属性

如果客户端在一个事务中运行，并调用了企业 bean 的方法，该方法将在这个客户端的事务中执行。如果客户端未与事务关联，那么容器在运行这个方法之前会开始一个新事务。

对于采用容器托管事务界定方式运行的所有企业 bean 方法，Required 属性是隐含的事务属性。通常不需要特别设置 Required 属性，除非需要覆盖另一个事务属性。由于事务属性是声明式的，所以以后很容易修改。

22.3.1.2 RequiresNew 属性

如果客户端在一个事务中运行，并调用了企业 bean 的方法，容器会完成以下步骤：

1. 挂起客户端的事务
2. 开始一个新事务
3. 委托方法调用
4. 方法完成后恢复客户端的事务

如果客户端未与一个事务关联，容器在运行这个方法之前会开始一个新事务。

如果希望确保方法总在一个新事务中运行，就应当使用 RequiresNew 属性。

22.3.1.3 Mandatory 属性

如果客户端在一个事务中运行，并调用了企业 bean 的方法，这个方法将在客户端的事务中执行。如果客户端未与一个事务关联，容器会抛出一个 TransactionRequiredException。

如果企业 bean 的方法必须使用客户端的事务，就要使用 Mandatory 属性。

22.3.1.4 NotSupported 属性

如果客户端在一个事务中运行，并调用了企业 bean 的方法，容器在调用这个方法之前会挂起客户端的事务。方法完成之后，容器将恢复客户端的事务。

如果客户端未与一个事务关联，容器在运行这个方法之前不会开始一个新事务。

不需要事务的方法可以使用 NotSupported 属性。由于事务会带来开销，所以这个属性可以提高性能。

22.3.1.5 Supports 属性

如果客户端在一个事务中运行，并调用了企业 bean 的方法，这个方法会在客户端的事务中执行。如果客户端未与一个事务关联，容器在运行方法之前不会开始一个新事务。

由于方法的事务行为可能不同，所以使用 Supports 属性时要特别当心。

22.3.1.6 Never 属性

如果客户端在一个事务中运行，并调用了企业 bean 的方法，容器会抛出一个 RemoteException。如果客户端未与一个事务关联，容器在运行方法之前不会开始一个新事务。

22.3.1.7 事务属性小结

表 22-1 对事务属性的影响做了一个小结。T1 和 T2 事务都由容器控制。T1 事务与客户端关联，这个客户端调用了一个企业 bean 的方法。大多数情况下，客户端是另一个企业 bean。方法执行前由容器开始 T2 事务。

表 22-1 事务属性和作用域

事务属性	客户端的事务	业务方法的事务
Required	无	T2
Required	T1	T1
RequiresNew	无	T2
RequiresNew	T1	T2
Mandatory	无	错误
Mandatory	T1	T1
NotSupported	无	无
NotSupported	T1	无
Supports	无	无
Supports	T1	T1
Never	无	无
Never	T1	错误

在表 22-1 的最后一列中，“无”表示业务方法不在由容器控制的事务中执行。不过，这个业务方法中的数据库调用可能会受数据库管理系统的事务管理器控制。

22.3.1.8 设置事务属性

要指定事务属性，需要用 javax.ejb.TransactionAttribute 注解装饰企业 bean 类或方法，并将它设置为某个 javax.ejb.TransactionAttributeType 常量。

如果用 @TransactionAttribute 装饰企业 bean 类，指定的 TransactionAttributeType 会应用到该类中的所有业务方法。如果用 @TransactionAttribute 装饰一个业务方法，则只对这个方法应用这个 TransactionAttributeType。如果一个 @TransactionAttribute 注解同时装饰了类和方法，那么方法的 TransactionAttributeType 会覆盖类的 TransactionAttributeType。

表 22-2 中所示的 TransactionAttributeType 常量封装了这一节前面介绍的事务属性。

表 22-2 TransactionAttributeType 常量

事务属性	TransactionAttributeType 常量
Required	TransactionAttributeType.REQUIRED
RequiresNew	TransactionAttributeType.REQUIRES_NEW
Mandatory	TransactionAttributeType.MANDATORY
NotSupported	TransactionAttributeType.NOT_SUPPORTED
Supports	TransactionAttributeType.SUPPORTS
Never	TransactionAttributeType.NEVER

下面的代码段展示了如何使用 @TransactionAttribute 注解：

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() { ... }

    @TransactionAttribute(REQUIRED)
    public void secondMethod() { ... }

    public void thirdMethod() { ... }

    public void fourthMethod() { ... }
}
```

在这个例子中，TransactionBean 类的事务属性设置为 NotSupported，firstMethod 的事务属性设置为 RequiresNew，secondMethod 的事务属性设置为 Required。由于方法上设置的 @TransactionAttribute 会覆盖类的 @TransactionAttribute，所以调用 firstMethod 时会创建一个新的事务，而 secondMethod 调用可能在当前事务中运行，也可能开始一个新事务。thirdMethod 或 fourthMethod 调用不会在事务中发生。

22.3.2 回滚容器托管事务

回滚一个容器托管事务有两种方法。首先，如果抛出一个系统异常，容器会自动回滚事务。其次，通过调用 EJBContext 接口的 setRollbackOnly 方法，bean 方法会指示容器回滚这个事务。如果 bean 抛出一个应用异常，则不会自动回滚，不过可以通过一个 setRollbackOnly 调用启动回滚。

22.3.3 同步会话 bean 的实例变量

SessionSynchronization 接口（可选）允许有状态会话 bean 实例接收事务同步通知。例如，可以将一个企业 bean 的实例变量与数据库中的相应值同步。容器会在事务的各个主阶段调用 SessionSynchronization 方法（afterBegin、beforeCompletion 和 afterCompletion）。

afterBegin 方法通知实例开始了一个新事务。容器在调用业务方法之前会立即调用 afterBegin。

在业务方法完成但事务提交之前，容器会调用 beforeCompletion 方法。beforeCompletion 方法是会话 bean 回滚事务的最后机会（通过调用 setRollbackOnly）。

afterCompletion 方法指示事务已经完成。这个方法有一个 boolean 参数，如果事务提交，这个参数值为 true，如果事务回滚，这个参数值为 false。

22.3.4 容器托管事务中不允许的方法

有些方法可能影响容器设置的事务边界，所以不允许调用这些方法，这包括：

- `java.sql.Connection` 的 `commit`、`setAutoCommit` 和 `rollback` 方法
- `javax.ejb.EJBContext` 的 `getUserTransaction` 方法
- `javax.transaction.UserTransaction` 的所有方法

不过，可以使用这些方法来设置应用托管事务的边界。

22.4 bean 托管事务

bean 托管事务界定（bean-managed transaction demarcation）中，由会话 bean 或消息驱动 bean 中的代码显式地标志事务的边界。尽管使用容器托管的 bean 需要的代码比较少，但存在一个限制：执行一个方法时，它只能与一个事务关联，或者根本不与事务关联。如果这个限制会增加编写 bean 的难度，就要考虑使用 bean 托管事务。

下面的伪代码展示了利用应用托管事务可以实现细粒度的控制。通过检查不同的条件，伪代码可以确定是否在业务方法中开始或结束事务：

```
begin transaction
...
    update table-a
...
    if (condition-x)
   commit transaction
    else if (condition-y)
   update table-b
   commit transaction
    else
   rollback transaction
   begin transaction
   update table-c
   commit transaction
```

为会话 bean 或消息驱动 bean 编写应用托管事务时，必须确定使用 Java 数据库互连（Java Database Connectivity）还是 JTA 事务。下面的小节将讨论这两类事务。

22.4.1 JTA 事务

利用 JTA 或 Java 事务 API（Java Transaction API），可以采用一种独立于事务管理器实现的方式界定事务。GlassFish 服务器为 Java 事务服务（Java Transaction Service，JTS）实现了事务管理器。不过，你的代码并不直接调用 JTS 方法，而是要调用 JTA 方法，这些方法再调用更底层的 JTS 例程。

JTA 事务由 Java EE 事务管理器控制。你可能希望使用 JTA 事务，因为它可以对不同开

发商的多个不同数据库完成更新，而特定 DBMS 的事务管理器可能无法处理异构数据库。不过，Java EE 事务管理器也有一个限制：它不支持嵌套事务。换句话说，在前一个事务结束之前，不能为一个实例开始一个新事务。

要界定一个 JTA 事务，可以调用 `javax.transaction.UserTransaction` 接口的 `begin`、`commit` 和 `rollback` 方法。

22.4.2 不提交返回

在一个采用 bean 托管事务的无状态会话 bean 中，业务方法必须在返回之前提交或回滚事务。不过，有状态会话 bean 没有这个限制。

在一个采用 JTA 事务的有状态会话 bean 中，bean 实例与事务之间的关联可以跨多个客户端调用。即使客户端调用的各个业务方法会打开和关闭数据库连接，这个关联仍然保持，直到 bean 实例完成事务。

在一个采用 JDBC 事务的有状态会话 bean 中，JDBC 连接会跨多个调用保持 bean 实例与事务之间的关联。如果这个连接关闭，关联就不再保持。

22.4.3 bean 托管事务中不允许的方法

bean 托管事务中不能调用 `EJBContext` 接口的 `getRollbackOnly` 和 `setRollbackOnly` 方法。这些方法只能在容器托管事务中使用。对于 bean 托管事务，应当调用 `UserTransaction` 接口的 `getStatus` 和 `rollback` 方法。

22.5 事务超时

对于容器托管事务，可以使用管理控制台来配置事务超时时间。更多信息参见 2.3 节。

对于采用 bean 托管 JTA 事务的企业 bean，要调用 `UserTransaction` 接口的 `setTransactionTimeout` 方法。

22.5.1 设置事务超时

1. 在管理控制台中，展开 Configurations（配置）节点，然后展开 server-config（服务器配置）节点并选择 Transaction Service（事务服务）。

2. 在事务服务页面上，为 Transaction Timeout 域设置你选择的值（例如，5）。

对于这个设置，如果事务在 5 秒内没有完成，EJB 容器就会将它回滚。

默认值为 0，这表示事务不会超时。

3. 单击 Save（保存）。

22.6 更新多个数据库

Java EE 事务管理器可以控制除了 bean 托管 JDBC 事务以外的所有其他企业 bean 事务。Java EE 事务管理器允许企业 bean 在一个事务中更新多个数据库。图 22-2 和图 22-3 显示了一个事务中更新多个数据库的两种情况。

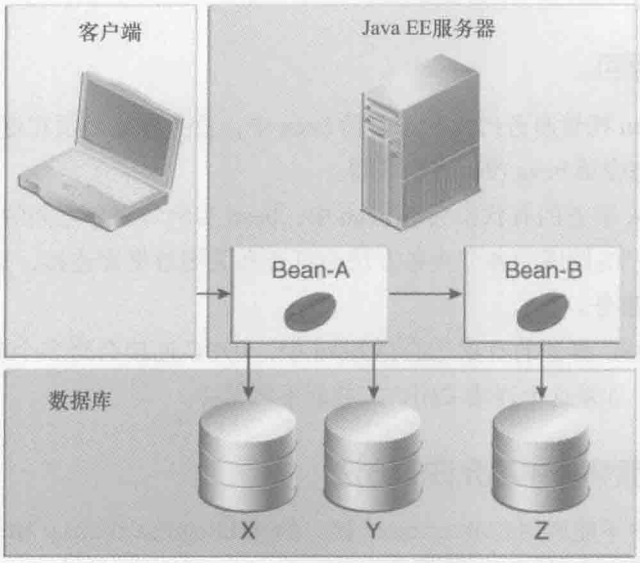


图 22-2 更新多个数据库

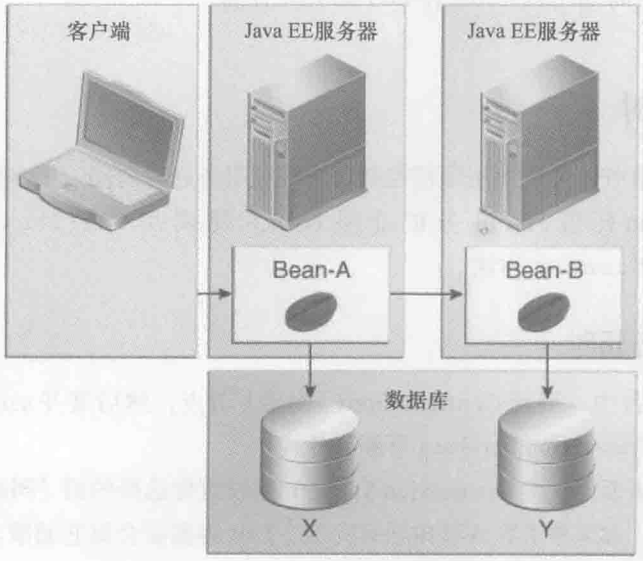


图 22-3 跨 Java EE 服务器更新多个数据库

图 22-2 中，客户端调用 Bean-A 的一个业务方法。这个业务方法开始一个事务，更新

数据库 X，更新数据库 Y，然后调用 Bean-B 的一个业务方法。第二个业务方法更新数据库 Z，再将控制返回 Bean-A 的业务方法，这会提交事务。所有这 3 个数据库更新都发生在同一个事务中。

图 22-3 中，客户端调用 Bean-A 的一个业务方法，这会开始一个事务，并更新数据库 X。然后 Bean-A 调用 Bean-B 的一个方法，这个 bean 位于一个远程 Java EE 服务器中。Bean-B 的方法更新数据库 Y。Java EE 服务器的事务管理器确保两个数据库都在同一个事务中更新。

22.7 Web 组件中的事务

可以使用 java.sql.Connection 或 javax.transaction.UserTransaction 接口界定 Web 组件中的事务。采用 bean 托管事务的会话 bean 也使用同样的接口。用 UserTransaction 接口界定的事务在 22.4.1 节中讨论。

22.8 关于事务的更多信息

要了解关于事务的更多信息，参见 Java 事务 API 1.2 规范：https://www.jcp.org/en/jsr/detail?id=907。

Chapter 23

第 23 章

资源适配器和契约

本章研究资源适配器，并解释资源适配器如何调解 Java EE 服务器和 EIS 系统之间的通信，包括以下内容：

- 什么是资源适配器
- 元数据注解
- 公共客户端接口
- 对 Java EE 上下文和依赖注入（CDI）使用资源适配器
- 关于资源适配器的更多信息

23.1 什么是资源适配器

资源适配器是为一个特定 EIS 实现 Java EE 连接器架构的 Java EE 组件。EIS 包括企业资源计划（ERP）、大型机事务处理（MTP）和数据库系统等。在一个 Java EE 服务器中，Java 消息服务器和 JavaMail 也相当于 EIS，可以使用资源适配器来访问。如图 23-1 所示，资源适配器可以调解 Java EE 应用和 EIS 之间的通信。

由于资源适配器存储在一个资源适配器归档（Resource Adapter Archive，RAR）文件中，所以它可以部署在任意的 Java EE 服务器上，这与 Java EE 应用很类似。RAR 文件可以包含在一个企业归档（Enterprise Archive，EAR）文件中，也可以作为一个单独的文件。

资源适配器类似于一个 JDBC 驱动程序。二者都提供了一个标准 API，通过这个 API，应用可以访问 Java EE 服务器之外的资源。对于资源适配器来说，目标系统是一个 EIS；对于 JDBC 驱动程序，目标系统则是一个 DBMS。资源适配器和 JDBC 驱动程序很少由应用开发人

员创建。大多数情况下，这两种软件都由出售工具、服务器或集成软件的开发商来构建。

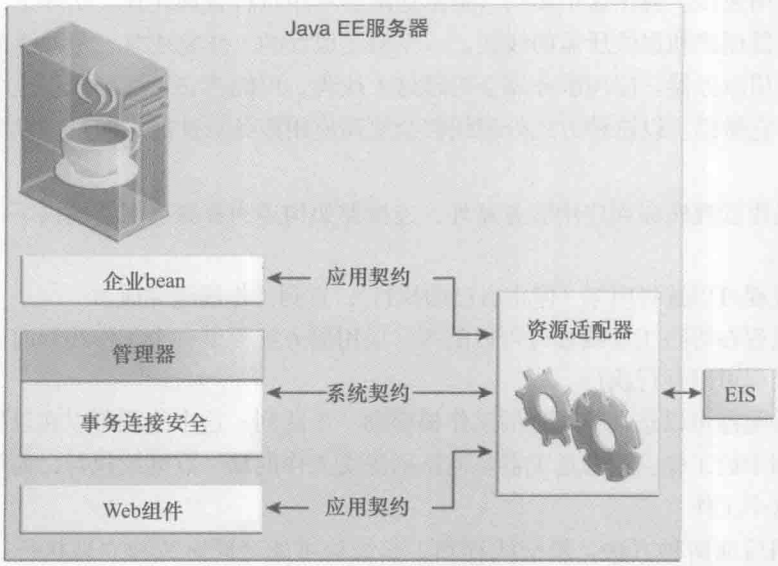


图 23-1 资源适配器

资源适配器利用契约来调解 Java EE 服务器和 EIS 之间的通信。应用契约定义了 Java EE 组件（如一个企业 bean）访问 EIS 时使用的 API。这个 API 是组件对 EIS 的唯一视图。系统契约将资源适配器链接到由 Java EE 服务器管理的重要服务。资源适配器本身及其系统契约对 Java EE 组件是透明的。

23.1.1 管理契约

Java EE 连接器架构定义了系统契约来支持资源适配器生命周期和线程管理。

23.1.1.1 生命周期管理

连接器架构（Connector Architecture）指定了一个生命周期管理契约，允许应用服务器管理一个资源适配器的生命周期。这个契约为应用服务器提供了一个机制，可以在部署或应用服务器启动时建立一个资源适配器实例。这个契约还提供了一种方法，使得应用服务器在取消部署或应用服务器依次关闭时会通知资源适配器。

23.1.1.2 工作管理契约

连接器架构工作管理契约确保资源适配器以一种正确的推荐方式使用线程。这个契约还支持应用服务器为资源适配器管理线程。

如果资源适配器不能正确地使用线程，这可能会危害整个应用服务器环境。例如，一个资源适配器可能创建太多的线程，或者可能没有适当地释放所创建的线程。如果线程处理不当，会阻止应用服务器关闭，并影响应用服务器的性能，因为创建和撤销线程是开销很大的操作。

工作管理契约为应用服务器建立一种方法，允许建立线程池并重用线程，这类似于建立连接池和重用连接。遵循这个契约，资源适配器不用自行管理线程。实际上，资源适配器会让应用服务器创建和提供所需的线程。一个给定线程的工作完成时，资源适配器会把这个线程返回给应用服务器。应用服务器会管理这个线程，可能将它返回到线程池中，以备以后重用，或者将它撤销。以这种方式处理线程会提高应用服务器性能，而且可以更高效地使用资源。

除了将线程管理转移到应用服务器外，连接器架构还为资源适配器提供了一个灵活的模型来使用线程。

- 请求线程可以选择阻塞（停止自己的执行），直到工作线程完成。
- 请求线程在等待工作线程时可以阻塞。应用服务器提供一个工作线程时，请求线程和工作线程可以并行执行。
- 资源适配器可以选择把线程的工作提交到一个队列。这个线程可以在以后某个时间执行队列中的工作。资源适配器从向队列提交工作的那一点继续执行，而不论线程何时执行这个工作。

如果采用后面两种方法，提交线程和工作线程可能会同时或独立地执行。对于这些方法，契约指定了一个监听器机制，来通知资源适配器线程已经完成其操作。资源适配器还可以为线程指定执行上下文，另外工作管理契约会控制执行线程的上下文。

23.1.2 通用工作上下文契约

基于应用服务器和资源适配器之间的工作管理契约，资源适配器可以通过传送要执行的 `Work` 实例来完成一个任务，如与 EIS 通信或传送消息。

通用工作上下文契约允许资源适配器控制上下文，所提交的 `Work` 实例将在这个上下文中由应用服务器的 `WorkManager` 执行。通用工作上下文机制还允许应用服务器支持消息流入和传送机制。它还为资源适配器提供一个更丰富的上下文 `Work` 执行环境，同时仍维持对托管环境中并发行为的控制。

通用工作上下文契约实现了事务上下文和安全上下文的标准化。

23.1.3 出站和入站契约

连接器架构定义以下出站契约，这是应用服务器与 EIS 之间的系统级契约，这些契约支持与 EIS 的出站连接。

- 连接管理契约支持连接池，这种技术可以提升应用性能和可扩展性。连接池对应用是透明的，只是得到与 EIS 的一个连接。
- 事务管理契约扩展了连接管理契约，可以为本地和 XA 事务的管理提供支持。

本地事务仅限于一个 EIS 系统范围内，这样一个事务由 EIS 资源管理器本身来管理。XA 事务或全局事务可以跨多个资源管理器。这种形式的事务需要由一个外部事务管理器完

成事务协调（通常与一个应用服务器一同打包）。事务管理器使用一个两阶段提交协议来管理跨多个资源适配器或 EIS 的事务，如果只有一个资源适配器参与 XA 事务，则使用一阶段提交优化。

- 安全管理契约提供了 Java EE 服务器和 EIS 之间的认证、授权和安全通信机制，以保护 EIS 中的信息。

工作安全映射可以完成 EIS 身份与应用服务器域身份的匹配。

入站契约是 Java EE 服务器与 EIS 之间的系统契约，支持 EIS 的入站连接：包括消息提供者的可插拔性契约和导入事务的契约。

23.2 元数据注解

Java EE 连接器架构提供了一组注解，从而尽可能不使用部署描述文件。

- 资源适配器开发人员可以使用 `@Connector` 注解来指定 JavaBeans 组件是一个资源适配器 JavaBeans 组件。这个注解用于提供资源适配器功能的有关元数据。可以提供一个实现 `ResourceAdapter` 接口的 JavaBeans 组件（可选），如下例所示：

```
@Connector(
    displayName = "TrafficResourceAdapter",
    vendorName = "Java EE Tutorial",
    version = "7.0"
)
public class TrafficResourceAdapter implements ResourceAdapter,
                                               Serializable {
    ...
}
```

- `@ConnectionDefinition` 注解定义了与一个特定连接类型相关的一组连接接口和类，如下例所示：

```
@ConnectionDefinition(
    connectionFactory = ConnectionFactory.class,
    connectionFactoryImpl = TradeConnectionFactory.class,
    connection = Connection.class,
    connectionImpl = TradeConnection.class
)
public class TradeManagedConnectionFactory ... {
    ...
}
```

- `@AdministeredObject` 注解指定一个 JavaBeans 组件作为受管理对象。
- `@Activation` 注解包含与一个 EIS 实例入站连接相关的配置信息，如下例所示：

```
@Activation(
        messageListeners = { TrafficListener.class }
)
```

```
public class TrafficActivationSpec implements ActivationSpec,
                                              Serializable {
    ...
    @ConfigProperty()
    /* port to listen to requests from the EIS */
    private String port;
    ...
}
```

- @ConfigProperty 注解用于 JavaBeans 组件，用来提供可由部署人员和资源适配器提供者使用的额外的配置信息。前面的示例代码中显示了多个 @ConfigProperty 注解。
- @ConnectionFactoryDefinition 注解是一个资源定义注解，用来定义一个连接器连接工厂，并以指定的名在 JNDI 中注册（这个名由必要的 name 注解元素指定）。必要的 interfaceName 注解元素指定连接工厂接口类的完全限定名。transactionSupport 注解元素指定连接工厂需要支持的事务支持级别。对于作为连接工厂资源的后备连接池，minPoolSize 和 maxPoolSize 注解元素指定了为这个连接池分配的最小和最大连接数。对于所定义的连接工厂，有关的其他属性可以通过 properties 元素指定。

  由于不允许有重复的注解，所以 @ConnectionFactoryDefinitions 注解相当于一个容器，用于包含多个连接器连接工厂定义。value 注解元素包含多个连接器连接工厂定义。
- @AdministeredObjectDefinition 注解是一个资源定义注解，用来定义一个受管理对象，并以指定的名在 JNDI 中注册（这个名由必要的 name 注解元素指定）。受管理对象类的必要的完全限定名必须由 className 元素指示。受管理对象中配置的其他属性可以通过 properties 元素指定。

由于不允许有重复的注解，所以 @AdministeredObjectDefinitions 注解相当于多个受管理对象定义的容器。value 注解元素包含多个受管理对象定义。

这个规范允许采用一种混合模式来开发资源适配器，也就是说，资源适配器开发人员可以在应用中同时使用元数据注解和部署描述文件。应用装配人员或部署人员可以使用部署描述文件来覆盖资源适配器开发人员指定的元数据注解。

资源适配器的部署描述文件（如果有）名为 ra.xml。metadata-complete 属性定义了资源适配器模块的部署描述文件是否完整，或者对于指定部署信息的注解，是否需要检查模块可用并与资源适配器一同打包的类文件。

要全面了解 Java EE 7 平台中提供的注解和 JavaBeans 组件，可参见 Java EE 连接器架构 1.7 规范。

23.3 公共客户端接口

这一节解释组件如何使用连接器架构公共客户端接口（Common Client Interface，CCI）

API 和一个资源适配器来访问 EIS 的数据。CCI API 定义了一组接口和类，利用这些 API 方法，客户端可以完成典型的数据访问操作。CCI 接口和类如下：

- ConnectionFactory: 为应用组件提供一个 EIS 的一个 Connection 实例。
- Connection：表示与底层 EIS 的连接。
- ConnectionSpec: 为应用组件提供了一种方法，可以在建立一个连接请求时向 ConnectionFactory 传递连接请求特定的属性。
- Interaction：为应用组件提供一种执行 EIS 函数的方法，如数据库存储过程。
- InteractionSpec:：应用组件与一个 EIS 交互的有关属性。
- Record：不同类型记录实例的超接口。记录实例可以是 MappedRecord、IndexedRecord 或 ResultSet 实例，所有这些实例都继承自 Record 接口。
- RecordFactory：为应用组件提供一个 Record 实例。
- IndexedRecord：表示基于 java.util.List 接口的 Record 实例的一个有序集合。

客户端或应用组件使用 CCI 与一个底层 EIS 交互时，会以一种预定的方式完成工作。这个组件必须建立与 EIS 资源适配器的一个连接，这可以使用 ConnectionFactory 来做到。Connection 对象表示与 EIS 的连接，用于与这个 EIS 的后续交互。

组件使用一个 Interaction 对象完成与 EIS 的交互，如从一个特定的表访问数据。应用组件使用一个 InteractionSpec 对象来定义这个交互对象。从 EIS 读取数据时（如从数据库表读取数据），或者写入数据库表时，应用组件可以使用一个特定类型的 Record 实例：MappedRecord、IndexedRecord 或 ResultSet 实例。

还需要说明一点，依赖于 CCI 资源适配器的客户端应用与使用企业 bean 方法的其他 Java EE 客户端很相似。

23.4 对 Java EE 上下文和依赖注入（CDI）使用资源适配器

关于 CDI 的详细信息，参见本书第 1 卷中的第 23 章和第 25 章。

不要在资源适配器中将以下类指定为 CDI 托管 bean（也就是说，不要注入这些类），因为这些类作为 CDI 托管 bean 时，其行为的定义是不可移植的。

- **资源适配器 bean**: 这些 bean 是增加了 javax.resource.spi.Connector 注解的类，或者在资源适配器部署描述文件 ra.xml 中声明为相应元素。
- **托管连接工厂 bean**: 这些 bean 是增加了 javax.resource.spi.ConnectorDefinition 注解或 javax.resource.spi.ConnectorDefinitions 注解的类，或者在 ra.xml 中声明为相应元素。
- **激活规范 bean**: 这些 bean 是增加了 javax.resource.spi.Activation 注解的类，或者在 ra.xml 中声明为相应元素。
- **受管理对象 bean**: 这些 bean 是增加了 javax.resource.spi.AdministeredObject

注解的类，或者在 ra.xml 中声明为相应元素。

资源适配器中的其他类型的类可以是 CDI 托管 bean，其行为可移植。

23.5 关于资源适配器的更多信息

关于资源适配器和注解的更多信息，参见：

- Java EE 7 平台规范（JSR 342）：

  http://jcp.org/en/jsr/detail?id=342

- Java EE 连接器架构 1.7 规范：

  http://jcp.org/en/jsr/detail?id=322

- EJB 3.2 规范：

  http://jcp.org/en/jsr/detail?id=345

- Java 平台的公共注解：

  http://www.jcp.org/en/jsr/detail?id=250

第 24 章 Chapter 20

资源适配器示例

本章介绍两个例子，展示如何在 Java EE 应用中使用资源适配器，以及如何实现简单的资源适配器。trading 示例展示如何在一个 Web 应用中使用简单的定制客户端接口连接一个 EIS。这个例子中的资源适配器实现了出站契约和定制客户端接口。traffic 示例展示了如何使用消息驱动 bean（MDB）处理一个 EIS 的交通信息更新。这个例子中的资源适配器实现了入站契约和工作管理契约。

本章将介绍以下内容：

- trading 示例
- traffic 示例

24.1 trading 示例

trading 示例展示了如何实现和使用一个简单的出站资源适配器，它使用 TCP 套接字向一个遗留 EIS 提交请求。这个例子展示了图 24-1 所示的场景，包括以下模块：

- trading-eis：这是一个 Java SE 程序，模拟一个遗留 EIS。
- trading-rar：这是一个出站资源适配器实现。

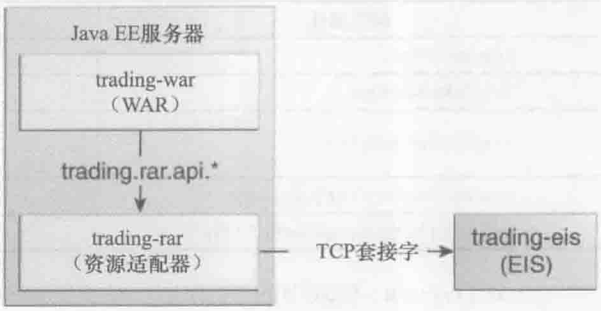


图 24-1 trading 示例

- trading-war：这是一个使用资源适配器的 Web 应用。
- trading-ear：这是一个包含资源适配器和 Web 应用的企业归档。

trading-eis 模块是一个辅助项目，用来模拟一个遗留的股票交易执行平台。它包含一个 Java SE 程序，监听一个 TCP 套接字上的明文交易请求。作为对交易请求的响应，程序会给出一个状态值、一个确认号、所请求份额的金额以及手续费。例如，下面是一个请求－响应对：

```
>> BUY 1000 ZZZZ MARKET
<< EXECUTED #1234567 TOTAL 50400.00 FEE 252.00
```

trading-rar 模块实现了 Java EE 连接器架构的出站契约，以此来提交请求，并从遗留股票交易执行平台得到响应。trading-rar 模块提供并实现了一个定制客户端接口，该接口将由 Java EE 应用使用。这个接口比公共客户端接口（CCI）简单。

trading-war 模块是一个 Web 应用，包含一个 JSF 接口和一个托管 bean。这个应用允许客户端使用 trading-rar 模块提供的资源适配器向 EIS 提交交易。trading-war 模块使用资源适配器提供的定制客户端接口来得到与 EIS 的连接。

24.1.1 使用出站资源适配器

大多数情况下，Java EE 应用开发人员会使用第三方开发的出站资源适配器。出站资源适配器可实现公共客户端接口（Common Client Interface，CCI），或者为应用提供一个定制接口与 EIS 交互。出站资源适配器为 Java EE 应用提供以下元素：

- 连接工厂
- 连接句柄
- 特定于 EIS 域的其他接口和对象

Java EE 应用通过资源注入得到连接工厂的一个实例，然后使用这个工厂对象得到 EIS 的连接句柄。利用这个连接句柄，应用可以做出请求并从 EIS 得到信息。

trading-rar 模块提供了一个定制客户端接口，包括表 24-1 所列的类。

表 24-1 javaeetutorial.trading.rar.api 包中的类和接口

API 组件	描 述
TradeOrder	表示 EIS 的一个交易订单
TradeResponse	表示 EIS 对一个交易请求的响应
TradeConnection	表示 EIS 的一个连接句柄 为应用提供一个方法向 EIS 提交交易
TradeConnectionFactory	使应用得到 EIS 的连接句柄
TradeProcessingException	指示处理一个交易请求时出现一个问题

trading-war 模块中的 ResourceAccessBean 托管 bean 为 trading-rar 资源适配器配置了一个连接工厂，这里使用了 @ConnectionFactoryDefinition 注解，如下所示：

```
@Named
@SessionScoped
@ConnectionFactoryDefinition(
    name = "java:comp/env/eis/TradeConnectionFactory",
    interfaceName = "javaeetutorial.trading.rar.api.TradeConnectionFactory",
    resourceAdapter = "#trading-rar",
    minPoolSize = 5,
    transactionSupport =
            TransactionSupport.TransactionSupportLevel.NoTransaction
)
public class ResourceAccessBean implements Serializable { ... }
```

name 参数指定了连接工厂的 JNDI 名。这个例子在 java:comp 作用域中注册这个连接工厂。也可以使用 ConnectionFactoryDefinition 注解指定一个不同的作用域，如 java:global、java:app 或 java:module。AdministeredObjectDefinition 注解还允许在 JNDI 命名空间注册受管理连接器对象。

interfaceName 参数指定由资源适配器中包括的连接工厂所实现的接口。在这个例子中，这是一个定制接口。

resourceAdapter 参数指定包含连接工厂实现的资源适配器名。#trading-rar 中的 # 前缀指示 trading-rar 是一个内嵌资源适配器，与这个 Web 应用打包在同一个 EAR 中。

说明 还可以从应用服务器使用管理命令为之前部署的出站资源适配器配置一个连接工厂。不过，这取决于具体的供应商。

托管 bean 使用资源注入包含一个连接工厂对象，如下所示：

```
...
public class ResourceAccessBean implements Serializable {
    @Resource(lookup = "java:comp/env/eis/TradeConnectionFactory")
    private TradeConnectionFactory connectionFactory;
    ...
}
```

这个托管 bean 使用连接工厂得到连接句柄，如下所示：

```
TradeConnection connection = connectionFactory.getConnection();
```

资源适配器返回一个与 EIS 物理连接关联的连接句柄。一旦得到连接句柄，这个托管 bean 将提交一个交易，并得到响应，如下所示：

```
TradeOrder order = new TradeOrder();
order.setNShares(1000);
order.setTicker(TradeOrder.Ticker.YYYY);
order.setOrderType(TradeOrder.OrderType.BUY);
order.setOrderClass(TradeOrder.OrderClass.MARKET);
...
try {
```

```
    TradeResponse response = connection.submitOrder(order);
    ...
} catch (TradeProcessingException ex) { ... }
```

24.1.2 实现出站资源适配器

trading-rar 模块为这个例子中使用的简单遗留股票交易平台 EIS 实现了出站契约和一个定制客户端接口。出站资源适配器的架构如图 24-2 所示。

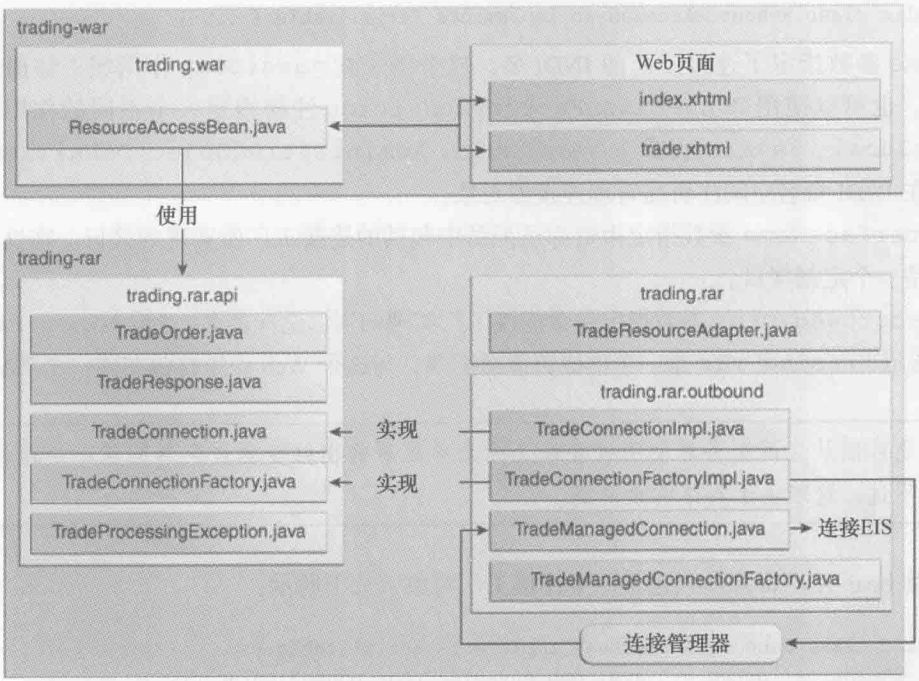


图 24-2 trading 示例的架构

trading-rar 模块实现了表 24-2 所列的接口。

表 24-2 trading-rar 模块中实现的接口

包	接 口	描 述
javax.resource.spi	ResourceAdapter	定义资源适配器的生命周期方法
javax.resource.spi	ManagedConnectionFactory	定义一个连接工厂，应用服务器的连接管理器使用这个连接工厂来得到 EIS 的物理连接
javax.resource.spi	ManagedConnection	定义 EIS 的一个物理连接，由连接管理器管理
trading.rar.api	TradeConnectionFactory	定义一个连接工厂，应用使用连接工厂得到连接句柄
trading.rar.api	TradeConnection	定义一个连接句柄，应用使用连接句柄与 EIS 交互

部署 trading-ear 归档并配置一个连接池资源（如 24.1 节所述）时，应用服务器会创建 TradeConnectionFactory 对象，应用可以使用资源注入得到这个连接工厂对象。TradeConnectionFactory 实现将委托应用服务器提供的连接管理器来创建连接。

连接管理器使用 ManagedConnectionFactory 实现得到与 EIS 的物理连接，并维护一个活动的物理连接池。应用请求一个连接句柄时，连接管理器将池中的一个连接与应用可用的一个新连接句柄关联。连接池可以提高应用性能，并简化资源适配器开发。

有关的更多详细信息，参见 trading-rar 模块中的代码和注释。

24.1.3 运行 trading 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 trading 示例。

24.1.3.1 使用 NetBeans IDE 运行 trading 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/connectors
```

4. 选择 trading 文件夹。
5. 单击 Open Project。
6. 在 Projects（工程）标签页中，展开 trading 节点。
7. 右键单击 trading-eis 模块并选择 Open Project。
8. 右键单击 trading-eis 工程并选择 Run（运行）。

EIS 的消息出现在 Output（输出）标签页中：

```
Trade execution server listening on port 4004.
```

9. 右键单击 trading-ear 工程并选择 Build（构建）。

这个命令将资源适配器和 Web 应用打包在一个 EAR 文件中，并将其部署到 GlassFish 服务器。

10. 在 Web 浏览器中打开以下 URL：

```
http://localhost:8080/trading/
```

这个 Web 界面允许你连接到 EIS，并提交交易。服务器日志显示了 Web 应用的请求和从资源适配器提供连接句柄的调用序列。

11. 取消部署 trading-ear 应用之前，从状态条关闭 trading-eis 应用。

24.1.3.2 使用 Maven 运行 trading 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 在一个终端窗口中导航至：

tut-install/examples/connectors/trading/

3. 输入以下命令：

```
mvn install
```

这个命令构建并将资源适配器和 Web 应用打包到一个 EAR 归档，并将它部署到 GlassFish 服务器。

4. 在同一个终端窗口中，进入 trading-eis 目录：

```
cd trading-eis
```

5. 输入以下命令运行交易执行平台：

```
mvn exec:java
```

EIS 的消息会出现在这个终端窗口中：

```
Trade execution server listening on port 4004.
```

6. 在 Web 浏览器中打开以下 URL：

```
http://localhost:8080/trading/
```

这个 Web 界面允许你连接到 EIS，并提交交易。服务器日志显示了 Web 应用的请求和从资源适配器提供连接句柄的调用序列。

7. 取消部署 trading-ear 应用之前，在终端窗口按下 Ctrl+C 关闭 trading-eis 应用。

24.2 traffic 示例

traffic 示例展示了如何实现和使用一个简单的入站资源适配器，它使用 TCP 套接字从一个遗留 EIS 接收数据。

这个例子存放在 *tut-install*/examples/connectors/traffic 目录。参见第 2 章来了解构建和运行示例应用的基本信息。

这个例子展示了图 24-3 中所示的场景，包括以下模块：

- traffic-eis：这是一个 Java SE 程序，模拟一个 EIS。
- traffic-rar：这是一个入站资源适配器实现。
- traffic-ejb：这是一个消息驱动 bean，该端点接收到来的消息。

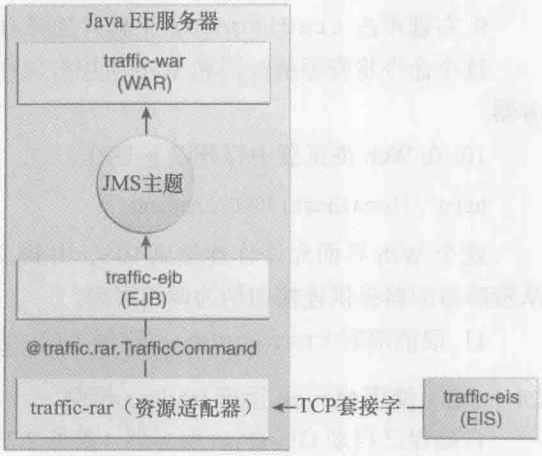


图 24-3 traffic 示例

- traffic-war：这是一个 Web 应用，显示来自消息驱动 bean 的信息。
- traffic-ear：这个企业归档包含资源适配器、消息驱动 bean 和 Web 应用。

traffic-eis 模块是一个辅助项目，模拟一个遗留的交通信息系统。它包含一个 Java SE 程序，向所有订阅客户端发送多个城市的交通状态更新。这个程序在一个 TCP 套接字上采用 JSON 格式发送更新。例如，下面给出一个交通更新示例：

```
{"report":[
    {"city":"City1", "access":"AccessA", "status":"GOOD"},
    {"city":"City1", "access":"AccessB", "status":"CONGESTED"},
    ...
    {"city":"City5", "access":"AccessE", "status":"SLOW"}
 ]}
```

traffic-rar 模块实现 Java EE 连接器架构的入站契约。这个模块使用 TCP 端口订阅交通信息系统（这个端口由 MDB 提供的配置来指示），并调用 MDB 的方法处理交通信息更新。

traffic-ejb 模块包含一个消息驱动 bean，它会启动资源适配器并指定一个配置参数（订阅交通信息系统的 TCP 端口）。MDB 包含一个处理交通信息更新的方法。这个方法会过滤特定城市的更新信息，并把结果发布到一个 Java 消息服务（JMS）主题。

traffic-war 模块包含一个消息驱动 bean，它会异步地从 JMS 主题接收过滤的交通信息更新，并使用一个 WebSocket 端点把它们发送到客户端。

24.2.1　使用入站资源适配器

大多数情况下，Java EE 应用开发人员会使用第三方开发的入站资源适配器。要使用一个入站资源适配器，Java EE 应用应包括有以下特点的消息驱动 bean：

- MDB 要实现资源适配器定义的业务接口。
- MDB 指定了配置参数来激活资源适配器。

资源适配器定义的业务接口并非在 Java EE 连接器架构中指定，而是特定于 EIS。

这个例子中的 MDB 定义如下：

```
@MessageDriven(
    activationConfig = {
      @ActivationConfigProperty(propertyName = "port",
                                propertyValue = "4008")
    }
)
public class TrafficMdb implements TrafficListener { ... }
```

TrafficListener 接口定义位于资源适配器的 API 包中。资源适配器要求这个 MDB 提供 port 属性。

部署 MDB 时，它会激活 traffic-rar 资源适配器，资源适配器调用 MDB 的方法来处理交通信息更新。然后 MDB 为一个特定城市过滤这些更新，并把结果发布到一个 JMS 主题。

在这个例子中，TrafficListener 接口为空。除了这个接口，资源适配器还提供了

TrafficCommand 注解，并使用反射来发现 MDB 中的哪些方法装饰了这个注解：

```
@MessageDriven(...)
public class TrafficMdb implements TrafficListener {
    @TrafficCommand(name="report", info="Process report")
    public void processReport(String jsonReport) { ... }
    ...
}
```

利用这个方法，可以调整 MDB 来支持 EIS 中的新特性，而不必修改 TrafficListener 接口或资源适配器模块。

24.2.2 实现入站资源适配器

对于这个示例中使用的简单交通信息系统，traffic-rar 模块为它实现了 Java EE 连接器架构的入站资源适配器契约。入站资源适配器的架构如图 24-4 所示。

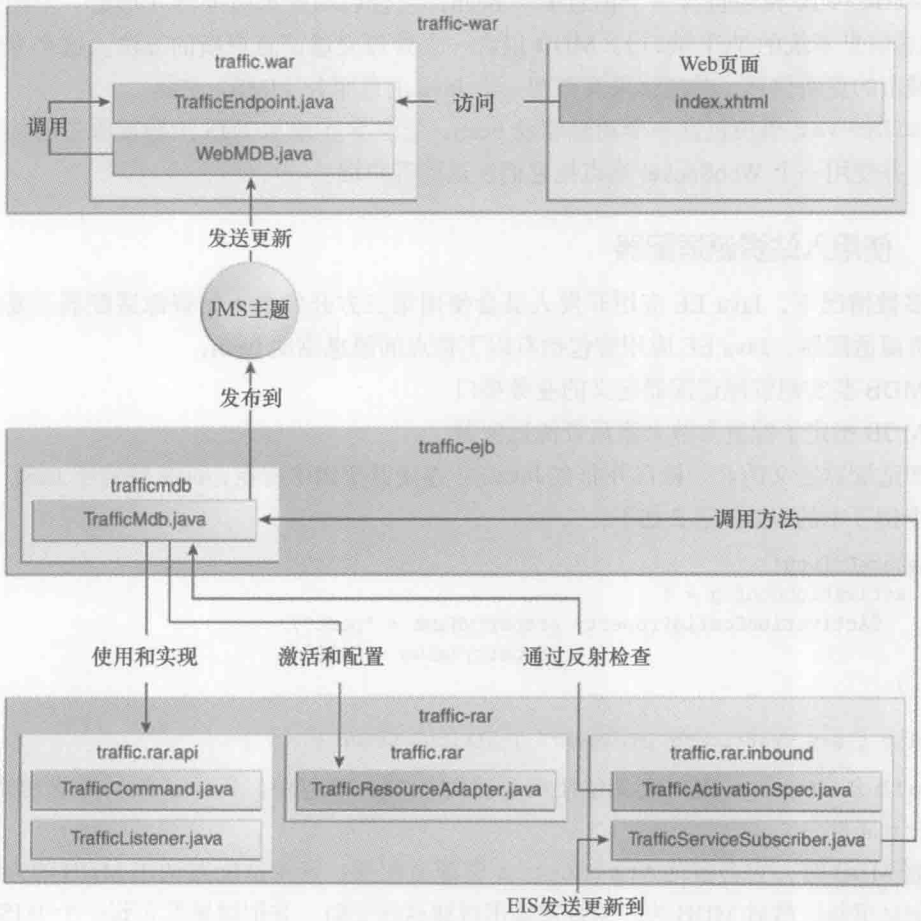


图 24-4 traffic 示例的架构

traffic-rar 模块实现了表 24-3 所列的接口。

表 24-3 traffic-rar 模块中实现的接口

包	接 口	描 述
javax.resource.spi	ResourceAdapter	定义资源适配器的生命周期方法
javax.resource.spi	ActivationSpec	定义配置参数，MDB 提供这些配置参数来激活入站资源适配器
javax.resource.spi	Work	交通服务订阅者要实现工作管理契约的这个接口，在一个单独的线程上等待交通信息更新

MDB 激活入站资源适配器时，容器调用 TrafficResourceAdapter 类中的 endpointActivation 方法：

```
@Connector(...)
public class TrafficResourceAdapter implements ResourceAdapter, Serializable {
    ...
    @Override
    public void endpointActivation(MessageEndpointFactory endpointFactory,
                                   ActivationSpec spec)
                                   throws ResourceException {
        ...
        Class endpointClass = endpointFactory.getEndpointClass();
        /* this method is called from a new thread in the example:
        MessageEndpoint endpoint = endpointFactory.createEndpoint(null); */
    }
}
```

getEndpointClass 方法返回完成激活的 MDB 的 Class 类型，从而允许资源适配器使用反射来查找 MDB 中有 @TrafficCommand 注解的方法。

createEndpoint 方法返回 MDB 的一个实例。资源适配器接收到来自 EIS 的请求时，会使用这个实例调用 MDB 的方法。

得到消息端点实例（MDB）后，资源适配器使用工作管理契约创建交通服务订阅者线程，从 EIS 接收交通更新。资源适配器从 Bootstrap 上下文得到 WorkManager 实例，如下所示：

```
WorkManager workManager;
...
@Override
public void start(BootstrapContext ctx) ... {
    workManager = ctx.getWorkManager();
}
```

资源适配器使用工作管理器来调度交通服务订阅者线程：

```
tSubscriber = new TrafficServiceSubscriber(tSpec, endpoint);
workManager.scheduleWork(tSubscriber);
```

TrafficServiceSubscriber 类实现了工作管理契约的 javax.resource.spi.

Work 接口。

交通服务订阅者线程使用反射来调用 MDB 中的方法：

```
private String callMdb(MessageEndpoint mdb, Method command,
                       String... params) ... {
    String resp;
    /* this code contains proper exception handling in the sources */
    mdb.beforeDelivery(command);
    Object ret = command.invoke(mdb, (Object[]) params);
    resp = (String) ret;
    mdb.afterDelivery();
    return resp;
}
```

有关的更多详细信息，参见 traffic-rar 模块中的代码和注释。

24.2.3 运行 traffic 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 traffic 示例。

24.2.3.1 使用 NetBeans IDE 运行 traffic 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/connectors
```

4. 选择 traffic 文件夹。
5. 单击 Open Project。
6. 在 Projects（工程）标签页中，展开 traffic 节点。
7. 右键单击 traffic-eis 模块并选择 Open Project。
8. 右键单击 traffic-eis 工程并选择 Run（运行）。

EIS 的消息出现在 Output（输出）标签页中：

```
Traffic EIS accepting connections on port 4008
```

9. 在 Projects（工程）标签页中，右键单击 traffic 工程并选择 Clean and Build（清理和构建）。

这个命令会构建资源适配器、MDB 和 Web 应用，将其打包到一个 EAR 归档并完成部署。服务器日志会显示激活资源适配器的调用序列以及为 City1 过滤得到的交通更新信息。

10. 在 Web 浏览器中打开以下 URL：

```
http://localhost:8080/traffic/
```

Web 界面每几秒显示一次为 City1 过滤的交通更新信息。

11. 取消部署 traffic-ear 应用之后，从状态条关闭 traffic-eis 应用。

24.2.3.2　使用 Maven 运行 traffic 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在一个终端窗口中导航至：

```
tut-install/examples/connectors/traffic/traffic-eis
```

3. 在终端窗口中输入以下命令：

```
mvn install
```

这个命令构建并打包交通 EIS。

4. 在终端窗口中输入以下命令：

```
mvn exec:java
```

EIS 的消息出现在终端窗口中：

```
Traffic EIS accepting connections on port 4008
```

保留这个终端窗口打开。

5. 打开一个新的终端窗口，并进入以下目录：

```
tut-install/examples/connectors/traffic/
```

6. 输入以下命令：

```
mvn install
```

这个命令会构建资源适配器、MDB 和 Web 应用，将其打包到一个 EAR 归档并完成部署。服务器日志会显示激活资源适配器的调用序列以及为 City1 过滤得到的交通更新信息。

7. 在一个 Web 浏览器中打开以下 URL：

```
http://localhost:8080/traffic/
```

Web 界面每几秒显示一次为 City1 过滤的交通更新信息。

8. 取消部署 `traffic-ear` 应用之后，在第一个终端窗口按下 Ctrl+C 关闭 `traffic-eis` 应用。

Chapter 25

第 25 章

使用 Java EE 拦截器

本章讨论如何创建拦截器类和方法，这些方法可以拦截目标类上的方法调用或生命周期事件。

本章将介绍以下内容：

- 拦截器概述
- 使用拦截器
- interceptor 示例应用

25.1 拦截器概述

拦截器与 Java EE 托管类结合使用，允许开发人员在关联的**目标类**上结合方法调用或生命周期事件调用拦截器方法。拦截器常用来完成日志记录、审计和性能分析。

尽管拦截器是 Enterprise JavaBeans 3.2 以及 Java EE 1.1 上下文和依赖注入规范的一部分，不过 Interceptors 1.2 规范也可以作为 JSR 318（Enterprise JavaBeans 3.1）维护版本的一部分下载，下载地址为 `http://jcp.org/en/jsr/detail?id=318`。可以对会话 bean、消息驱动 bean 和 CDI 托管 bean 使用拦截器。在所有这些情况下，拦截器目标类是 bean 类。

可以在目标类中将拦截器定义为一个**拦截器方法**（interceptor method），或者在一个关联类中定义，这称为一个**拦截器类**（interceptor class）。拦截器类包含的方法可以结合目标类的方法或生命周期事件调用。

拦截器类和方法使用元数据注解定义，或者对于包含拦截器和目标类的应用，可以在应用的部署描述文件中定义。

说明 使用部署描述文件定义拦截器的应用不能跨 Java EE 服务器移植。

要为目标类中的拦截器方法或拦截器类中的拦截器方法增加表 25-1 中定义的一个元数据注解。

表 25-1 拦截器元数据注解

拦截器元数据注解	描 述
javax.interceptor.AroundConstruct	指定这个方法为拦截器方法，构造目标类之后接收一个回调
javax.interceptor.AroundInvoke	指定这个方法为一个拦截器方法
javax.interceptor.AroundTimeout	指定这个方法为超时拦截器，拦截企业 bean 定时器的超时方法
javax.annotation.PostConstruct	指定这个方法为拦截器方法，拦截构造之后的生命周期事件
javax.annotation.PreDestroy	指定这个方法为拦截器方法，拦截撤销之前的生命周期事件

25.1.1 拦截器类

可以用 javax.interceptor.Interceptor 注解指定拦截器类（可选），不过拦截器类并不要求一定要加注解。拦截器类必须有一个公共的无参数构造函数。

目标类可以有任意多个拦截器类与之关联。调用拦截器类的顺序由 javax.interceptor.Interceptors 注解中定义的拦截器类顺序来确定。不过，可以在部署描述文件中覆盖这个顺序。

拦截器类可以作为依赖注入的目标。创建拦截器类实例时，会在调用 @PostConstruct 回调之前发生依赖注入，这里会使用关联目标类的命名上下文。

25.1.2 拦截器生命周期

拦截器类与其关联目标类有相同的生命周期。创建一个目标类实例时，会在目标类中为声明的各个拦截器类创建一个拦截器类实例。也就是说，如果目标类声明了多个拦截器类，创建目标类实例时，会为每一个类创建一个实例。目标类实例和所有拦截器类实例会在调用 @PostConstruct 回调之前完全实例化，在目标类和拦截器类实例撤销之前会调用 @PreDestroy 回调。

25.1.3 拦截器和 CDI

Java EE 上下文和依赖注入（CDI）建立在 Java EE 拦截器的基本功能之上。本书第 1 卷中的 25.6 节介绍了有关 CDI 拦截器的更多信息，包括对拦截器绑定类型的讨论。

25.2 使用拦截器

定义一个拦截器时，要在目标类中使用表 25-1 中所列的一个拦截器元数据注解，或者

在一个单独的拦截器类中使用注解。下面的代码在目标类中声明了一个 @AroundTimeout 拦截器方法：

```
@Stateless
public class TimerBean {
    ...
    @Schedule(minute="*/1", hour="*")
    public void automaticTimerMethod() { ... }

    @AroundTimeout
    public void timeoutInterceptorMethod(InvocationContext ctx) { ... }
    ...
}
```

如果使用拦截器类，可以使用 javax.interceptor.Interceptors 注解，在目标类的类级或方法级声明一个或多个拦截器。下面的代码在类级声明拦截器：

```
@Stateless
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
public class OrderBean { ... }
```

下面的代码声明一个方法级拦截器类：

```
@Stateless
public class OrderBean {
    ...
    @Interceptors(OrderInterceptor.class)
    public void placeOrder(Order order) { ... }
    ...
}
```

25.2.1 拦截方法调用

使用 @AroundInvoke 注解为托管对象方法指定拦截器方法。每个类只允许有一个 around-invoke 拦截器方法。around-invoke 拦截器方法有以下形式：

```
@AroundInvoke
visibility Object method-name(InvocationContext) throws Exception { ... }
```

例如：

```
@AroundInvoke
public void interceptOrder(InvocationContext ctx) { ... }
```

around-invoke 拦截器方法允许公共、私有、保护或包级访问，而且不能声明为静态或最终方法。

around-invoke 拦截器可以调用它拦截的目标方法所能调用的任何组件或资源，与目标方法有相同的安全和事务上下文，可以在目标方法所在的 Java 虚拟机调用栈中运行。

around-invoke 拦截器可以抛出运行时异常和目标方法的 throws 子句所允许的任何异常。它们可以捕获和抑制异常，然后通过调用 InvocationContext.proceed

方法恢复。

25.2.1.1 使用多个方法拦截器

使用 @Interceptors 注解为目标方法或类声明多个拦截器：

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
        LastInterceptor.class})
public void updateInfo(String info) { ... }
```

@Interceptors 注解中的拦截器顺序就是调用拦截器的顺序。

还可以在部署描述文件中定义多个拦截器。部署描述文件中拦截器的顺序将作为调用这些拦截器的顺序：

```
...
<interceptor-binding>
    <target-name>myapp.OrderBean</target-name>
    <interceptor-class>myapp.PrimaryInterceptor.class</interceptor-class>
    <interceptor-class>myapp.SecondaryInterceptor.class</interceptor-class>
    <interceptor-class>myapp.LastInterceptor.class</interceptor-class>
    <method-name>updateInfo</method-name>
</interceptor-binding>
...
```

要显式地将控制传递到链中的下一个拦截器，可以调用 InvocationContext.proceed 方法。

数据可以跨拦截器共享。

- 同一个 InvocationContext 实例可以作为输入参数传递到特定目标方法拦截器链中的每一个拦截器方法。可以使用 InvocationContext 实例的 contextData 属性跨拦截器方法传递数据。contextData 属性是一个 java.util.Map<String,Object> 对象。存储在 contextData 中的数据可以由拦截器链中后续的拦截器方法访问。
- contextData 中存储的数据不能跨不同的目标类方法调用共享。也就是说，会为目标类中的各个方法调用创建一个不同的 InvocationContext 对象。

25.2.1.2 从拦截器类访问目标方法参数

可以使用传入各个 around-invoke 方法的 InvocationContext 实例来访问和修改目标方法的参数。InvocationContext 的 parameters 属性是一个 Object 实例数组，与目标方法的参数顺序对应。例如，对于以下目标方法，在传入 PrimaryInterceptor 的 around-invoke 拦截器方法的 Invocation Context 实例中，parameters 属性是一个 Object 数组，其中包含两个 String 对象（firstName 和 lastName）和一个 Date 对象（date）：

```
@Interceptors(PrimaryInterceptor.class)
public void updateInfo(String firstName, String lastName, Date date) { ... }
```

可以分别使用 InvocationContext.getParameters 和 InvocationContext.setParameters 方法访问和修改这些参数。

25.2.2 拦截生命周期回调事件

生命周期回调事件（around-construct、post-construct 和 pre-destroy）的拦截器可以在目标类中定义，也可以在拦截器类中定义。javax.interceptor.AroundConstruct 注解指定方法作为一个拦截器方法，可以拦截目标类构造函数的调用。javax.annotation.PostConstruct 注解用来指定一个方法作为 post-construct 生命周期事件拦截器。javax.annotation.PreDestroy 注解用来指定一个方法作为 pre-destroy 生命周期事件拦截器。

目标类中定义的生命周期事件拦截器有以下形式：

```
void method-name() { ... }
```

例如：

```
@PostConstruct
void initialize() { ... }
```

拦截器类中定义的生命周期事件拦截器有以下形式：

```
void method-name(InvocationContext) { ... }
```

例如：

```
@PreDestroy
void cleanup(InvocationContext ctx) { ... }
```

生命周期拦截器方法允许公共、私有、保护或包级访问，而且不能声明为静态或最终方法。生命周期拦截器可以抛出运行时异常，但是不能抛出受查异常。

生命周期拦截器方法在未指定的安全和事务上下文中调用。也就是说，可移植的 Java EE 应用不能假设生命周期事件拦截器方法一定能访问某个安全或事务上下文。每个类中对于各个生命周期事件（post-create 和 pre-destroy）只允许有一个拦截器方法。

25.2.2.1 使用 AroundConstruct 拦截器方法

@AroundConstruct 方法拦截目标类构造函数的调用。只能在拦截器类或拦截器类超类中定义用 @AroundConstruct 装饰的方法。不能在目标类中使用 @AroundConstruct 方法。

对目标类关联的所有拦截器完成依赖注入之后才会调用 @AroundConstruct 方法。所有关联的 @AroundConstruct 方法调用了 Invocation.proceed 方法之后会创建目标类并完成目标类的构造函数注入。此时，目标类的依赖注入已完成，然后调用 @PostConstruct 回调方法（如果有）。

@AroundConstruct 方法调用 Invocation.proceed 之后可以访问所构造的目标实例，为此要调用 InvocationContext.getTarget 方法。

注意 要想从 @AroundConstruct 方法调用目标实例的方法，这会很危险，因为目标实例的依赖注入可能还没有完成。

@AroundConstruct 方法必须调用 Invocation.proceed 来创建目标实例。如果 @AroundConstruct 方法没有调用 Invocation.proceed，目标实例就不会创建。

25.2.2.2 使用多个生命周期回调拦截器

通过在 @Interceptors 注解中指定拦截器类，可以为一个目标类定义多个生命周期拦截器：

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
        LastInterceptor.class})
@Stateless
public class OrderBean { ... }
```

存储在 InvocationContext 的 contextData 属性中的数据不能跨不同生命周期事件共享。

25.2.3 拦截超时事件

通过对目标类或拦截器类中的方法使用 @AroundTimeout 注解，可以为 EJB 定时器服务超时方法定义拦截器。每个类只允许有一个 @AroundTimeout 方法。

超时拦截器有以下形式：

```
Object method-name(InvocationContext) throws Exception { ... }
```

例如：

```
@AroundTimeout
protected void timeoutInterceptorMethod(InvocationContext ctx) { ... }
```

超时拦截器方法允许公共、私有、保护或包级访问，但不能声明为静态或最终方法。

超时拦截器可以调用目标超时方法可调用的任何组件或资源，并在目标方法所在的相同事务和安全上下文中调用。

通过 InvocationContext 实例的 getTimer 方法，超时拦截器可以访问与目标超时方法关联的定时器对象。

25.2.3.1 使用多个超时拦截器

在 @Interceptors 注解中，通过在类级指定包含 @AroundTimeou 拦截器方法的拦截器类，可以为一个给定的目标类定义多个超时拦截器。

如果目标类在一个拦截器类中指定了超时拦截器，而且在目标类本身也有一个 @AroundTimeout 拦截器方法，那么会先调用拦截器类中的超时拦截器，然后再调用目标类中定义的超时拦截器。例如，在下面的例子中，假设 PrimaryInterceptor 和 SecondaryInterceptor 类都有超时拦截器方法：

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
@Stateful
public class OrderBean {
```

```
    ...
    @AroundTimeout
    private void last(InvocationContext ctx) { ... }
    ...
}
```

PrimaryInterceptor 中的超时拦截器会先调用，然后是 SecondaryInterceptor 中的超时拦截器，最后才调用目标类中定义的 last 方法。

25.2.4 为组件绑定拦截器

拦截器绑定类型是应用于组件的注解，将组件与一个特定的拦截器关联。拦截器绑定类型通常是定制的运行时注解类型，会指定拦截器目标。在定制注解定义上使用 javax.interceptor.InterceptorBinding 注解，并指定目标，为此要使用 @Target，并设置一个或多个 TYPE（类级拦截器）、METHOD（方法级拦截器）、CONSTRUCTOR（around-construct 拦截器）或任何其他合法目标：

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Inherited
pubic @interface Logged { ... }
```

拦截器绑定类型还可以应用到其他拦截器绑定类型：

```
@Logged
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Inherited
public @interface Secured { ... }
```

25.2.4.1 在拦截器类上声明拦截器绑定

为拦截器类增加拦截器绑定类型和 @Interceptor 注解，将拦截器绑定与拦截器类关联：

```
@Logged
@Interceptor
public class LoggingInterceptor {
   @AroundInvoke
   public Object logInvocation(InvocationContext ctx) throws Exception { ... }
   ...
}
```

拦截器类可以声明多个拦截器绑定类型，多个拦截器类也可以声明一个拦截器绑定类型。

如果拦截器类拦截生命周期回调，则只能用 Target(TYPE) 声明拦截器绑定类型，而对于 @AroundConstruct 生命周期回调，要使用 Target(CONSTRUCTOR)。

25.2.4.2　将组件绑定到拦截器

为目标组件的类、方法或构造函数增加拦截器绑定类型注解。拦截器绑定类型与 @Interceptor 注解的应用规则相同：

```
@Logged
public class Message {
    ...
    @Secured
    public void getConfidentialMessage() { ... }
    ...
}
```

如果组件有一个类级拦截器绑定，则该组件不能是 final，或者有任何非 static、非 private final 方法。如果一个非 static、非 private 方法应用了一个拦截器绑定，则这个方法不能是 final，组件类也不能是 final。

25.2.5　拦截器排序

调用多个拦截器时，由以下规则确定调用顺序：

- 默认拦截器在部署描述文件中定义，会首先调用。它们可能指定调用顺序，或者覆盖注解指定的顺序。默认拦截器会按其在部署描述文件中定义的顺序来调用。
- @Interceptors 注解中所列的拦截器类顺序定义了这些拦截器的调用顺序。对于 @Interceptors 注解中列出的拦截器，会忽略其 @Priority 设置。
- 如果拦截器类有超类，会首先调用超类中定义的拦截器（从最一般的超类开始）。
- 拦截器类可以在 javax.annotation.Priority 注解中设置一个值，来设置拦截器方法的优先级。
- 拦截器类中定义的所有拦截器都调用之后，会按 @Interceptors 注解中的拦截器顺序调用目标类的构造函数、around-invoke 或 around-timeout 拦截器。
- 如果目标类有超类，会首先调用超类中定义的所有拦截器（从最一般的超类开始）。

@Priority 注解需要一个 int 值作为一个元素。这个数越小，关联拦截器的优先级越高。

说明　对于有相同优先级值的拦截器，其调用顺序由具体实现决定。

javax.interceptor.Interceptor.Priority 类定义了表 25-2 所列的优先级常量。

表 25-2　拦截器优先级常量

优先级常量	值	描　述
PLATFORM_BEFORE	0	如果拦截器由 Java EE 平台定义，并在调用链早期调用，则要使用 PLATFORM_BEFORE 和 LIBRARY_BEFORE 之间的值。这些拦截器优先级最高
LIBRARY_BEFORE	1000	如果拦截器由扩展库定义，并且应当在拦截器链早期调用，则应当使用 LIBRARY_BEFORE 和 APPLICATION 之间的值

（续）

优先级常量	值	描　述
APPLICATION	2000	由应用定义的拦截器应当使用 APPLICATION 和 LIBRARY_AFTER 之间的值
LIBRARY_AFTER	3000	由扩展库定义的低优先级拦截器应当使用 LIBRARY_AFTER 和 PLATFORM_AFTER 之间的值
PLATFORM_AFTER	4000	Java EE 平台定义的低优先级拦截器应当使用大于 PLATFORM_AFTER 的值

说明 拦截器规范保留了负优先级值，以备将来使用，所以目前不要使用负值。

下面的代码段显示了如何在应用定义的一个拦截器中使用优先级常量：

```
@Interceptor
@Priority(Interceptor.Priority.APPLICATION+200
public class MyInterceptor { ... }
```

25.3 interceptor 示例应用

interceptor 示例展示了如何对一个无状态会话 bean 使用拦截器类（包含一个 @AroundInvoke 拦截器方法）。

HelloBean 无状态会话 bean 是一个简单的企业 bean，有两个业务方法 getName 和 setName，可以获取和修改一个字符串。setName 业务方法有一个 @Interceptors 注解，为这个方法指定了一个拦截器类 HelloInterceptor：

```
@Interceptors(HelloInterceptor.class)
public void setName(String name) {
    this.name = name;
}
```

HelloInterceptor 类定义了一个 @AroundInvoke 拦截器方法 modifyGreeting，将传入 HelloBean.setName 的字符串转换为小写：

```
@AroundInvoke
public Object modifyGreeting(InvocationContext ctx) throws Exception {
    Object[] parameters = ctx.getParameters();
    String param = (String) parameters[0];
    param = param.toLowerCase();
    parameters[0] = param;
    ctx.setParameters(parameters);
    try {
        return ctx.proceed();
    } catch (Exception e) {
        logger.warning("Error calling ctx.proceed in modifyGreeting()");
        return null;
    }
}
```

通过调用 InvocationContext.getParameters 方法获取 HelloBean.setName 的参数，并存储在一个 Object 数组中。由于 setName 只有一个参数，因此它是数组中第一个也是唯一一个元素。这个字符串设置为小写，并存储在 parameters 数组中，然后传入 InvocationContext.setParameters。为了把控制返回给会话 bean，要调用 InvocationContext.proceed。

interceptor 的用户界面是一个 JSF Web 应用，由两个 Facelets 视图组成：index.xhtml 包含一个表单用来输入名字，response.xhtml 会显示最终的名字。

25.3.1　运行 interceptor 示例

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 interceptor 示例。

25.3.1.1　使用 NetBeans IDE 运行 interceptor 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/ejb
```

4. 选择 interceptor 文件夹并单击 Open Project。
5. 在 Projects（工程）标签页中，右键单击 interceptor 工程并选择 Run（运行）。

这会编译、部署和运行 interceptor 示例，在 Web 浏览器中打开以下 URL：

```
http://localhost:8080/interceptor/
```

6. 在表单中输入一个名字，并单击 Submit（提交）。

这个名字会由 HelloInterceptor 类中定义的方法拦截器转换为小写。

25.3.1.2　使用 Maven 运行 interceptor 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 进入以下目录：

```
tut-install/examples/ejb/interceptor/
```

3. 要编译源文件，并打包这个应用，使用以下命令：

```
mvn install
```

这个命令会构建应用并打包到一个 WAR 文件 interceptor.war，这个文件位于 target 目录。然后将这个 WAR 文件部署到 GlassFish 服务器。

4. 在 Web 浏览器中打开以下 URL：

```
http://localhost:8080/interceptor/
```

5. 在表单中输入一个名字并单击 Submit（提交）。

这个名字会由 HelloInterceptor 类中定义的方法拦截器转换为小写。

Chapter 26

第 26 章

批 处 理

本章介绍 Java 平台批处理应用（JSR 352），这为定义、实现和运行批处理作业提供了支持。批处理作业是无需用户交互就可以执行的任务。批处理框架由一个基于 XML 的作业规范语言、一个 Java API 和一个批处理运行时环境组成。

有些企业应用包含无需用户交互就可以执行的任务。这些任务定期执行，或者在资源利用率低时执行，它们通常会处理大量信息，如日志文件、数据库记录或图像。这样的例子包括计费、报告生成、数据格式转换和图像处理。这些任务就称为**批处理作业**（batch job）。

批处理（batch processing）是指在一个计算机系统上运行批处理作业。Java EE 包括一个批处理框架，提供了所有批处理应用共同的批处理执行基础设施，允许开发人员集中精力实现其批处理应用的业务逻辑。这个批处理框架由一个基于 XML 的作业规范语言、为实现业务逻辑的应用类提供的一组批处理注解和接口、一个管理批处理作业执行的批处理容器以及与批处理容器交互的一些支持类和接口组成。

本章将介绍以下内容：

- 批处理介绍
- Java EE 中的批处理
- 简单用例
- 使用作业规范语言
- 创建批处理工件
- 向批处理运行时环境提交作业
- 打包批处理应用
- webserverlog 示例应用

- phonebilling 示例应用
- 关于批处理的更多信息

26.1 批处理介绍

批处理作业无需用户干预就可以完成。例如，考虑一个电话计费应用，它从企业信息系统读取电话拨号记录，每月为每个账户生成一个账单。由于这个应用不需要任何用户交互，因此可以作为一个批处理作业运行。

这个电话计费应用包括两个阶段：第一个阶段将登记表中的每个拨号记录与一个月账单关联，第二个阶段计算各个账单的税费和总金额。这两个阶段都作为批处理作业中的一个**步骤**（step）。

批处理应用指定一组步骤和这些步骤的执行顺序。不同的批处理框架可能还会指定额外的元素，如判定元素或并行运行的步骤组。下面各小节将更详细地介绍步骤，并提供批处理框架其他公共特点的有关信息。

26.1.1 批处理作业中的步骤

步骤是批处理作业中一个独立的顺序阶段。批处理作业包含面向块的步骤和面向任务的步骤。

- **面向块的步骤**（chunk-oriented step）也称为块步骤，可以处理数据，包括从数据源读取项，对每一项应用某个业务逻辑，并存储结果。块步骤一次读取和处理一项，并把结果成组为一个块。这个块达到一个可配置的大小时，就存储结果。面向块的处理可以更高效地存储结果，并有利于事务界定。

  块步骤有 3 个部分：
  - 输入获取部分一次从数据源读取一项，如数据库中的一个记录、目录中的一个文件或日志文件中的一个日志记录。
  - 业务处理部分使用由应用定义的业务逻辑一次处理一项。例如过滤、格式化和访问项中的数据来计算一个结果。
  - 输出书写部分一次存储一个块所处理的项。

块步骤通常运行时间很长，因为块步骤会处理大量数据。批处理框架允许块步骤使用**检查点**（checkpoint）对其进展建立书签。被中断的块步骤可以从上一个检查点重新开始。块步骤的输入获取和输出书写部分会保存每个块处理之后的当前位置，这样在步骤重新开始时可以从这个位置恢复。

图 26-1 显示了一个批处理作业中两个步骤的 3 个部分。

- **面向任务的步骤**（task-oriented step）即任务步骤，除了处理数据源的项外，还要执行任务。这包括创建或删除目录、移动文件、创建或删除数据库表、配置资源等。与块

步骤相比，任务步骤通常不会长时间运行。

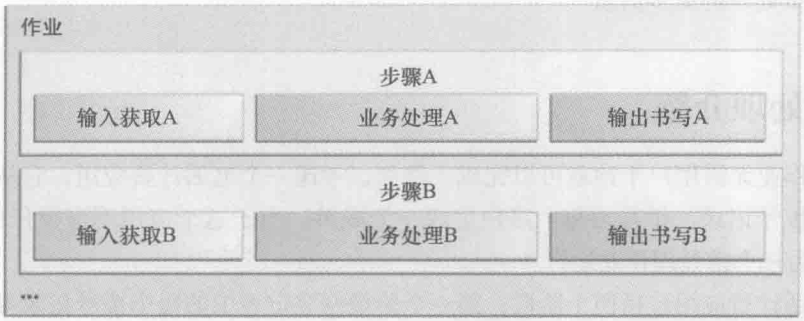


图 26-1 批处理作业中的块步骤

例如，电话计费应用包括两个块步骤：

- 在第一个步骤中，输入获取部分从登记表中读取拨号记录；业务处理部分将各个拨号记录与一个账单关联，如果一个账户还没有账单，则创建一个账单；输出书写部分将各个账单存储在一个数据库中。
- 在第二个步骤中，输入获取部分从数据库读取账单；业务处理部分计算每个账单的税费和总金额；输出书写部分更新数据库记录，并为每个账单生成可打印的版本。

这个应用还可以包含一个任务步骤，从上个月生成的账单清理文件。

26.1.2 并行处理

批处理作业通常要处理大量数据，或者要完成计算开销很大的操作。在这两种情况下，并行处理对批处理应用很有好处。

- 不相互依赖的步骤可以在不同的线程上运行。
- 面向块的步骤中，如果各个项的处理不依赖于之前项的处理结果，这些步骤可以在多个线程中运行。

批处理框架为开发人员提供了一些机制来定义独立的步骤组，将面向块的步骤分解为可以并行运行的部分。

26.1.3 状态和判定元素

批处理框架会跟踪作业中每一个步骤的**状态**。状态指示一个步骤正在运行还是已经完成。如果步骤已经完成，状态则会指示：

- 步骤执行成功。
- 步骤被中断。
- 步骤执行中出现一个错误。

除了步骤以外，批处理作业还可以包含**判定元素**（decision element）。判定元素使用前一

个步骤的退出状态来确定下一个步骤，或终止这个批处理作业。终止一个批处理作业时，判定元素会设置这个批处理作业的状态。与步骤类似，批处理作业可以成功结束、被中断或失败。

图 26-2 显示了一个作业的例子，其中包含块步骤、任务步骤和一个判定元素。

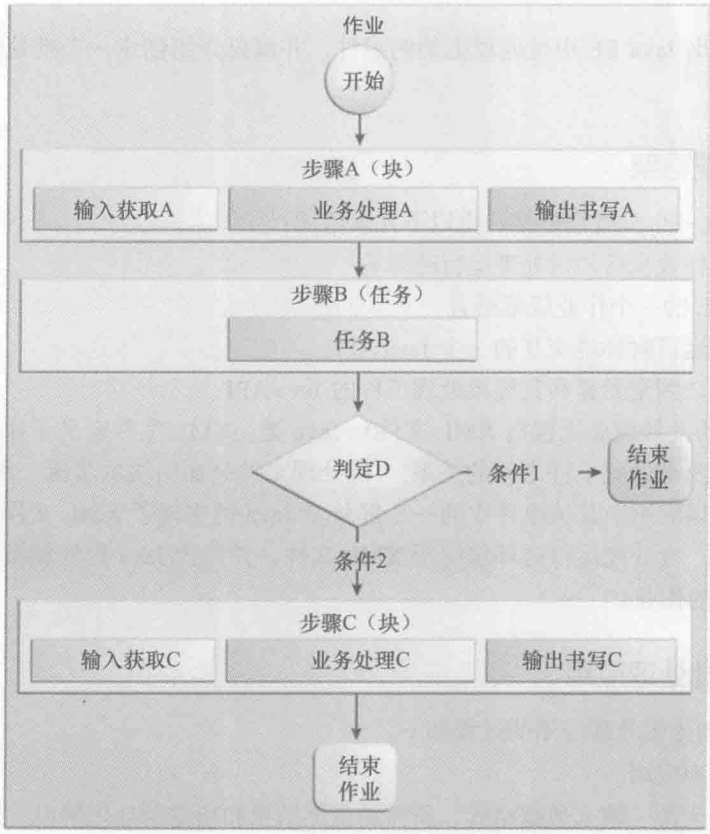


图 26-2 作业中的步骤和判定元素

26.1.4 批处理框架功能

批处理应用有以下常见的需求：

- 定义作业、步骤、判定元素以及它们之间的关系。
- 并行执行步骤组或步骤中的部分。
- 为作业和步骤维护状态信息。
- 执行作业和恢复被中断的作业。
- 处理错误。

批处理框架提供了批处理执行基础设施，可以解决所有批处理应用中的这些常见需求，允许开发人员集中精力处理应用的业务逻辑。批处理框架包括一个指定作业和步骤的格式、

一个应用编程接口（API）和一个可以在运行时管理批处理作业执行的服务。

26.2 Java EE 中的批处理

这一节将列出 Java EE 中批处理框架的组件，并概要介绍创建一个批处理应用所需遵循的步骤。

26.2.1 批处理框架

Java EE 包含一个批处理框架，由以下元素组成：

- 一个管理作业执行的批处理运行时环境
- 基于 XML 的一个作业规范语言
- 与批处理运行时环境交互的一个 Java API
- 实现步骤、判定元素和其他批处理工件的 Java API

Java EE 中的批处理应用包括 XML 文件和 Java 类。XML 文件定义了作业的结构，会指定批处理工件以及批处理工件之间的关系。（批处理工件是面向块的步骤、面向任务的步骤、判定元素或批处理应用中其他组件中的一个部分。）Java 类实现了 XML 文件中定义的批处理工件的应用逻辑。批处理运行时环境解析 XML 文件，并作为 Java 类加载批处理工件，来运行批处理应用中的作业。

26.2.2 创建批处理应用

Java EE 中创建批处理应用的过程如下。

1. 设计批处理应用。
   a. 明确输入源、输入数据格式、所需的最终结果和所需的处理阶段。
   b. 将应用组织为一个包含面向块的步骤、面向任务的步骤和判定元素的作业。确定它们之间的依赖关系。
   c. 按步骤之间的迁移来确定执行顺序。
   d. 明确可以并行运行的步骤以及可以在多个线程中运行的步骤。

2. 实现框架为步骤、判定元素等指定的接口，将批处理工件创建为 Java 类。这些 Java 类包含相应的代码，可以从输入源读取数据、格式化项、处理项，并存储结果。批处理工件可以使用依赖注入从批处理运行时环境访问上下文对象。

3. 在 XML 文件中使用作业规范语言（Job Specification Language）定义作业、步骤及其执行流。XML 文件中的元素会引用批处理工件（实现为 Java 类）。这些批处理工件可以访问 XML 文件中声明的属性，如文件和数据库名。

4. 使用批处理运行时环境提供的 Java API 来启动批处理应用。

下面各小节将详细介绍如何使用 Java EE 中的批处理框架的组件来创建批处理应用。

26.2.3 批处理作业的元素

批处理作业可以包含一个或多个以下元素：

- 步骤（Step）
- 流（Flow）
- 分支（Split）
- 判定元素（Decision element）

步骤在26.1节中介绍过，可以是面向块的步骤或面向任务的步骤。面向块的步骤可以是**划分步骤**（partitioned step）。在一个划分块步骤中，一个项的处理不依赖于其他项，所以这些步骤可以在多个线程中运行。

流（flow）是作为一个单元执行的一个步骤序列。相关步骤的一个序列可以成组为一个流。一个流中的步骤不能转移到流外的步骤。这个流中最后一个步骤完成时会转移到下一个元素。

分支（split）是一组并行执行的流；每个流在一个单独的线程中运行。分支中所有流都完成时，会转移到下一个元素。

判定元素使用前一个步骤的退出状态来确定下一个步骤，或者终止这个批处理作业。

26.2.4 属性和参数

作业和步骤可以有多个与之关联的**属性**（property）。属性在作业定义文件中定义，批处理工件从批处理运行时环境使用上下文对象访问这些属性。以这种方式使用属性，可以将作业的静态参数与业务逻辑解耦合，并在不同的作业定义文件中重用批处理工件。

26.4节中会介绍如何指定属性，而如何访问批处理工件中的属性则在26.5节中讨论。

Java EE应用向批处理运行时环境提交作业时，还可以向作业传递**参数**（parameter）。这允许你指定只在运行时才知道的动态参数。参数对于划分步骤也是必要的，因为每个划分需要知道一些信息，例如要处理的项范围。

26.6节中会介绍提交作业时指定参数的有关内容。26.9节中会介绍如何为划分步骤指定参数并在批处理工件中访问这些参数。

26.2.5 作业实例和作业执行

一个作业定义可以有多个**实例**（instance），每个实例有不同的参数。作业**执行**（execution）就是尝试运行一个作业实例。批处理运行时环境维护了有关作业实例和作业执行的信息，见26.6.2节中的介绍。

26.2.6 批处理和退出状态

作业、步骤、分支和流的状态在批处理运行时环境中表示为一个**批处理状态**（batch

status）值。批处理状态值见表 26-1，这些值表示为字符串。

表 26-1 批处理状态值

值	描 述
STARTING	作业已经提交到批处理运行时环境
STARTED	作业正在运行
STOPPING	已经请求这个作业停止
STOPPED	作业已经停止
FAILED	作业由于一个错误而结束执行
COMPLETED	作业成功结束执行
ABANDONED	作业已经标志为放弃

Java EE 应用可以提交作业，并使用 JobOperator 接口访问一个作业的批处理状态，见 26.6 节中的介绍。作业定义文件可以使用作业规范语言（JSL）引用批处理状态值，参见 26.4 节。批处理工件可以使用上下文对象访问批处理状态值，见 26.5.3 节的介绍。

对于流，其批处理状态是流中最后一步的状态。对于分支，批处理状态可以是以下状态：

- COMPLETED：分支中所有流的批处理状态都为 COMPLETED。
- FAILED：某个流的批处理状态是 FAILED。
- STOPPED：某个流的批处理状态是 STOPPED，而且所有流的批处理状态都不是 FAILED。

作业、步骤、分支和流的批处理状态由批处理运行时环境设置。作业、步骤、分支和流还可以有一个**退出状态**（exit statu），这是基于批处理状态的一个用户自定义值。可以在批处理工件中设置退出状态，或者在作业定义文件中设置。如上所述，类似于访问批处理状态，可以采用同样的方式访问退出状态。退出状态的默认值与批处理状态的默认值相同。

26.3 简单用例

这一节介绍如何使用作业规范语言（JSL）定义一个简单的作业，以及如何实现相应的批处理工件。有关批处理框架中元素的详细描述，参见这一章的其余小节。

下面的作业定义指定了一个块步骤和一个任务步骤，如下所示：

```
<job id="simplejob" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
                    version="1.0">
  <properties>
    <property name="input_file" value="input.txt"/>
    <property name="output_file" value="output.txt"/>
  </properties>
  <step id="mychunk" next="mytask">
    <chunk>
      <reader ref="MyReader"></reader>
```

```
      <processor ref="MyProcessor"></processor>
      <writer ref="MyWriter"></writer>
    </chunk>
  </step>
  <step id="mytask">
    <batchlet ref="MyBatchlet"></batchlet>
    <end on="COMPLETED"/>
  </step>
</job>
```

26.3.1 块步骤

大多数情况下，必须为面向块的步骤实现一个检查点类。下面的类会跟踪记录一个文本文件中的行号：

```
public class MyCheckpoint implements Serializable {
    private long lineNum = 0;
    public void increase() { lineNum++; }
    public long getLineNum() { return lineNum; }
}
```

下面是一个项阅读器实现，会在作业重新开始时从所提供的检查点继续读取输入文件。项是文本文件中的各行（在更复杂的情况下，项可以是定制的 Java 类型，其输入源可以是数据库）：

```
@Dependent
@Named("MyReader")
public class MyReader implements javax.batch.api.chunk.ItemReader {
    private MyCheckpoint checkpoint;
    private BufferedReader breader;
    @Inject
    JobContext jobCtx;

    public MyReader() {}

    @Override
    public void open(Serializable ckpt) throws Exception {
        if (ckpt == null)
            checkpoint = new MyCheckpoint();
        else
            checkpoint = (MyCheckpoint) ckpt;
        String fileName = jobCtx.getProperties()
                                .getProperty("input_file");
        breader = new BufferedReader(new FileReader(fileName));
        for (long i = 0; i < checkpoint.getLineNum(); i++)
            breader.readLine();
    }

    @Override
    public void close() throws Exception {
```

```
        breader.close();
    }

    @Override
    public Object readItem() throws Exception {
        String line = breader.readLine();
        return line;
    }
}
```

在这里，项处理器只是把这些行转换为大写。更复杂的例子还可能采用不同的方式处理项，或者将它们转换为定制的输出 Java 类型：

```
@Dependent
@Named("MyProcessor")
public class MyProcessor implements javax.batch.api.chunk.ItemProcessor {

    public MyProcessor() {}

    @Override
    public Object processItem(Object obj) throws Exception {
        String line = (String) obj;
        return line.toUpperCase();
    }
}
```

说明 批处理 API 不支持泛型。大多数情况下，在处理项之前需要将其转换为特定的类型。

项书写器将所处理的项写入输出文件。如果没有提供检查点，它会覆盖输出文件；否则，会从文件末尾继续写。项会按块写出：

```
@Dependent
@Named("MyWriter")
public class MyWriter implements javax.batch.api.chunk.ItemWriter {
    private BufferedWriter bwriter;
    @Inject
    private JobContext jobCtx;

    @Override
    public void open(Serializable ckpt) throws Exception {
        String fileName = jobCtx.getProperties()
                                .getProperty("output_file");
        bwriter = new BufferedWriter(new FileWriter(fileName,
                                                    (ckpt != null)));
    }

    @Override
    public void writeItems(List<Object> items) throws Exception {
```

```
        for (int i = 0; i < items.size(); i++) {
            String line = (String) items.get(i);
            bwriter.write(line);
            bwriter.newLine();
        }
    }

    @Override
    public Serializable checkpointInfo() throws Exception {
        return new MyCheckpoint();
}
```

26.3.2 任务步骤

任务步骤显示输出文件的长度。对于更复杂的情况，任务步骤可以完成块处理编程模型中无法实现的任务：

```
@Dependent
@Named("MyBatchlet")
public class MyBatchlet implements javax.batch.api.chunk.Batchlet {
    @Inject
    private JobContext jobCtx;

    @Override
    public String process() throws Exception {
        String fileName = jobCtx.getProperties()
                                .getProperty("output_file");
        System.out.println(""+(new File(fileName)).length());
        return "COMPLETED";
    }
}
```

26.4 使用作业规范语言

作业规范语言（Job Specification Language，JSL）允许使用 XML 文件定义作业中的步骤以及这些步骤的执行顺序。下面的例子显示了如何定义一个简单的作业，其中包含一个块步骤和一个任务步骤：

```
<job id="loganalysis" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
                      version="1.0">
  <properties>
    <property name="input_file" value="input1.txt"/>
    <property name="output_file" value="output2.txt"/>
  </properties>

  <step id="logprocessor" next="cleanup">
    <chunk checkpoint-policy="item" item-count="10">
      <reader ref="com.example.pkg.LogItemReader"></reader>
```

```
      <processor ref="com.example.pkg.LogItemProcessor"></processor>
      <writer ref="com.example.pkg.LogItemWriter"></writer>
    </chunk>
  </step>

  <step id="cleanup">
    <batchlet ref="com.example.pkg.CleanUp"></batchlet>
    <end on="COMPLETED"/>
  </step>
</job>
```

这个例子定义了批处理作业 loganalysis，它由块步骤 logprocessor 和任务步骤 cleanup 组成。Logprocessor 步骤会转移到 cleanup 步骤，cleanup 步骤完成时会终止作业。

job 元素定义了两个属性 input_file 和 output_file。通过这种方式指定属性，允许使用不同的配置参数运行一个批处理作业，而不必重新编译它的 Java 批处理工件。批处理工件可以从批处理运行时环境使用上下文对象访问这些属性。

Logprocessor 步骤是一个块步骤，将为阅读器（LogItemReader）、处理器（LogItemProcessor）和书写器（LogItemWriter）指定批处理工件。每处理 10 个项，这个步骤会创建一个检查点。

Cleanup 步骤是一个任务步骤，指定 CleanUp 类作为它的批处理工件。这个步骤完成时，作业终止。

下面各小节将更详细地介绍作业规范语言（JSL）的元素，并展示最常用的属性和子元素。

26.4.1 job 元素

job 元素是作业定义文件中的顶层元素。它的主要属性是 id 和 restartable。job 元素可以包含一个 properties 元素以及 0 个或多个以下元素：listener、step、flow 和 split。例如：

```
<job id="jobname" restartable="true">
  <listeners>
    <listener ref="com.example.pkg.ListenerBatchArtifact"/>
  </listeners>
  <properties>
    <property name="propertyName1" value="propertyValue1"/>
    <property name="propertyName2" value="propertyValue2"/>
  </properties>
  <step ...> ... </step>
  <step ...> ... </step>
  <decision ...> ... </decision>
  <flow ...> ... </flow>
  <split ...> ... </split>
</job>
```

listener 元素指定一个批处理工件，将在作业执行前和执行后调用这个监听器的方法。这个批处理工件是 javax.batch.api.listener.JobListener 接口的一个实现。更多信息参见 26.8.1.4 节，其中给出了一个作业监听器实现的例子。

job 元素中的第一个 step、flow 或 split 元素最先执行。

26.4.2 step 元素

step 元素可以是 job 和 flow 元素的一个子元素。它的主要属性是 id 和 next。step 元素可以包含以下元素：

- 一个面向块步骤的 chunk 元素或一个面向任务步骤的 batchlet 元素。
- 一个 properties 元素（可选）。
  这个元素可以指定一组属性，批处理工件可以使用批处理上下文对象访问这些属性。
- 一个 listener 元素（可选）；如果指定了多个监听器，便会有一个 listeners 元素。
  这个元素指定拦截步骤执行不同阶段的监听器工件。

对于块步骤，监听器的批处理工件可以是以下接口的实现：StepListener、ItemReadListener、ItemProcessListener、ItemWriteListener、ChunkListener、RetryReadListener、RetryProcessListener、RetryWriteListener、SkipReadListener、SkipProcessListener 和 SkipWriteListener。

对于任务步骤，监听器的批处理工件必须是 StepListener 接口的实现。

参见 26.8.1.4 节，其中提供了一个项处理器监听器实现的例子。

- 一个 partition 元素（可选）
  对于可以在多个线程中执行的划分步骤，可以使用这个元素。
- 如果这是作业中的最后一个步骤，则有一个 end 元素。
  这个元素将批处理状态设置为 COMPLETED。
- 一个 stop 元素（可选），在这个步骤停止作业。
  这个元素将批处理状态设置为 STOPPED。
- 一个 fail 元素（可选），在这个步骤终止作业。
  这个元素将批处理状态设置为 FAILED。
- 如果没有指定 next 属性，则有一个或多个 next 元素。
  这个元素与一个退出状态关联，指示另一个步骤、流、分支或判定元素。

下面是一个块步骤的例子：

```
<step id="stepA" next="stepB">
  <properties> ... </properties>
  <listeners>
    <listener ref="MyItemReadListenerImpl"/>
    ...
  </listeners>
  <chunk ...> ... </chunk>
```

```
  <partition> ... </partition>
  <end on="COMPLETED" exit-status="MY_COMPLETED_EXIT_STATUS"/>
  <stop on="MY_TEMP_ISSUE_EXIST_STATUS" restart="step0"/>
  <fail on="MY_ERROR_EXIT_STATUS" exit-status="MY_ERROR_EXIT_STATUS"/>
</step>
```

下面是一个任务步骤的例子：

```
<step id="stepB" next="stepC">
  <batchlet ...> ... </batchlet>
  <properties> ... </properties>
  <listener ref="MyStepListenerImpl"/>
</step>
```

26.4.2.1 chunk 元素

chunk 元素是面向块步骤 step 元素的子元素。这个元素的属性如表 26-2 所示。

表 26-2 chunk 元素的属性

属性名	描述	默认值
checkpoint-policy	指定如何提交处理各个块的结果： ● "item"：块在处理 item-count 项后提交 ● "custom"：根据用 checkpoint-algorithm 元素指定的一个检查点算法提交块 提交块结果时会更新检查点。 每个块都在一个全局 Java EE 事务中处理。如果块中某一项的处理失败，这个事务就会回滚，这个块已处理的所有项都不会存储	"item"
item-count	指定提交块并建立检查点之前要处理的项数	10
time-limit	指定 checkpoint-policy="item" 时提交块并建立检查点之前的秒数。 如果 time-limit 秒前还没有处理 item-count 项，会提交块并建立检查点	0（无限制）
buffer-items	指定所处理的项是否缓存（直到建立检查点）。如果为 true，会在提交块并建立检查点之前对所缓存的项调用项书写器	true
skip-limit	指定块处理期间这个步骤中跳过的可跳过异常数。可跳过异常类用 skippable-exception-classes 元素指定	无限制
retry-limit	指定出现可重试的异常时执行这个步骤的重试次数。可重试的异常类用 retryable-exception-classes 元素指定	无限制

chunk 元素可以包含以下元素：

- 一个 reader 元素。

  这个元素指定一个实现 ItemReader 接口的批处理工件。
- 一个 processor 元素。

  这个元素指定一个实现 ItemProcessor 接口的批处理工件。
- 一个 writer 元素。

  这个元素指定一个实现 ItemWriter 接口的批处理工件。

- 一个 checkpoint-algorithm 元素（可选）。

  这个元素指定一个实现 CheckpointAlgorithm 接口的批处理工件，并提供一个定制检查点策略。

- 一个 skippable-exception-classes 元素（可选）。

  这个元素指定从阅读器、书写器和处理器批处理工件抛出的一组异常，块处理要跳过这些异常。chunk 元素的 skip-limit 属性指定了最大可跳过异常数。

- 一个 retryable-exception-classes 元素（可选）。

  这个元素指定从阅读器、书写器和处理器批处理工件抛出的一组异常，对于这些异常，块处理要重试。chunk 元素的 retry-limit 属性指定了最大重试次数。

- 一个 no-rollback-exception-classes 元素（可选）。

  这个元素指定从阅读器、书写器和处理器批处理工件抛出的一组异常，这些异常应当不会导致批处理运行时环境回滚当前块，而是重试当前操作（不回滚）。

  默认地，对于未在这个元素中指定的异常类型，出现相应异常时当前块会回滚。

下面是一个面向块的步骤的例子：

```
<step id="stepC" next="stepD">
  <chunk checkpoint-policy="item" item-count="5" time-limit="180"
         buffer-items="true" skip-limit="10" retry-limit="3">
    <reader ref="pkg.MyItemReaderImpl"></reader>
    <processor ref="pkg.MyItemProcessorImpl"></processor>
    <writer ref="pkg.MyItemWriterImpl"></writer>
    <skippable-exception-classes>
      <include class="pkg.MyItemException"/>
      <exclude class="pkg.MyItemSeriousSubException"/>
    </skippable-exception-classes>
    <retryable-exception-classes>
      <include class="pkg.MyResourceTempUnavailable"/>
    </retryable-exception-classes>
  </chunk>
</step>
```

这个例子定义了一个块步骤，并指定了它的阅读器、处理器和书写器工件。这个步骤更新了一个检查点，并在处理 5 个数据项之后提交各个块。它会跳过 MyItemException 异常及其所有子类型，但 MyItemSeriousSubException 除外，最多跳过 10 个异常。出现 MyResourceTempUnavailable 异常时，这个步骤会重试一个块，最多重试 3 次。

26.4.2.2 batchlet 元素

batchlet 元素是面向任务步骤 step 元素的一个子元素。这个元素只有 ref 属性，它指定一个实现 Batchlet 接口的批处理工件。batch 元素可以包含一个 properties 元素。

下面是一个面向任务的步骤的例子：

```
<step id="stepD" next="stepE">
  <batchlet ref="pkg.MyBatchletImpl">
```

```
    <properties>
      <property name="pname" value="pvalue"/>
    </properties>
  </batchlet>
</step>
```

这个例子定义了一个批处理步骤，并指定了它的批处理工件。

26.4.2.3 partition 元素

partition 元素是 step 元素的一个子元素，指示一个步骤是划分步骤。大多数划分步骤都是块步骤，其中各个项的处理不依赖于之前项的处理结果。要指定步骤中的划分个数，为各个划分提供具体处理信息（要处理哪些项），如下所示：

- 项的范围。例如，划分 1 处理第 1 项到第 500 项，划分 2 处理第 501 项到第 1000 项。
- 输入源。例如，划分 1 处理 input1.txt 中的项，划分 2 处理 input2.txt 中的项。

开发时或部署时如果知道一个划分步骤的划分数、项数和输入源，可以使用作业定义文件中的划分属性来指定划分的特定信息，并从步骤批处理工件访问这些属性。运行时环境会根据划分数创建相同数目的步骤批处理工件实例（阅读器、处理器和书写器），每个工件实例会得到其划分的特定属性。

大多数情况下，一个划分步骤的划分数、项数或输入源只能在运行时确定。并不是在作业定义文件中静态地指定划分的特定属性，而是可以提供一个批处理工件在运行时访问数据源，并确定需要多少个划分以及每个划分处理的项范围。这个批处理工件是 PartitionMapper 接口的一个实例。批处理运行时环境调用这个工件，然后使用它提供的信息为各个划分完成步骤批处理工件（阅读器、处理器和书写器）的实例化，并作为参数向这些步骤批处理工件传递划分特定的数据。

这一节后面会详细地介绍 partition 元素，并提供作业定义文件的两个例子：一个使用划分属性为每个划分指定一个项范围，另一个依赖于 PartitionMapper 实现来确定划分的特定信息。

参见 26.9.1.4 节，其中提供了一个划分块步骤的完整例子。

partition 元素可以包含以下元素。

- 一个 plan 元素（如果未指定 mapper 元素）。

  这个元素为作业定义文件中的各个划分定义划分数、线程数和属性。如果可以在开发时或部署时知道这些信息，plan 元素会很有用。

- 一个 mapper 元素（如果未指定 plan 元素）。

  这个元素指定一个批处理工件为各个划分提供划分数、线程数和属性。这个批处理工件是 PartitionMapper 接口的一个实现。如果只有在运行时才能知道每个划分所需的信息，就要使用这个选项。

- 一个 reducer 元素（可选）。

  这个元素指定一个批处理工件，划分步骤开始、结束或回滚时将由这个工件控

制。利用这个批处理工件，可以合并不同划分的结果，并完成其他相关的操作。这个批处理工件是 PartitionReducer 接口的一个实现。

- 一个 collector 元素（可选）。

  这个元素指定一个批处理工件将各个划分的中间结果发送到一个划分分析器。在块步骤的各个检查点之后以及任务步骤的步骤结束时，这个批处理工件会发送中间结果。这个批处理工件是 PartitionCollector 接口的一个实现。

- 一个 analyzer 元素（可选）。

  这个元素指定一个批处理工件分析从划分收集器实例得到的中间结果。这个批处理工件是 PartitionAnalyzer 接口的一个实现。

下面是使用 plan 元素的一个划分步骤的例子：

```
<step id="stepE" next="stepF">
  <chunk>
    <reader ...></reader>
    <processor ...></processor>
    <writer ...></writer>
  </chunk>
  <partition>
    <plan partitions="2" threads="2">
      <properties partition="0">
        <property name="firstItem" value="0"/>
        <property name="lastItem" value="500"/>
      </properties>
      <properties partition="1">
        <property name="firstItem" value="501"/>
        <property name="lastItem" value="999"/>
      </properties>
    </plan>
  </partition>
  <reducer ref="MyPartitionReducerImpl"/>
  <collector ref="MyPartitionCollectorImpl"/>
  <analyzer ref="MyPartitionAnalyzerImpl"/>
</step>
```

在这个例子中，plan 元素为作业定义文件中的各个划分指定了属性。

下面的例子使用了一个 mapper 元素而不是 plan 元素。这个 PartitionMapper 实现会动态提供作业定义文件中 plan 元素所提供的相同信息：

```
<step id="stepE" next="stepF">
  <chunk>
    <reader ...></reader>
    <processor ...></processor>
    <writer ...></writer>
  </chunk>
  <partition>
    <mapper ref="MyPartitionMapperImpl"/>
    <reducer ref="MyPartitionReducerImpl"/>
```

```
    <collector ref="MyPartitionCollectorImpl"/>
    <analyzer ref="MyPartitionAnalyzerImpl"/>
  </partition>
</step>
```

参见 26.9 节，其中给出了 PartitionMapper 接口的一个示例实现。

26.4.3 flow 元素

flow 元素可以是 job、flow 和 split 元素的子元素，其属性为 id 和 next。流可以转移到流、步骤、分支和判定元素。flow 元素可以包含以下元素：

- 一个或多个 step 元素
- 一个或多个 flow 元素（可选）
- 一个或多个 split 元素（可选）
- 一个或多个 decision 元素（可选）

流中最后一个 step 元素没有 next 属性或 next 元素。流中的步骤和其他元素不能转移到流外的元素。

下面是 flow 元素的一个例子：

```
<flow id="flowA" next="stepE">
  <step id="flowAstepA" next="flowAstepB">...</step>
  <step id="flowAstepB" next="flowAflowC">...</step>
  <flow id="flowAflowC" next="flowAsplitD">...</flow>
  <split id="flowAsplitD" next="flowAstepE">...</split>
  <step id="flowAstepE">...</step>
</flow>
```

这个示例流包含 3 个步骤、一个流和一个分支。最后一个步骤没有 next 属性。流的最后一个步骤完成时会转移到 stepE。

26.4.4 split 元素

split 元素可以是 job 和 flow 元素的子元素，其属性为 id 和 next。分支可以转移到分支、步骤、流和判定元素。split 元素只能包含一个或多个 flow 元素，这些元素只能转移到分支中的其他 flow 元素。

下面是一个分支的例子，其中包含 3 个并发执行的流：

```
<split id="splitA" next="stepB">
  <flow id="splitAflowA">...</flow>
  <flow id="splitAflowB">...</flow>
  <flow id="splitAflowC">...</flow>
</split>
```

26.4.5 decision 元素

decision 元素可以是 job 和 flow 元素的子元素，其属性为 id 和 next。步骤、流和

分支可以转移到 decision 元素。这个元素指定一个批处理工件，它根据上一个步骤、流或分支执行的信息来确定下一个步骤、流或分支。这个批处理工件实现了 Decider 接口。decision 元素可以包含以下元素：

- 一个或多个 end 元素（可选）。
  这个元素将批处理状态设置为 COMPLETED。
- 一个或多个 stop 元素（可选）。
  这个元素将批处理状态设置为 STOPPED。
- 一个或多个 fail 元素（可选）。
  这个元素将批处理状态设置为 FAILED。
- 一个或多个 next 元素（可选）。
- 一个 properties 元素（可选）。

下面是 decider 元素的一个例子：

```
<decision id="decisionA" ref="MyDeciderImpl">
  <fail on="FAILED" exit-status="FAILED_AT_DECIDER"/>
  <end on="COMPLETED" exit-status="COMPLETED_AT_DECIDER"/>
  <stop on="MY_TEMP_ISSUE_EXIST_STATUS" restart="step2"/>
</decision>
```

26.5 创建批处理工件

使用作业规范语言（JSL）用批处理工件定义一个作业后，将工件创建为 Java 类，这些类要实现 javax.batch.api 包及其子包中的接口。

这一节将列出主要的批处理工件接口，介绍如何从批处理运行时环境访问上下文对象，另外还会提供一些例子。

26.5.1 批处理工件接口

下表列出了创建批处理工件要实现的接口。26.4 节介绍的元素要引用这些接口实现。

表 26-3 列出为块步骤、任务步骤和判定元素实现批处理工件的接口。

表 26-3 主要批处理工件接口

包	接 口	描 述
javax.batch.api	Batchlet	实现一个面向任务步骤的业务逻辑。由 batchlet 元素引用
javax.batch.api	Decider	根据上一个步骤、流或分支执行的信息来确定下一个要执行的步骤、流或分支。由 decision 元素引用
javax.batch.api.chunk	CheckPointAlgorithm	为块步骤实现一个定制检查点策略。由 chunk 元素中的 checkpoint-algorithm 元素引用

（续）

包	接 口	描 述
javax.batch.api.chunk	ItemReader	在块步骤中从一个输入源读取项。由 chunk 元素中的 reader 元素引用
javax.batch.api.chunk	ItemProcessor	在块步骤中处理输入项来得到输出项。由 chunk 元素中的 processor 元素引用
javax.batch.api.chunk	ItemWriter	块步骤中写出输出项。由 chunk 元素中的 writer 元素引用

表 26-4 列出为划分步骤实现批处理工件的接口。

表 26-4 划分批处理工件接口

包	接 口	描 述
javax.batch.api.partition	PartitionPlan	提供有关如何执行一个划分步骤的详细信息，如划分数、线程数和各个划分的参数。这个工件不在作业定义文件中直接引用
javax.batch.api.partition	PartitionMapper	提供一个 PartitionPlan 对象。由 partition 元素中的 mapper 元素引用
javax.batch.api.partition	PartitionReducer	划分步骤开始、结束或回滚时接收控制。由 partition 元素中的 reducer 元素引用
javax.batch.api.partition	PartitionCollector	将各个划分的中间结果发送到一个划分分析器。由 partition 元素中的 collector 元素引用
javax.batch.api.partition	PartitionAnalyzer	处理各个划分的数据和最终结果。由 partition 元素中的 analyzer 元素引用

表 26-5 列出为作业和步骤监听器实现批处理工件的接口。

表 26-5 监听器批处理工件接口

包	接 口	描 述
javax.batch.api.listener	JobListener	运行一个作业之前和之后拦截作业执行。由 job 元素中的 listener 元素引用
javax.batch.api.listener	StepListener	运行一个步骤之前和之后拦截步骤执行。由 step 元素中的 listener 元素引用
javax.batch.api.chunk.listener	ChunkListener	处理各个块之前和之后以及出现错误时，拦截块步骤中的块处理。由 step 元素中的 listener 元素引用
javax.batch.api.chunk.listener	ItemReadListener	读取各个项之前和之后以及出现错误时，拦截块步骤中的项读取。由 step 元素中的 listener 元素引用
javax.batch.api.chunk.listener	ItemProcessListener	处理各个项之前和之后以及出现错误时，拦截块步骤中的项处理。由 step 元素中的 listener 元素引用

（续）

包	接 口	描 述
javax.batch.api.chunk.listener	ItemWriteListener	写出各个项之前和之后以及出现错误时，拦截块步骤中的项写出。由 step 元素中的 listener 元素引用
javax.batch.api.chunk.listener	RetryReadListener	出现异常时拦截块步骤中的重试项读取。由 step 元素中的 listener 元素引用
javax.batch.api.chunk.listener	RetryProcessListener	出现异常时拦截块步骤中的重试项处理。由 step 元素中的 listener 元素引用
javax.batch.api.chunk.listener	RetryWriteListener	出现异常时拦截块步骤中的重试项写出。由 step 元素中的 listener 元素引用
javax.batch.api.chunk.listener	SkipReadListener	拦截块步骤中项阅读器的可跳过异常处理。由 step 元素中的 listener 元素引用
javax.batch.api.chunk.listener	SkipProcessListener	拦截块步骤中项处理器的可跳过异常处理。由 step 元素中的 listener 元素引用
javax.batch.api.chunk.listener	SkipWriteListener	拦截块步骤中项书写器的可跳过异常处理。由 step 元素中的 listener 元素引用

26.5.2 批处理工件中的依赖注入

为了确保批处理工件中上下文和依赖注入（CDI）能正常工作，要完成以下步骤：

1. 使用 Named 注解将批处理工件实现定义为 CDI 命名 bean。

例如，在一个块步骤中定义一个项阅读器实现，如下所示：

```
@Named("MyItemReaderImpl")
public class MyItemReaderImpl implements ItemReader {
    /* ... Override the ItemReader interface methods ... */
}
```

2. 为批处理工件提供一个空的公共无参数构造函数。

例如，为以上工件提供下面的构造函数：

```
public MyItemReaderImpl() {}
```

3. 为作业定义文件中的批处理工件提供 CDI 名，而不是使用类的完全限定名。

例如，为上面的工件定义步骤，如下所示：

```
<step id="stepA" next="stepB">
  <chunk>
    <reader ref="MyItemReaderImpl"></reader>
    ...
  </chunk>
</step>
```

这个例子使用 CDI 名（MyItemReaderImpl）而不是类的完全限定名（com.example.pkg.MyItemReaderImpl）来指定一个批处理工件。

4. 确保模块是一个 CDI bean 归档，为此要为批处理工件增加 javax.enterprise.context.Dependent 注解，或者为应用包含一个空的 beans.xml 部署描述文件。例如，下面的批处理工件增加了 @Dependent 注解：

```
@Dependent
@Named("MyItemReaderImpl")
public class MyItemReaderImpl implements ItemReader { ... }
```

关于 bean 归档的更多信息，参见本书第 1 卷中 25.1 节。

说明 批处理工件中要从批处理运行时环境访问上下文对象，必须使用上下文和依赖注入（CDI）。

如果没有遵循这个过程，可能会遇到以下错误：

- 批处理运行时环境无法找到某些批处理工件。
- 访问注入的对象时，批处理工件会抛出 null 指针异常。

26.5.3 从批处理运行时环境使用上下文对象

批处理运行时环境提供了上下文对象，这些对象实现了 javax.batch.runtime.context 包中的 JobContext 和 StepContext 接口。这些对象分别与当前作业和步骤关联，允许完成以下工作：

- 从当前作业或步骤得到信息，如它的名字、实例 ID、执行 ID、批处理状态和退出状态。
- 设置用户定义的退出状态。
- 存储用户数据。
- 从作业或步骤定义获得属性值。

在批处理工件实现中可以从批处理运行时环境注入上下文对象，如项阅读器、项处理器、项书写器、任务步骤（batchlet）、监听器等。下面的例子展示了如何在一个项阅读器实现中从作业定义文件访问属性值：

```
@Dependent
@Named("MyItemReaderImpl")
public class MyItemReaderImpl implements ItemReader {
    @Inject
    JobContext jobCtx;

    public MyItemReaderImpl() {}

    @Override
    public void open(Serializable checkpoint) throws Exception {
        String fileName = jobCtx.getProperties()
                                .getProperty("log_file_name");
```

```
        ...
    }
    ...
}
```

要了解如何定义批处理工件来使用依赖注入，可以参见26.5.2节。

说明 不要在工件构造函数中访问批处理上下文对象。因为在提交到批处理运行时环境之前，作业并不会真正运行。加载应用时，CDI要实例化工件，此时批处理上下文对象还不可用。这些bean的实例化会失败，应用提交作业时，批处理运行时环境将无法找到你的批处理工件。

26.6 向批处理运行时环境提交作业

利用javax.batch.operations包中的JobOperator接口，可以向批处理运行时环境提交作业，并得到现有作业的信息。这个接口提供了以下功能：

- 得到所有已知作业的作业名。
- 开始、停止、重启和放弃作业。
- 获得作业实例和作业执行。

javax.batch.runtime包中的BatchRuntime类提供了getJobOperator工厂方法来得到JobOperator对象。

26.6.1 开始作业

下面的示例代码展示了如何得到一个JobOperator对象并提交一个批处理作业：

```
JobOperator jobOperator = BatchRuntime.getJobOperator();
Properties props = new Properties();
props.setProperty("parameter1", "value1");
...
long execID = jobOperator.start("simplejob", props);
```

JobOperator.start方法的第一个参数是作业定义文件中指定的作业名。第二个参数是一个Properties对象，表示这个作业执行的参数。可以使用作业参数传递运行时才知道的作业信息。

26.6.2 检查作业的状态

javax.batch.runtime包中的JobExecution接口提供了一些方法，可以获得所提交作业的有关信息。这个接口提供了以下功能：

- 获得一个作业执行的批处理状态和退出状态。

- 获得作业执行开始、更新或结束的时间。
- 获得作业名。
- 获得执行 ID。

下面的示例代码展示了如何使用一个作业的执行 ID 得到它的批处理状态：

```
JobExecution jobExec = jobOperator.getJobExecution(execID);
String status = jobExec.getBatchStatus().toString();
```

26.6.3 在应用中调用批处理运行时环境

要在哪一个组件中调用批处理运行时环境，这取决于具体应用的架构。例如，可以从一个企业 bean、servlet、托管 bean 等调用批处理运行时环境。

参见 26.8 节和 26.9 节来详细了解如何从一个 JSF 用户界面驱动的托管 bean 调用批处理运行时环境。

26.7 打包批处理应用

作业定义文件和批处理工件不需要单独打包，可以包含在任意 Java EE 应用中。

将批处理工件类与应用中其余的类一同打包，并将作业定义文件包含在以下目录中：

- 对于 jar 包，包含在 META-INF/batch-jobs/ 目录中
- 对于 war 包，包含在 WEB-INF/classes/META-INF/batch-jobs/ 目录中

各个作业定义文件的名必须与作业 ID 匹配。例如，如果如下定义一个作业，并将应用打包为一个 WAR 文件，要将作业定义文件包含在 WEB-INF/classes/META-INF/batch-jobs/simplejob.xml：

```
<job id="simplejob" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
                    version="1.0">
  ...
</job>
```

26.8 webserverlog 示例应用

Webserverlog 示例应用位于 *tut-install*/examples/batch/webserverlog/ 目录，展示了如何在 Java EE 中使用批处理框架来分析 Web 服务器的日志文件。这个示例应用读取一个日志文件，分析用户从平板设备查看的页面中商品销售页面占多大比例。

26.8.1 webserverlog 示例应用架构

Webserverlog 示例应用包含以下元素：

- 一个作业定义文件（webserverlog.xml），使用作业规范语言（JSL）定义一个批处理作业，其中包含一个块步骤和一个任务步骤。这个块步骤相当于一个过滤器，任务步骤对其余的日志记录计算统计结果。
- 一个日志文件（log1.txt），这个文件将作为这个批处理作业的输入数据。
- 两个 Java 类（LogLine 和 LogFilteredLine），表示块步骤的输入项和输出项。
- 实现应用块步骤的 3 个批处理工件（LogLineReader、LogLineProcessor 和 LogFilteredLineWriter）。这个步骤从 Web 服务器日志文件读取项，根据客户端使用的 Web 浏览器过滤这些项，并把结果写入一个文本文件。
- 两个批处理工件（InfoJobListener 和 InfoItemProcessListener），实现两个简单的监听器。
- 一个批处理工件（MobileBatchlet.java），对过滤得到的项计算统计结果。
- 两个 Facelets 页面（index.xhtml 和 jobstarted.xhtml），提供这个批处理应用的前端。第一个页面显示将由批处理作业处理的日志文件，第二个页面允许用户检查作业的状态并显示结果。
- Facelets 页面访问的一个托管 bean（JsfBean）。这个 bean 向批处理运行时环境提交作业，检查作业的状态，并从一个文本文件读取结果。

26.8.1.1 作业定义文件

webserverlog.xml 作业定义文件位于 WEB-INF/classes/META-INF/batch-jobs/ 目录。这个文件指定了 7 个作业级属性和两个步骤：

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="webserverlog" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
     version="1.0">
    <properties>
        <property name="log_file_name" value="log1.txt"/>
        <property name="filtered_file_name" value="filtered1.txt"/>
        <property name="num_browsers" value="2"/>
        <property name="browser_1" value="Tablet Browser D"/>
        <property name="browser_2" value="Tablet Browser E"/>
        <property name="buy_page" value="/auth/buy.html"/>
        <property name="out_file_name" value="result1.txt"/>
    </properties>
    <listeners>
        <listener ref="InfoJobListener"/>
    </listeners>
    <step id="mobilefilter" next="mobileanalyzer"> ... </step>
    <step id="mobileanalyzer"> ... </step>
</job>
```

第一个步骤定义如下：

```
<step id="mobilefilter" next="mobileanalyzer">
    <listeners>
```

```
        <listener ref="InfoItemProcessListeners"/>
    </listeners>
    <chunk checkpoint-policy="item" item-count="10">
        <reader ref="LogLineReader"></reader>
        <processor ref="LogLineProcessor"></processor>
        <writer ref="LogFilteredLineWriter"></writer>
    </chunk>
</step>
```

这个步骤是一个正常的块步骤，指定了实现步骤各个阶段的批处理工件。批处理工件名不是完全限定类名，所以这些批处理工件是有 @Named 注解的 CDI bean。

第二个步骤定义如下：

```
<step id="mobileanalyzer">
    <batchlet ref="MobileBatchlet"></batchlet>
    <end on="COMPLETED"/>
</step>
```

这个步骤是一个任务步骤，指定了实现这个任务的批处理工件。这是这个作业的最后一个步骤。

26.8.1.2 LogLine 和 LogFilteredLine 项

LogLine 类表示 Web 服务器日志文件中的日志记录，定义如下：

```
public class LogLine {
    private final String datetime;
    private final String ipaddr;
    private final String browser;
    private final String url;

    /* ... Constructor, getters, and setters ... */
}
```

LogFileteredLine 类与 LogLine 类很类似，不过只有两个字段：客户端的 IP 地址和 URL。

26.8.1.3 块步骤批处理工件

第一个步骤由 LogLineReader、LogLineProcessor 和 LogFilteredLineWriter 批处理工件组成。

LogLineReader 工件从 Web 服务器日志文件读取记录：

```
@Dependent
@Named("LogLineReader")
public class LogLineReader implements ItemReader {
    private ItemNumberCheckpoint checkpoint;
    private String fileName;
    private BufferedReader breader;
    @Inject
    private JobContext jobCtx;
```

```
    public LogLineReader() { }
    /* ... Override the open, close, readItem, and
     *     checkpointInfo methods ... */
}
```

open 方法读取 log_file_name 属性，并用一个缓冲阅读器打开日志文件。在这个例子中，日志文件包含在应用的 webserverlog/WEB-INF/classes/log1.txt 中：

```
fileName = jobCtx.getProperties().getProperty("log_file_name");
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
InputStream iStream = classLoader.getResourceAsStream(fileName);
breader = new BufferedReader(new InputStreamReader(iStream));
```

如果提供了一个检查点对象，open 方法会让阅读器前进到上一个检查点。否则，这个方法将创建一个新的检查点对象。检查点对象会跟踪上一次提交的块的行号。

readItem 方法返回一个新 LogLine 对象，或者当到达日志文件末尾时返回 null：

```
@Override
public Object readItem() throws Exception {
    String entry = breader.readLine();
    if (entry != null) {
        checkpoint.nextLine();
        return new LogLine(entry);
    } else {
        return null;
    }
}
```

LogLineProcessor 工件从作业属性得到一个浏览器列表，并根据这个列表过滤日志记录：

```
@Override
public Object processItem(Object item) {
    /* Obtain a list of browsers we are interested in */
    if (nbrowsers == 0) {
        Properties props = jobCtx.getProperties();
        nbrowsers = Integer.parseInt(props.getProperty("num_browsers"));
        browsers = new String[nbrowsers];
        for (int i = 1; i < nbrowsers + 1; i++)
            browsers[i - 1] = props.getProperty("browser_" + i);
    }

    LogLine logline = (LogLine) item;
    /* Filter for only the mobile/tablet browsers as specified */
    for (int i = 0; i < nbrowsers; i++) {
        if (logline.getBrowser().equals(browsers[i])) {
            return new LogFilteredLine(logline);
        }
    }
    return null;
}
```

LogFilteredLineWriter 工件从作业属性读取输出文件名。open 方法打开这个文件来完成写操作。如果提供了一个检查点对象，这个工件会继续在文件末尾写；否则，它会覆盖原有的文件。writeItems 方法将过滤得到的项写入输出文件：

```
@Override
public void writeItems(List<Object> items) throws Exception {
    /* Write the filtered lines to the output file */
    for (int i = 0; i < items.size(); i++) {
        LogFilteredLine filtLine = (LogFilteredLine) items.get(i);
        bwriter.write(filtLine.toString());
        bwriter.newLine();
    }
}
```

26.8.1.4 监听器批处理工件

InfoJobListener 批处理工件实现了一个简单的监听器，作业开始和结束时会写入日志消息：

```
@Dependent
@Named("InfoJobListener")
public class InfoJobListener implements JobListener {
    ...
    @Override
    public void beforeJob() throws Exception {
        logger.log(Level.INFO, "The job is starting");
    }

    @Override
    public void afterJob() throws Exception { ... }
}
```

InfoItemProcessListener 批处理工件实现了 ItemProcessListener 接口来完成块步骤：

```
@Dependent
@Named("InfoItemProcessListener")
public class InfoItemProcessListener implements ItemProcessListener {
    ...
    @Override
    public void beforeProcess(Object o) throws Exception {
        LogLine logline = (LogLine) o;
        logger.log(Level.INFO, "Processing entry {0}", logline);
    }
    ...
}
```

26.8.1.5 任务步骤批处理工件

任务步骤由 MobileBatchlet 工件实现，它会计算过滤的日志记录中购买商品的记录占多大比例：

```
@Override
public String process() throws Exception {
    /* Get properties from the job definition file */
    ...
    /* Count from the output of the previous chunk step */
    breader = new BufferedReader(new FileReader(fileName));
    String line = breader.readLine();
    while (line != null) {
        String[] lineSplit = line.split(", ");
        if (buyPage.compareTo(lineSplit[1]) == 0)
            pageVisits++;
        totalVisits++;
        line = breader.readLine();
    }
    breader.close();
    /* Write the result */
    ...
}
```

26.8.1.6 JSF 页面

index.xhtml 页面包含一个文本域，将显示 Web 服务器日志。这个页面提供了一个按钮，允许用户提交批处理作业，并导航到下一个页面：

```
<body>
    ...
    <textarea cols="90" rows="25"
              readonly="true">#{jsfBean.getInputLog()}</textarea>
    <p> </p>
    <h:form>
        <h:commandButton value="Start Batch Job"
                         action="#{jsfBean.startBatchJob()}" />
    </h:form>
</body>
```

这个页面调用托管 bean 的方法来显示日志文件，并提交批处理作业。

jobstarted.xhtml 页面提供了一个按钮，可以检查批处理作业的当前状态，并在作业完成时显示结果：

```
<p>Current Status of the Job: <b>#{jsfBean.jobStatus}</b></p>
<p>#{jsfBean.showResults()}</p>
<h:form>
    <h:commandButton value="Check Status"
                     action="jobstarted"
                     rendered="#{jsfBean.completed==false}" />
</h:form>
```

26.8.1.7 托管 bean

JsfBean 托管 bean 向批处理运行时环境提交作业，检查作业的状态，并从一个文本文件读取结果。

startBatchJob 方法向批处理运行时环境提交作业：

```
/* Submit the batch job to the batch runtime.
 * JSF Navigation method (return the name of the next page) */
public String startBatchJob() {
    jobOperator = BatchRuntime.getJobOperator();
    execID = jobOperator.start("webserverlog", null);
    return "jobstarted";
}
```

getJobStatus 方法检查作业的状态：

```
/* Get the status of the job from the batch runtime */
public String getJobStatus() {
    return jobOperator.getJobExecution(execID).getBatchStatus().toString();
}
```

showResults 方法从一个文本文件读取结果。

26.8.2 运行 webserverlog 示例应用

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 webserverlog 示例应用。

26.8.2.1 使用 NetBeans IDE 运行 webserverlog 示例应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

tut-install/examples/batch

4. 选择 webserverlog 文件夹。
5. 单击 Open Project。
6. 在 Projects（工程）标签页中，右键单击 webserverlog 工程并选择 Run（运行）。

这个命令会构建应用并将其打包到一个 WAR 文件 webserverlog.war，这个文件位于 target/ 目录；将它部署到服务器，并打开一个 Web 浏览器窗口访问以下 URL：

http://localhost:8080/webserverlog/

26.8.2.2 使用 Maven 运行 webserverlog 示例应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 在一个终端窗口中导航至：

tut-install/examples/batch/webserverlog/

3. 输入以下命令部署这个应用：

```
mvn install
```

4. 打开一个 Web 浏览器窗口访问以下 URL：

http://localhost:8080/webserverlog/

26.9 phonebilling 示例应用

Phonebilling 示例应用位于 *tut-install*/examples/batch/phonebilling/ 目录，展示了如何使用 Java EE 中的批处理框架来实现一个电话计费系统。这个示例应用会处理一个电话拨号日志文件，并为每个顾客创建一个账单。

26.9.1 phonebilling 示例应用架构

Phonebilling 示例应用由以下元素组成：

- 一个作业定义文件（phonebilling.xml），使用作业规范语言（JSL）来定义一个批处理作业，其中包括两个块步骤。第一个步骤从一个日志文件读取拨号记录，并将它们与一个账单关联。第二个步骤计算费用，把各个账单写至一个文本文件。
- 一个 Java 类（CallRecordLogCreator），为这个批处理作业创建日志文件。这是一个辅助组件，没有展示这个示例的关键功能。
- 两个 Java 持久化 API（JPA）实体（CallRecord 和 PhoneBill），表示拨号记录和顾客账单。应用使用一个 JPA 实体管理器将这些实体的实例存储在一个数据库中。
- 实现应用的第一个步骤的 3 个批处理工件（CallRecordReader、CallRecordProcessor 和 CallRecordWriter）。这个步骤从日志文件读取拨号记录，将它们与一个账单关联，并存储在一个数据库中。
- 实现应用的第二个步骤的 4 个批处理工件（BillReader、BillProcessor、BillWriter 和 BillPartitionMapper）。这个步骤是一个划分步骤，从数据库得到各个账单，计算费用，并写至一个文本文件。
- 两个 Facelets 页面（index.xhtml 和 jobstarted.xhtml），提供这个批处理应用的前端。第一个页面显示将由批处理作业处理的日志文件，第二个页面允许用户检查作业的状态，并显示各个顾客的最后账单。
- 从 Facelets 页面访问的一个托管 bean（JsfBean）。这个 bean 向批处理运行时环境提交作业，检查作业的状态，并读取各个账单的文本文件。

26.9.1.1 作业定义文件

phonebilling.xml 作业定义文件位于 WEB-INF/classes/META-INF/batch-jobs/ 目录。这个文件指定了 3 个作业级属性和两个步骤：

```
<job id="phonebilling" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
     version="1.0">
    <properties>
        <property name="log_file_name" value="log1.txt"/>
        <property name="airtime_price" value="0.08"/>
        <property name="tax_rate" value="0.07"/>
    </properties>
    <step id="callrecords" next="bills"> ... </step>
```

```
    <step id="bills"> ... </step>
</job>
```

第一个步骤定义如下：

```
<step id="callrecords" next="bills">
    <chunk checkpoint-policy="item" item-count="10">
        <reader ref="CallRecordReader"></reader>
        <processor ref="CallRecordProcessor"></processor>
        <writer ref="CallRecordWriter"></writer>
    </chunk>
</step>
```

这个步骤是一个普通的块步骤，指定了实现步骤各个阶段的批处理工件。批处理工件名不是完全限定类名，所以这些批处理工件是有 @Named 注解的 CDI bean。

第二个步骤定义如下：

```
<step id="bills">
    <chunk checkpoint-policy="item" item-count="2">
        <reader ref="BillReader"></reader>
        <processor ref="BillProcessor"></processor>
        <writer ref="BillWriter"></writer>
    </chunk>
    <partition>
        <mapper ref="BillPartitionMapper"/>
    </partition>
    <end on="COMPLETED"/>
</step>
```

这个步骤是一个划分块步骤。划分计划通过 BillPartitionMapper 工件指定而不是使用 plan 元素来指定。

26.9.1.2 CallRecord 和 PhoneBill 实体

CallRecord 实体定义如下：

```
@Entity
public class CallRecord implements Serializable {
    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.DATE)
    private Date datetime;
    private String fromNumber;
    private String toNumber;
    private int minutes;
    private int seconds;
    private BigDecimal price;

    public CallRecord() { }
    public CallRecord(String datetime, String from,
            String to, int min, int sec)
            throws ParseException { ... }
```

```
    public CallRecord(String jsonData) throws ParseException { ... }

    /* ... Getters and setters ... */
}
```

id 字段由 JPA 实现自动生成，用于在数据库中存储和从数据库获取 CallRecord 对象。

第二个构造函数使用 JSON 处理 API 从日志文件中的 JSON 数据记录创建一个 CallRecord 对象。日志记录如下所示：

```
{"datetime":"03/01/2013 04:03","from":"555-0101",
"to":"555-0114","length":"03:39"}
```

PhoneBill 实体定义如下：

```
@Entity
public class PhoneBill implements Serializable {
    @Id
    private String phoneNumber;
    @OneToMany(cascade = CascadeType.PERSIST)
    @OrderBy("datetime ASC")
    private List<CallRecord> calls;
    private BigDecimal amountBase;
    private BigDecimal taxRate;
    private BigDecimal tax;
    private BigDecimal amountTotal;

    public PhoneBill() { }

    public PhoneBill(String number) {
        this.phoneNumber = number;
        calls = new ArrayList<>();
    }

    public void addCall(CallRecord call) {
        calls.add(call);
    }
    public void calculate(BigDecimal taxRate) { ... }

    /* ... Getters and setters ... *
}
```

OneToMany 注解定义了一个账单和它的拨号记录之间的关系。CascadeType.PERSIST 参数指示持久存储电话账单时应当自动地持久存储拨号列表中的元素。OrderBy 注解定义了从数据库获取拨号列表元素的顺序。

批处理工件使用这两个实体的实例作为读取、处理和写出的项。

关于 Java 持久化 API 的更多信息，参见第 8 章。关于 JSON 处理 API 的更多信息，参见本书第 1 卷第 19 章。

26.9.1.3 拨号记录块步骤

第一个步骤由 CallRecordReader、CallRecordProcessor 和 CallRecordWriter

批处理工件组成。

CallRecordReader 工件从日志文件读取拨号记录：

```
@Dependent
@Named("CallRecordReader")
public class CallRecordReader implements ItemReader {
    private ItemNumberCheckpoint checkpoint;
    private String fileName;
    private BufferedReader breader;
    @Inject
    JobContext jobCtx;

    /* ... Override the open, close, readItem,
     *     and checkpointInfo methods ... */
}
```

open 方法读取 log_filename 属性，并用一个缓冲阅读器打开日志文件：

```
fileName = jobCtx.getProperties().getProperty("log_file_name");
breader = new BufferedReader(new FileReader(fileName));
```

如果提供了一个检查点对象，open 方法会让阅读器前进到上一个检查点。否则，这个方法会创建一个新的检查点对象。检查点对象会跟踪上一次提交的块的行号。

readItem 方法返回一个新的 CallRecord 对象，或者到达日志文件末尾时返回 null：

```
@Override
public Object readItem() throws Exception {
    /* Read a line from the log file and
     * create a CallRecord from JSON */
    String callEntryJson = breader.readLine();
    if (callEntryJson != null) {
        checkpoint.nextItem();
        return new CallRecord(callEntryJson);
    } else
        return null;
}
```

CallRecordProcessor 工件从作业属性获得通话费、计算每次拨号的费用，并返回拨号对象。这个工件只覆盖了 processItem 方法。

CallRecordWriter 工件将各个拨号记录与一个账单关联，并把账单存储在数据库中。这个工件覆盖了 open、close、writeItems 和 checkpointInfo 方法。writeItems 方法如下所示：

```
@Override
public void writeItems(List<Object> callList) throws Exception {

    for (Object callObject : callList) {
        CallRecord call = (CallRecord) callObject;
        PhoneBill bill = em.find(PhoneBill.class, call.getFromNumber());
        if (bill == null) {
```

```
            /* No bill for this customer yet, create one */
            bill = new PhoneBill(call.getFromNumber());
            bill.addCall(call);
            em.persist(bill);
        } else {
            /* Add call to existing bill */
            bill.addCall(call);
        }
    }
}
```

26.9.1.4 电话计费块步骤

第二个步骤由 BillReader、BillProcessor、BillWriter 和 BillPartitionMapper 批处理工件组成。这个步骤从数据库得到电话账单，计算税费和总费用，并把各个账单写到一个文本文件。由于各个账单的处理相互独立，所以这个步骤可以划分，在多个线程中运行。

BillPartitionMapper 工件指定划分数以及每个划分的参数。在这个例子中，参数表示各个划分要处理的项范围。这个工件获得数据库中的账单数来计算这些范围。它提供了一个划分计划对象，覆盖了 PartitionPlan 接口的 getPartitions 和 getPartitionProperties 方法。getPartitions 方法如下所示：

```
@Override
public Properties[] getPartitionProperties() {
    /* Assign an (approximately) equal number of elements
     * to each partition. */
    long totalItems = getBillCount();
    long partItems = (long) totalItems / getPartitions();
    long remItems = totalItems % getPartitions();

    /* Populate a Properties array. Each Properties element
     * in the array corresponds to a partition. */
    Properties[] props = new Properties[getPartitions()];

    for (int i = 0; i < getPartitions(); i++) {
        props[i] = new Properties();
        props[i].put("firstItem", i * partItems);
        /* Last partition gets the remainder elements */
        if (i == getPartitions() - 1) {
            props[i].put("numItems", partItems + remItems);
        } else {
            props[i].put("numItems", partItems);
        }
    }
    return props;
}
```

BillReader 工件获得划分参数，如下所示：

```
@Dependent
@Named("BillReader")
public class BillReader implements ItemReader {
    ...
    @Inject
    JobContext jobCtx;
    private Properties partParams;
    ...
    @Override
    public void open(Serializable ckpt) throws Exception {
        /* Get the parameters for this partition */
        JobOperator jobOperator = BatchRuntime.getJobOperator();
        long execID = jobCtx.getExecutionId();
        partParams = jobOperator.getParameters(execID);

        /* Get the range of items to work on in this partition */
        long firstItem0 = ((Long) partParams.get("firstItem")).longValue();
        long numItems0 = ((Long) partParams.get("numItems")).longValue();
        ...
    }
    ...
}
```

这个工件还得到一个迭代器，从 JPA 实体管理器读取项：

```
/* Obtain an iterator for the bills in this partition */
String query = "SELECT b FROM PhoneBill b ORDER BY b.phoneNumber";
Query q = em.createQuery(query)
        .setFirstResult((int)firstItem).setMaxResults((int)numItems);
iterator = q.getResultList().iterator();
```

BillProcessor 工件迭代处理账单中的拨号记录列表，并计算每个账单的税费和总费用。BillWriter 工件将各个账单写至一个纯文本文件。

26.9.1.5 JSF 页面

index.xhtml 页面包含一个文本域，用来显示拨号记录日志文件。这个页面还为用户提供了一个按钮，可以提交这个批处理作业，并导航到下一个页面：

```
<body>
    <h1>The Phone Billing Example Application</h1>
    <h2>Log file</h2>
    <p>The batch job analyzes the following log file:</p>
    <textarea cols="90" rows="25"
              readonly="true">#{jsfBean.createAndShowLog()}</textarea>
    <p> </p>
    <h:form>
        <h:commandButton value="Start Batch Job"
                         action="#{jsfBean.startBatchJob()}" />
    </h:form>
</body>
```

这个页面调用托管 bean 的方法来显示日志文件，并提交批处理作业。

jobstarted.xhtml 页面提供一个按钮，可以检查批处理作业的当前状态，并且在作业完成时显示账单：

```
<p>Current Status of the Job: <b>#{jsfBean.jobStatus}</b></p>
<h:dataTable var="_row" value="#{jsfBean.rowList}"
             border="1" rendered="#{jsfBean.completed}">
    <!-- ... show results from jsfBean.rowList ... -->
</h:dataTable>
<!-- Render the check status button if the job has not finished -->
<h:form>
    <h:commandButton value="Check Status"
                     rendered="#{jsfBean.completed==false}"
                     action="jobstarted" />
</h:form>
```

26.9.1.6 托管 bean

JsfBean 托管 bean 向批处理运行时环境提交作业，检查作业的状态，并为各个账单读取文本文件。

bean 的 startBatchJob 方法向批处理运行时环境提交作业：

```
/* Submit the batch job to the batch runtime.
 * JSF Navigation method (return the name of the next page) */
public String startBatchJob() {
    jobOperator = BatchRuntime.getJobOperator();
    execID = jobOperator.start("phonebilling", null);
    return "jobstarted";
}
```

bean 的 getJobStatus 方法检查作业的状态：

```
/* Get the status of the job from the batch runtime */
public String getJobStatus() {
    return jobOperator.getJobExecution(execID).getBatchStatus().toString();
}
```

bean 的 getRowList 方法创建一个账单列表，将在 jobstarted.xhtml JSF 页面上用一个表格显示。

26.9.2 运行 phonebilling 示例应用

可以使用 NetBeans IDE 或 Maven 来构建、打包、部署和运行 phonebilling 示例应用。

26.9.2.1 使用 NetBeans IDE 运行 phonebilling 示例应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

tut-install/examples/batch

4. 选择 phonebilling 文件夹。

5. 单击 Open Project。

6. 在 Projects（工程）标签页中，右键单击 phonebilling 工程并选择 Run（运行）。

这个命令会构建应用并打包到一个 WAR 文件 phonebilling.war，这个文件位于 target/ 目录；把它部署到服务器，并打开一个 Web 浏览器窗口访问以下 URL：

http://localhost:8080/phonebilling/

26.9.2.2 使用 Maven 运行 phonebilling 示例应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在一个终端窗口中导航至：

tut-install/examples/batch/phonebilling/

3. 输入以下命令部署这个应用：

```
mvn install
```

4. 打开一个 Web 浏览器窗口访问以下 URL：

http://localhost:8080/phonebilling/

26.10 关于批处理的更多信息

关于 Java EE 中批处理的更多信息，参见 Java 平台批处理应用规范：

http://www.jcp.org/en/jsr/detail?id=352

第 27 章 Chapter 27

Java EE 的并发工具

本章介绍 Java EE 的并发工具（由 JSR 236 指定），包括以下内容：

- 并发基础
- 并发工具的主要组件
- 并发和事务
- 并发和安全
- jobs 并发示例
- taskcreator 并发示例
- 关于并发工具的更多信息

27.1 并发基础

并发概念是指同时（并行）执行两个或多个任务。任务可能包括方法（函数）、程序的部分，或者甚至可以是其他程序。对于当前的计算机架构，一个 CPU 中支持多个内核和多个处理器是相当常见的。

Java 平台提供了并发编程支持，这是实现 Java EE 容器提供的众多服务的基础。Java SE 5 中，由 `java.util.concurrent` 包提供对并发的高层 API 支持。

在 Java EE 7 之前，没有专门的 API 支持企业开发人员以一种安全的标准方式使用并发工具。Java EE Web 和 EJB 容器使用容器托管线程池实例化对象。因此，强烈建议不要使用 Java SE 并发 API 实例化 `Thread` 对象。如果开发人员创建了一个新的（非托管）`Thread` 对象，则容器不能保证这个 `Thread` 支持其他 Java EE 平台服务（例如，事务和安全性）。

27.1.1 线程和进程

有两个主要的并发概念：**进程**（process）和**线程**（thread）。

进程主要与操作系统（operating system，OS）中运行的应用关联。进程有特定的运行时资源来与底层 OS 交互和分配其他资源（如它自己的内存），这与 JVM 进程类似。JVM 实际上就是一个进程。

Java 编程语言和平台主要考虑线程。

线程与进程有一些相同的特性，因为二者都要使用 OS 或执行环境的资源。不过，线程更易于创建，使用的资源也远远少于进程使用的资源。

由于线程是轻量级的，因此对于有多个内核和几 GB RAM 的现代 CPU，可以在一个 JVM 进程中处理成千上万个线程。线程的具体数目取决于 CPU、OS 和可用 RAM 共同作用的结果，另外还依赖于 JVM 的正确配置（优化）。

尽管并发编程可以解决很多问题，并且可以改善大多数应用的性能，不过有些情况下多个执行线程或进程可能会带来严重的问题。这些情况包括：

- 死锁
- 线程饥饿
- 共享资源的并发访问
- 程序生成不正确的数据

27.2 并发工具的主要组件

并发资源是为 Java EE 应用提供并发功能的托管对象。在 GlassFish 服务器中，应配置并发资源，以便应用组件使用（如 servlet 和企业 bean）。并发资源通过 JNDI 查找或资源注入来访问。

并发工具的主要组件包括：

- `ManagedExecutorService`：这是一个托管执行器服务，应用使用这个服务异步地执行所提交的任务。任务在由容器启动和管理的线程上执行。容器上下文会传播到执行任务的线程。

  例如，通过使用 `ManagedExecutorService.submit()` 调用，可以提交一个任务（如 GenerateReportTask）在以后某个时间执行，然后通过使用 `Future` 对象回调，在结果可用时获取结果。
- `ManagedScheduledExecutorService`：这是一个托管调度执行器服务，应用使用这个服务在指定时间异步地执行所提交的任务。任务在由容器启动和管理的线程上执行。容器上下文会传播到执行任务的线程。这个 API 提供了调度功能，允许用户通过编程在应用中为任务执行设置一个特定的日期 / 时间。

- ContextService：这是一个上下文服务，用来创建动态代理对象，这些对象将描述一个容器的上下文，使应用可以在稍后某个时间在这个上下文中运行，或者提交到一个托管执行器服务。容器上下文会传播到执行任务的线程。
- ManagedThreadFactory：这是一个托管线程工厂，应用使用这个线程工厂来创建托管线程。这些线程由容器启动和管理。容器上下文会传播到执行任务的线程。这个对象也可以用来为特定用例（定制线程）提供定制工厂，另外还可以有其他用法，例如，将特定 / 专用属性设置为这些这些工厂对象。

27.3 并发和事务

事务最基本的操作是提交和回滚。不过，在一个完成并发处理的分布式环境中，可能很难保证成功地处理提交或回滚操作，事务可能分布在不同的线程、CPU 内核、物理主机和网络。

在这种情况下，确保回滚操作成功执行至关重要。通过 javax.transaction.UserTransaction，并发工具依赖于 Java 事务 API（JTA）在其组件上实现和支持事务，允许应用开发人员显式地管理事务边界。更多有关信息可以参见 JTA 规范。

上下文对象可以开始、提交或回滚事务（可选），不过这些对象不能加入父组件事务。

下面的代码段展示了一个 Runnable 任务，首先得到一个 UserTransaction，然后开始并提交一个事务，同时与其他事务性组件交互，如一个企业 bean 和一个数据库：

```
public class MyTransactionalTask implements Runnable {

    UserTransaction ut = ... // obtained through JNDI or injection

    public void run() {

        // Start a transaction
        ut.begin();

        // Invoke a Service or an EJB
        myEJB.businessMethod();

        // Update a database entity using an XA JDBC driver
        myEJB.updateCustomer(customer);

        // Commit the transaction
        ut.commit();

    }
}
```

27.4 并发和安全

Java EE 的并发工具将大部分安全决策都留给应用服务器实现来完成。不过，如果容器支持一个安全上下文，这个上下文就可以传播到执行线程。ContextService 可以支持多个运行时行为，如果启用了 security 属性，它会传播容器安全主体。

27.5 jobs 并发示例

这一节介绍一个非常基本的例子，展示如何在企业应用中使用一些基本的并发特性。特别地，这个例子使用了 Java EE 并发工具的一个主要组件：托管执行器服务（Managed Executor Service）。

这个例子中，使用了一个 RESTful Web 服务（提供为一个公共 API）来提交要执行的通用作业。这些作业在后台处理。每个作业分别在执行开始和结束时打印一个 "Starting" 和一个 "Finished" 消息。另外，为了模拟后台处理，每个作业需要 10 秒的执行时间。

这个 RESTful 服务提供了两个方法：

- /token: 提供为一个 GET 方法，注册并返回合法的 API token。
- /process: 提供为一个 POST 方法，接收一个 jobID 查询参数，这是要执行的作业的标识符，另外还要接收一个定制 HTTP 首部，名为 X-REST-API-Key，可以在内部使用这个首部来根据 token 验证请求。

这个 token 用来区分 API 提供的服务质量（Quality of Service，QoS）。如果用户在服务请求中提供了一个 token，那么可以处理多个并发作业。不过，如果用户没有提供 token，则只能一次处理一个作业。由于执行每个作业需要花费 10 秒时间，因此未提供 token 的用户只能每 10 秒执行一个服务调用。对于提供了 token 的用户，处理则会快得多。

之所以存在这种差别，是因为使用了两个不同的托管执行器服务，分别对应一种类型的请求。

27.5.1 运行 jobs 示例

增加两个托管执行器服务并配置 GlassFish 服务器之后，可以使用 NetBeans IDE 或 Maven 构建、打包、部署和运行 jobs 示例。

27.5.1.1 为基本并发示例配置 GlassFish 服务器

要配置 GlassFish 服务器，需要完成以下步骤：

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 打开管理控制台 http://localhost:4848。
3. 展开 Resources（资源）节点。

4. 展开 Concurrent Resources（并发资源）节点。

5. 单击 Managed Executor Services（托管执行器服务）。

6. 在托管执行器服务页面上，单击 New（新建）打开新建托管执行器服务页面。

7. 在 JNDI Name（JNDI 名）域中，输入 `MES_High` 来创建高优先级托管执行器服务。使用以下设置（其他设置保留默认值）：

- **线程优先级**：10
- **内核大小**：2
- **最大池大小**：5
- **任务队列容量**：2

8. 单击 OK。

9. 在托管执行器服务页面上，再次单击 New。

10. 在 JNDI Name 域中，输入 `MES_Low` 创建低优先级托管执行器服务。使用以下设置（其他设置保留默认值）：

- **线程优先级**：1
- **内核大小**：1
- **最大池大小**：1
- **任务队列容量**：0

11. 单击 OK。

27.5.1.2 使用 NetBeans IDE 构建、打包和部署 jobs 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 从 File（文件）菜单选择 Open Project（打开工程）。

3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/concurrency
```

4. 选择 `jobs` 文件夹。

5. 单击 Open Project。

6. 在 Projects（工程）标签页中，右键单击 `jobs` 工程并选择 Build（构建）。

这个命令会构建并部署这个应用。

27.5.1.3 使用 Maven 构建、打包和部署 jobs 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。

2. 在一个终端窗口中导航至：

```
tut-install/examples/concurrency/jobs/
```

3. 输入以下命令构建并部署这个应用：

```
mvn install
```

27.5.1.4 运行 jobs 示例并提交低优先级作业

作为提交低优先级作业的用户，要运行示例，需要遵循以下步骤：

1. 在 Web 浏览器中输入以下 URL：

```
http://localhost:8080/jobs
```

2. 在作业客户端页面，在 Enter a JobID（输入一个作业 ID）域输入值 1，在 Enter a Token（输入一个 Token）域中不输入任何内容，然后单击 Submit Job（提交作业）。

页面下方会显示以下消息：

```
Job 1 successfully submitted
```

服务器日志包括以下消息：

```
INFO:   Invalid or missing token!
INFO:   Task started LOW-1
INFO:   Job 1 successfully submitted
INFO:   Task finished LOW-1
```

这样就提交了一个低优先级作业。这说明，10 秒内不能提交另一个作业。如果试图这么做，RESTful API 会返回一个服务器不可用（HTTP 503）响应，另外页面下方会显示以下消息：

```
Job 2 was NOT submitted
```

服务器日志包括以下消息：

```
INFO:   Invalid or missing token!
INFO:   Job 1 successfully submitted
INFO:   Task started LOW-1
INFO:   Invalid or missing token!
INFO:   Job 2 was NOT submitted
INFO:   Task finished LOW-1
```

27.5.1.5 运行 jobs 示例并提交高优先级作业

作为提交高优先级作业的用户，要运行示例，需要遵循以下步骤。

1. 在 Web 浏览器中输入以下 URL：

```
http://localhost:8080/jobs
```

2. 在作业客户端页面，在 Enter a JobID（输入一个作业 ID）域输入一个 1 到 10 位的值。

3. 单击 Get a token here 行上的 here 链接得到一个 token。在一个新标签页中会显示 token 的页面。

4. 复制这个 token，并返回到作业客户端页面。

5. 在 Enter a Token(输入一个 Token）域粘贴这个 token，然后单击 Submit Job(提交作业)。

页面下方会显示以下消息：

```
Job 11 successfully submitted
```

服务器日志包括以下消息：

```
INFO:   Token accepted. Execution with high priority.
INFO:   Task started HIGH-11
INFO:   Job 11 successfully submitted
INFO:   Task finished HIGH-11
```

这样就提交了一个高优先级作业。这意味着，你可以提交多个有 token 的作业，而不会像提交低优先级作业那样受到每个作业 10 秒的限制。如果快速连续地提交 3 个有 token 的作业，则会在页面下方显示如下消息：

```
Job 1 was submitted
Job 2 was submitted
Job 3 was submitted
```

服务器日志包括以下消息：

```
INFO:   Token accepted. Execution with high priority.
INFO:   Task started HIGH-1
INFO:   Job 1 successfully submitted
INFO:   Token accepted. Execution with high priority.
INFO:   Task started HIGH-2
INFO:   Job 2 successfully submitted
INFO:   Task finished HIGH-1
INFO:   Token accepted. Execution with high priority.
INFO:   Task started HIGH-3
INFO:   Job 3 successfully submitted
INFO:   Task finished HIGH-2
INFO:   Task finished HIGH-3
```

27.6 taskcreator 并发示例

`Taskcreator` 示例将展示如何使用 Java EE 并发工具立即、定期或在一个固定延迟之后运行任务。这个示例提供了一个 JSF 界面，允许用户提交要执行的任务，并显示每个任务的信息消息。这个例子使用托管执行器服务立即运行任务，另外使用托管调度执行器服务定期或在一个固定延迟之后运行任务。(参见 27.2 节来了解有关这些服务的信息。)

`Taskcreator` 示例包括以下组件：

- 一个 JSF 页面（`index.xhtml`），包含 3 个元素：一个用来提交任务的表单，一个任务执行日志和一个用来取消定期任务的表单。这个页面提交 Ajax 请求来创建和取消任务，另外还会接收 WebSocket 消息，这里使用 JavaScript 代码更新任务执行日志。
- 一个 CDI 托管 bean（`TaskCreatorBean`），处理来自 JSF 页面的请求。这个 bean 调用 `TaskEJB` 中的方法提交新任务，还可以取消定期任务。
- 一个企业 bean（`TaskEJB`），使用资源注入获得执行器服务实例，并提交要执行的任务。这个 bean 也是一个 JAX-RS Web 服务端点。任务向这个端点发送信息消息。
- 一个 WebSocket 端点（`InfoEndpoint`），企业 bean 使用这个端点向客户端发送信息消息。

- 一个实现 Runnable 接口的任务类（Task）。这个类中的 run 方法向 TaskEJB 中的 Web 服务端点发送信息消息，并休眠 1.5 秒。

图 27-1 显示了这个 taskcreator 示例的架构。

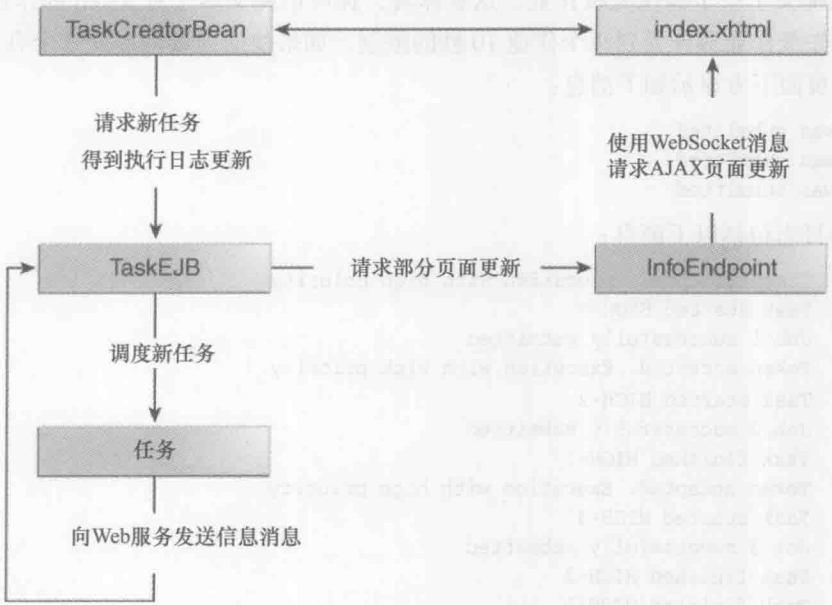


图 27-1 taskcreator 示例的架构

TaskEJB 类从应用服务器得到默认的执行器服务对象，如下所示：

```
@Resource(name="java:comp/DefaultManagedExecutorService")
ManagedExecutorService mExecService;

@Resource(name="java:comp/DefaultManagedScheduledExecutorService")
ManagedScheduledExecutorService sExecService;
```

TaskEJB 中的 submitTask 方法使用这些对象来提交要执行的任务，如下所示：

```
public void submitTask(Task task, String type) {
    /* Use the managed executor objects from the app server */
    switch (type) {
        case "IMMEDIATE":
            mExecService.submit(task);
            break;
        case "DELAYED":
            sExecService.schedule(task, 3, TimeUnit.SECONDS);
            break;
        case "PERIODIC":
            ScheduledFuture<?> fut;
            fut = sExecService.scheduleAtFixedRate(task, 0, 8,
                    TimeUnit.SECONDS);
```

```
            periodicTasks.put(task.getName(), fut);
            break;
        }
    }
```

对于定期任务，TaskEJB 维护了 ScheduledFuture 对象的一个引用，使用户可以在任意时间取消这个任务。

27.6.1 运行 taskcreator 示例

这一节介绍如何使用 NetBeans IDE 或 Maven 构建、打包、部署和运行 taskcreator 示例。

27.6.1.1 使用 NetBeans IDE 构建、打包和部署 taskcreator 示例

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/concurrency
```

4. 选择 taskcreator 文件夹。
5. 单击 Open Project。
6. 在 Projects（工程）标签页中，右键单击 taskcreator 工程并选择 Build。

这个命令会构建并部署这个应用。

27.6.1.2 构建、打包和部署 taskcreator 示例使用 Maven

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 在一个终端窗口中导航至：

```
tut-install/examples/concurrency/taskcreator/
```

3. 输入以下命令来构建和部署这个应用：

```
mvn install
```

27.6.1.3 运行 taskcreator 示例

1. 在 Web 浏览器中打开以下 URL：

```
http://localhost:8080/taskcreator/
```

这个页面包含一个提交任务的表单、一个任务执行日志和一个取消定期任务的表单。

2. 选择 Immediate（立即）任务类型，输入一个任务名，并单击 Submit（提交）按钮。任务执行日志中会出现类似下面的消息：

```
12:40:47 - IMMEDIATE Task TaskA finished
12:40:45 - IMMEDIATE Task TaskA started
```

3. 选择 Delayed(3 sec)(延迟（3 秒)) 任务类型，输入一个任务名，并单击 Submit 按钮。任务执行日志中会出现类似下面的消息：

```
12:43:26 - DELAYED Task TaskB finished
12:43:25 - DELAYED Task TaskB started
12:43:22 - DELAYED Task TaskB submitted
```

4. 选择 Periodic(8 sec)(定期（8 秒)) 任务类型，输入一个任务名，并单击 Submit 按钮。任务执行日志中会出现类似下面的消息：

```
12:45:25 - PERIODIC Task TaskC finished run #2
12:45:23 - PERIODIC Task TaskC started run #2
12:45:17 - PERIODIC Task TaskC finished run #1
12:45:15 - PERIODIC Task TaskC started run #1
```

可以增加多个定期任务。要取消一个定期任务，可以在表单上选择这个任务并单击 Cancel Task（取消任务）。

27.7 关于并发工具的更多信息

关于并发的更多信息，参见：

- Java EE 并发工具规范：

  http://jcp.org/en/jsr/detail?id=236

- 并发工具规范：

  http://jcp.org/en/jsr/detail?id=166

- 《Java 教程》中的并发课程：

  http://docs.oracle.com/javase/tutorial/essential/concurrency/

第七部分 *Part 7*

案例研究

第七部分将提供一些使用了多种 Java EE 技术的案例研究，包含以下各章：

- 第 28 章　Duke 书店案例研究示例
- 第 29 章　Duke 辅导案例研究示例
- 第 30 章　Duke 综合案例研究示例

Chapter 28 第 28 章

Duke 书店案例研究示例

Duke 书店示例是一个简单的电子商务应用，展示了 JSF 技术的一些高级特性，并结合了 Java EE 的上下文和依赖注入（CDI）、企业 bean 以及 Java 持久化 API。用户可以从一个图像地图选择图书，查看书店商品目录，还可以购买图书。这个应用中没有使用安全特性。

本章将介绍以下内容：

- Duke 书店的设计和架构
- Duke 书店接口
- 运行 Duke 书店案例研究应用

28.1 Duke 书店的设计和架构

Duke 书店是一个简单的 Web 应用，使用了 JSF 技术的很多特性，另外还结合使用了其他 Java EE 7 特性：

- JSF 技术以及 Java EE 上下文和依赖注入（CDI）
  - 一组 Facelets 页面（以及一个模板）提供应用的用户界面。
  - CDI 托管 bean 与各个 Facelets 页面关联。
  - 前端页面上的定制图像地图组件允许用户选择一本书进入商店。地图上的每个区域分别由一个 JSF 托管 bean 表示。另外为提高可访问性，还提供了文本超链接。
  - 在图像地图和文本链接上注册了动作监听器。这些监听器获取所选图书的 ID 值，将其存储在会话映射中，以便为下一个页面的托管 bean 获取。
  - 使用 `h:dataTable` 标记动态地呈现图书目录和购物车内容。

- 结账页面 bookcashier.xhtml 的信用卡域上注册一个定制转换器，另外还使用了一个 f:validateRegEx 标记确保输入能够正确地格式化。
- 在 bookcashier.xhtml 的名字域上注册一个值改变监听器。这个监听器将名字保存在一个参数中，以便后续页面 bookreceipt.xhtml 访问这个名字。

- 企业 bean：本地无接口视图无状态会话 bean 和单例 bean
- Java 持久化 API 实体

Duke 书店应用包位于 *tut-install*/examples/case-studies/dukes-bookstore/src/main/java/javaeetutorial/dukesbookstore/ 目录，包括：

- components：包含定制 UI 组件类 MapComponent 和 AreaComponent
- converters：包含定制转换器类 CreditCardConverter
- ejb：包含两个企业 bean：
  - 一个单例 bean ConfigBean，用来初始化数据库中的数据
  - 一个无状态会话 bean BookRequestBean，包含管理实体的业务逻辑
- entity：包含 Book 实体类
- exceptions：包含 3 个异常类
- listeners：包含事件处理器和事件监听器类
- model：包含一个模型 JavaBeans 类
- renderers：包含定制 UI 组件类的定制呈现器
- web.managedbeans：包含 Facelets 页面的托管 bean
- web.messages：包含本地化消息的资源包文件

28.2 Duke 书店接口

这一节介绍 Duke 书店示例组件的详细内容以及这些组件如何交互。

28.2.1 Book Java 持久化 API 实体

Book 实体位于 dukesbookstore.entity 包，封装了 Duke 书店存储的图书数据。Book 实体定义了示例中使用的属性：

- 图书 ID
- 作者的名
- 作者的姓
- 书名
- 价格
- 这本书是否在售
- 出版年份

- 图书简介
- 库存数量

Book 实体还定义了一个简单的命名查询 findBooks。

28.2.2 Duke 书店中使用的企业 bean

Duke 书店中使用的两个企业 bean 位于 dukesbookstore.ejb 包，提供了 Duke 书店的业务逻辑。

- BookRequestBean 是一个有状态会话 bean，包含应用的业务方法。这些方法可以创建、获取和购买图书，以及更新一本书的库存。要获取图书，getBooks 方法会调用 Book 实体中定义的 findBooks 命名查询。
- ConfigBean 是一个单例会话 bean，用于在最初部署应用时创建图书目录中的图书。它会调用 BookRequestBean 中定义的 createBook 方法。

28.2.3 Duke 书店中使用的 Facelets 页面和托管 bean

Duke 书店应用使用 Facelets 及其模板特性来显示用户界面。Facelets 页面与一组 CDI 托管 bean 交互，这些 bean 相当于后备 bean，为用户界面提供底层属性和方法。前端页面还要与应用使用的定制组件交互。

这个应用使用以下 Facelets 页面，它们位于 *tut-install*/examples/case-studies/dukes-bookstore/src/main/webapp/ 目录。

- bookstoreTemplate.xhtml：模板文件指定了每个页面上使用的一个首部以及所有页面使用的样式表。这个模板还会获取 Web 浏览器中的语言集。
  使用 LocaleBean 托管 bean。
- index.xhtml：登录页面，使用 faces-config.xml 文件中配置的托管 bean 摆放定制地图和区域组件，允许用户选择一本书，并进入 bookstore.xhtml 页面。
- bookstore.xhtml：这个页面允许用户得到所选图书或特色图书的详细信息，可以向购物车增加所选图书或特色图书，并进入 bookcatalog.xhtml 页面。
  使用 BookstoreBean 托管 bean。
- bookdetails.xhtml：这个页面会显示从 bookstore.xhtml 或其他页面选择的图书的详细信息，允许用户将图书增加到购物车，并/或进入 bookcatalog.xhtml 页面。
  使用 BookDetailsBean 托管 bean。
- bookcatalog.xhtml：这个页面显示图书目录中的图书，允许用户向购物车增加图书，查看图书的详细信息，查看购物车，清空购物车，或者购买购物车中的图书。
  使用 BookstoreBean、CatalogBean 和 ShoppingCart 托管 bean。
- bookshowcart.xhtml：这个页面显示购物车的内容，允许用户删除购物车中的商品，

查看一个商品的详细信息，清空购物车，购买购物车中的图书，或者返回图书目录。

使用 ShowCartBean 和 ShoppingCart 托管 bean。

- bookcashier.xhtml：这个页面允许用户购买图书，指定一个送货方式选项，订阅快讯，或者通过购买一定金额的商品加入 Duke 粉丝俱乐部。

  使用 CashierBean 和 ShoppingCart 托管 bean。

- bookreceipt.xhtml：这个页面确认用户购买的商品，并允许用户返回到图书目录继续购物。

  使用 CashierBean 托管 bean。

- bookordererror.xhtml：对于所订购的图书，如果书店没有更多库存，就会由 CashierBean 呈现这个页面。

这个应用使用以下托管 bean，它们位于 *tut-install*/examples/case-studies/dukes-bookstore/src/main/java/javaeetutorial/dukesbookstore/web/managedbeans/ 目录。

- AbstractBean：包含其他托管 bean 调用的工具方法。
- BookDetailsBean：bookdetails.xhtml 页面的后备 bean。指定名 details。
- BookstoreBean：bookstore.xhtml 页面的后备 bean。指定名 store。
- CashierBean：bookcashier.xhtml 和 bookreceipt.xhtml 页面的后备 bean。
- CatalogBean：bookcatalog.xhtml 页面的后备 bean。指定名 catalog。
- LocaleBean：获取当前本地化环境的托管 bean；每个页面上都会使用。
- ShoppingCart：bookcashier.xhtml、bookcatalog.xhtml 和 bookshowcart.xhtml 页面使用的后备 bean。指定名 cart。
- ShoppingCartItem：包含 ShoppingCart、CatalogBean 和 ShowCartBean 调用的方法。
- ShowCartBean：bookshowcart.xhtml 页面的后备 bean。指定名 showcart。

28.2.4 Duke 书店中使用的定制组件和其他定制对象

Duke 书店的地图和区域定制组件以及相关的呈现器、监听器和模型类在 *tut-install*/examples/case-studies/dukes-bookstore/src/main/java/javaeetutorial/dukesbookstore/ 目录的以下包中定义。

- components：包含 MapComponent 和 AreaComponent 类。更多信息参见本书第 1 卷 15.4 节。
- listeners：包含 AreaSelectedEvent 类以及其他监听器类。更多信息参见本书第 1 卷 15.7 节。
- model：包含 ImageArea 类。更多信息参见本书第 1 卷 15.2.4 节。

- renderers：包含 MapRenderer 和 AreaRenderer 类。更多信息参见本书第 1 卷 15.5 节。

tut-install/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/ 目录还包含一个定制转换器以及没有绑定到定制组件的其他定制监听器。

- converters：包含 CreditCardConverter 类。更多信息参见本书第 1 卷 15.10 节。
- listeners：包含 LinkBookChangeListener、MapBookChangeListener 和 NameChanged 类。更多信息参见本书第 1 卷 15.6 节。

28.2.5 Duke 书店中使用的属性文件

Duke 书店应用中使用的字符串封装在资源包中，从而可以用多个不同的本地化环境显示本地化字符串。这些属性文件位于 *tut-install*/examples/case-studies/dukes-bookstore/src/main/java/javaeetutorial/dukesbookstore/web/messages/ 目录，其中包括一个包含英语字符串的默认文件和 3 个对应其他本地化环境的文件，具体为：

- Messages.properties：默认文件，包含英语字符串
- Messages_de.properties：包含德语字符串的文件
- Messages_es.properties：包含西班牙语字符串的文件
- Messages_fr.properties：包含法语字符串的文件

用户 Web 浏览器中的语言设置确定了要使用哪个本地化环境。bookstoreTemplate.xhtml 中的 html 标记从 LocaleBean 的 language 属性获取语言设置：

```
<html lang="#{localeBean.language}"
...
```

关于资源包的更多信息，参见本书第 1 卷第 20 章。

这个资源包在 faces-config.xml 文件中配置如下：

```
<application>
    <resource-bundle>
        <base-name>
            javaeetutorial.dukesbookstore.web.messages.Messages
        </base-name>
        <var>bundle</var>
    </resource-bundle>
    <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>de</supported-locale>
        <supported-locale>es</supported-locale>
        <supported-locale>fr</supported-locale>
    </locale-config>
</application>
```

从这个配置可以看到，在 Facelets 页面中，获取消息时要使用前缀 bundle，并指定

Messages_*locale*.properties 文件中找到的键，如 index.xhtml 页面中的下例所示：

```
<h:outputText style="font-weight:bold"
              value="#{bundle.ChooseBook}" />
```

在 Messages.properties 中，这个键字符串定义如下：

```
ChooseBook=Choose a Book from our Catalog
```

28.2.6 Duke 书店中使用的部署描述文件

Duke 书店中使用了以下部署描述文件：

- src/main/resources/META-INF/persistence.xml：Java 持久化 API 配置文件
- src/main/webapp/WEB-INF/bookstore.taglib.xml：定制组件的标记库描述文件
- src/main/webapp/WEB-INF/faces-config.xml：JSF 配置文件，这个文件配置地图组件的托管 bean 以及应用的资源包
- src/main/webapp/WEB-INF/web.xml：Web 应用配置文件

28.3 运行 Duke 书店案例研究应用

可以使用 NetBeans IDE 或 Maven 构建、打包、部署和运行 Duke 书店应用。

28.3.1 使用 NetBeans IDE 构建和部署 Duke 书店

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

tut-install/examples/case-studies

4. 选择 dukes-bookstore 文件夹。
5. 单击 Open Project。
6. 在 Projects（工程）标签页中，右键单击 dukes-bookstore 工程并选择 Build（构建）。

这会构建和打包 Duke 书店应用，并部署到 GlassFish 服务器。

28.3.2 使用 Maven 构建和部署 Duke 书店

1. 确保已启动 GlassFish 服务器（参见 2.2 节）以及数据库服务器（参见 2.4 节）。
2. 在一个终端窗口中导航至：

tut-install/examples/case-studies/dukes-bookstore/

3. 输入以下命令：

```
mvn install
```

这个命令会构建应用，并打包到一个 WAR 文件中，这个文件位于 *tut-install*/examples/case-studies/dukes-bookstore/target/ 目录。然后将应用部署到 GlassFish 服务器。

28.3.3 运行 Duke 书店应用

1. 在 Web 浏览器中输入以下 URL：

```
http://localhost:8080/dukes-bookstore/
```

2. 在 Duke 书店主页面上，单击图片中的一本书，或者单击页面下方的某个链接。
3. 使用应用中的页面查看和购买图书。

第 29 章 Chapter 29

Duke 辅导案例研究示例

Duke 辅导示例应用是一个学生辅导中心的跟踪系统。学生可以登记和离开，另外可以参观公园。辅导中心可以记录考勤和状态更新，另外可以存储监护人和学生的联系信息。管理员可以维护这个辅导中心系统。

本章将介绍以下内容：

- Duke 辅导应用的设计和架构
- 主界面
- 管理界面
- 运行 Duke 辅导案例研究应用

29.1 Duke 辅导应用的设计和架构

Duke 辅导应用是一个结合使用了多个 Java EE 技术的 Web 应用。它提供了一个主界面（面向学生、监护人和辅导中心人员），另外提供了一个管理界面（面向维护这个系统的人员）。两个界面的业务逻辑都由企业 bean 提供。这些企业 bean 使用 Java 持久化 API 来创建应用的数据，并存储在数据库中。图 29-1 展示了这个应用的架构。

Duke 辅导应用组织为两个主工程：`dukes-tutoring-common` 库和 `dukes-tutoring-war` Web 应用。`dukes-tutoring-common` 库工程包含 `dukes-tutoring-war` Web 应用使用的实体类和辅助类，`dukes-tutoring-common` 与 `dukes-tutoring-war` 一同打包和部署。这个库 JAR 文件很有用，允许其他应用重用这些实体类和辅助类，如某个 JavaFX 客户端应用。

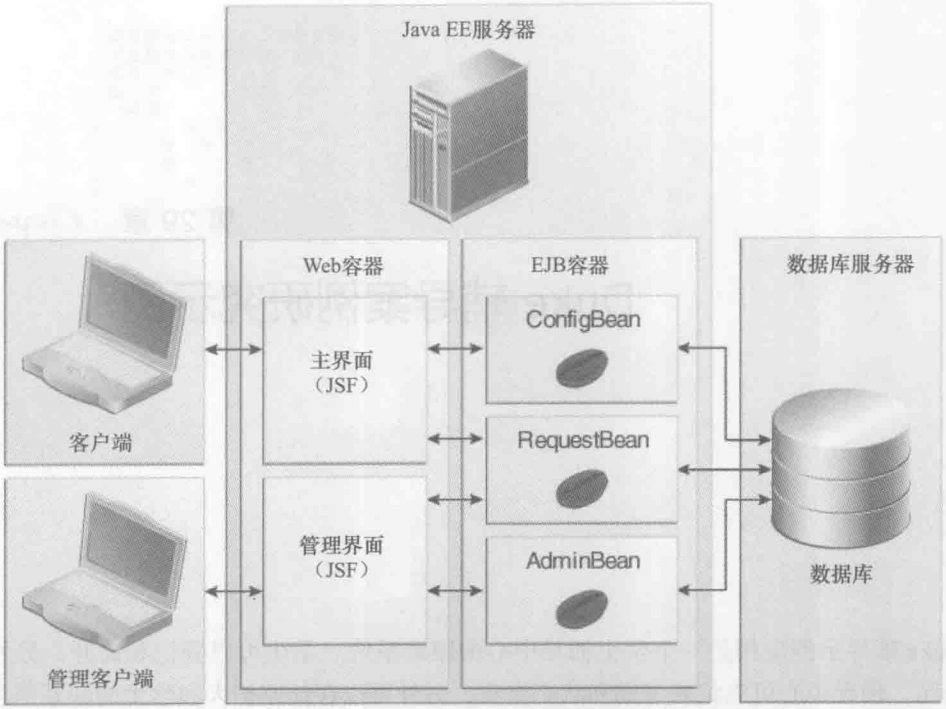


图 29-1 Duke 辅导示例应用架构

Duke 辅导应用使用以下 Java EE 7 平台特性：

- Java 持久化 API 实体
  - 一个定制 bean 验证注解 `@Email`，用于验证 email 地址
  - 一个标准 `jta-data-source` 定义，将在部署时创建 JDBC 资源
  - `persistence.xml` 部署描述文件中的一个标准属性，可以采用可移植方式在 `jta-data-source` 中自动地创建和删除表
- 企业 bean
  - 本地无接口视图会话和单例 bean
  - 会话 bean 中的 JAX-RS 资源
  - 管理接口业务方法上的 Java EE 安全约束
  - WAR 中打包的所有企业 bean
- WebSocket
  - 一个 WebSocket 服务器端点，自动向客户端端点发布学生的状态
- 上下文和依赖注入
  - 一个 CDI 事件，学生状态改变时会触这个事件
  - 处理器方法，一旦触发状态事件会更新应用

  - Facelets 页面的 CDI 托管 bean
  - CDI 托管 bean 中的 bean 验证注解
- JSF 技术，使用 Facelets 作为 Web 前端
  - 模板
  - 复合组件
  - 定制格式化器 PhoneNumberFormatter
  - 管理接口上的安全约束
  - 启用 Ajax 的 Facelets 组件

Duke 辅导应用有两个主要的用户界面，这两个用户界面都打包在一个 WAR 文件中：

- 主界面：面向学生、监护人和辅导中心人员
- 管理界面，辅导中心人员使用这个界面管理学生和监护人，还可以生成考勤报告

29.2　主界面

主界面允许学生和辅导中心人员检查进出的学生，并记录学生什么时间去操场。

29.2.1　主界面中使用的 Java 持久化 API 实体

主界面中使用了以下实体，它们封装了 Duke 辅导应用存储和管理的数据，位于 dukes-tutoring-common 工程的 dukestutoring.entity 包中。

- Person：Person 实体定义了应用记录的学生和监护人共同的属性。这些属性包括人名和联系信息（联系信息包括电话号码和 email 地址）。这个实体有两个子类：Student 和 Guardian。
- PersonDetails：PersonDetails 实体用于存储所有人共同的额外数据，如照片和生日等属性，出于性能方面的原因，这些属性未包含在 Person 实体中。
- Student 和 Guardian：Student 实体存储来辅导中心的学生的特定属性，包括学生的年级和学校等信息。Guardian 实体存储 Student 的父母或监护人的特定属性。学生和监护人有一种多对多的关系。也就是说，一个学生可以有一个或多个监护人，一个监护人也可以有一个或多个学生。
- Address：Address 实体表示一个邮寄地址，与 Person 实体关联。地址与人有一种多对一的关系。也就是说，一个人可以有多个地址。
- TutoringSession：TutoringSession 实体表示辅导中心的某一天。这个辅导班实体会记录这一天哪些学生在位，哪些学生去了公园。
- StatusEntry：学生的状态改变时，StatusEntry 实体将状态记入日志，这个实体与 TutoringSession 实体关联。学生在辅导班登记、去公园以及离开时，他的状态就会改变。利用状态记录，辅导中心人员就能跟踪哪些学生参加了辅导班，他们

何时登记和离开，哪些学生去了公园，以及他们何时去公园，另外何时从公园回来。

关于创建 Java 持久化 API 实体的更多信息，参见第 8 章。有关验证实体数据的更多信息，参见 8.1.2.4 节以及本书第 1 卷第 22 章。

29.2.2 主界面中使用的企业 bean

主界面中使用了以下企业 bean，这些企业 bean 提供了 Duke 辅导应用的业务逻辑，位于 `dukes-tutoring-war` 工程的 `dukestutoring.ejb` 包中：

- `ConfigBean` 是一个单例会话 bean，用来在应用最初部署时创建默认的学生，并创建一个自动 EJB 定时器，它会在每个工作日自动创建辅导班实体。
- `RequestBean` 是一个无状态会话 bean，包含主界面的业务方法。这个 bean 还提供了一些业务方法来获取学生名单。这些业务方法使用强类型的 Criteria API 查询从数据库获取数据。`RequestBean` 还注入一个 CDI 事件实例 `StatusEvent`。当学生的状态改变时，会从业务方法触发这个事件。

关于创建和使用企业 bean 的信息，参见第二部分。关于创建强类型 Criteria API 查询的信息，参见第 11 章。关于 CDI 事件的信息，参见本书第 1 卷 25.5 节。

29.2.3 主界面中使用的 WebSocket 端点

`javaeetutorial.dukestutoring.web.websocket.StatusEndpoint` 类是一个 WebSocket 服务器端点，可以将学生及其状态返回给客户端端点。`StatusEndpoint.updateStatus` 方法是对应 `StatusEvent` 事件的一个 CDI 观察者方法。主界面中一个学生的状态改变时，会触发一个 `StatusEvent` 事件。`updateStatus` 观察者方法由容器调用，将状态改变压入到所有向 `StatusEndpoint` 注册的客户端端点。

`index.xhtml` JSF 页面包含连接 WebSocket 端点的 JavaScript 代码。这个页面上的 `onMessage` 方法单击一个 JSF 按钮，这会建立一个 Ajax 请求，刷新显示学生当前状态的表格。

关于 WebSocket 端点的更多信息，参见本书第 1 卷第 18 章。关于 CDI 事件的更多信息，参见本书第 1 卷 25.5 节。

29.2.4 主界面中使用的 Facelets 文件

Duke 辅导应用使用 Facelets 显示用户界面，大量使用了 Facelets 的模板特性。Facelets 是 JSF 技术的默认显示技术，相应的 XHTML 文件位于 *tut-install*`/examples/case-studies/dukes-tutoring-war/src/main/webapp/` 目录。

主界面中使用了以下 Facelets 文件。

- `template.xhtml`：主界面的模板文件
- `error.xhtml`：出错时的错误文件
- `index.xhtml`：主界面的登录页面

- park.xhtml：这个页面显示目前谁在公园
- current.xhtml：这个页面显示谁参加了今天的辅导班
- statusEntries.xhtml：这个页面显示今天辅导班的详细状态记录日志
- resources/components/allStudentsTable.xhtml：这是一个复合组件，表示一个表，显示所有活动的学生
- resources/components/allInactiveStudentsTable.xhtml：这是一个复合组件，表示一个表，显示所有不活动学生
- resources/components/currentSessionTable.xhtml：这是一个复合组件，表示一个表，显示今天辅导班的所有学生
- resources/components/parkTable.xhtml：这是一个复合组件，表示一个表，显示目前所有在公园的学生
- WEB-INF/includes/mainNav.xhtml：主界面导航条的 XHTML 片段

关于使用 Facelets 的更多信息，参见本书第 1 卷第 8 章。

29.2.5 主界面中使用的辅助类

主界面中使用了以下辅助类，这些类放在 dukes-tutoring-common 工程的 dukestutoring.util 包中。

- CalendarUtil：这个类提供了一个方法，可以从 java.util.Calendar 实例去除不必要的时间数据。
- Email：这个定制 bean 验证注解类用于验证 Person 实体中的 email 地址。
- StatusType：这个枚举类型定义了学生的不同状态，可取值包括 IN、OUT 和 PARK。整个应用中都会使用 StatusType，这包括 StatusEntry 实体中以及整个主界面中。StatusType 还定义了一个 toString 方法，会根据本地化环境返回状态的一个本地化转换版本。

29.2.6 属性文件

主界面中使用的字符串封装在资源包中，从而可以用多个不同的本地化环境显示本地化字符串。对应每一个属性文件，会有一些相应的本地化环境特定文件（文件名后追加了本地化环境代码），包含对应各个本地化环境的已转换的字符串。例如，Messages_es.properties 包含对应西班牙本地化环境的本地化字符串。

dukes-tutoring-common 工程有以下资源包，位于 src/main/resources/ 目录下：

- javaeetutorial/dukestutoring/util/StatusMessages.properties：包含对应默认本地化环境中 StatusType 枚举类型中定义的各个状态类型的字符串。每一个得到支持的本地化环境有一个形如 StatusMessages_*localeprefix*.properties 的包含本地化环境字符串的属性文件。例如，对应西班牙语本地化环

境的字符串位于 StatusMessages_es.properties。

dukes-tutoring-war 工程有以下资源包，位于 src/main/resources/ 目录下：

- ValidationMessages.properties：这个资源包对应 bean 验证运行时环境所用的默认本地化环境，包含显示验证消息的字符串。这个文件必须命名为 ValidationMessages.properties，并位于 bean 验证规范所要求的默认包中。每一个得到支持的本地化环境有一个形如 ValidationMessages_*locale prefix*.properties 的包含本地化环境字符串的属性文件。例如，对应德语本地化环境的字符串位于 ValidationMessages_de.properties。
- javaeetutorial/dukestutoring/web/messages/Messages.properties：这个资源包对应主界面和管理 Facelets 界面的默认本地化环境，包含相应的字符串。每一个得到支持的本地化环境有一个形如 Messages_*locale prefix*.properties 的包含本地化环境字符串的属性文件。例如，对应简体中文本地化环境的字符串位于 Messages_zh.properties。

关于本地化 Web 应用的更多信息，参见本书第 1 卷 16.5 节。

29.2.7 Duke 辅导应用中使用的部署描述文件

Duke 辅导应用使用以下部署描述文件，位于 dukes-tutoring-war 工程的 src/main/webapp/WEB-INF 目录：

- faces-config.xml：JSF 配置文件
- glassfish-web.xml：特定于 GlassFish 服务器的配置文件，定义了安全角色映射
- web.xml：Web 应用配置文件

Duke 辅导应用还使用了以下部署描述文件，位于 dukes-tutoring-common 工程的 src/main/resources/META-INF 目录：

- persistence.xml: Java 持久化 API 配置文件

Duke 辅导应用中没有使用企业 bean 部署描述文件。这个应用在企业 bean 类文件中使用注解来完成企业 bean 的配置。

29.3 管理界面

Duke 辅导应用的管理界面由辅导中心人员用来管理主界面使用的数据：学生、学生的监护人和地址。管理界面使用的很多组件与主界面中的相同，这里只介绍仅在管理界面中使用的组件。

29.3.1 管理界面中使用的企业 bean

管理界面中使用了以下企业 bean，位于 dukes-tutoring-war 工程的 dukestutoring.

ejb 包中。

- AdminBean：这是一个无状态会话 bean，表示管理界面中使用的所有业务逻辑。调用安全方法来保证只能由授权用户调用业务方法。

29.3.2 管理界面中使用的 Facelets 文件

管理界面中使用了以下 Facelets 文件，位于 src/main/webapp/ 目录下：

- admin/adminTemplate.xhtml：管理界面的模板
- admin/index.xhtml：管理界面的登陆页面
- login.xhtml：有安全约束的管理界面的登录页面
- loginError.xhtml：认证管理用户出现错误时所显示的页面
- admin/address 目录：允许创建、编辑和删除 Address 实体的页面
- admin/guardian 目录：允许创建、编辑和删除 Guardian 实体的页面
- admin/student 目录：允许创建、编辑和删除 Student 实体的页面
- resources/components/formLogin.xhtml：这是一个复合组件，表示使用 Java EE 安全的一个登录表单
- WEB-INF/includes/adminNav.xhtml：表示管理界面导航条的 XHTML 片段

29.3.3 管理界面中使用的 CDI 托管 bean

管理界面中使用的 CDI 托管 bean 位于 dukes-tutoring-war 工程的 dukestutoring.web 包中。

- StudentBean.java：这是一个托管 bean，对应用来创建和编辑学生的 Facelets 页面。名和姓有 bean 验证注解，要求必须在相应的名和姓域中填入内容。电话号码的 bean 验证注解要求确保所提交的数据有正确的格式。
- GuardianBean.java：这是一个托管 bean，对应用来创建监护人以及向学生指定监护人的 Facelets 页面。名和姓有 bean 验证注解，要求必须在相应的名和姓域中填入内容。电话号码的 bean 验证注解要求确保所提交的数据有正确的格式。
- AddressBean.java：这是一个托管 bean，对应用来创建学生地址的 Facelets 页面。街道、城市、省和邮政编码属性有 bean 验证注解，要求必须在相应域中填入内容。邮政编码属性还有一个注解，要求确保所提交的数据有正确的格式。

29.3.4 管理界面中使用的辅助类

管理界面中使用了以下辅助类，位于 dukes-tutoring-war 工程的 dukestutoring.web.util 包中。

- EntityConverter：StudentConverter 和 GuardianConverter 的父类，定义了一个缓存，在转换 JSF 用户界面组件使用的实体时可以用来存储实体类。这个缓

存有助于提高性能，存储在 JSF 上下文中。

- StudentConverter：这是 Student 实体类的一个 JSF 转换器。这个类包含一些方法，可以完成 Student 实例与字符串的来回转换，从而可以在应用的用户界面组件中使用。
- GuardianConverter：类似于 StudentConverter，这个类是 Guardian 实体类的一个转换器。

29.4 运行 Duke 辅导案例研究应用

这一节介绍如何构建、打包、部署和运行 Duke 辅导应用。

29.4.1 运行 Duke 辅导应用

可以使用 NetBeans IDE 或 Maven 构建、打包、部署和运行 Duke 辅导应用。

29.4.1.1 使用 NetBeans IDE 构建和部署 Duke 辅导应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 如果数据库服务器还未运行，按照 2.4 节中的说明启动该服务器。
3. 从 File（文件）菜单选择 Open Project（打开工程）。
4. 在 Open Project 对话框中，导航至：

tut-install/examples/case-studies

5. 选择 dukes-tutoring 文件夹。
6. 选择 Open Required Projects（打开必要工程）复选框并单击 Open Project。

说明 第一次在 NetBeans 中打开 Duke 辅导应用时，会看到 Projects（工程）标签页中有一个错误符号。这是正常的，因为对应 Criteria API 查询的企业 bean 所用的元模型文件还没有生成。

---

7. 在 Projects（工程）标签页中，右键单击 dukes-tutoring 工程并选择 Build（构建）。

这个命令会创建一个 JDBC 安全域，名为 tutoringRealm，构建并打包 dukes-tutoring-common 和 dukes-tutoring-war 工程，并将 dukes-tutoring-war 部署到 GlassFish 服务器，启动 Java DB 数据库和 GlassFish 服务器（如果还未启动）。

29.4.1.2 使用 Maven 构建和部署 Duke 辅导应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 如果数据库服务器还未运行，按照 2.4 节中的说明启动该服务器。

3. 在一个终端窗口中导航至：

```
tut-install/examples/case-studies/dukes-tutoring/
```

4. 输入以下命令：

```
mvn install
```

这个命令会创建一个 JDBC 安全域，名为 tutoringRealm，构建并打包 dukes-tutoring-common 和 dukes-tutoring-war 工程，并将 dukes-tutoring-war 部署到 GlassFish 服务器。

29.4.1.3　使用 Duke 辅导应用

一旦 Duke 辅导应用在 GlassFish 服务器上运行，就可以使用主界面来检查学生的进出情况，或者让他们前往公园。

使用 Duke 辅导应用的主界面

1. 在一个 Web 浏览器中，访问以下 URL 打开主界面：

```
http://localhost:8080/dukes-tutoring-war/
```

2. 使用主界面来检查学生的进出，学生前往公园时记入日志。

使用 Duke 辅导应用的管理界面

按以下说明登录 Duke 辅导应用的管理界面，增加新的学生、监护人和地址。

1. 从主界面单击右边菜单中的 Administration main page（管理主页面），打开管理界面。这会重定向到登录页面，URL 如下：

```
http://localhost:8080/dukes-tutoring-war/admin/index.xhtml
```

2. 在这个登录页面上，在 User name（用户名）域中输入 admin@example.com，并在 Password（密码）域中输入 javaee。

3. 使用管理界面增加或修改学生、增加监护人，或者增加地址。

- 要增加一个新的学生，单击左边菜单中的 Create new student（创建新学生），在打开的表单中填写表单域（有两个必填域），并单击 Submit（提交）。Email、Home phone（家庭电话）和 Mobile phone（移动电话）域有格式要求，这些由 HTML5 直通约束或 bean 验证约束来保证。
- 要修改一个学生，单击学生名旁边的 Edit（编辑），修改所打开表单中的域，并单击 Submit（提交）。要编辑另一个学生，从页面上方下拉菜单中选择这个学生，并单击 Change student（更换学生）。
- 要删除一个学生，单击学生名旁边的 Remove（删除），然后单击所出现页面中的 Confirm（确认）。这个动作会从辅导班删除这个学生，但并不是从数据库中将这个学生删除。要把这个学生再次增加到辅导班，单击左边菜单中的 Activate student（激活学生），然后在出现的页面中单击学生名旁边的 Activate（激活）。

- 要为学生增加一个监护人，单击学生名旁的 Add guardian（增加监护人）。这会出现一个页面，其中显示学生的名字、可以指定的监护人以及这个学生的当前监护人（如果有）。要为这个学生增加一个已有的监护人，可以从列表中选择该监护人，并单击 Add guardian（增加监护人）。要为学生创建一个新监护人，需要填写相应的域，并单击 Submit（提交）。要删除一个学生的一个监护人，可以从列表中选择学生的一个前监护人，并单击 Remove guardian（删除监护人）。
- 要为学生增加一个地址，单击学生名旁边的 Add address（增加地址）。在出现的页面中，在表单中填写适当的域，然后单击 Submit（提交）。这里有 4 个必填域。

管理界面并没有完全实现。还不能编辑监护人，也不能查看或编辑地址，不过已经有提供这些特性的 Facelets 页面。另外这个应用没有使用 `PersonDetails` 实体中的属性，你可以自己修改应用来增加这些特性。

第 30 章 Chapter 30

Duke 综合案例研究示例

Duke 综合应用是一个简单的电子商务应用，包含多个 Web 应用，展示了多个 Java EE 7 API 的使用：

- JSF 技术，包括 Ajax
- Java EE 上下文和依赖注入（CDI）
- RESTful Web 服务 Java API（JAX-RS）
- Java 持久化 API（JPA）
- JavaBeans 验证 Java API（bean 验证）
- 企业 JavaBeans（EJB）技术
- Java 消息服务（JMS）

这个应用由以下工程组成：

- Duke 商店：这个 Web 应用包含一个产品目录、顾客自助注册和一个购物车。它还有一个管理界面，用于完成商品、类别和用户的管理。这个工程名为 `dukes-store`。
- Duke 货运：这个 Web 应用提供一个界面来完成订单货运管理。这个工程名为 `dukes-shipment`。
- Duke 支付：这个 Web 服务应用有一个 RESTful Web 服务，用于完成订单支付。这个工程名为 `dukes-payment`。
- Duke 资源：这是一个简单的 Java 归档工程，包含这些 web 工程使用的所有资源。它包括消息、CSS 样式表、图像、JavaScript 文件和 JSF 复合组件。这个工程名为 `dukes-resources`。
- Entities（实体）：这是一个简单的 Java 归档工程，包含所有 JPA 实体。使用这些实体

的其他工程会共享这个工程。这个工程名为 entities。

- Events（事件）：这是一个简单的 Java 归档工程，包含用作 CDI 事件的 POJO 类。这个工程名为 events。

本章将介绍以下内容：

- Duke 综合应用的设计和架构
- 构建和部署 Duke 综合案例研究应用
- 运行 Duke 综合应用

30.1 Duke 综合应用的设计和架构

Duke 综合应用是一个复杂的应用，由 3 个主工程和 3 个子工程组成。图 30-1 显示了要部署的 3 个主工程的架构：Duke 商店、Duke 货运和 Duke 支付。这里还显示了 Duke 商店如何利用 Events（事件）和 Entities（实体）工程。

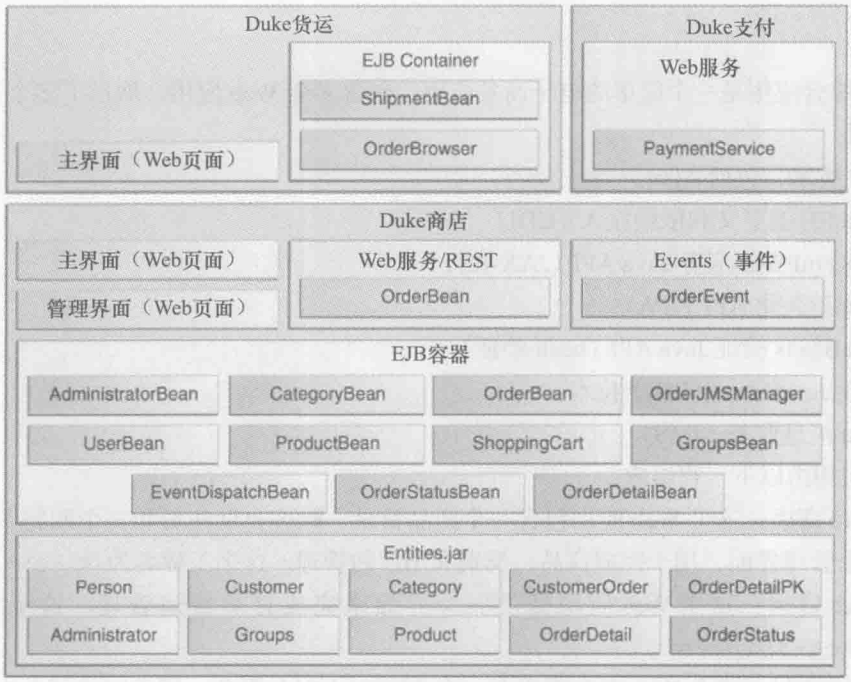


图 30-1 Duke 综合示例应用的架构

Duke 综合应用使用了以下 Java EE 7 平台特性：

- Java 持久化 API 实体
  - 在实体上增加 bean 验证注解来验证数据
  - 实现 Java XML 绑定（JAXB）串行化 API 的 XML 注解

- Web 服务
  - 一个用于支付的 JAX-RS Web 服务（有安全约束）
  - 一个基于 EJB 的 JAX-RS Web 服务
- 企业 bean
  - 本地会话 bean
  - 打包在 WAR 中的所有企业 bean
- 上下文和依赖注入（CDI）
  - JSF 组件的 CDI 注解
  - 一个用作购物车的 CDI 托管 bean（使用了对话作用域）
  - 限定符
  - 事件和事件处理器
- servlet
  - 一个用于动态图像表示的 servlet
- JSF 2.2 技术，使用 Facelets 作为 Web 前端
  - 模板
  - 复合组件
  - 文件上传
  - 资源打包在一个 JAR 文件中，从而可以在类路径上找到
- 安全
  - 管理界面业务方法上增加 Java EE 安全约束（企业 bean）
  - 顾客和管理员的安全约束（Web 组件）
  - 单点登录（Single Sign-On，SSO），将认证的用户身份从 Duke 商店传播到 Duke 货运

Duke 综合应用有两个主要的用户界面，都打包在 Duke 商店 WAR 文件中：

- 主界面，用于顾客和游客访问
- 管理界面，用于完成后台操作，如向商品目录增加新商品

Duke 货运应用还有一个由管理员访问的用户界面。

图 30-2 显示了这些 Web 应用与 Web 服务如何交互。

如图 30-2 所示，顾客与 Duke 商店主界面交互，而管理员与管理界面交互。两个界面都会访问一个外观（façade），它由托管 bean 和无状态会话 bean 组成，这些 bean 进一步与表示数据库表的实体交互。façade 还与访问 Duke 支付 Web 服务的 Web 服务 API 交互。一个订单支付通过审核时，Duke 商店将这个订单发送到一个 JMS 队列。管理员还与 Duke 货运的界面交互，可以通过 Duke 货运应用直接访问，或者从 Duke 商店的管理界面利用一个 Web 服务来访问。管理员批准可以为一个订单送货时，Duke 货运将使用 JMS 队列中的订单。

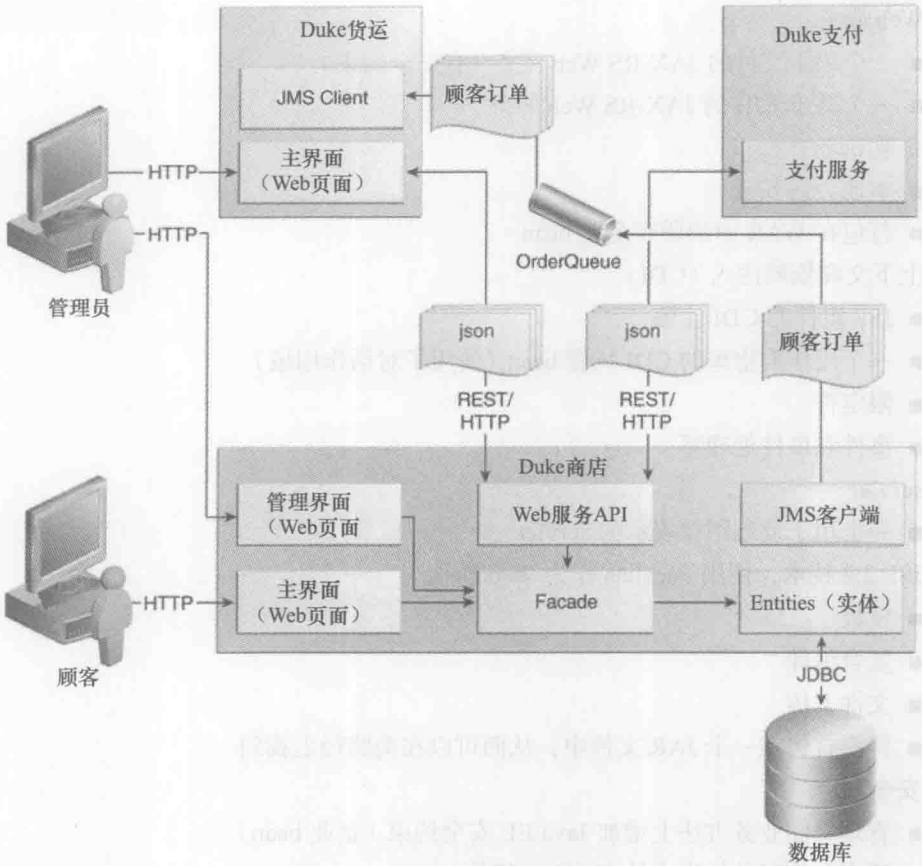


图 30-2 Duke 综合应用组件之间的交互

这个应用中最基础的构建模块是 Events（事件）和 Entities（实体）工程，它们与 Duke 资源工程一同打包到 Duke 商店和 Duke 货运。

30.1.1 events 工程

事件（event）是 Duke 综合应用的核心组件之一。所有 3 个主工程中都包含 `Events` 工程，这是应用中最简单的工程。它只有一个类 `OrderEvent`，不过这个类要负责应用中对象之间的大部分消息。

这个应用可以根据事件向不同组件发送消息，并根据事件的限定情况做出反应。应用支持以下限定符：

- `@LoggedIn`: 对应经过认证的用户
- `@New`: 购物车创建一个新订单时
- `@Paid`: 一个订单已经付费并准备送货时

下面是 Duke 商店 PaymentHandler 类中的一个代码段，显示了如何处理 @Paid 事件：

```
@Inject @Paid Event<OrderEvent> eventManager;

...
public void onNewOrder(@Observes @New OrderEvent event) {

    if (processPayment(event)) {
        orderBean.setOrderStatus(event.getOrderID(),
              String.valueOf(OrderBean.Status.PENDING_PAYMENT.getStatus()));
        logger.info("Payment Approved");
        eventManager.fire(event);
    } else {
        orderBean.setOrderStatus(event.getOrderID(),
              String.valueOf(OrderBean.Status.CANCELLED_PAYMENT.getStatus()));
        logger.info("Payment Denied");
    }
}
```

为了让用户轻松地向工程增加更多事件，或者允许为一个新客户端更新事件类来增加更多字段，这个组件要作为应用中一个单独的工程。

30.1.2　entities 工程

Entities 工程是一个 Java 持久化 API（JPA）工程，Duke 商店和 Duke 货运都使用了这个工程。它由图 30-3 所示的数据库模式生成，另外还用作实体的基础，Web 服务将通过 JAXB 使用和生成这些实体。根据业务需求，每个实体有一些验证规则，使用 bean 验证来指定。

这个数据库模式包含 8 个表：

- PERSON，与 PERSON_GROUPS 和 CUSTOMER_ORDER 有一对多关系
- GROUPS，与 PERSON_GROUPS 有一对多关系
- PERSON_GROUPS，与 PERSON 和 GROUPS 有多对一关系（这是这两个表之间的联接表）
- PRODUCT，与 CATEGORY 有多对一关系，另外与 ORDER_DETAIL 有一对多关系
- CATEGORY，与 PRODUCT 有一对多关系
- CUSTOMER_ORDER，与 ORDER_DETAIL 有一对多关系，并与 PERSON 和 ORDER_STATUS 有多对一关系
- ORDER_DETAIL，与 PRODUCT 和 CUSTOMER_ORDER 有多对一关系（这是这两个表之间的联接表）
- ORDER_STATUS，与 CUSTOMER_ORDER 有一对多关系

对应于这些表的实体类如下：

- Person，定义顾客和管理员共有的属性。这些属性包括人名和联系信息，联系信息包括街道和 email 地址。email 地址有一个 bean 验证注解，确保所提交的数据有正确的格式。为 Person 实体生成的表还有一个 DTYPE 字段，表示鉴别器列。这个字段

值标识这个人属于哪个子类（Customer 或 Administrator）。

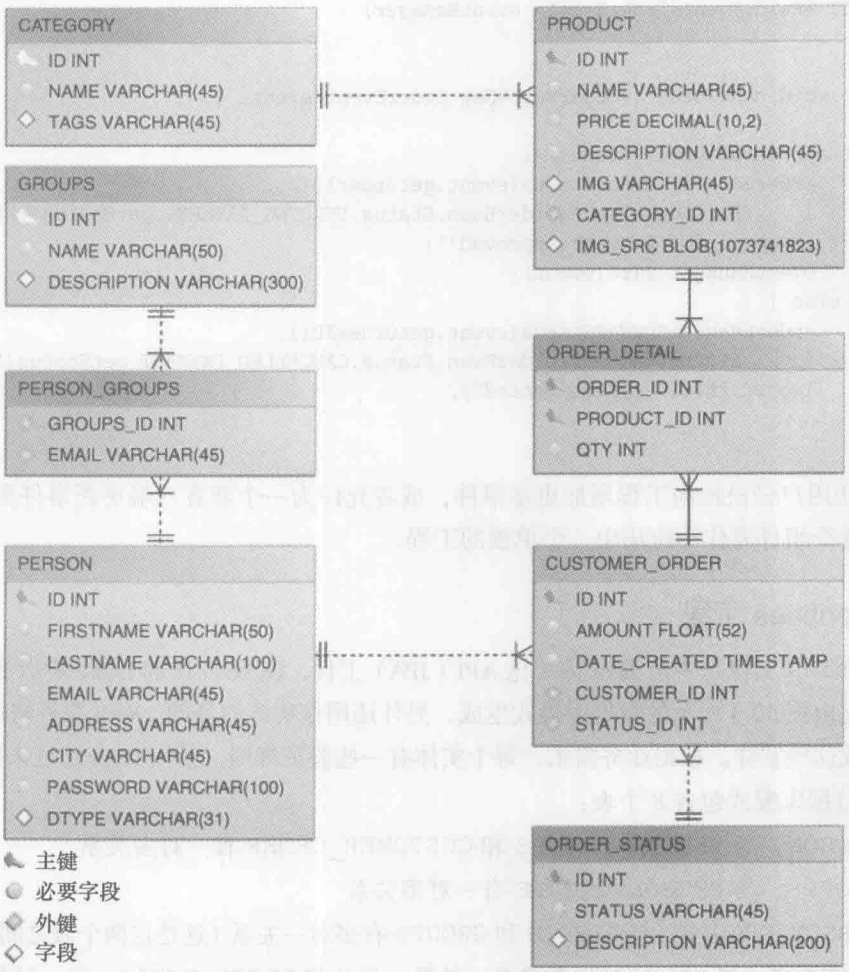


图 30-3 Duke 综合应用数据库表及其关系

- Customer，Person 的一个特殊化类，有一个对应 CustomerOrder 对象的特定字段。
- Administrator，Person 的一个特殊化类，有一些对应管理员权限的字段。
- Groups，表示用户所属的组（USERS 或 ADMINS）。
- Product，定义产品的属性。属性包括产品名、价格、描述、相关图片和类别。
- Category，定义产品类别的属性。属性包括类别名和一组标记。
- CustomerOrder，定义顾客所下订单的属性。属性包括数量和日期，以及顾客和订单明细的 ID 值。
- OrderDetail，定义订单明细的属性。属性包括产品的数量以及产品和顾客的 ID 值。
- OrderStatus，定义每个订单的状态属性。

30.1.3　dukes-payment 工程

dukes-payment 工程是一个 web 工程，包含一个简单的支付 Web 服务。由于这是一个示例应用，因此并不包含真实的信用卡信息，甚至没有顾客状态来验证支付情况。对现在来说，这个支付系统唯一的规则是拒绝所有超过 $1000 的订单。这个应用可以展示一种常见的情况：使用一个第三方支付服务来验证信用卡或银行支付。

这个工程使用 HTTP 基本认证和 JAAS（Java 认证和授权服务）向一个 JAX-RS Web 服务认证顾客。这个实现本身提供了一个简单的方法 processPayment，它接收一个 OrderEvent 来完成计算，然后批准或拒绝订单支付。这个方法将由 Duke 商店的结账过程调用。

30.1.4　dukes-resources 工程

dukes-resources 工程包含 Duke 商店和 Duke 货运使用的一些文件，打包在位于类路径的一个 JAR 文件中。这些资源放在 src/main/resources 目录下：

- META-INF/resources/css：两个样式表 default.css 和 jsfcrud.css
- META-INF/resources/img：工程使用的图像
- META-INF/resources/js：一个 JavaScript 文件 util.js
- META-INF/resources/util：工程使用的复合组件
- bundles/Bundle.properties：应用消息（英语）
- bundles/Bundle_es.properties：应用消息（西班牙语）
- ValidationMessages.properties：bean 验证消息（英语）
- ValidationMessages_es.properties：bean 验证消息（西班牙语）

30.1.5　Duke 商店工程

Duke 商店是一个 Web 应用，这是 Duke 综合应用的核心应用。它负责为顾客提供商店主界面，另外提供管理界面。

Duke 商店的主界面允许用户完成以下任务：

- 浏览产品目录
- 新顾客注册
- 向购物车增加产品
- 结账
- 查看订单状态

Duke 商店的管理界面允许管理员完成以下任务：

- 产品维护（创建、编辑、更新、删除）
- 类别维护（创建、编辑、更新、删除）

- 顾客维护（创建、编辑、更新、删除）
- 组维护（创建、编辑、更新、删除）

这个工程还使用无状态会话 bean 作为 façades 外观与 30.1.2 节中描述的 JPA 实体交互，另外使用 CDI 托管 bean 作为控制器完成与 Facelets 页面的交互。因此这个工程遵循 MVC（Model-View-Controller，模型－视图－控制器）模式，并对所有实体和页面应用相同的模式，如下例所示。

- AbstractFacade 是一个抽象类，接收一个 Type<T>，并为这个类型实现常用操作（CRUD：创建、读取、更新和删除），这里 <T> 是一个 JPA 实体。
- ProductBean 是一个无状态会话 bean，扩展了 AbstractFacade，将 Product 用作 Type<T>，并为 EntityManager 注入 PersistenceContext。这个 bean 实现了与 Product 实体交互或调用定制查询所需的定制方法。
- ProductController 是一个 CDI 托管 bean，与必要的企业 bean 和 Facelets 页面交互，可以控制数据如何显示。

ProductBean 开始部分如下：

```
@Stateless
public class ProductBean extends AbstractFacade<Product> {
    private static final Logger logger =
        Logger.getLogger(ProductBean.class.getCanonicalName());

    @PersistenceContext(unitName="forestPU")
    private EntityManager em;
@Override
protected EntityManager getEntityManager() {
    return em;
}
...
```

30.1.5.1 Duke 商店中使用的企业 bean

Duke 商店中使用的企业 bean 为应用提供业务逻辑，这些企业 bean 位于 com.forest.ejb 包。所有这些 bean 都是无状态会话 bean。

AbstractFacade 不是一个企业 bean，而是一个抽象类，为 Type<T> 实现常用操作，其中 <T> 是一个 JPA 实体。

大多数其他 bean 都扩展 AbstractFacade，注入 PersistenceContext，并实现所需的定制方法：

- AdministratorBean
- CategoryBean
- EventDispatcherBean
- GroupsBean
- OrderBean

- OrderDetailBean
- OrderJMSManager
- OrderStatusBean
- ProductBean
- ShoppingCart
- UserBean

尽管 ShoppingCart 类在 ejb 包中，但这是一个对话作用域（conversation scope）CDI 托管 bean，这意味着，请求信息会跨多个请求持久存储。另外，ShoppingCart 负责为顾客订单开始一个事件链，这会调用 dukes-payment 中的 RESTful Web 服务，并向 JMS 队列发布一个订单，如果支付成功则批准送货。

30.1.5.2　Duke 商店主界面中使用的 Facelets 文件

与其他案例研究示例类似，Duke 商店使用 Facelets 来显示用户界面。主界面使用大量 Facelets 页面来显示不同的区域。根据这些页面所处理的模块将它们分组到不同的目录。

- template.xhtml：模板文件，用于主界面和管理界面。首先完成一个浏览器检查，验证用户的浏览器是否支持 HTML 5，这是 Duke 综合应用要求的必要条件。它将屏幕划分为多个区域，并为各个区域指定相应的客户端页面。
- topbar.xhtml：对应屏幕上方登录区域的页面。
- top.xhtml：对应标题区域的页面。
- left.xhtml：对应左边栏的页面。
- index.xhtml：对应主屏幕内容的页面。
- login.xhtml：web.xml 中指定的登录页面。主登录界面在 topbar.xhtml 中提供，不过如果有登录错误则会出现这个页面。
- admin 目录：与管理界面相关的页面，参见 30.1.5.3 节。
- customer 目录：与顾客相关的页面（Create.xhtml、Edit.xhtml、List.xhtml、Profile.xhtml、View.xhtml）。
- order 目录：与订单相关的页面（Create.xhtml、List.xhtml、MyOrders.xhtml、View.xhtml）。
- orderDetail 目录：弹出式页面，允许用户查看一个订单的详细信息（View_popup.xhtml）。
- product 目录：与产品相关的页面（List.xhtml、ListCategory.xhtml、View.xhtml）。

30.1.5.3　Duke 商店管理界面中使用的 Facelets 文件

Duke 商店管理界面中的 Facelets 页面放在 web/admin 目录中：

- administrator 目录：与管理员管理相关的页面（Create.xhtml、Edit.xhtml、

List.xhtml、View.xhtml)

- category 目录：与产品目录管理相关的页面（Create.xhtml、Edit.xhtml、List.xhtml、View.xhtml）
- customer 目录：与顾客管理相关的页面（Create.xhtml、Edit.xhtml、List.xhtml、Profile.xhtml、View.xhtml）
- groups 目录：与组管理相关的页面（Create.xhtml、Edit.xhtml、List.xhtml、View.xhtml）
- order 目录：与订单管理相关的页面（Create.xhtml、Edit.xhtml、List.xhtml、View.xhtml）
- orderDetail 目录：弹出式页面，允许管理员查看订单的详细信息（View_popup.xhtml）
- product 目录：与产品管理相关的页面（Confirm.xhtml、Create.xhtml、Edit.xhtml、List.xhtml、View.xhtml）

30.1.5.4 Duke 商店中使用的托管 bean

Duke 商店使用了以下 CDI 托管 bean，与企业 bean 对应。这些 bean 位于 com.forest.web 包中：

- AdministratorController
- CategoryController
- CustomerController
- CustomerOrderController
- GroupsController
- OrderDetailController
- OrderStatusController
- ProductController
- UserController

30.1.5.5 Duke 商店中使用的辅助类

Duke 商店主界面中的 CDI 托管 bean 使用了以下辅助类，这些类在 com.forest.web.util 包中：

- AbstractPaginationHelper：这是一个抽象类，提供托管 bean 使用的方法
- ImageServlet：这是一个 servlet 类，从数据库获取图像内容并显示
- JsfUtil：这是用于 JSF 操作的类，如在 FacesContext 实例上将消息入队
- MD5Util：CustomerController 托管 bean 使用这个类为用户生成加密密码

30.1.5.6 Duke 商店中使用的限定符

Duke 商店定义了以下限定符，位于 com.forest.qualifiers 包：

- @LoggedIn: 限定用户为刚登录的用户
- @New: 限定订单为新订单
- @Paid: 限定订单已支付

30.1.5.7 Duke 商店中使用的事件处理器

对于打包在 events 工程中的 OrderEvent 类，Duke 商店定义了与这个类相关的事件处理器（参见 30.1.1 节）。这些事件处理器放在 com.forest.handlers 包中。

- IOrderHandler：IOrderHandler 接口定义了一个方法 onNewOrder，两个处理器类都要实现这个方法。
- PaymentHandler: ShoppingCart bean 触发一个限定为 @New 的 OrderEvent。PaymentHandler 的 onNewOrder 方法观察这些事件，拦截到这些事件时，会使用 Duke 支付 Web 服务处理支付。得到 Web 服务的成功响应之后，PaymentHandler 再次触发 OrderEvent，这一次限定为 @Paid。
- DeliveryHandler：DeliveryHandler 的 onNewOrder 方法观察限定为 @Paid 的 OrderEvent 对象（已支付订单并准备发货），将订单状态修改为 PENDING_SHIPMENT。管理员访问 Duke 货运应用时，它会调用订单服务（这是一个 RESTful Web 服务），并请求数据库中所有准备发货的订单。

30.1.5.8 Duke 商店中使用的部署描述文件

Duke 商店使用以下部署描述文件，这些文件位于 web/WEB-INF 目录：

- faces-config.xml：JSF 配置文件
- glassfish-web.xml：GlassFish 服务器特定的配置文件
- web.xml：Web 应用配置文件

30.1.6 Duke 货运工程

Duke 货运是一个 Web 应用，包含一个登录页面、一个主 Facelets 页面和另外一些对象。这个应用只能由管理员访问，会使用 JMS 队列的订单，并调用 Duke 商店提供的 RESTful Web 服务来更新订单状态。Duke 货运的主页面显示一个已经批准将要发货的订单列表以及一个已发货订单列表。管理员可以批准或拒绝订单发货。如果批准，就会为这个订单发货，它会出现在 Shipped（已发货）标题下。如果拒绝，这个订单会从页面上消失，而在顾客的订单列表中它会显示为已取消。

未发货列表上还有一个齿轮图标，这会向订单服务发出一个 Ajax 调用，将刷新列表而不会刷新整个页面。代码如下所示：

```
<h:commandLink>
    <h:graphicImage library="img" title="Check for new orders"
                    style="border:0px" name="refresh.png"/>
    <f:ajax execute="@form" render="@form" />
</h:commandLink>
```

30.1.6.1 Duke 货运中使用的企业 bean

Duke 货运中使用的 UserBean 无状态会话 bean 为应用提供业务逻辑，位于 com.forest.shipment.session 包中。

与 Duke 商店一样，Duke 货运使用了 AbstractFacade 类。这个类不是一个企业 bean，而是一个抽象类，实现了 Type<T> 的常用操作，这里 <T> 是一个 JPA 实体。

OrderBrowser 无状态会话 bean 位于 com.forest.shipment.ejb 包，它的一个方法可以浏览 JMS 订单队列，另外一个方法可以在管理员批准或拒绝订单发货之后使用一个订单消息。

30.1.6.2 Duke 货运中使用的 Facelets 文件

Duke 货运只有一个页面，所以它的 Facelets 文件比 Duke 商店的少得多。

- template.xhtml：模板文件，与 Duke 商店中的模板文件类似，首先完成一个浏览器检查，验证用户的浏览器是否支持 HTML 5，这是 Duke 综合应用要求的必要条件。它将屏幕划分为多个区域，并为各个区域指定相应的客户端页面。
- topbar.xhtml：对应屏幕上方登录区域的页面。
- top.xhtml：对应标题区域的页面。
- index.xhtml：对应初始主屏幕内容的页面。
- login.xhtml：web.xml 中指定的登录页面。主登录界面在 topbar.xhtml 中提供，不过如果有登录错误则会出现这个页面。
- admin/index.xhtml：对应认证后主屏幕内容的页面。

30.1.6.3 Duke 货运中使用的托管 bean

Duke 货运使用以下 CDI 托管 bean，位于 com.forest.shipment 包中：

- web.ShippingBean：相当于订单服务客户端的托管 bean
- web.UserController：对应于 UserBean 会话 bean 的托管 bean

30.1.6.4 Duke 货运中使用的辅助类

Duke 货运托管 bean 只使用了一个辅助类，在 com.forest.shipment.web.util 包中：

- JsfUtil：用于 JSF 操作的类，如在 FacesContext 实例上将消息入队

30.1.6.5 Duke 货运中使用的限定符

Duke 货运包含 30.1.5.6 节介绍的 @LoggedIn 限定符。

30.1.6.6 Duke 货运中使用的部署描述文件

Duke 货运使用了以下部署描述文件：

- faces-config.xml：JSF 配置文件
- glassfish-web.xml：GlassFish 服务器特定的配置文件
- web.xml：Web 应用配置文件

30.2 构建和部署 Duke 综合案例研究应用

可以使用 NetBeans IDE 或 Maven 构建和部署 Duke 综合应用。

30.2.1 使用 NetBeans IDE 构建和部署 Duke 综合应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）。
2. 从 File（文件）菜单选择 Open Project（打开工程）。
3. 在 Open Project 对话框中，导航至：

```
tut-install/examples/case-studies
```

4. 选择 dukes-forest 文件夹。
5. 选择 Open Required Projects（打开必要工程）复选框并单击 Open Project。
6. 在 Projects（工程）标签页中，右键单击 dukes-forest 工程并选择 Build（构建）。

这个任务会配置服务器，创建和填充数据库，构建所有子工程，把它们装配到 JAR 和 WAR 文件，然后部署 dukes-payment、dukes-store 和 dukes-shipment 应用。

为了配置服务器，这个任务创建一个 JDBC 安全域，名为 jdbcRealm，启用默认的主体 – 角色映射，并为 HTTP 服务启用单点登录。

30.2.2 使用 Maven 构建和部署 Duke 综合应用

1. 确保已启动 GlassFish 服务器（参见 2.2 节）以及数据库服务器（参见 2.4 节）。
2. 在一个终端窗口中，进入：

```
tut-install/examples/case-studies/dukes-forest/
```

3. 输入以下命令配置服务器，创建并填充数据库，构建所有子工程，把它们装配到 JAR 和 WAR 文件，然后部署 dukes-payment、dukes-store 和 dukes-shipment 应用。

```
mvn install
```

为了配置服务器，这个任务创建一个 JDBC 安全域，名为 jdbcRealm，启用默认的主体 – 角色映射，并为 HTTP 服务启用单点登录。

30.3 运行 Duke 综合应用

运行 Duke 综合应用包括以下任务：

- 注册为 Duke 商店的一个顾客
- 作为顾客购买商品
- 作为管理员批准或拒绝产品送货
- 作为管理员创建一个新产品、顾客、组或类别

30.3.1 注册为 Duke 商店顾客

1. 在 Web 浏览器中输入以下 URL：

```
http://localhost:8080/dukes-store
```

这会打开 Duke 综合应用的商店页面。

2. 单击页面上方的 Sign Up（注册）。

3. 填写表单域，然后单击 Save（保存）。

所有域都是必填域，另外 Password（密码）值长度必须至少为 7 个字符。

30.3.2 购买产品

1. 要作为所创建的用户登录，或者作为数据库中已有的两个用户之一登录，输入用户名和密码，并单击 Log In（登录）。

已有用户的用户名为 `jack@example.com` 和 `robert@example.com`，他们的密码都是 `1234`。

2. 单击左边栏上的 Products（产品）。

3. 在出现的页面中，单击某个类别（Plants、Food、Services 或 Tools）。

4. 选择一个产品，并单击 Add to Cart（加入购物车）。

只能订购其中一个产品，不过可以订购多个类别中的多个不同的产品。产品和累加值会出现在左边栏的购物车中。

5. 选择了产品后，单击 Checkout（结账）。

这会出现一个消息："Your order is being processed. Check the Orders page to see the status of your order."（您的订单已处理。请检查订单页面查看您的订单状态。）

6. 单击左边栏的 Orders（订单）来检查订单。

如果订单总金额超过 $1000，订单的状态则为 "Order cancelled"（取消订单），因为支付 Web 服务会拒绝超过这个上限的订单。否则，订单的状态为 "Ready to ship"（准备发货）。

7. 下订单之后，单击页面上方的 Logout（注销）。

30.3.3 批准产品发货

1. 作为管理员登录 Duke 商店。

你的用户名是 `admin@example.com`，密码是 `1234`。

主管理页面允许你查看类别、顾客、管理员、组、产品和订单，还可以创建除订单以外所有类型的新对象。

2. 在页面下方，单击 Approve Shipment（批准发货）。

这个动作会带你进入 Duke 货运应用，仍保持作为管理员登录。

3. 在 Pending（未发货）列表中，单击 Approve（批准）来批准一个订单，将它移到页面

的 Shipped（已发货）区域。

如果单击 Deny（拒绝），这个订单会从页面上消失。如果再作为一个顾客登录 Duke 商店，它会作为“已取消订单”出现在 Orders（订单）列表中。

要从 Duke 货运返回 Duke 商店，单击 Return to Duke’s Store“返回 Duke 商店”。

30.3.4　创建新产品

可以创建其他类型的对象以及产品。创建产品比其他创建过程更复杂，所以下面来介绍如何创建新产品。

1. 作为管理员登录 Duke 商店。
2. 在主管理页面上，单击 Create New Product（创建新产品）。
3. 在 Name（产品名）、Price（价格）和 Description（描述）域中输入值。
4. 选择一个类别，然后单击 Next（下一步）。
5. 在上传产品图片页面中，单击 Browse（浏览）使用文件选择器在你的文件系统上找到一个图片。
6. 单击 Next（下一步）。
7. 在下一个页面上，查看产品域，然后单击 Done（完成）。
8. 单击左边栏的 Products（产品），然后单击类别来验证这个产品确实已经增加。
9. 单击页面上方的 Administration（管理），返回到主管理页面，或者单击 Logout 注销。

封面
书名
版权
前言
目录
第一部分　引言／2
第1章　概述／2
1.1　Java EE 7平台新增特性／3
1.2　Java EE应用模型／3
1.3　分布式多层应用／4
1.3.1　安全／4
1.3.2　Java EE组件／5
1.3.3　Java EE客户端／6
1.3.4　Web组件／7
1.3.5　业务组件／8
1.3.6　企业信息系统层／8
1.4　Java EE容器／9
1.4.1　容器服务／9
1.4.2　容器类型／10
1.5　Web服务支持／11
1.5.1　XML／11
1.5.2　SOAP传输协议／12
1.5.3　WSDL标准格式／12
1.6　Java EE应用装配与部署／12
1.7　开发角色／13
1.7.1　Java EE产品提供商／13
1.7.2　工具提供商／13
1.7.3　应用组件提供商／13
1.7.4　应用装配人员／14
1.7.5　应用部署人员和管理员／14
1.8　Java EE 7 API／14
1.8.1　EJB技术／16
1.8.2　Java servlet技术／17
1.8.3　JSF技术／18
1.8.4　JSP技术／18
1.8.5　JSP标准标记库／19

1.8.6　Java持久化API／19
1.8.7　Java事务API／19
1.8.8　RESTful Web服务JavaAPI／19
1.8.9　托管bean／20
1.8.10　JavaEE的上下文和依赖注入／20
1.8.11　Java的依赖注入／20
1.8.12　bean验证／20
1.8.13　Java消息服务API／20
1.8.14　Java EE连接器架构／21
1.8.15　JavaMail API／21
1.8.16　Java容器授权契约／21
1.8.17　Java容器认证服务提供者接口／21
1.8.18　WebSocket Java API／22
1.8.19　JSON处理Java API／22
1.8.20　Java EE的并发工具／22
1.8.21　Java平台的批处理应用／22
1.9　Java平台标准版7的Java EE 7 API／23
1.9.1　Java数据库互连API／23
1.9.2　Java命名和目录接口API／23
1.9.3　JavaBeans激活框架／24
1.9.4　XML处理Java API／24
1.9.5　Java XML绑定架构／24
1.9.6　XML Web服务Java-API／24
1.9.7　带附件SOAP Java API／25
1.9.8　Java认证和授权服务／25
1.9.9　Java平台的公共注解／25
1.10　GlassFish服务器工具／25
第2章　使用教程示例／27
2.1　必要软件／27
2.1.1　Java EE 7软件开发包／28
2.1.2　Java平台标准版／28
2.1.3　Java EE 7教程组件／28
2.1.4　NetBeans IDE／29
2.1.5　Apache Maven／30
2.2　启动和停止GlassFish服务器／30
2.2.1　使用NetBeans IDE启动GlassFish服务器／30

2.2.2　使用NetBeans IDE停止GlassFish服务器／30
2.2.3　使用命令行启动GlassFish服务器／30
2.2.4　使用命令行停止GlassFish服务器／31
2.3　启动管理控制台／31
2.3.1　使用NetBeans IDE启动管理控制台／31
2.4　启动和停止Java DB服务器／32
2.4.1　使用NetBeans IDE启动数据库服务器／32
2.5　构建示例／32
2.6　教程示例目录结构／32
2.7　教程中的Java EE 7 Maven原型／33
2.7.1　安装教程Maven原型／33
2.8　获取教程最新更新／34
2.8.1　使用NetBeans IDE更新教程／34
2.8.2　使用命令行更新教程／34
2.9　调试Java EE应用／34
2.9.1　使用服务器日志／34
2.9.2　使用调试器／35
第二部分　企业bean／38
第3章　企业bean／38
3.1　什么是企业bean／38
3.1.1　企业bean的好处／39
3.1.2　何时使用企业bean／39
3.1.3　企业bean类型／39
3.2　什么是会话bean／39
3.2.1　会话bean类型／40
3.2.2　何时使用会话bean／41
3.3　什么是消息驱动bean／41
3.3.1　消息驱动bean与会话bean的区别／41
3.3.2　何时使用消息驱动bean／42
3.4　访问企业bean／42
3.4.1　在客户端使用企业bean／43
3.4.2　确定远程或本地访问／44
3.4.3　本地客户端／45
3.4.4　远程客户端／46
3.4.5　Web服务客户端／47
3.4.6　方法参数和访问／47

3.5　企业bean的内容／48
3.6　企业bean的命名约定／48
3.7　企业bean的生命周期／48
3.7.1　有状态会话bean的生命周期／48
3.7.2　无状态会话bean的生命周期／49
3.7.3　单例会话bean的生命周期／50
3.7.4　消息驱动bean的生命周期／50
3.8　有关企业bean的更多信息／51
第4章　企业bean入门／52
4.1　创建企业bean／52
4.1.1　编写企业bean类／53
4.1.2　创建converter Web客户端／53
4.1.3　运行converter示例／54
4.2　修改Java EE应用／55
4.2.1　修改类文件／55
第5章　运行企业bean示例／56
5.1　cart示例／56
5.1.1　业务接口／57
5.1.2　会话bean类／57
5.1.3　@Remove方法／61
5.1.4　辅助类／61
5.1.5　运行cart示例／61
5.2　一个单例会话bean示例：counter／62
5.2.1　创建一个单例会话bean／63
5.2.2　countei示例的架构／67
5.2.3　运行counter示例／69
5.3　一个Web服务示例：helloservice／69
5.3.1　Web服务端点实现类／69
5.3.2　无状态会话bean实现类／70
5.3.3　运行helloservice示例／71
5.4　使用定时器服务／72
5.4.1　创建基于日历的定时器表达式／72
5.4.2　编程定时器／74
5.4.3　自动定时器／76
5.4.4　取消和保存定时器／76
5.4.5　获得定时器信息／77

5.4.6　事务和定时器／77
5.4.7　timersession示例／77
5.4.8　运行timersession示例／80
5.5　处理异常／81
第6章　使用嵌入式企业bean容器／82
6.1　嵌入式企业bean容器概述／82
6.2　开发嵌入式企业bean应用／82
6.2.1　运行嵌入式应用／83
6.2.2　创建企业bean容器／83
6.2.3　查找会话bean引用／84
6.2.4　关闭企业bean容器／85
6.3　standalone示例应用／85
6.3.1　使用NetBeans IDE运行standalone示例应用／86
6.3.2　使用Maven运行standalone示例应用／87
第7章　在会话bean中使用异步方法调用／88
7.1　异步方法调用／88
7.1.1　创建异步业务方法／88
7.1.2　从企业bean客户端调用异步方法／89
7.2　async示例应用／90
7.2.1　async-war模块的架构／91
7.2.2　运行async示例／92
第三部分　持久化／96
第8章　Java持久化API介绍／96
8.1　实体／96
8.1.1　实体类的需求／97
8.1.2　实体类中的持久化字段和属性／97
8.1.3　实体的主键／101
8.1.4　实体关系中的多重性／103
8.1.5　实体关系中的方向／103
8.1.6　实体中的可嵌入类／105
8.2　实体继承／106
8.2.1　抽象实体／106
8.2.2　映射超类／106
8.2.3　非实体超类／107
8.2.4　实体继承映射策略／107
8.3　管理实体／109

8.3.1　EntityManager接口／109
8.3.2　持久化单元／113
8.4　查询实体／114
8.5　数据库模式创建／114
8.5.1　配置应用以创建或删除数据库表／115
8.5.2　使用SQL脚本加载数据／116
8.6　有关持久化的更多信息／116
第9章　运行持久化示例／118
9.1　order应用／118
9.1.1　order应用中的实体关系／119
9.1.2　order应用中的主键／121
9.1.3　实体映射到多个数据库表／125
9.1.4　order应用中的层叠操作／125
9.1.5　order应用中的BLOB和CLOB数据库类型／126
9.1.6　order应用中的时态类型／126
9.1.7　管理order应用的实体／127
9.1.8　运行order示例／129
9.2　roster应用／129
9.2.1　roster应用中的关系／130
9.2.2　roster应用中的实体继承／131
9.2.3　roster应用中的Criteria查询／132
9.2.4　roster应用中的自动表生成／134
9.2.5　运行roster示例／135
9.3　address-book应用／136
9.3.1　address-book中的bean验证约束／136
9.3.2　在address-book中为约束指定错误消息／137
9.3.3　验证JSF应用的联系信息输入／138
9.3.4　运行address-book示例／139
第10章　Java持久化查询语言／140
10.1　查询语言术语／140
10.2　使用Java持久化查询语言创建查询／141
10.2.1　查询中的命名参数／142
10.2.2　查询中的位置参数／142
10.3　简化的查询语言语法／142
10.3.1　选择语句／142
10.3.2　更新和删除语句／143

10.4　示例查询／143
10.4.1　简单查询／143
10.4.2　导航到相关实体的查询／144
10.4.3　包含其他条件表达式的查询／146
10.4.4　批处理更新和删除／147
10.5　完整的查询语言语法／148
10.5.1　BNF符号／148
10.5.2　Java持久化查询语言的BNF文法／148
10.5.3　FROM子句／152
10.5.4　路径表达式／157
10.5.5　WHERE子句／158
10.5.6　SELECT子句／165
10.5.7　ORDERBY子句／167
10.5.8　GROUP BY和HAVING子句／167
第11章　使用Criteria API创建查询／168
11.1　Criteria和Metamodel API概述／168
11.2　使用Metamodel API为实体类建模／170
11.2.1　使用元模型类／170
11.3　使用Criteria API和Metamodel API创建基本的类型安全查询／171
11.3.1　创建Criteria查询／171
11.3.2　查询根／172
11.3.3　使用联接查询关系／172
11.3.4　Criteria查询中的路径导航／173
11.3.5　限制Criteria查询结果／173
11.3.6　管理Criteria查询结果／175
11.3.7　执行查询／176
第12章　创建和使用基于字符串的Criteria查询／178
12.1　基于字符串的Criteria API查询概述／178
12.2　创建基于字符串的查询／178
12.3　执行基于字符串的查询／179
第13章　使用锁定控制对实体数据的并发访问／180
13.1　实体锁定和并发概述／180
13.1.1　使用乐观锁定／181
13.2　锁模式／181
13.2.1　设置锁模式／182
13.2.2　使用悲观锁定／183

第14章　用实体图创建获取计划／185
14.1　实体图基础／185
14.1.1　默认实体图／186
14.1.2　在持久化操作中使用实体图／186
14.2　使用命名实体图／187
14.2.1　对实体类应用命名实体图注解／187
14.2.2　从命名实体图获得EntityGraph实例／188
14.3　在查询操作中使用实体图／189
第15章　对Java持久化API应用使用二级缓存／190
15.1　二级缓存概述／190
15.1.1　控制实体是否可以缓存／191
15.2　指定缓存模式设置来提高性能／192
15.2.1　设置缓存获取和存储模式／192
15.2.2　通过编程方式控制二级缓存／194
第四部分　消息传送／198
第16章　Java消息服务概念／198
16.1　JMS API概述／198
16.1.1　什么是消息传送／198
16.1.2　什么是JMS API／199
16.1.3　何时使用JMS API／199
16.1.4　Java EE平台如何使用JMS API／200
16.2　基本JMS API概念／201
16.2.1　JMS API架构／201
16.2.2　消息传送方式／201
16.2.3　消息使用／203
16.3　JMS API编程模型／204
16.3.1　JMS受管理对象／205
16.3.2　连接／206
16.3.3　会话／206
16.3.4　JMSContext对象／207
16.3.5　JMS消息生产者／208
16.3.6　JMS消息消费者／208
16.3.7　JMS消息／213
16.3.8　JMS队列浏览器／215
16.3.9　JMS异常处理／216
16.4　使用高级JMS特性／216

16.4.1　控制消息确认／217
16.4.2　为发送消息指定选项／218
16.4.3　创建临时目的地／220
16.4.4　使用JMS本地事务／221
16.4.5　异步发送消息／223
16.5　在Java EE应用中使用JMS API／223
16.5.1　为Java EE应用创建资源／223
16.5.2　在企业bean或Web组件中使用资源注入／225
16.5.3　使用Java EE组件生成和同步接收消息／226
16.5.4　使用消息驱动bean异步接收消息／227
16.5.5　管理JTA事务／229
16.6　关于JMS的更多信息／231
第17章　Java消息服务示例／232
17.1　JMS示例概述／233
17.2　编写简单的JMS应用／233
17.2.1　启动JMS提供者／234
17.2.2　创建JMS受管理对象／234
17.2.3　构建所有简单示例／235
17.2.4　发送消息／235
17.2.5　同步接收消息／238
17.2.6　使用消息监听器完成异步消息传送／240
17.2.7　浏览队列中的消息／242
17.2.8　在同一个目的地运行多个消费者／245
17.2.9　确认消息／245
17.3　编写更高级的JMS应用／247
17.3.1　使用持久订阅／247
17.3.2　使用本地事务／249
17.4　编写高性能和可扩展的JMS应用／254
17.4.1　使用共享非持久订阅／254
17.4.2　使用共享持久订阅／256
17.5　使用一个简单Web应用发送和接收消息／257
17.5　1　websimplemessage Facelets页面／257
17.5.2　websimplemessage托管bean／258
17.5.3　运行websimplemessage示例／259
17.6　使用消息驱动bean异步接收消息／261
17.6.1　simplemessage示例概述／261

17.6.2　simplemessage应用客户端／261
17.6.3　simplemessage消息驱动bean类／262
17.6.4　运行simplemessage示例／263
17.7　从会话bean向MDB发送消息／264
17.7.1　为clientsessionmdb示例编写应用组件／265
17.7.2　运行clientsessionmdb示例／267
17.8　使用实体联接两个MDB的消息／268
17.8.1　clientmdbentity示例应用概述／269
17.8.2　为clientmdbentity示例编写应用组件／270
17.8.3　运行clientmdbentity示例／272
17.9　使用NetBeansIDE创建JMS资源／274
17.9.1　使用NetBeans IDE创建JMS资源／274
17.9.2　使用NetBeans IDE删除JMS资源／275
第五部分　安全／278
第18章　Java EE平台安全介绍／278
18.1　Java EE安全概述／278
18.1.1　简单的应用安全演示／279
18.1.2　安全机制特性／281
18.1.3　应用安全特点／281
18.2　安全机制／282
18.2.1　Java SE安全机制／282
18.2.2　Java EE安全机制／283
18.3　保护容器安全／285
18.3.1　使用注解指定安全信息／285
18.3.2　使用部署描述文件提供声明式安全／285
18.3.3　使用编程式安全／286
18.4　保护GlassFish服务器安全／286
18.5　使用安全域、用户、组和角色／286
18.5.1　什么是安全域、用户、组和角色／287
18.5.2　在GlassFish服务器中管理用户和组／289
18.5.3　建立安全角色／290
18.5.4　角色映射到用户和组／291
18.6　使用SSL建立安全连接／292
18.6.1　验证和配置SSL支持／293
18.7　有关安全的更多信息／293
第19章　Web应用安全入门／295

19.1　Web应用安全概述／295
19.2　保护Web应用安全／296
19.2.1　指定安全约束／297
19.2.2　指定认证机制／300
19.2.3　在部署描述文件中指定认证机制／302
19.2.4　声明安全角色／303
19.3　Web应用使用编程式安全／304
19.3.1　通过编程方式认证用户／304
19.3.2　通过编程方式检查调用者身份／306
19.3.3　编程式安全的示例代码／306
19.3.4　声明和链接角色引用／308
19.4　示例：保护Web应用安全／309
19.4.1　设置系统来运行安全示例／309
19.4.2　hello2-basicauth示例：对servlet使用基本认证／310
19.4.3　hellol-formauth示例：对JSF应用使用基于表单的认证／312
第20章　企业应用安全入门／316
20.1　企业应用的基本安全任务／316
20.2　保护企业bean安全／316
20.2.1　使用声明式安全保护企业bean安全／318
20.2.2　通过编程方式保护企业bean安全／321
20.2.3　传播安全身份（Run-As）／323
20.2.4　部署安全企业bean／324
20.3　示例：保护企业bear安全／324
20.3.1　cart-secure示例：使用声明式安全保护企业bean安全／324
20.3.2　converter-secure示例：使用编程式安全保护企业bean安全／328
第21章　JavaEE安全：高级主题／331
21.1　使用数字证书／331
21.1.1　创建服务器证书／332
21.1.2　向证书安全域增加用户／334
21.1.3　为GlassFish服务器使用一个不同的服务器证书／334
21.2　认证机制／335
21.2.1　客户端认证／335
21.2.2　相互认证／335
21.3　使用JDBC安全域完成用户认证／338
21.3.1　配置JDBC认证安全域／339
21.4　保护HTTP资源安全／340

21.5　保护应用客户端安全／343
21.5.1　使用登录模块／343
21.5.2　使用编程式登录／344
21.6　保护企业信息系统应用安全／344
21.6.1　容器托管登录／344
21.6.2　组件托管登录／345
21.6.3　配置资源适配器安全／345
21.6.4　将应用主体映射到EIS主体／346
21.7　使用部署描述文件配置安全／347
21.7.1　在部署描述文件中指定基本认证安全／347
21.7.2　在部署描述文件中指定非默认主体-角色映射／348
21.8　关于高级安全主题的更多信息／348
第六部分　Java EE支持技术／352
第22章　事务／352
22.1　Java EE应用中的事务／352
22.2　什么是事务／353
22.3　容器托管事务／353
22.3.1　事务属性／354
22.3.2　回滚容器托管事务／357
22.3.3　同步会话bean的实例变量／357
22.3.4　容器托管事务中不允许的方法／358
22.4　bean托管事务／358
22.4.1　JTA事务／358
22.4.2　不提交返回／359
22.4.3　bean托管事务中不允许的方法／359
22.5　事务超时／359
22.5.1　设置事务超时／359
22.6　更新多个数据库／360
22.7　Web组件中的事务／361
22.8　关于事务的更多信息／361
第23章　资源适配器和契约／362
23.1　什么是资源适配器／362
23.1.1　管理契约／363
23.1.2　通用工作上下文契约／364
23.1.3　出站和入站契约／364
23.2　元数据注解／365

23.3　公共客户端接口／366
23.4　对Java EE上下文和依赖注入（CDI）使用资源适配器／367
23.5　关于资源适配器的更多信息／368
第24章　资源适配器示例／369
24.1　trading示例／369
24.1.1　使用出站资源适配器／370
24.1.2　实现出站资源适配器／372
24.1.3　运行trading示例／373
24.2　traffic示例／374
24.2.1　使用入站资源适配器／375
24.2.2　实现入站资源适配器／376
24.2.3　运行traffic示例／378
第25章　使用Java EE拦截器／380
25.1　拦截器概述／380
25.1.1　拦截器类／381
25.1.2　拦截器生命周期／381
25.1.3　拦截器和CDI／381
25.2　使用拦截器／381
25.2.1　拦截方法调用／382
25.2.2　拦截生命周期回调事件／384
25.2.3　拦截超时事件／385
25.2.4　为组件绑定拦截器／386
25.2.5　拦截器排序／387
25.3　interceptor示例应用／388
25.3.1　运行interceptor示例／389
第26章　批处理／390
26.1　批处理介绍／391
26.1.1　批处理作业中的步骤／391
26.1.2　并行处理／392
26.1.3　状态和判定元素／392
26.1.4　批处理框架功能／393
26.2　Java EE中的批处理／394
26.2.1　批处理框架／394
26.2.2　创建批处理应用／394
26.2.3　批处理作业的元素／395
26.2.4　属性和参数／395

26.2.5　作业实例和作业执行／395
26.2.6　批处理和退出状态／395
26.3　简单用例／396
26.3.1　块步骤／397
26.3.2　任务步骤／399
26.4　使用作业规范语言／399
26.4.1　job元素／400
26.4.2　step元素／401
26.4.3　flow元素／406
26.4.4　split元素／406
26.4.5　decision元素／406
26.5　创建批处理工件／407
26.5.1　批处理工件接口／407
26.5.2　批处理工件中的依赖注入／409
26.5.3　从批处理运行时环境使用上下文对象／410
26.6　向批处理运行时环境提交作业／411
26.6.1　开始作业／411
26.6.2　检查作业的状态／411
26.6.3　在应用中调用批处理运行时环境／412
26.7　打包批处理应用／412
26.8　webserverlog示例应用／412
26.8.1　webserverlog示例应用架构／412
26.8.2　运行webserverlog示例应用／418
26.9　phonebilling示例应用／419
26.9.1　phonebilling示例应用架构／419
26.9.2　运行phonebilling示例应用／425
26.10　关于批处理的更多信息／426
第27章　Java EE的并发工具／427
27.1　并发基础／427
27.1.1　线程和进程／428
27.2　并发工具的主要组件／428
27.3　并发和事务／429
27.4　并发和安全／430
27.5　jobs并发示例／430
27.5.1　运行jobs示例／430
27.6　taskcreator并发示例／433

27.6.1　运行taskcreator示例／435
27.7　关于并发工具的更多信息／436
第七部分　案例研究／438
第28章　Duke书店案例研究示例／438
28.1　Duke书店的设计和架构／438
28.2　Duke书店接口／439
28.2.1　BookJava持久化API实体／439
28.2.2　Duke书店中使用的企业bean／440
28.2.3　Duke书店中使用的Facelets页面和托管bean／440
28.2.4　Duke书店中使用的定制组件和其他定制对象／441
28.2.5　Duke书店中使用的属性文件／442
28.2.6　Duke书店中使用的部署描述文件／443
28.3　运行Duke书店案例研究应用／443
28.3.1　使用NetBeans IDE构建和部署Duke书店／443
28.3.2　使用Maven构建和部署Duke书店／443
28.3.3　运行Duke书店应用／444
第29章　Duke辅导案例研究示例／445
29.1　Duke辅导应用的设计和架构／445
29.2　主界面／447
29.2.1　主界面中使用的Java持久化API实体／447
29.2.2　主界面中使用的企业bean／448
29.2.3　主界面中使用的WebSocket端点／448
29.2.4　主界面中使用的Facelets文件／448
29.2.5　主界面中使用的辅助类／449
29.2.6　属性文件／449
29.2.7　Duke辅导应用中使用的部署描述文件／450
29.3　管理界面／450
29.3.1　管理界面中使用的企业bean／450
29.3.2　管理界面中使用的Facelets文件／451
29.3.3　管理界面中使用的CDI托管bean／451
29.3.4　管理界面中使用的辅助类／451
29.4　运行Duke辅导案例研究应用／452
29.4.1　运行Duke辅导应用／452
第30章　Duke综合案例研究示例／455
30.1　Duke综合应用的设计和架构／456
30.1.1　events工程／458

30.1.2 entities工程／459
30.1.3 dukes-payment工程／461
30.1.4 dukes-resources工程／461
30.1.5 Duke商店工程／461
30.1.6 Duke货运工程／465
30.2 构建和部署Duke综合案例研究应用／467
30.2.1 使用NetBeans IDE构建和部署Duke综合应用／467
30.2.2 使用Maven构建和部署Duke综合应用／467
30.3 运行Duke综合应用／467
30.3.1 注册为Duke商店顾客／468
30.3.2 购买产品／468
30.3.3 批准产品发货／468
30.3.4 创建新产品／469