

前 言

本书基于GlassFish服务器开源版讲述如何在Java EE 6平台上开发企业应用。

Oracle GlassFish Server是一个Java EE兼容的应用服务器，它基于GlassFish服务器开源版开发而成。GlassFish服务器开源版是目前业界领先的开源项目和开放平台，可用来构建和部署下一代应用和服务。GlassFish服务器开源版是由GlassFish项目开源社区研发的，其官方网站为<http://glassfish.java.net>，它是第一个兼容Java EE 6平台规范的应用服务器，具备轻量、灵活和开源等特点，这使得企业不仅可以利用Java EE 6带来的新功能，还能通过更快捷的开发和部署流程在已有功能的基础上添加新功能。Oracle GlassFish Server又称GlassFish服务器商业版，它和GlassFish服务器开源版在本书中统称为GlassFish服务器。

前言内容

- ☐ 阅读须知
- ☐ Oracle GlassFish Server文档集
- ☐ 相关文档
- ☐ 符号约定
- ☐ 字体约定
- ☐ 默认路径和文件名方面的约定
- ☐ 文档、支持以及培训
- ☐ 查找Oracle产品文档
- ☐ 第三方网站参考内容

阅读须知

读者应当事先熟悉Java这门编程语言，如果并不熟悉，建议大家先通读Addison-Wesley在2006年出版、由Sharon Zakhour等人所著的*The Java Tutorial, Fourth Edition*。

Oracle GlassFish Server文档集

GlassFish服务器文档集讲述的是部署计划以及系统的安装。GlassFish服务器文档的网址是<http://docs.sun.com/coll/1343.13>。有关GlassFish服务器的介绍，可以按顺序参阅表1列出的文档。

表1 GlassFish服务器文档集里的文档

文 档 名	描述信息
Release Notes (版本说明)	提供有关软件和文档的最新信息,以表格的形式全面列出它所支持的硬件、操作系统、Java开发包(JDK)以及数据库的驱动程序等信息
Quick Start Guide (快速入门指南)	讲述如何使用GlassFish服务器
Installation Guide (安装指南)	讲述如何安装GlassFish服务器软件以及它的各个组件
Upgrade Guide (升级指南)	讲述如何升级到最新版本的GlassFish服务器,并介绍相邻版本之间的差异、配置选项的差异,这些差异可能会导致与产品规范不兼容
Administration Guide (管理指南)	讲述如何使用asadmin(1M)实用工具,以命令行的方式配置、监控和管理GlassFish服务器的各个子系统和组件。管理控制台(Administration Console)的在线帮助里介绍了如何执行这些任务
Application Deployment Guide (应用部署指南)	讲述如何把应用装配和部署到GlassFish服务器上,还提供部署描述文件的相关信息
Your First Gup: An Introduction to the Java EE Platform (Java EE平台简介)	为Java EE的初学者提供简短的教程,讲述开发简单的企业应用的完整过程。示例程序是一个Web程序,包括一个基于EJB规范的组件、一个基于JAX-RS的Web服务,以及一个用于Web前端的JavaServer Faces组件
Application Development Guide (应用开发指南)	讲述如何创建和实现运行在GlassFish服务器上的Java EE应用。这些应用遵循开放Java标准模型,使用的是Java EE组件以及API(Application Programming Interface,应用编程接口)。本指南还提供有关开发者工具、安全以及调试的信息
Add-On Component Development Guide (插件开发指南)	讲述如何使用GlassFish服务器已发布的接口为GlassFish服务器开发插件。本文档只讲述那些能让插件应用于GlassFish服务器的必要步骤
Embedded Server Guide (嵌入式服务器指南)	讲述如何在嵌入式GlassFish服务器环境下运行应用及如何开发嵌入GlassFish服务器的应用
Scripting Framework Guide (脚本框架指南)	讲述如何使用Ruby on Rails和Groovy on Grails等脚本语言为GlassFish服务器开发脚本应用
Troubleshooting Guide (疑难排解指南)	讲述使用GlassFish服务器过程中的常见问题及解决办法
Error Message Reference (出错消息参考)	讲述在使用GlassFish服务器过程中可能遇到的出错消息
Reference Manual (参考手册)	提供手册页(manual page)格式的参考信息,讲述GlassFish服务器的管理命令、实用工具命令以及相关的概念
Domain File Format Reference (域配置文件格式参考)	讲述GlassFish服务器的配置文件domain.xml的格式
Java EE 6 Tutorial (Java EE 6教程)	讲述如何使用Java EE 6平台技术和API开发Java EE应用
Message Queue Release Notes (消息队列版本说明)	讲述GlassFish消息队列的新特性、兼容性问题以及已知bug
Message Queue Administration Guide (消息队列管理指南)	讲述如何创建和管理消息队列系统
Message Queue Developer's Guide for JMX Clients (JMX客户端的消息队列开发指南)	讲述消息队列相关的API,以便开发人员在遵循JMX(Java Management Extensions, Java管理扩展)的前提下,通过编程的方式配置和监控消息队列的资源情况

相关文档

GlassFish服务器还提供了基于Javadoc工具的开发包参考文档，可以以如下方式访问。

- ❑ Java EE 6的API的规范位于<http://docs.oracle.com/javaee/6/api/>。
- ❑ GlassFish服务器 3.0.1版的API规范，包括Java EE 6 平台的包以及GlassFish服务器专用的、非平台的包，位于<http://glassfish.java.net/nonav/docs/v3/api/>。

此外，Java EE规范也有可能用得着，其网址为<http://www.oracle.com/technetwork/java/javaee/tech/index.html>。

关于如何用NetBeans这个IDE（Integrated Development Environment，集成开发环境）创建企业应用，参见<http://www.netbeans.org/kb/>。

关于如何为GlassFish服务器采用Java DB数据库，参见<http://www.oracle.com/technetwork/java/javadb/overview/index.html>。

GlassFish的示例项目包括许多应用，这些应用展示了广泛的Java EE技术。GlassFish示例项目和Java EE SDK打包到了一起，可以在GlassFish示例项目页<http://glassfish-samples.java.net/>获得。

符号约定

表2展示了本书中可能用到的符号。

表2 符号约定

符 号	描 述	例 子	意 义
[]	括号里是可选参数或者命令的选项	ls [-l]	-l选项不是必需的，可以不用
{ }	对于一个必选的命令选项，必须从括号里的几个备选项中选择一个	-d {y n}	-d选项要求必须用y或n作为参数
\${ }	指对变量的引用	\${com.sun.javaRoot}	表示com.sun.javaRoot这个变量的值
-	同时按下多个按键	Control+A	同时按下Ctrl键和A键
+	顺序按下多个按键	Ctrl+A+N	按下Ctrl键，然后松开，接着顺序按下A键和N键
→	指的是在图形用户界面里菜单项的选择	File→New→Templates	从File菜单里选择New，在New子菜单里选择Templates

字体约定

表3展示的是字体约定。

表3 字体约定

字 体	意 义	例 子
AaBbCc123	命令以及计算机屏幕上的输出信息	使用ls -a列出所有文件 machine_name% you have mail.

(续)

字 体	意 义	例 子
AaBbCc123	用户的输入用粗体，以区别于计算机的输出	machine_name% su Password:
AaBbCc123 楷体	占位符，会被真正的名字或者值替换掉 新的术语以及需要强调的重点术语（注意，有一些强调的项以粗体形式出现）	删除文件的命令是rm filename 缓存是指本地保存的一份数据副本 不要保存文件

默认路径和文件名方面的约定

表4列出了本书会用到的默认路径和文件名。

表4 默认路径和文件名

占 位 符	描述信息	默 认 值
as-install	表示GlassFish服务器的安装路径或者SDK的安装路径（如果GlassFish服务器打包至SDK中）	在Solaris操作系统、Linux操作系统或者Mac操作系统下时为： user's-home-directory/glassfishv3/glassfish 在Windows系统下时为： SystemDrive:\glassfishv3\glassfish
as-install-parent	表示GlassFish服务器安装路径的上一级路径	在Solaris操作系统、Linux操作系统或者Mac操作系统下时为： user's-home-directory/glassfishv3 Windows系统下时为： SystemDrive:\glassfishv3
tut-install	表示安装完GlassFish服务器或者SDK后运行Update工具时，“Java EE Tutorial”（Java EE教程）所在的路径	as-install/docs/javaee-tutorial
domain-root-dir	表示创建域（domain）的默认保存位置	as-install/domains/
domain-dir	保存域配置文件的地方 在配置文件里，domain-dir表示为： \$(com.sun.aas.instanceRoot)	domain-root-dir/domain-name

文档、支持以及培训

Oracle网站提供了如下一些辅助资源：

- ❑ 文档信息，参见<http://docs.sun.com/>；
- ❑ 支持信息，参见<http://www.sun.com/support/>；
- ❑ 培训信息，参见<http://education.oracle.com/>。

查找Oracle产品文档

除了可以在<http://docs.sun.com>网站查找Oracle产品的文档以外，还可以使用搜索引擎，在搜索框里按如下语法格式输入信息：

搜索关键词 **site:docs.sun.com**

举例来说，想查找“broker”的相关内容，就在搜索框里输入：

broker site:docs.sun.com

为了查找到Oracle其他网站的信息，比如说Oracle技术网络中的Java开发者网站（<http://www.oracle.com/technetwork/java/index.html>），就在输入框里把上述docs.sun.com替换成oracle.com。

第三方网站参考内容

本书会引用一些第三方网站的参考内容，主要是为了提供相关的辅助信息。

注意 Oracle不对本书所引用第三方网站的可访问性负责。对于第三方网站上及可通过其访问的任何内容、广告、产品以及其他材料，Oracle并不表示认可，也不对其负责。任何对上述内容（内容、产品或者服务）的使用，Oracle不对其导致的实际或者潜在的损害或损失负任何责任。

致 谢

Java EE教程团队想要向Java EE 规范的领导者致谢，他们是Roberto Chinnici、Bill Shannon、Kenneth Saks、Linda DeMichiel、Ed Burns、Roger Kitain、Ron Monzillo、Dhiru Pandey、SanKara Rao、Binod PG、Sivakumar Thygarajan、Kin-Man Chung、Jan Luehe、Jitendra Kotamraju、Marc Hadley、Paul Sandoz、Gavin King、Emmanuel Bernard、Rod Johnson、Bob Lee以及Rajiv Mordani。

我们还要向Java EE 6 SDK团队致谢，特别是Carla Carlson、Snjezana Sevo-Zenzerovic、Adam Leftik以及John Clingan。

JavaServer Faces技术以及Facelets章节从Jim Driscoll与Ryan Lubke的文档审阅以及贡献的代码中获益良多。

EJB技术、Java Persistence API以及Criteria API章节的内容来自于EJB和持久化团队所提供的丰富材料，包括Marina Vatkina和Mitesh Meswani的努力。

我们还要感谢Pete Muir对CDI章节的审阅工作，感谢Tim Quinn所提供的关于应用客户端容器的工作。同时要感谢NetBeans工程和文档团队，特别是Petr Jiricka、John Jullion-Ceccarelli以及Troy Giniupero。由于他们的帮助，这些代码示例才能用在NetBeans IDE里。

还要向我们的经理Alan Sommerer致谢，多谢他的支持和长期的激励。

我们还要向为本书绘制图形的Dwayne Wolff致谢，向修改图形的Jordan Douglas致谢。Julie Bettis做了大量的编辑工作，为本书的可读性以及章节设计的合理性提供了有力保证。Sheila Cepero在许多方面提供了帮助，帮忙理顺了本书的出版流程。Steve Cogorno 为我们的工具提供了非常宝贵的帮助。

最后，我们还要向Greg Doench、John Fuller、Vicki Rowland、Evelyn Pyle以及Addison-Wesley出版社的产品团队表示深深的谢意，是他们把篇幅如此之大、内容如此复杂的书稿变成了真正的出版物。

Part 1

第一部分

简 介

本部分简单介绍平台、教程和示例。

本 部 分 内 容

- 第 1 章 综述
- 第 2 章 使用教程示例

今天，开发人员越发认识到市场对分布性、事务性和移动性应用的需求，这种应用能发挥服务器端技术在速度、安全和可靠性方面的优势。企业业务运营需要企业应用作为支撑，而这些应用通常是集中管理的，并且经常需要与其他的企业软件进行协作。在信息技术时代，必须本着投资少、见效快、省资源的原则来设计、开发和构建企业应用。

随着Java EE（Java Platform, Enterprise Edition，Java平台企业版）的出现，基于Java的企业应用开发变得前所未有的简单，开发周期也明显缩短。Java EE的目标是为开发人员提供一组强大的API，以缩短开发周期，降低应用的复杂度，并改善应用的性能。

JCP（Java Community Process）负责Java EE的研发工作，管理所有Java相关技术。由相关各方组成的专家组编制JSR（Java Specification Request，Java规范请求），以此定义Java EE相关技术规范。JCP的工作就是确保Java技术标准满足稳定性和跨平台兼容性的要求。

Java EE平台使用简化的编程模型。XML部署描述文件不再是强制性的要求。取而代之的是，开发人员可以以注解的方式，在Java源文件中直接写入这些信息。Java EE服务器会在部署和运行的时候配置相关组件。注解（annotation）通常嵌入在代码中，以替代那些原先需要在部署描述文件中声明的对组件的配置信息。注解使得组件的逻辑和配置信息紧密地结合在一起，从而更加直观和便于阅读。

在Java EE平台中，可以通过依赖注入（dependency injection）提供组件需要的所有资源，因而不必通过应用程序代码创建和查找资源。依赖注入可以应用于EJB容器、Web容器和应用客户端。依赖注入允许Java EE容器以注解的方式自动插入对其他必要组件和资源的引用。

本书通过示例讲解了开发企业应用时可以使用的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，本书使用的示例和通俗的讲解都能为你创建自己的解决方案提供帮助。

如果你刚刚开始学习Java EE企业应用开发，本章是一个很好的起点。在这里可以学习基础开发知识，了解Java EE架构和API，熟悉重要的术语和概念，并且学会如何编写、装配和部署Java EE应用。

本章内容

- Java EE 6平台新特性
- Java EE应用模型

- ❑ 分布式多层应用
- ❑ Java EE容器
- ❑ Web 服务支持
- ❑ Java EE应用程序装配与部署
- ❑ 打包应用程序
- ❑ 开发中的角色分工
- ❑ Java EE 6 API
- ❑ Java标准版6.0中的Java EE 6 API
- ❑ GlassFish服务器工具

1.1 Java EE 6 平台新特性

Java EE 6平台最重要的目标是简化开发过程。Java EE 6以平台的方式，为不同类型的组件提供了一个通用的基础框架。开发人员开发效率的提升得益于更多的注解、更少的XML配置、更多的POJO（Plain Old Java Object，简单Java对象），以及简化的程序打包方式。Java EE 6平台包括如下新特性。

- ❑ 配置文件，它是为特定应用程序类型设定的Java EE平台配置。具体来说，Java EE 6为下一代Web应用程序引入了一种轻量级的Web Profile，为企业应用提供了包括全部Java EE技术、涵盖Java EE 6平台所有功能的Full Profile。
- ❑ 新技术，包括：
 - JAX-RS（构建REST式 Web服务的Java API）；
 - 托管Bean；
 - Java EE平台的上下文和依赖注入（JSR 299），又称CDI；
 - Java依赖注入（JSR 330）；
 - Bean 验证规范（JSR 303）；
 - JASPIC（Java Authentication Service Provider Interface for Containers，Java容器认证服务接口）。
- ❑ EJB（Enterprise JavaBeans，企业JavaBeans）组件新特性（参见1.9.1节）。
- ❑ Servlet新特性（参见1.9.2节）。
- ❑ JavaServer Faces组件新特性（参见1.9.3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，它们提供的应用可移植性强、安全性好、开发效率高，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服务应用，以便为对企业有要求或贡献的客户、员工、供应商、合作伙伴以及其他角色提供帮助。这些应用本

身复杂度高，涉及的数据源多，服务受众面广。

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位于模型的中间层。中间层是一种受到企业IT部门严格控制的运行环境。中间层通常运行在专用的服务器硬件上，拥有访问企业全部服务的权限。

Java EE应用模型定义了一种架构，可以实现企业级应用系统所需的，具备可扩展性、可操作性以及可管理性的多层次应用。这个模型将一个多层次的服务划分为以下几个部分：

- 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。

借助于Java EE平台，开发人员可以为多层服务开发所面临的复杂的系统级问题提供解决方案。

1.3 分布式多层应用

Java EE平台使用分布式多层应用模型构建企业应用。应用逻辑根据功能划分为不同的组件，而应用组件被安装至不同的主机。在主机上部署的原则取决于应用组件在Java EE多层环境中所在的层次。

在图1-1中，两个多层Java EE应用按照下面的方式被划分为多个层次。图1-1中展示的Java EE应用部分将在1.3.2节介绍。

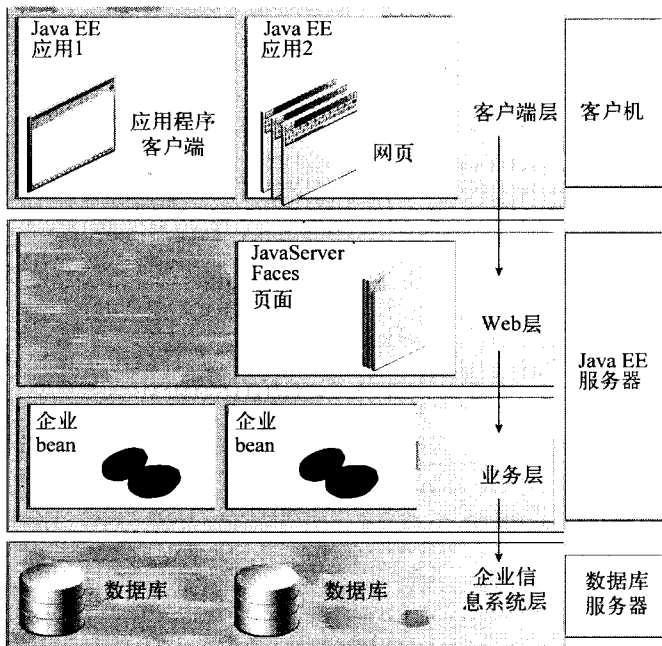


图1-1 多层应用

- 客户端层组件运行在客户机上。
- 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。
- EIS (Enterprise Information System, 企业信息系统) 层软件运行在EIS服务器上。

虽然如图1-1所示, 一个Java EE应用可能由3或4层构成, 但Java EE多层应用通常被认为是三层结构。其原因是这类应用通常部署在3个物理设备上: 客户端主机、Java EE服务器以及数据库服务器(或后端的遗留系统主机)。以这种方式运行的三层应用扩展了标准的两层应用模式(即客户机-服务器模式), 它在客户端应用与后端存储之间增加了一个支持多线程的应用服务器。

1.3.1 安全

基于其他企业应用模型开发的企业应用, 都需要考虑与特定平台相关的安全问题, 而Java EE的安全环境使得安全约束可在部署阶段才设定。Java EE平台让开发人员不用考虑安全性带来的复杂实现细节, 使得基于Java EE平台的应用移植性更好, 能够适应各种不同的安全场合。

Java EE平台提供了标准的声明式访问控制规则。这些规则是由开发人员定义的, 仅在将应用部署到服务器上的时候才会被解析。Java EE也提供了标准的登录机制(login mechanism), 因此应用开发人员无需在自己的应用中专门实现它。无需修改源代码, 同样的应用就可以在不同的安全环境下正常工作。

1.3.2 Java EE组件

Java EE应用由组件构成。Java EE组件是一个可以装配至Java EE应用中, 具备完整功能的软件单元。Java EE组件包括与之相关的类和文件, 可与其他组件通信。

Java EE规范定义了如下Java EE组件:

- 应用客户端以及运行于客户端上的小程序(applet);
- Java Servlet、JavaServer Faces、JavaServer Pages是运行于Java服务器上的Web组件;
- Enterprise JavaBeans(企业bean)是运行于服务器上的业务逻辑组件。

Java EE组件用Java语言编写而成, 以与该语言编写的其他程序一样的方式进行编译。Java EE组件与其他标准Java类的区别在于, Java EE组件将装配至一个Java EE应用中并通过验证, 以确保其满足Java EE规范的要求, 从而可以部署至生产环境中。生产环境是一个Java EE服务器运行并管理Java EE组件的地方。

1.3.3 Java EE客户端

Java EE客户端通常是Web客户端或应用程序客户端。

1. Web客户端

Web客户端由两部分构成:

- 包含不同标记语言（HTML、XML等）的动态网页，它们是由运行于Web层的Web组件生成的；
- Web浏览器，展示来自服务器的页面内容。

Web客户端通常称为瘦客户端。瘦客户端通常无需查询数据库、执行复杂的业务逻辑或连接过时的应用。当使用瘦客户端时，重量级的操作通常转移至Java EE服务器上的企业 bean执行，从而充分利用Java EE服务器端技术在安全、速度、服务性和可靠性上的优势。

2. 应用程序客户端

应用程序客户端运行在客户机上。它为用户提供了一种处理丰富人机交互任务的方式，而这种能力是标记语言无法提供的。应用程序客户端通常有基于Swing或AWT（Abstract Window Toolkit，抽象窗口工具包）API实现的GUI（Graphic User Interface，图形用户界面）。当然，也可能是命令行界面。

应用程序客户端直接访问运行于业务层的企业bean。然而，如果确有需要，应用程序客户端可以打开一个HTTP连接，以连接到运行于Web层的servlet。使用非Java的其他语言编写的应用程序客户端可以与Java EE服务器交互，这使得Java EE平台具备与遗留系统、客户端以及非Java语言交互的能力。

3. applet

Web层的网页中可以嵌入applet。用Java语言编写的applet是一种小型客户端程序，它能够运行于客户端浏览器上安装的Java虚拟机中。然而，客户端系统可能需要一个Java插件以及一个安全策略文件，才能使applet能够在Web浏览器中正常运行。

Web组件无需插件和安全策略文件，因而是创建Web客户端程序的首选API。同时，由于Web组件可将应用程序开发与网页设计相隔离，因此它使更加简洁和更加模块化的应用程序设计成为可能。因此，网页设计人员无需理解Java编程语言的语法即可完成其工作。

4. JavaBeans组件架构

服务器端与客户端也同样可包含基于JavaBeans架构的组件，以实现下面这些元素间数据流的管理：

- 应用程序客户端或applet与运行于Java EE服务器上的组件；
- 服务器端组件与数据库。

在Java EE规范中，JavaBeans未被认作是一种Java EE组件。

JavaBeans组件有特定的属性参数，并通过get和set方法进行访问和设置。以这种方式使用的JavaBeans组件通常设计和实现起来比较简单，但应该满足JavaBeans组件架构中定义的命名和设计规范。

5. Java EE服务器通信

图1-2展示了构成客户端层的不同元素。客户端可以直接与运行在Java EE服务器上的业务组件通信，或在客户端运行于浏览器中的前提下，通过网页或运行于Web层的servlet进行通信。

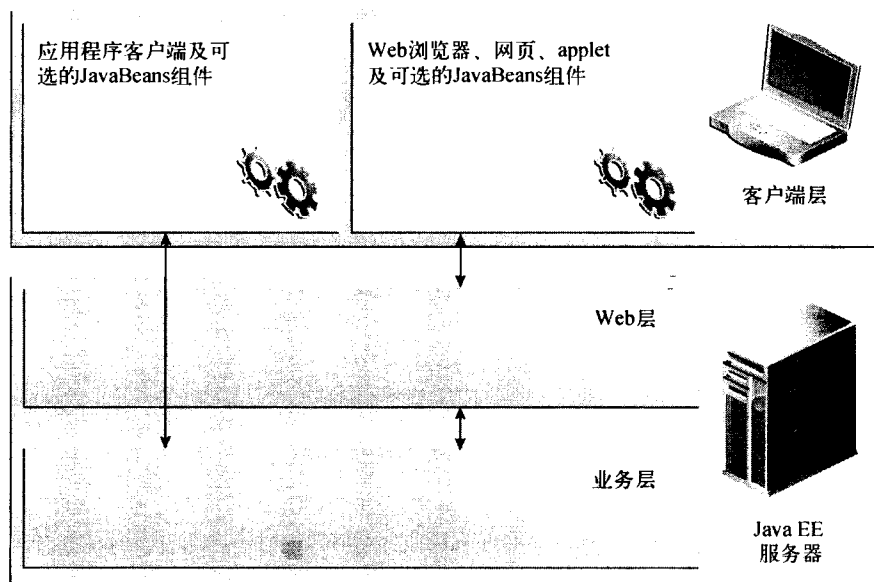


图1-2 服务器通信

1.3.4 Web组件

Java EE Web组件或是servlet，或是由JavaServer Faces和（或）JSP技术创建的网页。servlet是Java语言里的类，能够动态地处理请求并生成响应。JSP页面是基于文本的文档，能够以servlet的方式执行，且能以一种更加自然的方式创建静态内容。JavaServer Faces技术构建于servlet和JSP技术之上，为Web应用程序提供用户界面组件框架。

在应用程序装配时，静态HTML页面及applet将与Web组件打包在一起，但它们不被Java EE规范认定为Web组件。服务器端的工具类也可以与Web组件打包在一起，如同HTML页面一样，也不被认作是Web组件。

如图1-3所示，Web层（如同客户端层）也可包含JavaBeans组件，以管理用户的输入，并将其转发至运行于业务层的企业bean进行处理。

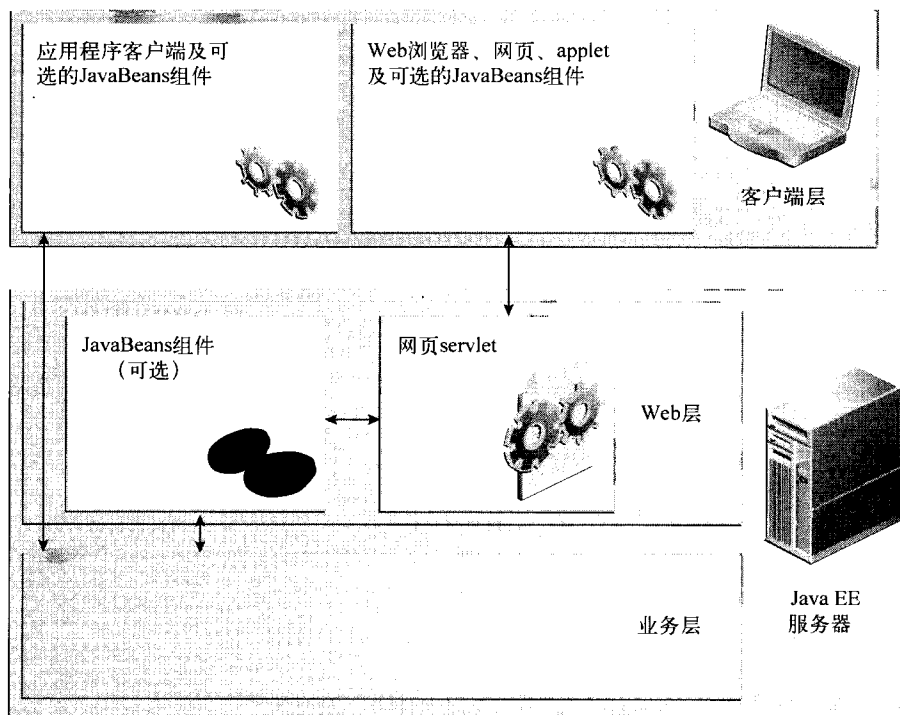


图1-3 Web层与Java EE应用程序

1.3.5 业务组件

业务代码（即业务逻辑）通常解决特定行业（如银行、零售、金融）中的业务问题，是由运行于业务层或Web层的企业bean处理的。图1-4展示了企业bean如何接收来自客户端程序的数据，进而处理（如果需要的话），并将其发送至企业信息系统中进行存储的完整过程。企业bean也从数据存储中获取数据，进而处理（如果需要的话），并将其送回至客户端程序。

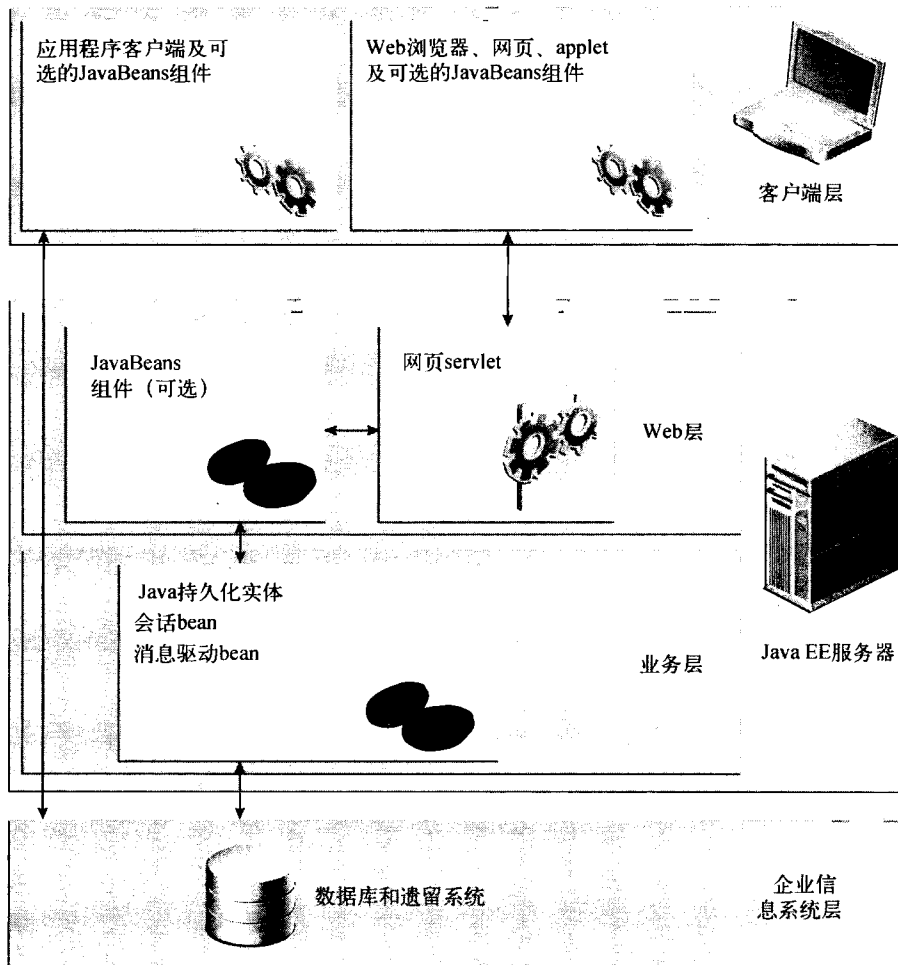


图1-4 业务和企业信息系统层

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软件),它包括企业基础设施系统,如ERP(Enterprise Resource Planning, 企业资源计划)、主机事务处理、数据库系统,以及其他遗留的信息系统。例如,Java EE应用的组件可能需要访问企业信息系统以获取数据库连接。

1.4 Java EE 容器

通常情况下,开发人员需要引入复杂的代码解决事务及状态管理问题,处理多线程和资源轮

询等复杂的底层细节，因此瘦客户端多层应用程序不易编写。而在Java EE架构下，由于业务逻辑被封装至可重用的组件之中，基于组件和平台独立的架构特性使得上述问题得以缓解。除此之外，Java EE服务器为每一种组件都提供了基于容器的底层服务，因此开发人员无需自行开发这些服务，可以轻松地将注意力集中在解决业务问题上。

1.4.1 容器服务

容器是组件与底层平台功能间的接口。在Web组件、企业bean或应用程序客户端运行之前，必须将其装配至Java EE模块中，并部署至其容器中。

装配的过程涉及为Java EE应用中的每个组件所对应的容器以及应用自身所对应的容器进行设定。容器的设定将明确Java EE服务器的底层支持方式，如安全服务、事务管理、JNDI（Java Naming and Directory Interface，Java命名与目录接口）API查询和远程连接等。以下是其中比较重要的几项。

- Java EE安全模型可用于配置Web组件或企业bean，从而使企业资源仅被授权用户访问。
- Java EE事务模型可用于指定一个完整事务中的多个过程间的关系，从而使得一次事务中的所有过程能够被认作是同一个处理单元。
- JNDI查询服务为企业中的不同命名和目录服务提供了统一的接口，从而使得应用程序组件可以轻松地访问这些服务。
- Java EE远程连接模型管理客户端与企业bean之间的底层通信服务。当一个企业bean创建之后，客户端调用bean中的方法，就如同两者在同一个虚拟机中。

由于Java EE架构提供可配置的服务，同一Java EE应用程序中的应用组件可以基于它们被部署的位置有不同的行为表现。例如，企业bean可以通过其安全性设置允许以某种级别的权限访问一个生产环境中的数据库数据，而以另一个级别的权限访问另一个生产环境中的数据库数据。

容器还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、数据持久性，以及对Java EE平台API的访问（参见1.9节）。

1.4.2 容器类型

如图1-5所示，在部署的过程中，Java EE应用程序组件将会安装到Java EE容器里。

- **Java EE服务器** Java EE产品的运行环境。Java EE服务器提供EJB和Web容器。
- **EJB（Enterprise JavaBeans）容器** 为Java EE应用程序管理企业bean的运行。企业bean及其容器运行在Java EE服务器上。
- **Web容器** 为Java EE应用程序管理网页、servlet和一些EJB组件的运行。Web组件及其容器运行在Java EE服务器上。
- **应用程序客户端容器** 管理应用程序客户端组件的运行。应用程序客户端及其容器运行于客户端。

□ applet容器 管理applet的运行，由运行于客户端的Web浏览器和Java插件组成。

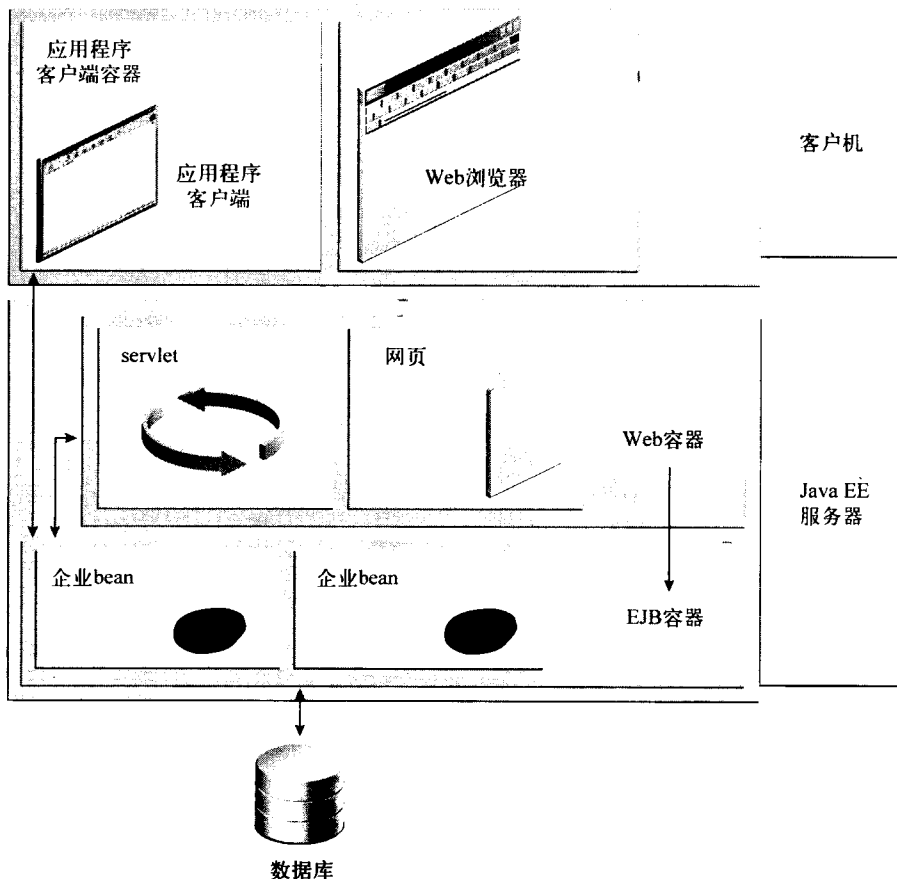


图1-5 Java EE服务器和容器

1.5 Web 服务支持

Web 服务是基于Web的企业应用程序，它使用开放的、基于XML的标准和传输协议与其客户端进行数据交换。我们需要使用Java EE平台提供的XML API及工具，以实现快速设计、开发、测试和部署完全可与其他Web服务及客户端互操作的Web服务及其客户端，而无论后者是否运行在Java平台上。

利用Java EE XML API开发Web 服务和客户端，开发人员只需为调用的方法传递参数并处理返回的数据。对于面向文档的Web服务，仅需要发送和接收包含数据的文档。XML API代为完成了在标准的XML传输协议中，应用数据与XML数据流间的翻译工作，因此无需编写底层代码。

基于XML的标准和协议将在随后的章节里介绍。

将数据翻译成标准的、基于XML的数据流，使得用Java EE XML API编写的Web服务和客户端间具有完全的互操作性。这种特性并不意味着传输的数据必须包括XML标签，也就是说，传输的数据可以是普通的文本、XML数据以及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（CAD）文档等。下面的内容将介绍XML，并解释不同的实体间如何通过XML标签和模式（schema），实现包含特定含义的数据交换。

1.5.1 XML

XML（Extensible Markup Language，可扩展标记语言）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签（tag）以描述数据，设定模式来说明哪些标签可以用于特定类型的XML文档中，并使用XML样式表管理数据的显示和处理方式。

例如，Web服务可以使用XML和一个模式生成价格清单，而收到价格清单和模式的公司可以自己定制XML样式表，以最能满足自己需要的方式处理数据。下面是一些XML的应用场景。

- ❑ 一家公司可以将以XML方式记录的报价信息通过一个程序转化为HTML格式，从而将价格信息公布在内部网络中。
- ❑ 一家合作伙伴公司可以将XML格式的报价信息通过工具转化为一个市场宣传材料。
- ❑ 另一家公司可以将XML格式的报价信息读入一个程序进行处理。

1.5.2 SOAP传输协议

客户端请求和Web服务响应作为SOAP（Simple Object Access Protocol，简单对象访问协议）的消息，通过HTTP协议进行传输。SOAP协议使得客户端与Web服务间的信息传递过程具备完全的互操作性，而无论消息传递的双方运行于什么样的平台上或位于因特网的哪个地方。HTTP协议是我们熟知的标准，它通过因特网以请求-响应模式传送数据。SOAP协议基于XML，参照HTTP的请求-响应模型。

传输消息中的SOAP部分包括如下内容：

- ❑ 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；
- ❑ 包括一系列基于XML的编码规则，描述消息内部应用程序定义的数据类型实例；
- ❑ 定义一个基于XML的规范，描述远程服务请求及结果响应的表示方式。

1.5.3 WSDL标准格式

WSDL（Web Services Description Language，Web服务描述语言）是用XML描述网络服务的标准格式。这种描述包括服务的名称、服务的位置以及与服务通信的方式。WSDL服务描述可以发布至网络中。GlassFish服务器提供了一个工具，可以生成Web服务的WSDL描述，并以远程过程调用的方式与客户端通信。

1.6 Java EE 应用程序装配与部署

我们可以将Java EE应用程序打包至一个或多个标准的单元中，以便部署至任何与Java EE平台兼容的系统中。每个单元包括：

- 一个或多个功能组件，如企业bean、网页、servlet或applet；
- 一个可选的、描述单元内容的部署描述文件。

一旦创建好Java EE单元，便可用于部署。部署时，通常使用平台提供的工具指定与位置有关的特定信息，如可以访问应用程序的本地用户列表及本地数据库名称。一经在本地平台部署，应用程序即可运行。

1.7 打包应用程序

一个Java EE应用程序可以以JAR（Java Archive）文件、WAR（Web Archive）文件或EAR（Enterprise Archive）文件3种形式交付。WAR和EAR文件是以.war或.ear为扩展名的标准JAR文件（.jar）。JAR、WAR和EAR文件及模块的使用，使得多个Java EE应用程序可共享一些组件。无需额外编码，而仅将各种不同的Java EE模块装配（打包）进JAR、WAR或EAR文件即可。

一个EAR文件（参见图1-6）包括Java EE模块以及一个可选的部署描述文件。部署描述文件是一个以.xml为扩展名的XML文档，用以描述应用程序、模块或组件的部署方式。由于部署描述文件是以声明的方式使用，因此无需修改源代码就可更改它的内容。在运行状态下，Java EE服务器读取部署描述文件，并按照其设定的方式执行应用程序、模块或组件。

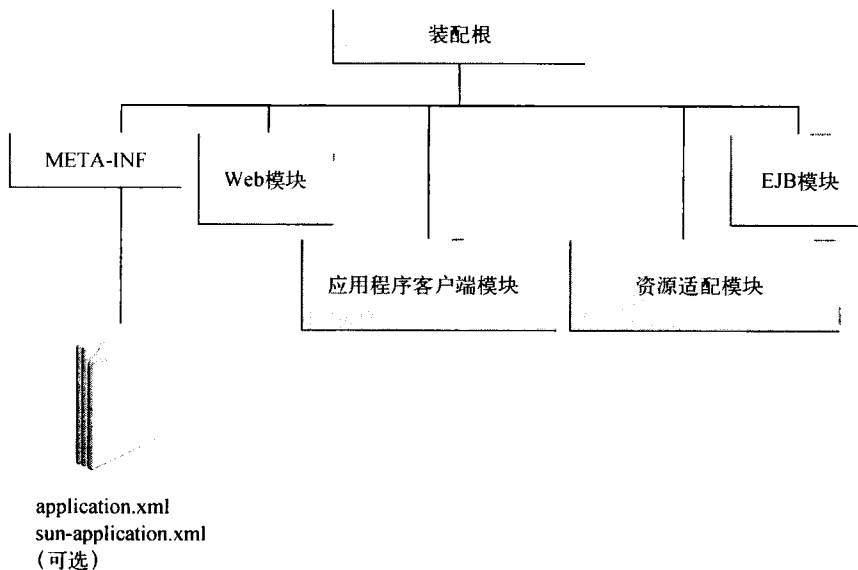


图1-6 EAR文件结构

部署描述文件有两种类型，分别是Java EE通用部署描述文件和服务器专用部署描述文件。Java EE通用部署描述文件在Java EE规范中定义，可用于在任何Java EE兼容的服务器实现上进行部署设置。服务器专用部署描述文件用于某个特定的Java EE服务器的参数配置。例如，GlassFish服务器专用部署描述文件包括Web应用程序的上下文环境以及GlassFish服务器的特定参数，如缓存指令。GlassFish服务器专用部署描述文件的文件名为`sun-moduleType.xml`，与Java EE部署描述文件一同位于META-INF目录中。

Java EE模块由一个或多个使用同种容器的Java EE组件，及一个可选的该类组件的部署描述文件构成。例如，企业bean组件模块部署描述文件声明了事务属性及安全认证机制。Java EE模块可以以单机方式部署。

Java EE模块包括如下类型。

- ❑ EJB模块，包含企业bean的类文件及一个部署描述文件。EJB模块打包至以`.jar`为扩展名的JAR文件中。
- ❑ Web模块，包含servlet类文件、Web文件、工具类文件、GIF文件、HTML文件以及一个Web应用程序部署描述文件。Web模块打包至以`.war`（web archive）为扩展名的JAR文件中。
- ❑ 应用程序客户端模块，包含类文件和应用程序客户端部署描述文件。应用程序客户端模块打包至以`.jar`为扩展名的JAR文件中。
- ❑ 资源适配模块，包含所有Java接口、类、本地库包，以及其他文档和资源适配部署描述文件。它们在一起，实现了特定企业信息系统的连接器架构（参见1.9.14节）。资源适配模块打包至以`.rar`（resource adapter archive）为扩展名的JAR文件中。

1.8 开发中的角色分工

模块可重用的特性实现了应用程序开发和部署过程两个阶段的角色划分，使得不同的人或公司能够在这个过程中发挥不同的作用。

前两个角色，即Java EE产品提供商和工具提供商，涉及购买和安装Java EE产品和工具。当软件购买并安装之后，Java EE组件可以由应用程序组件的开发人员、装配人员和部署人员，分别实现应用的开发、配置和部署。在大型的组织内部，每一种角色可能是由不同的个人或团队承担。这种工作划分的原则源于这样一个理念，即前期角色的输出物作为后续角色的输入。例如，在应用组件开发阶段，一名企业bean的软件开发人员将交付EJB JAR文件。作为应用程序装配角色，另一名软件开发人员可以将这些EJB的JAR文件合并至一个Java EE应用程序中，并保存为EAR文件。作为应用程序部署的角色，在客户现场的系统管理员使用EAR文件，将应用程序安装到Java EE服务器中。

不同角色的工作可能由同一个人来承担。例如，如果你为一家小型的公司工作，或者要为示例程序做原型开发，可能需要承担每个阶段的工作。

1.8.1 Java EE产品提供商

Java EE产品提供商设计和实现Java EE平台API及Java EE规范中定义的其他特性。产品提供商通常是按照Java EE 6平台规范，设计并实现应用服务器的开发厂商。

1.8.2 工具提供商

工具提供商是一些组织或个人，为组件提供商、编译人员和部署人员开发面向开发、编译和打包的工具。

1.8.3 应用组件提供商

应用组件提供商是一些组织或个人，为Java EE应用程序开发Web组件、企业bean、applet或应用程序客户端。

1. 企业Bean开发者

企业bean开发者完成如下任务，以交付一个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）；
- 打包.class文件和部署描述文件至EJB JAR文件中。

2. Web组件开发者

Web组件开发者完成如下任务，以交付一个包含一个或多个Web组件的WAR文件：

- 编写和编译servlet源代码；
- 编写JavaServer Faces、JSP和HTML文件；
- 指定部署描述文件（可选）；
- 打包.class文件、.jsp文件、.html文件及部署描述文件至WAR文件中。

3. 应用程序客户端开发者

应用程序客户端开发者完成如下任务，以交付一个包含应用程序客户端的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）；
- 打包.class文件及部署描述文件文件至JAR文件中。

1.8.4 应用程序装配者

应用程序装配者是一个组织或个人，接收来自组件提供商的应用程序模块，并且会将其装配至Java EE应用程序的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，或者利用工具以交互的方式正确地增加XML标签。

应用程序装配者完成如下工作，以交付一个包含Java EE应用程序的EAR文件：

- 装配前一阶段提交的EJB JAR文件和WAR文件至Java EE应用程序文件（EAR）中；

- ❑ 为Java EE应用程序指定部署描述文件（可选）；
- ❑ 验证EAR文件的内容是否正确地装配并满足Java EE规范。

1.8.5 应用程序部署者和管理员

应用程序部署者和管理员是一个组织或个人，配置和部署Java EE应用程序，管理Java EE应用程序运行的环境和网络资源并监督运行环境。职责包括设置事务控制和安全属性，以及指定到数据库的连接。

在配置的过程中，部署人员参照应用程序组件提供商提供的指导，解决外部依赖，指定安全设置及事务属性。在安装的过程中，部署人员将应用程序组件复制到服务器上，并生成特定容器的类和接口。

部署人员或系统管理员完成如下工作，以安装和配置Java EE应用程序：

- ❑ 为其所在的运行环境配置Java EE应用程序；
- ❑ 验证EAR文件的内容是否装配正确并满足Java EE规范；
- ❑ 部署（安装）Java EE应用程序的EAR文件至Java EE服务器。

1.9 Java EE 6 API

图1-7展示了Java EE容器间的关系。

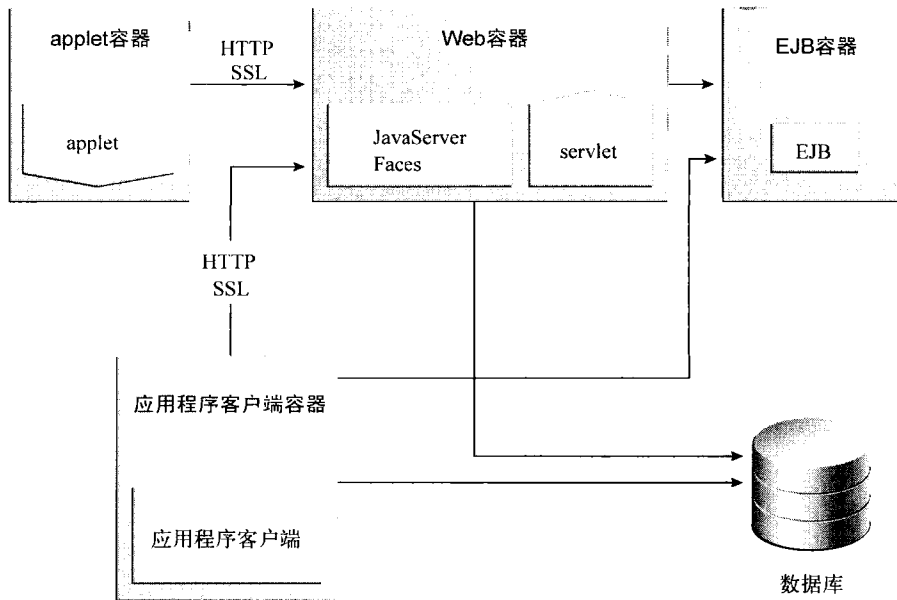


图1-7 Java EE容器

图1-8展示了Java EE 6中可用的Web容器API。

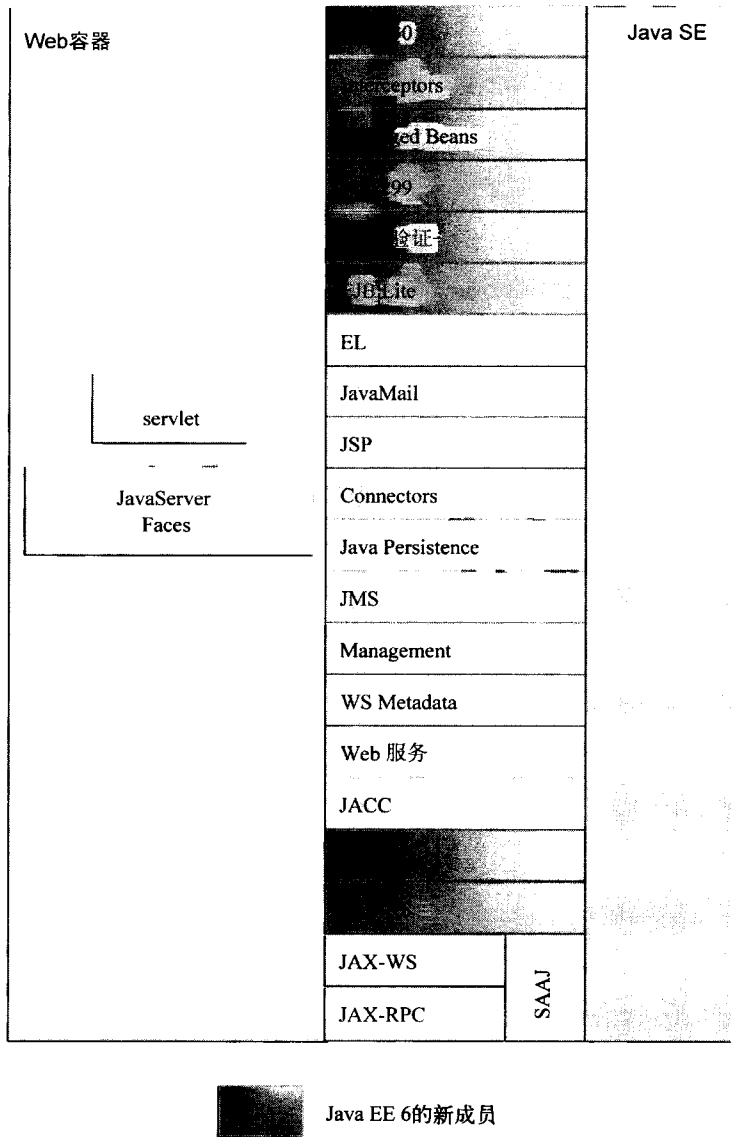
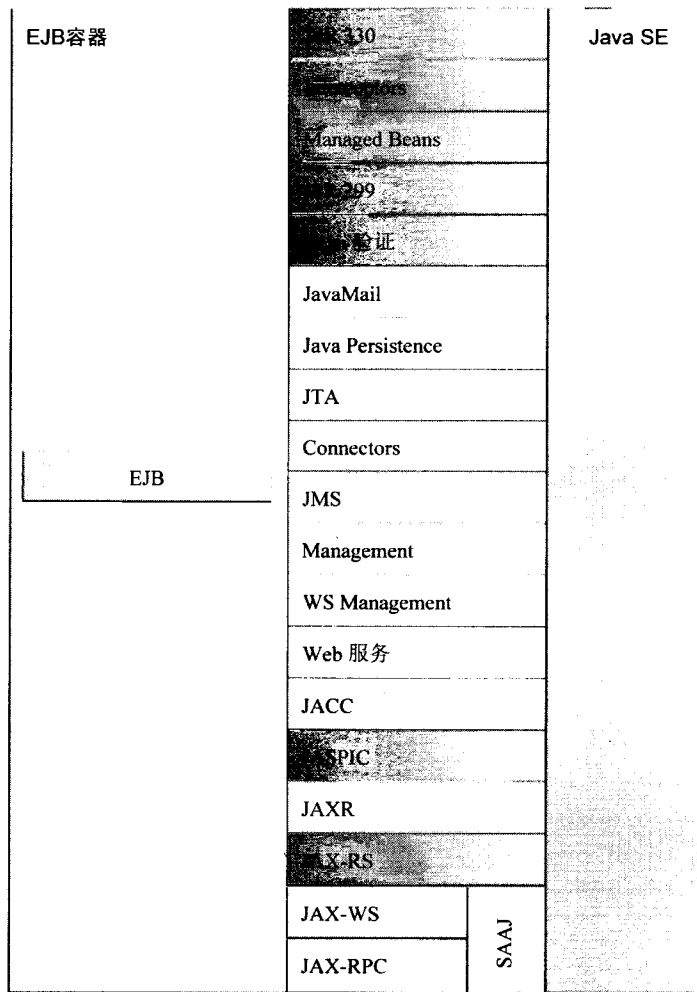


图1-8 Web容器中的Java EE API

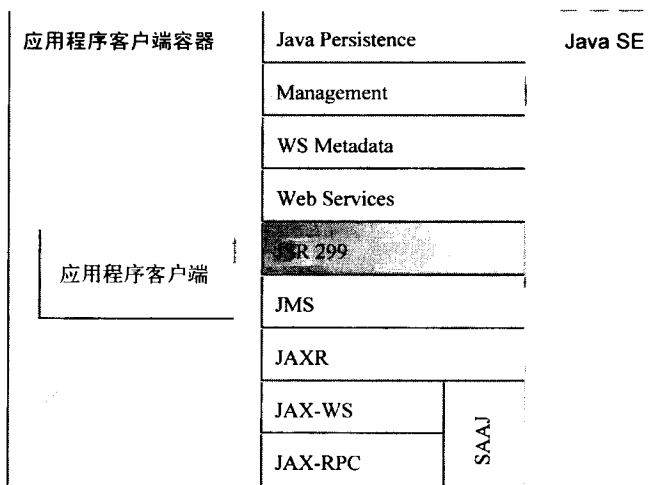
图1-9展示了Java EE 6中可用的EJB容器API。



Java EE 6的新成员

图1-9 EJB容器的Java EE API

图1-10展示了Java EE 6中可用的应用程序客户端容器API。
接下来的内容将对Java EE平台的相关技术和Java EE应用中使用的API进行简要介绍。



Java EE 6的新成员

图1-10 应用程序客户端容器中的Java EE API

1.9.1 企业JavaBeans技术

EJB（企业JavaBeans）组件，或称企业bean，是一段代码，包含了属性和方法，以实现业务逻辑。我们可以将企业bean看做建筑用的砖瓦，既可以单独使用，亦可与其他的企业bean一起在Java EE服务器上执行业务逻辑。

企业bean分为会话bean（session bean）和消息驱动bean（message-driven bean）两种。

- 会话bean代表服务器与客户端间的短暂会话。当客户端完成执行，会话bean及其数据将消失。
- 消息驱动bean结合了会话bean和消息监听器的特性，允许业务组件以异步的方式获取消息。这些消息通常是JMS（Java Message Service，Java消息服务）中生成的消息。

在Java EE 6平台中，企业bean包括如下新特性：

- 将本地企业bean打包至WAR文件中的能力；
- 单件模式会话bean，提供了轻松访问共享状态的能力；
- 在Java EE自定义规范（Profile）中提供轻量级的企业JavaBeans功能子集（EJB Lite），如Java EE Web自定义规范。

作为EJB 3.1规范中的一部分，拦截器规范使得在EJB 3.0中定义的拦截器工具应用范围更广。

1.9.2 Java Servlet技术

使用Java Servlet技术可以定义面向HTTP的servlet类。servlet类以请求-响应模式，扩展承载应用程序运行的服务器的能力。虽然servlet可以响应任何类型的请求，但通常用于扩展运行于Web服务器上的应用程序。

在Java EE 6平台中，Java Servlet技术包括如下新特性：

- 支持注解；
- 支持异步模式；
- 易配置；
- 对已有API进行增强；
- 即插即用能力。

1.9.3 JavaServer Faces技术

JavaServer Faces技术是构建Web应用程序用户界面的框架。JavaServer Faces技术包括如下主要组件。

- GUI组件框架。
- 具备在不同类型HTML、不同类型标记语言和技术中生成组件的灵活模型。Renderer对象产生生成组件的标记语言，并将存储于模型对象中的数据转换成可以被视图展示的类型。
- 产生HTML/4.01标记的标准RenderKit。

下述这些特性支持GUI组件：

- 输入校验；
- 事件处理；
- 模型对象与组件间的数据转换；
- 托管的模型对象创建；
- 页面导航配置；
- EL（Expression Language，表达式语言）。

这些功能均通过标准的Java API和基于XML的配置文件实现。

在Java EE 6平台中，JavaServer Faces的新特定包括：

- 使用注解代替配置文件设定托管bean；
- Facelets，一种使用XHTML文件替代JavaServer Pages（JSP）技术的数据表现技术；
- 支持Ajax；
- 复合（composite）组件；
- 隐式（implicit）导航。

1.9.4 JavaServer Pages技术

JSP（JavaServer Pages）技术允许将servlet的代码片段直接植入基于文本的文档中。JSP页面

是基于文本的文档，包括两种类型的文本：

- ❑ 静态数据，可以以任何基于文本的格式（如HTML或XML）展示数据；
- ❑ JSP元素，决定如何在页面中动态创建内容。

1.9.5 JavaServer Pages标准标签库

JSTL（JavaServer Pages Standard Tag Library，JavaServer Pages 标签库）封装了JSP应用程序常用的核心功能。取代JSP页面中来自不同厂商的混合标签，可以使用单一的、标准的标签集合。这种标准化确保能在任何支持JSTL的JSP容器中部署应用程序，并最大限度地使标签的实现得到优化。

JSTL有用于流程控制的迭代和条件控制标签、操作XML文档的标签、国际化标签、使用SQL访问数据库的标签，以及其他一些常用功能。

1.9.6 Java Persistence API

Java Persistence API是Java中基于标准的持久化解决方案。持久化使用对象/关系映射方法，架起面向对象模型与关系型数据库之间的桥梁。除了在Java EE环境中，Java Persistence API也可用在Java SE应用中。Java Persistence包括如下3个方面：

- ❑ Java Persistence API；
- ❑ 查询语言；
- ❑ 对象/关系映射元数据。

1.9.7 Java Transaction API

Java Transaction API，即JTA，提供了界定事务的标准接口。Java EE提供默认的自动提交功能，以处理提交和回滚。自动提交意味着正在查看数据的任何其他应用程序都将看到每一次数据库读写操作之后得到的最新数据。然而，如果应用程序执行了两个相互依赖但独立的数据库访问操作，可以使用JTA API实现完整事务的界定，包括两次操作的开始、回滚和提交。

1.9.8 支持REST式Web服务的Java API

支持JAX-RS(REST式 Web服务)的Java API定义了依据REST(Representational State Transfer)架构风格开发Web服务的API。JAX-RS应用程序是由类及所需库组成，并以WAR文件方式打包成servlet的Web应用程序。

JAX-RS API是Java EE 6平台的新成员。

1.9.9 Managed Beans

Managed Beans是一种轻量级的、由容器管理的对象（POJO）。它仅满足基本需求，支持少

量的基本服务，如资源注入、生命周期管理和拦截器。Managed Beans代表了JavaServer Faces技术所管理的托管bean的一般形态，可用于Java EE应用程序的各处，而不限于Web模块。

Managed Beans规范是Java EE 6平台规范（JSR 316）的一部分。

Managed Beans是Java EE 6平台的新成员。

1.9.10 Java EE平台（JSR 299）的上下文与依赖注入

Java EE平台的CDI（Contexts and Dependency Injection，上下文与依赖注入）定义了一系列由Java EE容器提供的上下文服务，使得开发人员在开发Web应用程序时可以轻松使用JavaServer Faces技术及企业bean。CDI针对有状态对象而设计，因而适用范围更广。CDI为开发人员提供了很大的自由度，从而使他们可以以松耦合但类型安全的方式集成不同类型的组件。

CDI是Java EE 6平台的新成员。

1.9.11 Java Dependency Injection（JSR 330）

Java Dependency Injection（依赖注入）定义了一系列标准的注解（和一个接口），用于可注入类。

在Java EE平台中，CDI提供了对Dependency Injection的支持。换句话说，仅能在支持CDI的应用程序中使用依赖注入点。

Dependency Injection是Java EE 6平台的新成员。

1.9.12 Bean Validation

Bean Validation（Bean验证）规范定义了用于验证JavaBeans组件中数据的元数据模型和API。取代在不同的层（如浏览器和服务端）上以分布式的方式验证数据，对于应用程序可以在一个地方定义验证限制规则，并在不同层间共享规则。

Bean Validation是Java EE 6平台的新成员。

1.9.13 Java Message Service API

JMS（Java Message Service，Java消息服务）API是一种消息标准，允许Java EE应用程序组件创建、发送、接收并读取消息。它使得松耦合、可靠、异步的分布式通信成为可能。

1.9.14 Java EE Connector架构

工具厂商和系统集成商使用Java EE Connector（连接器）架构，创建可以访问企业信息系统，并可植入任何Java EE产品中的资源适配器。资源适配器是一种软件组件，允许Java EE应用程序组件访问企业信息系统的底层资源管理器并与之交互。由于资源适配器面向特定的资源管理器，因此每一种数据库或企业信息系统都有专有的资源适配器。

Java EE Connector架构还提供基于Java EE的Web 服务与已有企业信息系统间的应用集成。该集成经过性能的优化,具备安全性和可扩展能力,并以消息的方式实现。应用集成支持同步和异步两种方式。已有应用程序,以及通过Java EE Connector架构集成至Java EE平台的企业信息系统,可以使用JAX-WS和Java EE组件模型,对外发布为基于XML的Web 服务。因此,JAX-WS和Java EEConnector架构是实现EAI (Enterprise Application Integration, 企业应用集成)和端到端业务集成的一组相互补充的技术。

1.9.15 JavaMail API

Java EE应用程序使用JavaMailAPI发送电子邮件。JavaMail API由两部分构成:

- 应用程序组件发送邮件时使用的应用程序级别的接口;
- 服务提供商接口。

Java EE平台集成了服务提供商提供的JavaMail API,使得应用组件可以通过因特网发送电子邮件。

1.9.16 Java容器授权合约

JACC (Java Authorization Contract for Containers, Java容器授权合约)规范定义了Java EE应用程序服务器与授权策略提供者间的合约。所有Java EE容器均支持这个合约。

JACC规范定义了一组满足Java EE授权模型要求的`java.security.Permission`类,定义了容器访问策略与这些权限类的实例之间的绑定关系。这一规范还定义了策略提供者的语法规则,从而使它们可以使用新的权限类去满足Java EE平台的授权需求,包括定义和使用角色。

1.9.17 Java容器认证服务提供商接口

JASPIC (Java Authentication Service Provider Interface for Containers, Java容器认证服务提供商接口)规范定义了一个SPI (Service Provider Interface, 服务提供商接口)。通过这个接口,实现消息认证机制的认证提供商可以集成至客户端或服务器端消息处理容器或运行环境。通过该接口集成的认证提供商可以操作网络消息,而消息来源于调用它们的容器。认证提供商转换外发消息,以使消息源能够被接收方的容器认证,同时消息的接收者可以被消息的发送者认证。认证提供商认证接收到的消息,并将认证建立时产生的标识作为消息认证结果返回给调用方的容器。

JASPIC是Java EE 6平台的新成员。

1.10 Java 标准版 6.0 中的 Java EE 6 API

Java EE 6平台中所需的一些API也包含在Java标准版6.0 (Java SE 6)平台中,并因此可供Java EE应用程序使用。

1.10.1 Java Database Connectivity API

JDBC (Java Database Connectivity, Java数据库连接) API使应用可以从Java编程语言的方法中调用SQL命令。当有一个会话bean访问数据库时,可以在企业bean中使用JDBC API。我们还可以在servlet或JSP页面中使用JDBC API直接访问数据库,而无需经由企业bean。

JDBC API由两部分构成:

- ❑ 应用程序组件访问数据库时使用的程序级接口;
- ❑ 为Java EE平台提供JDBC驱动的服务提供商接口。

1.10.2 Java命名和目录接口API

JNDI (Java Naming and Directory Interface, Java命名和目录接口) API提供了命名和目录访问功能,使得应用程序可以访问不同的命名和目录服务,包括已有的命名和目录服务,如LDAP、NDS、DNS和NIS。JNDI API为应用程序提供执行标准目录操作的方法,如将对象与属性关联以及使用属性查找对象。通过使用JNDI,Java EE应用程序可以存储和获取任何类型的、已命名的Java对象,从而使Java EE应用程序与其他遗留应用和系统共存。

Java EE命名服务为应用程序客户端、企业bean和Web组件提供了对JNDI命名环境的访问。命名环境 (naming environment) 使得我们不需要拿到或更改源代码就可以实现组件的客户化。容器实现了组件的环境,并将其以JNDI命名上下文的形式提供给组件。

Java EE组件可以通过使用JNDI接口定位其运行环境的命名上下文。组件可以创建 `javax.naming.InitialContext` 对象,并在 `java:comp/env` 下的 `InitialContext` 中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,或任何直接或间接的上下文子环境中。

Java EE组件可以访问具名的系统对象以及用户定义的对象。系统对象的名称,如JTA `UserTransaction` 对象,存储在环境命名上下文 `java:comp/env` 中。Java EE平台允许组件命名用户定义的对象,如企业bean、环境条目、JDBC `DataSource` 对象和消息连接。一个对象应该根据其类型,在命名环境上下文子环境中进行命名。例如,企业bean应该在环境 `java:comp/env/ejb` 中进行命名,而JDBC `DataSource` 引用则应该在子环境 `java:comp/env/jdbc` 中命名。

1.10.3 JavaBeans Activation Framework

JAF (JavaBeans Activation Framework, JavaBeans激活框架) 用于Java邮件API。JAF提供标准的服务,确定数据类型并封装对任意类型数据的访问操作,找到适用于这些数据的操作并创建合适的JavaBeans组件以实现这些操作。

1.10.4 Java XML API

JAXP (Java API for XML Processing) 作为Java标准版平台中的一部分,支持使用DOM (Document Object Model, 文档对象模型)、SAX (简单XML API) 和XSLT (Extensible Stylesheet

Language Transformations, 扩展样式表转换语言) 处理XML文档。JAXP能使应用程序以一种与特定XML处理实现机制无关的方式解析和转换XML文档。

JAXP还支持命名空间, 这使得应用可以使用可能存在命名冲突(若不使用命名空间的话)的模式。JAXP以灵活性为设计思想, 使我们可以在应用程序中使用任何与XML兼容的解析器或者XSL处理器, 并且支持W3C(WorldWide Web Consortium, 万维网联盟)模式(详见<http://www.w3.org/XML/Schema>)。

1.10.5 Java XML绑定架构

JAXB(Java Architecture for XML Binding, Java XML绑定架构) 提供一种方便的方式, 用于将XML模式与Java语言程序中的对象绑定在一起。JAXB可以单独使用, 也可与JAX-WS一起使用, 此时它为Web 服务消息提供一种标准的数据绑定方法。所有的Java EE应用程序客户端容器、Web容器和EJB容器均支持JAXB API。

1.10.6 支持带附件的SOAP消息API

SAAJ(SOAP with Attachments API for Java, 带附件的SOAP消息API) 是JAX-WS所依赖的底层API。遵从SOAP 1.1和1.2规范, 或带附件的SOAP协议的消息, 可以通过SAAJ创建并读取。大多数开发人员无需使用SAAJ API, 而是使用更高层的JAX-WS API。

1.10.7 基于XML的Java Web Services API

JAX-WS(Java API for XML Web Services) 规范提供了对使用JAXB API绑定XML数据到Java对象的Web 服务的支持。JAX-WS规范定义了访问Web 服务及实现Web 服务端点的技术的客户端API。Implementing Enterprise Web Services(实现企业Web 服务) 规范描述了基于JAX-WS的服务和客户端的部署方式。EJB和Java Servlet规范也涉及对部署方式的介绍。故而, 基于JAX-WS的应用必然可使用上述规范中定义的某种部署方式进行部署。

JAX-WS规范描述了对处理消息请求和响应的消息处理程序的支持。通常来说, 这些消息处理程序在相同的容器中执行, 并且作为相关的JAX-WS客户端与端点组件, 拥有相同的权限和执行上下文。这些消息处理程序拥有与其关联组件访问相同的JNDI `java:comp/env`命名空间的权限。如果支持定制的序列化与反序列化程序, 则它们与标准消息处理程序以相同的方式处理。

1.10.8 Java认证与授权服务

JAAS(Java Authentication and Authorization Service, Java认证与授权服务) 为Java EE应用程序提供了一种认证与授权特定用户或用户组的方式。

JAAS是标准的PAM(Pluggable Authentication Module, 可插拔认证模块) 框架的Java版。它扩展了Java平台的安全架构, 以支持基于用户的授权。

1.11 GlassFish 服务器工具

GlassFish Server是一个与Java EE 6平台相兼容的服务器实现。除了支持前文所述的全部API，GlassFish服务器还提供了一系列Java EE 6平台之外的面向开发人员的Java EE工具。

本节将简要概括构成GlassFish服务器的工具。启动和停止GlassFish服务器、启动管理控制台、启动和停止Java数据库服务器的方法将在第2章介绍。

Glass Fish服务器集成了表1-1所列的工具，本教程将穿插介绍其中一些工具的基本使用方法。关于工具的详细信息，参见GUI工具的在线帮助文档。

表1-1 GlassFish服务器工具

工 具	描 述
管理控制台 (Administration Console)	基于Web的GUI GlassFish服务器管理工具。用于停止GlassFish服务器并管理用户、资源和应用程序
asadmin	命令行方式的GlassFish服务器管理工具。用于启动和停止GlassFish服务器并管理用户、资源和应用程序
appclient	命令行工具，加载应用程序客户端容器，从应用程序客户端的JAR文件中调用客户端程序
capture-schema	命令行工具，从数据库中提取模式信息，产生GlassFish服务器容器持久化管理所需的模式文件
package-appclient	命令行工具，打包应用程序客户端的容器库和JAR文件
Java DB数据库	Java DB服务器的副本
xjc	命令行工具，用Java编程语言将源XML模式转换（或绑定）至一组JAXB内容类
schemagen	命令行工具，为Java类中所引用的命名空间创建模式文件
wsimport	命令行工具，为给定的WSDL文件产生可移植的JAX-WS中间文件（artifact）。在这些中间文件产生之后，它们将与WSDL、模式文档和端点实现在一起打包至WAR文件中并部署
wsgen	命令行工具，读取Web 服务端点类，并产生Web 服务部署和调用所需的全部可移植的JAX-WS中间文件



本章全面讲述关于安装、构建以及运行示例应用的相关内容。

本章内容

- ☐ 必备软件
- ☐ 启动和关闭GlassFish服务器
- ☐ 启动管理控制台
- ☐ 启动和关闭Java DB服务器
- ☐ 构建示例
- ☐ 教程示例的目录结构
- ☐ 获取最新版的教程
- ☐ 调试Java EE应用

2.1 必备软件

必须安装如下软件，才能运行示例应用：

- ☐ Java平台标准版（J2SE），参见2.1.1节；
- ☐ Java EE 6 SDK，参见2.1.2节；
- ☐ Java EE 6 教程组件，参见2.1.3节；
- ☐ NetBeans集成开发环境，参见2.1.4节；
- ☐ Apache Ant，参见2.1.5节。

2.1.1 Java 平台标准版（J2SE）

为了构建、部署以及运行示例应用，需要安装Java平台6（标准版）SDK（即JDK 6，可以从<http://www.oracle.com/technetwork/java/javase/downloads/index.html>处下载）。

下载JDK时要注意只下载JDK的最新更新，不要下载把NetBeans IDE或者Java EE SDK也打包进去的版本。

2.1.2 Java EE 6 SDK

GlassFish 服务器开源版3.0.1是教程里示例应用的构建和运行环境。为了构建、部署以及运行这些示例应用，需要安装GlassFish 服务器。至于集成开发环境NetBeans，则是可选的。为了能够安装和运行GlassFish 服务器，必须先安装Java EE 6的软件开发包（即SDK），这可以在<http://www.oracle.com/technetwork/java/javaee/overview/index.html>处下载，注意一定要选择Java EE 6的SDK，不能是Java EE 6的Web Profile SDK。

SDK 安装小贴士

在安装SDK的过程中，请执行如下操作。

- ☐ 配置GlassFish服务器管理员的用户名为admin，口令为空，这是默认的设置。
- ☐ 接受默认的端口号，即管理端口4848以及HTTP端口8080。
- ☐ 允许安装程序下载以及配置更新工具（Update Tool）。如果是通过防火墙上网，则需要提供代理服务的主机IP地址以及端口号。

本教程假定GlassFish服务器是安装到了`as-install-parent`路径下，所有相对路径都是以此为基准。例如，在微软的Windows操作系统上，默认的安装路径为C:\glassfishv3，那么`as-install-parent`就是C:\glassfishv3。GlassFish服务器安装在`as-install`，也就是`as-install-parent`下的那个叫做glassfish的目录中。因此，对于微软的Windows操作系统，`as-install`是C:\glassfishv3\glassfish。

安装好GlassFish服务器之后，需要把如下路径加到环境变量PATH中去，这样可不必在每次运行如下命令时都指定全路径：

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

2.1.3 Java EE 6 教程组件

本教程中示例应用的源代码已经包含在教程组件里了，要获取教程组件，可以使用更新工具Update Tool。

如果防火墙阻止你使用Update Tool获取组件，请访问java.net网站来获取本教程。

▼用更新工具Update Tool获取教程组件

- (1) 启动Update Tool。
 - ☐ 在命令行上输入命令`updatetool`。
 - ☐ 如果是运行在Windows操作系统上，则可以参照如下步骤：开始菜单→所有程序→Java EE 6 SDK→Start Update Tool。
- (2) 展开GlassFish Server Open Source Edition结点。
- (3) 选择Available Add-ons结点。
- (4) 从列表里找出Java EE 6 Tutorial，并勾选之。
- (5) 单击Install。

(6) 接受许可证协议。

安装之后, Java EE 6 Tutorial就会出现在已安装组件的列表中, 具体会安装到`as-install/docs/javace-tutorial`目录下。这个目录有两个子目录, 分别为`docs`和`examples`。其中`examples`包含了一系列子目录, 分别对应本教程所讲述的各项技术。

后续操作 Java EE 6 Tutorial会定期更新。有关获取更新的细节, 请参阅2.7节。

▼ 从java.net网站获取教程组件

仔细按照如下步骤来进行, 如果路径配置错误, 则示例应用会无法执行。

- (1) 打开浏览器, 访问<http://javaeetutorial.java.net/>。
- (2) 单击左侧的Documents & Files链接。
- (3) 在打开的Documents&Files页面的表格里, 找到Java EE 6 Tutorial压缩文件的最新稳定版本。
- (4) 右击这个压缩文件, 保存到本地。
- (5) 复制或者移动这个压缩文件到GlassFish SDK路径下(默认路径为`glassfishv3`)。
- (6) 解压这个压缩文件, 文件都被解压到`glassfish/docs/javace-tutorial`目录。

2.1.4 NetBeans 集成开发环境

NetBeans 集成开发环境 (IDE) 是一款免费且开源的软件, 用来开发Java应用, 包括企业应用。NetBeans IDE支持Java EE平台, 所以可以在NetBeans IDE里构建、打包、部署以及运行教程里的示例应用。

为了运行教程里的示例应用, 需要最新版的NetBeans IDE。这可以从<http://www.netbeans.org/downloads/index.html>处下载。

▼ 安装NetBeans IDE (无GlassFish服务器)

当安装NetBeans IDE的时候, 不要安装其中的GlassFish服务器。执行如下的步骤, 跳过GlassFish服务器的安装:

- (1) 在NetBeans IDE安装向导的第一个对话框里, 单击Customize;
- (2) 在Customize Installation对话框里, 去掉GlassFish Server的勾选, 并单击OK;
- (3) 执行NetBeans IDE后续的安装步骤。

▼ 把GlassFish服务器注册到NetBeans IDE中

为了在NetBeans IDE里运行教程里的示例应用, 我们必须把自己的GlassFish服务器注册到NetBeans IDE里去, 作为NetBeans 服务器的一个实例, 步骤如下:

- (1) 选择Tools→Servers打开Servers对话框;
- (2) 单击Add Server;
- (3) 在Choose Server下, 选择GlassFish v3, 然后单击Next;
- (4) 在Server Location下, 浏览并选择GlassFish服务器的安装路径, 然后单击Next;
- (5) 在Domain Location下, 选择Register Local Default Domain;
- (6) 单击Finish按钮。

2.1.5 Apache Ant

Ant是由Apache软件基金会开发的一个基于Java的构建工具, 其官方网站是<http://ant.apache.org/>。Ant可以用于教程示例应用的构建、打包以及部署。要想运行教程中的示例应用, 需要Ant 1.7.1。如果还没有安装Ant 1.7.1的话, 可以用Update Tool来安装。Update Tool已经集成至GlassFish服务器中。

▼ 获取Apache Ant

- (1) 启动更新工具Update Tool。
 - 在命令行上, 输入命令`updatetool`。
 - 如果是运行在Windows操作系统上, 则可以参照如下步骤: 开始菜单→所有程序→Java EE 6 SDK→Start Update Tool。
- (2) 展开GlassFish Server Open Source Edition结点。
- (3) 选择Available Add-ons结点。
- (4) 从列表里找出Apache Ant Build Tool, 并勾选之。
- (5) 单击Install。
- (6) 接受许可证协议。

安装之后, Apache Ant就会出现在已安装组件的列表中。其文件会安装到`as-install-parent/ant`路径下。

后续操作 为了便于使用ant命令, 需要把`as-install/ant/bin`加到PATH环境变量中。

2.2 启动和关闭 GlassFish 服务器

要启动GlassFish服务器, 需要打开一个终端窗口或者命令提示符, 并执行如下命令:

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

域(domain)是一个或多个GlassFish服务器的实例的集合, 由一个管理服务器来管理。与域相关的内容有:

- GlassFish服务器的端口号，默认值为8080；
- 管理服务器的端口号，默认值为4848；
- 管理员用户名和口令。

我们可以在安装GlassFish服务器的时候指定这些配置，本教程里的示例应用一律假设在安装服务器的过程中接受了默认值。

如果没有参数，`start-domain`命令会初始化默认的域，也就是`domain1`。`--verbose`参数使得所有的日志信息和调试信息都输出到终端窗口或命令提示符窗口上。同时，这些信息也会记录在服务器的日志文件里，对应的文件目录为`domain-dir/logs/server.log`。

如果使用的是Windows操作系统，则以如下方式启动：开始菜单→所有程序→Java EE 6 SDK→Start Application Server。

应用服务器完成了启动过程后，终端窗口会显示如下信息：

```
Domain domain1 started.
```

要关闭GlassFish服务器，打开一个终端窗口或者命令提示符，并执行如下命令：

```
asadmin stop-domain domain1
```

如果使用的是Windows操作系统，则以如下方式关闭：

开始菜单→所有程序→Java EE 6 SDK→Stop Application Server。

服务器关闭后，终端窗口会显示如下信息：

```
Domain domain1 stopped.
```

2.3 启动管理控制台

管理控制台可以用来管理GlassFish服务器、用户、资源以及Java EE应用。在使用管理控制台之前，GlassFish服务器必须处于运行状态。为了启动管理控制台，需要打开浏览器，访问<http://localhost:4848/>。

如果是运行在Windows操作系统上，则可以参照如下步骤：开始菜单→所有程序→Java EE 6 SDK→Administration Console。

▼ 在NetBeans IDE里启动管理控制台

- (1) 单击Services标签。
- (2) 展开Servers结点。
- (3) 右击GlassFish Server实例，选择View Admin Console。

注意 NetBeans IDE使用系统默认的Web浏览器打开管理控制台。

2.4 启动和关闭 Java DB 服务器

GlassFish服务器里包含了Java DB数据库服务器。

要启动Java DB服务器，打开一个终端窗口或者命令提示符，并执行：

```
asadmin start-database
```

要关闭Java DB服务器，打开一个终端窗口或者命令提示符，并执行：

```
asadmin stop-database
```

要想详细了解GlassFish服务器内置的Java DB数据库，可以访问<http://www.oracle.com/technet-work/java/javadb/overview/index.html>。

▼ 在NetBeans IDE里启动数据库服务器

要想在NetBeans IDE里启动数据库服务器，请参照如下步骤：

- (1) 单击Services标签；
- (2) 展开Databases结点；
- (3) 鼠标右击Java DB，选择Start Server。

后续操作 要想在NetBeans IDE里关闭数据库服务器，则用鼠标右击Java DB，选择Stop Server。

2.5 构建示例

教程的示例为NetBeans IDE和Ant分别提供了配置文件。构建这些示例的具体指令，会在对应章节里介绍。NetBeans IDE和Ant都可以用来构建、打包、部署以及运行这些示例。

2.6 教程示例的目录结构

本教程中的示例使用Java BluePrints应用目录结构。这样可以把应用的源代码和编译后的文件分离开来，从而为迭代式的开发过程提供方便。

对于每一个应用程序模块来说，其目录有着如下的结构：

- build.xml——Ant构建文件；
- src/java——模块的Java源代码；
- src/conf——除Web应用程序之外的模块配置文件；
- web——网页文件、CSS文件、标签文件以及图片文件（Web应用独有）；
- web/WEB-INF——Web应用的配置文件（Web应用独有）；
- nbproject——NetBeans项目文件。

有些示例有多个应用模块被打包到一个EAR文件（企业应用归档文件）中，其子模块遵循如下的目录命名规则：

- *example-name-app-client*——应用客户端；
- *example-name-ejb*——企业bean的JAR文件；
- *example-name-war*——Web应用。

每一个示例项目都有一个Ant构建文件（*build.xml*），每个构建文件可以有多个任务，具体功能有：创建*build*子目录，复制和编译文件到这个*build*目录里去；创建*dist*子目录，里面包含打包的模块文件；创建*client-jar*目录，这个目录包含了应用客户端的JAR包。

2.7 获取最新版的教程

使用Update Center（更新中心）可以检查教程的更新，这个Update Center工具已经包含在Java EE 6 SDK里了。

▼ 通过Update Center更新教程

- (1) 在NetBeans IDE里打开Services标签，展开Servers结点。
- (2) 右击GlassFish v3实例，选择View Update Center，以显示Update Tool。
- (3) 选择树结构里的Available Updates，查看已更新包的列表。
- (4) 找出Java EE 6教程对应的更新包（*javaee-tutorial*）。
- (5) 如果教程有新版本，选择Java EE 教程（*javaee-tutorial*）并单击Install。

2.8 调试 Java EE 应用

Java应用在部署或者在运行时，有可能会出现各种各样的问题。本节主要讲述如果出现了错误，应该采取哪些手段来查找出错原因。

2.8.1 服务器日志

调试应用的一个办法是查看服务器日志，具体文件位于*domain-dir/logs/server.log*。这个日志文件包含了GlassFish服务器和应用的输出信息。应用程序的Java类都可以用System.out.println输出日志信息，也可以采用Java的Logging API（文档参见<http://docs.oracle.com/javase/6/docs/api/java/util/logging/package-summary.html>）来实现。如果是Web组件，则采用ServletContext类的log方法。

如果启动GlassFish服务器时使用了--verbose参数，那么所有的日志信息和调试信息都会显示到终端窗口或命令提示符窗口上，同时也会记录在服务器的日志文件里。如果GlassFish服务器是在后台启动的，则调试信息仅出现在日志文件里。我们可以用文本编辑器打开服务器中的日志文件或者用管理控制台的日志查看器（log viewer）查看其内容。

▼ 日志查看器

- (1) 选择GlassFish Server结点。
- (2) 单击View Log Files按钮，日志查看器就会打开，并显示最新的40条记录。
- (3) 如果想显示其他的日志内容，则执行如下步骤：
 - 单击Modify Search按钮；
 - 针对想要查看的记录设定约束条件；
 - 单击位于日志查看器界面顶端的Search按钮。

2.8.2 调试器

GlassFish服务器支持JPDA (Java Platform Debugger Architecture, Java平台调试架构)。借助JPDA，我们可以配置GlassFish服务器，使用socket获取应用的调试信息。

▼ 使用调试器调试应用

- (1) 在管理控制台里打开GlassFish服务器的调试功能。

- 展开Configuration结点。
- 选择JVM Settings结点，默认的调试选项设为：

`-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=9009`

正如看到的那样，默认的调试端口号为9009。我们可以将其改成其他端口，只要这个端口没被GlassFish服务器或者其他服务占用即可。

- 勾选Debug Enabled的复选框。
- 单击Save按钮。

- (2) 关闭GlassFish服务器并重新启动它。

Part 2

第二部分

Web 层

本部分主要讲述 Web 层会用到的技术。

本 部 分 内 容

- 第 3 章 Web 应用初步
- 第 4 章 JSF 技术
- 第 5 章 Facelets 介绍
- 第 6 章 表达式语言
- 第 7 章 在网页中使用 JSF 技术
- 第 8 章 转换器、监听器和验证器
- 第 9 章 用 JSF 技术开发 Web 应用
- 第 10 章 Java Servlet 技术

Web应用是Web服务器或者应用服务器的动态扩展，Web应用有如下两种。

- 面向表现的Web应用 这种类型的Web应用能够生成可交互的网页，以便动态响应用户的请求。动态网页通常包含几种类型的标记语言，如HTML、XHTML、XML等。第4章 ~ 第9章将介绍如何开发面向表现的Web应用。
- 面向服务的Web应用 这种类型的Web应用实现了Web 服务的端点（endpoint）。面向表现的应用一般是通过调用面向服务的Web应用来实现自己的功能。第12章、第13章以及第三部分将介绍如何开发面向Web服务的Web应用。

本章内容

- Web应用
- Web应用的生命周期
- Web模块示例——hello1
- 配置Web应用之hello2示例
- Web应用的更多信息

3.1 Web 应用

Java EE平台上的Web组件为Web服务器提供了动态扩展能力。Web组件可能是Java Servlet、用JavaServer Faces技术实现的网页、Web service的端点，抑或是JSP页面。图3-1展示了Web客户端和一个使用了servlet技术的Web应用是如何交互的。客户端首先向Web服务器发送一个HTTP请求，实现了Java Servlet和JSP技术的Web服务器把此请求转化为一个HTTPServletRequest对象。这个对象会被传递给Web组件，该组件可以和JavaBeans组件或者数据库交互，以产生动态内容。这一Web组件可以直接生成一个HTTPServletResponse对象，也有可能把这一请求转发到另外一个Web组件。最后，Web组件总会生成一个HTTPServletResponse对象。Web服务器把这个HTTPServletResponse对象转化为一个HTTP应答消息，再返回给客户端。

servlet从编程的角度来说是Java语言提供的类，它可以动态处理请求并构造应答消息。Java的许多技术，比如JSF（JavaServer Faces）和Facelets以及各种框架，都可以用来创建可交互的Web

应用。虽然servlet、JSF和Facelets技术可以实现相似的功能，但是每一种技术都有着自己的适用场合。servlet适用于面向服务的应用（Web 服务的端点可以用servlet实现）以及面向表现的Web应用的控制功能，比如说消息分发以及处理非文本数据。JSF和Facelets技术适用于生成基于文本的标记语言页面，比如XHTML页面，一般用于面向表现的Web应用。

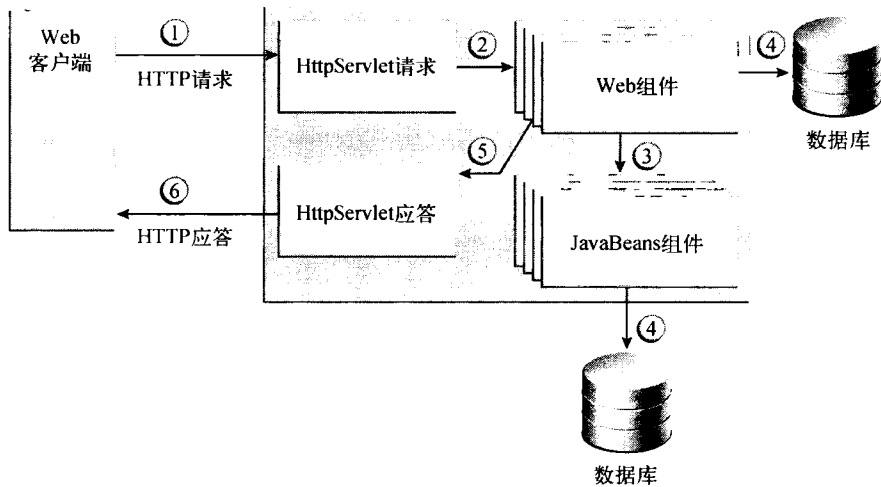


图3-1 Java Web应用请求的处理流程

Web组件功能的实现离不开为其运行提供各种服务的平台，即Web容器。Web容器提供诸如请求分发、安全、并发以及应用的生命周期管理等服务，同时也为Web组件提供命名服务、事务处理以及电子邮件等相关的API。

当Web应用安装或部署至Web容器之后，它的某些运行参数可通过配置的方式设定。配置信息可以用Java EE的注解功能来实现，也可以通过一个名为Web应用部署描述文件（deployment descriptor，也叫部署描述符，简称DD）来维护，且格式为XML。该文件的格式需要遵从Java Servlet规范所规定的格式。

本章会简要介绍一下开发Web应用的有关事项，首先介绍Web应用的生命周期，然后讲述如何把一个简单的Web应用打包以及部署到GlassFish服务器。后面的内容还会讲述如何配置GlassFish上的Web应用以及如何设定最常用的几个配置项。

3.2 Web 应用的生命周期

Web应用包含一系列的Web组件、静态资源文件（比如图片文件），以及一些工具类和类库。Web容器提供了许多基础服务，可以增强Web组件的功能，并使开发Web组件更容易。然而，正是由于Web应用必须调用这些服务，创建和运行一个Web应用有别于传统的单机应用。

创建、部署以及执行Web应用可以总结为以下过程：

- (1) 编写Web组件的代码;
- (2) 如果有必要的话, 编辑Web应用部署描述文件;
- (3) 编译Web组件以及它所引用的工具类;
- (4) 把应用打包到一个部署单元, 这一步是可选的;
- (5) 部署应用到Web容器里去;
- (6) 用相应的URL访问这个Web应用。

关于如何为Web组件编写代码会在以后几章里讲述。第(2)步~第(4)步会在下面的小节里讲述, 同时会引用一个Hello, World风格的、面向表现的Web应用的例子。这个应用允许用户在HTML表单里输入姓名(如图3-2所示), 并在用户提交表单后, 显示一个问候语(如图3-3所示)。

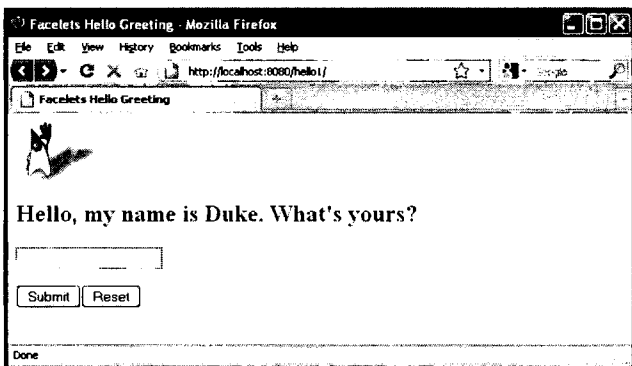


图3-2 hello1 Web应用的问候表单

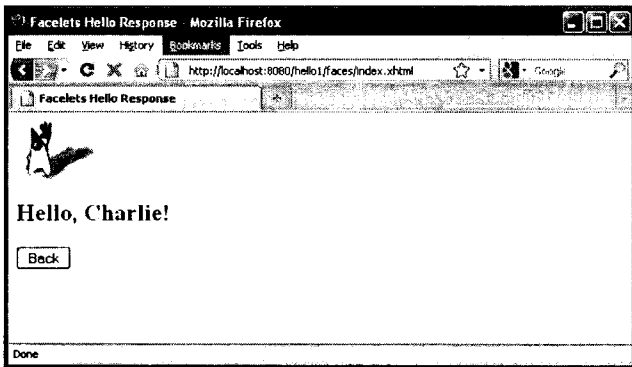


图3-3 hello1 Web应用的应答页面

Hello应用包括两个Web组件, 一个是生成问候语的组件, 一个是生成应答内容的组件。本章讨论如下两个简单应用:

- hello1应用, 这是一个基于JSF技术的Web应用, 使用了两个XHTML文件以及一个后台bean;

□ hello2应用，这是一个基于servlet技术的Web应用，它里面的Web组件是用两个servlet类实现的。

通过对这两个应用的介绍，我们将展示包含Web组件的应用所涉及的打包、部署、配置以及运行任务。对应的应用源代码分别位于目录 *tut-install/examples/web/hello1/* 和 *tut-install/examples/web/hello2/* 中。

3.3 Web 模块示例——hello1

3

在Java EE架构里，Web组件以及静态Web内容文件，比如图片文件，都叫做Web资源。Web模块是Web资源里可以部署和使用的最小可用单元。Java EE的Web模块就是Java Servlet规范里定义的Web应用。

除了Web组件和Web资源，Web模块还可以包括其他类型的文件，如下：

- 服务器端的工具类，如网上商城的购物车；
- 客户端的类，如Java applet以及工具类。

Web模块有特定的结构，其最上层是应用的根目录，这个目录存放了XHTML文件、客户端的类以及归档文件、静态Web资源（如图片文件）。

根目录包含了一个叫做WEB-INF的子目录，这里面包含了如下文件和目录：

- classes目录，该目录包含了服务器端的类，包括servlet、EJB（Enterprise Java Beans）、工具类以及JavaBeans组件等；
- tags目录，该目录包含了标签文件，也就是标签库的实现；
- lib目录，该目录包含了EJB类的JAR包以及其他服务器端应用所调用的JAR包；
- 部署描述文件，比如web.xml（Web应用部署描述文件）或者ejb-jar.xml（EJB部署描述文件）。

如果使用了JSF技术，或需要指定某种类型的安全信息，或者想覆盖掉Web组件里的注解，则web.xml文件必须存在。

我们也可以在应用的根目录下或者WEB-INF/classes/目录下创建应用专用的子目录，也就是说，Web组件可以以编程的方式访问这个目录（即包目录）。

Web模块可以部署为一个未打包的文件目录，也可以打包到一个JAR文件中，也就是所谓的WAR（Web）文件。因为WAR文件的内容以及用途与通常的JAR文件有所区别，所以WAR文件采用.war扩展名。刚才所说的Web模块是跨平台的，可以部署到任何Web容器里，只要该容器符合Java Servlet规范。

要部署WAR文件到GlassFish服务器，该文件必须有一个运行时需要的部署描述文件。后者是XML格式的文件，包含诸如Web应用的上下文根（context root）以及应用资源名称在GlassFish服务器上的映射。在GlassFish服务器上，Web应用的部署描述文件叫做sun-web.xml，位于WEB-INF目录下。可以部署到GlassFish服务器上的Web模块的结构如图3-4所示。

举例来说，hello1应用的sun-web.xml文件指定了如下的上下文根：

```
<context-root>/hello1</context-root>
```

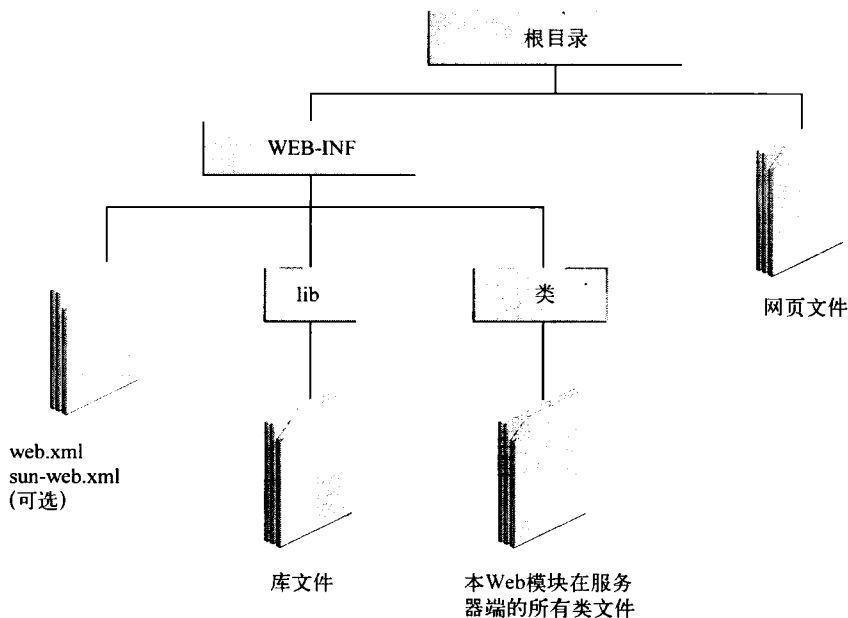


图3-4 hello1 Web模块目录结构

3.3.1 hello1 Web模块分析

hello1应用是由JSF技术实现的Web模块，用于在浏览器里显示问候语和应答消息。我们可以使用文本编辑器查看应用里的文件，也可以使用NetBeans IDE来查看。

▼用NetBeans IDE查看hello1 Web模块

- (1) 在NetBeans IDE里，选择File→Open Project。
- (2) 在Open Project对话框里，定位到*tut-install/examples/web/*。
- (3) 选择hello1目录。
- (4) 勾选Open as Main Project复选框。
- (5) 展开Web Pages结点，双击index.xhtml文件，在右侧窗格里查看其内容。

index.html是Facelets应用的默认首页。对于hello1应用来说，该页面使用简单标签标记来创建一个表单。这个表单带有一个图片，一个标题信息栏，一个文本框以及两个命令按钮，代码如下：

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Facelets Hello Greeting</title>
  
```

```

</h:head>
<h:body>
  <h:form>
    <h:graphicImage url="duke.waving.gif"/>
    <h2>Hello, my name is Duke. What's yours?</h2>
    <h:inputText id="username"
      value="#{hello.name}"
      required="true"
      requiredMessage="A name is required."
      maxLength="25">
    </h:inputText>
    <p></p>
    <h:commandButton id="submit" value="Submit" action="response">
    </h:commandButton>
    <h:commandButton id="reset" value="Reset" type="reset">
    </h:commandButton>
  </h:form>
</h:body>
</html>

```

本页面里最复杂的元素是inputText文本框，它的maxLength属性指定了文本框的最大长度，required属性指定了这个文本框是不是必填的，requiredMessage属性指定当文本框中没填东西时，界面上应显示的出错提示信息。最后，value属性包含一个表达式，该表达式的值由hello这个后台bean来提供。

值为Submit的commnadButton元素，其action属性为response。也就是说单击该按钮时，将显示response.xhtml这个页面。

(6) 双击response.xhtml文件以查看其内容。

应答页面出现了，其内容比刚才的问候页面简单一些，包含了一个图片文件和一个标题信息栏（标题信息栏显示的内容是由后台bean提供的表达式的值）以及一个按钮，该按钮的action属性被设置为index。也就是说，当此按钮被按下时，则返回到index.xhtml页面。

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Facelets Hello Response</title>
  </h:head>
  <h:body>
    <h:form>
      <h:graphicImage url="duke.waving.gif"/>
      <h2>Hello, #{hello.name}!</h2>
      <p></p>
      <h:commandButton id="back" value="Back" action="index" />
    </h:form>
  </h:body>
</html>

```

(7) 展开Source Packages结点，再展开hello1结点。

(8) 双击Hello.java以查看其内容。

Hello类是一个后台bean类，其name属性有两个方法——getName和setName。刚才Facelets页面里的表达式引用了name属性。默认情况下，表达式语言引用时需要指定类名，且第一个字母要用

小写，比如hello.name。

```
package hello1;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class Hello {

    private String name;

    public Hello() {
    }

    public String getName() {
        return name;
    }

    public void setName(String user_name) {
        this.name = user_name;
    }
}
```

(9) 在Web Pages结点下，展开WEB-INF结点，双击web.xml以查看其内容。

web.xml文件包含一些正常运行Facelets应用时必须定义的参数。当用NetBeans IDE新建一个应用时，这些内容是自动生成的。

□ 一个上下文参数指定项目所处的阶段：

```
<context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
```

上下文参数指定Web应用所需要的配置信息。Web应用可以定义自己的上下文参数。除此之外，JSF技术和Java Servlet技术还定义了Web应用可以使用的上下文参数。

□ servlet参数 以及其servlet-mapping参数则指定了FacesServlet的相关信息：

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

□ welcome-file-list参数指定应用的首页，注意是faces/index.xhtml，不是index.xhtml。

```
<welcome-file-list>
    <welcome-file>faces/index.xhtml</welcome-file>
</welcome-file-list>
```

3.3.2 打包Web模块

Web模块必须打包成WAR文件才能部署到服务器上。假如我们想分发Web模块给其他人使

用，也必须把Web模块打包成WAR文件。打包Web模块成WAR文件有几种方式，包括使用Ant工具或IDE工具，以及在Web模块的特定目录下执行jar命令。本书会讲述如何使用NetBeans IDE以及Ant来构建、打包以及部署hello1示例应用。

▼ 设置上下文根

上下文根（context root）是Web应用在Java EE服务器里的标识，它必须是一个以/开始的字符串。

对一个要部署到GlassFish服务器上的打包的Web模块来说，上下文根保存在sun-web.xml文件里。

我们可以按照如下步骤查看和编辑上下文根。

- (1) 展开hello1应用的Web Pages以及WEB-INF结点。
- (2) 双击sun-web.xml。
- (3) 在General标签下，查看 Context Root字段是否设置成/hello1了。

如果想要编辑该值，就可在此处修改。当创建一个新的应用时，可以在此处设置上下文根。

- (4) 单击XML标签。这一步是可选的。

请注意，上下文根的值/hello1在context-root元素之内。我们可以在此处修改这个值。

▼ 在NetBeans IDE里构建和打包hello1 Web模块

在NetBeans IDE里构建和打包hello1 Web模块，可以执行如下步骤：

- (1) 选择菜单File→Open Project；
- (2) 在Open Project对话框里导航到tut-install/examples/web/；
- (3) 选择hello1目录；
- (4) 勾选Open as Main Project复选框；
- (5) 单击Open Project；
- (6) 在Projects标签里，用鼠标右击hello1这个项目，选择Build。

▼ 用Ant工具构建和打包hello1模块

用Ant工具构建和打包hello1这个应用，可以参照如下步骤。

- (1) 打开一个终端窗口，切换目录到tut-install/examples/web/hello1/。
- (2) 执行ant命令。

该命令会执行一系列的必要任务，包括编译源文件、复制文件到tut-install/examples/web/hello1/build/、创建WAR文件，以及复制WAR文件到tut-install/examples/web/hello1/dist/目录下。

3.3.3 部署Web模块

部署WAR文件到GlassFish服务器有如下几种方式：

- 使用NetBeans IDE;
- 使用Ant工具;
- 运行asadmin命令;
- 使用管理控制台;
- 复制WAR文件到`domain-dir/autodeploy/`目录。

贯穿全书,我们会讲述如何使用NetBeans IDE和Ant打包以及部署应用。

▼ 用NetBeans IDE部署hello1应用

- 用鼠标右击hello1项目,然后选择Deploy。

▼ 用Ant部署hello1应用

(1) 打开一个终端窗口,切换目录到:

`tut-install/examples/web/hello1/`

(2) 执行如下命令:

```
ant deploy
```

3.3.4 运行已部署的 Web 模块

Web模块已经部署好了,现在可以在浏览器里打开并查看这个应用了。默认情况下,应用部署到localhost,端口号为8080。该应用对应的上下文根为hello1。

▼ 运行已部署的Web模块

要运行这个应用,需要参照如下步骤:

- (1) 打开浏览器;
- (2) 在浏览器的地址栏里输入如下网址:

`http://localhost:8080/hello1/`

- (3) 输入你的名字,然后单击Submit按钮。

应用(应答页面)会显示你刚才提交的名字。单击Back按钮可以重试一次。

3.3.5 查看已部署的 Web 模块

GlassFish服务器提供了两种查看已部署Web模块列表的方式,即使用管理控制台或者asadmin命令。

▼ 使用管理控制台查看已部署的Web模块

- (1) 在浏览器里打开网址`http://localhost:4848/`。
- (2) 选择Applications结点。

已部署的Web模块会显示在Deployed Applications表中。

▼使用asadmin命令查看已部署的Web模块

□ 执行如下命令：

```
asadmin list-applications
```

3.3.6 更新Web模块

在典型的迭代式开发周期里，通常在部署Web模块后，又会改动应用组件。更新Web模块可以按照如下步骤进行。

▼更新Web模块的步骤

- (1) 重新编译已改动的类。
- (2) 重新部署模块。
- (3) 在浏览器里重新打开网址。

3.3.7 动态加载

如果启用了动态加载功能，当改动了代码或者修改了部署描述文件后，无需重新部署应用或者模块。只要把改动的网页文件或者类文件复制到应用或模块对应的部署目录就可以了。Web模块的部署目录就叫`context-root`，具体路径为`domain-dir/applications/context-root`。服务器会定期检测它的变化，然后自动地重新部署应用。

这项功能对于开发人员来说非常有用，因为代码的改动马上就能被测试。但是，不推荐在生产环境中使用动态加载功能，因为这样会显著降低服务器的性能。此外，一旦重新加载，则所有当前会话都会无效，客户端必须重新建立会话。

对于GlassFish服务器来说，动态加载功能默认是打开的。

▼关掉或者修改动态加载功能

如果由于某种原因，我们不再需要默认的动态加载功能，可以在管理控制台里按照以下步骤把它关掉。

- (1) 打开浏览器，输入网址<http://localhost:4848/>。
- (2) 选择GlassFish Server结点。
- (3) 选择Advanced 标签。
- (4) 要想关掉动态加载功能，反选Reload Enabled复选框。
- (5) Reload Poll Interval字段表示动态加载的检测频率，可以修改此值以满足实际需要。默认值为2s。
- (6) 单击Save按钮。

3.3.8 卸载Web模块

卸载Web模块或其他类型的企业应用有两种方法，分别是使用NetBeans IDE和使用Ant工具。

▼ 使用NetBeans IDE卸载Web模块hello1

- (1) 首先确认GlassFish服务器已经启动。
- (2) 在Services窗口，依次展开Servers结点、GlassFish Server实例以及Applications结点。
- (3) 用鼠标右击hello1模块，然后执行Undeploy。
- (4) 要想删除类文件以及其他构建过程中生成的中间文件，用鼠标右击项目，然后执行Clean。

▼ 使用Ant工具卸载Web模块hello1

- (1) 打开一个终端窗口，并切换到如下目录：

tut-install/examples/web/hello1/

- (2) 执行如下命令：

```
ant undeploy
```

- (3) 要想删除类文件以及其他构建过程中生成的中间文件，执行如下命令：

```
ant clean
```

3.4 配置 Web 应用之 hello2 示例

Web应用的配置可以通过注解或者修改Web应用部署描述文件来实现。

接下来我们简要介绍Web应用里通常需要修改的条目。我们以hello2项目为例讲述配置Web应用的过程。

3.4.1 映射URL到Web组件

当收到一个请求后，Web容器必须决定由哪个Web组件处理这个请求。Web容器通过将请求里的URL路径映射到Web应用或者Web组件来实现这一目的。URL路径包括应用的上下文根，后面紧跟着一个URL模式。URL模式是可选的。

`http://host:port/context-root[/url-pattern]`

为servlet设置URL模式，可以在servlet的源代码文件里使用@WebServlet注解来实现。举例来说，hello2应用里的GreetingServlet.java文件包含了如下注解，它把URL模式设定为/greeting：

```
@WebServlet("/greeting")
public class GreetingServlet extends HttpServlet {
    ...
}
```

这个注解表明/greeting这个URL模式紧接着应用的上下文根。于是，当servlet在本地部署后，可以通过如下URL访问该servlet：

`http://localhost:8080/hello2/greeting`

要想仅用上下文根来访问该servlet，可以指定/作为URL模式。

3.4.2 hello2 Web模块分析

hello2的功能几乎和hello1一样，只不过它不使用JSF技术，而是用Java Servlet技术实现的。我们可以使用文本编辑器查看项目里的文件，当然也可以使用NetBeans IDE。

3

▼ 用NetBeans IDE查看hello2 Web模块

- (1) 在NetBeans IDE里，选择菜单File→Open Project。
- (2) 在Open Project对话框里，定位到*tut-install/examples/web*。
- (3) 选择hello2目录。
- (4) 勾选Open as Main Project复选框。
- (5) 依次展开Source Packages结点和servlets结点。
- (6) 双击GreetingServlet.java文件以查看内容。

这个servlet覆写了父类的doGet方法，实现了对HTTP里的GET请求的处理。该servlet显示一个简单问候语表单，这个表单的Submit按钮和hello1项目里的那个一样，在其action属性里指定了应答页面。在下面的代码片段中，第一行的@WebServlet注解指定了URL的模式，该路径是基于上下文根的相对路径：

```
@WebServlet("/greeting")
public class GreetingServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        response.setBufferSize(8192);
        PrintWriter out = response.getWriter();

        // then write the data of the response
        out.println("<html>"
            + "<head><title>Servlet Hello</title></head>");

        // then write the data of the response
        out.println("<body bgcolor=\"#ffffff\">"
            + "<img src=\"duke.waving.gif\" alt=\"Duke waving\">"
            + "<h2>Hello, my name is Duke. What's yours?</h2>"
            + "<form method=\"get\">"
            + "<input type=\"text\" name=\"username\" size=\"25\">"
            + "<p></p>"
            + "<input type=\"submit\" value=\"Submit\">"
            + "<input type=\"reset\" value=\"Reset\">"
            + "</form>");

        String username = request.getParameter("username");
```

```

    if (username != null && username.length() > 0) {
        RequestDispatcher dispatcher =
            getServletContext().getRequestDispatcher("/response");

        if (dispatcher != null) {
            dispatcher.include(request, response);
        }
    }
    out.println("</body></html>");
    out.close();
}
...

```

(7) 双击ResponseServlet.java以查看内容。

这个servlet也覆写了父类的doGet方法，只显示应答信息。下面的代码片段第一行的@WebServlet注解指定了URL的模式，该路径是基于上下文根的相对路径：

```

@WebServlet("/response")
public class ResponseServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();

        // then write the data of the response
        String username = request.getParameter("username");
        if (username != null && username.length() > 0) {
            out.println("<h2>Hello, " + username + "!</h2>");
        }
    }
    ...
}

```

(8) 在Web Pages结点下，展开WEB-INF结点，双击sun-web.xml文件以查看内容。

在General标签中可以看到Context Root字段设置成了/hello2。

对于这种简单的servlet，用不到web.xml这个文件。

3.4.3 构建、打包、部署以及运行hello2 应用

我们可以使用NetBeans IDE或者Ant工具来构建、打包、部署以及运行hello2应用。

▼使用NetBeans IDE构建、打包、部署以及运行hello2

- (1) 打开菜单File→Open Project。
- (2) 在Open Project对话框里，定位到*tut-install/examples/web/*。
- (3) 选择hello2目录。
- (4) 勾选Open as Main Project复选框。
- (5) 单击Open Project。
- (6) 在Projects标签下，用鼠标右击hello2项目，然后选择Build。

(7) 用鼠标右击hello2项目，然后选择Deploy。

(8) 打开Web浏览器，输入网址http://localhost:8080/hello2/greeting。

该网址包含了上下文根以及URL模式。

hello2的界面和hello1应用（如图3-2所示）很相像。主要的区别在于，当单击Submit按钮之后，应答信息显示在问候语下面，而不像hello1那样使用一个单独的网页。

▼使用Ant工具构建、打包、部署以及运行hello2

(1) 打开一个终端窗口，切换目录到`tut-install/examples/web/hello2/`。

(2) 执行命令`ant`。

该命令会生成WAR文件并把它复制到`tut-install/examples/web/hello2/dist/`目录中。

(3) 执行命令`ant deploy`。

终端会输出一个运行应用的URL，不过这个URL是错的，先不用管它。

(4) 打开Web浏览器，输入网址http://localhost:8080/hello2/greeting。

该网址包含了上下文根以及URL模式。

再次强调，hello2的界面和hello1应用（如图3-2所示）很相像，主要的区别在于，当单击Submit按钮之后，应答内容显示在问候语下面，而不像hello1那样使用一个单独的网页。

3.4.4 设置welcome文件

welcome文件机制可以指定一些文件。对于特定的URL请求（即部分有效请求，valid partial request），假如没有Web组件与之相对应，则Web容器可以把welcome文件的内容返回给浏览器作为应答信息。举例来说，假设我们定义了一个welcome文件welcome.html，当浏览器请求的URL为`host:port/webapp/directory`，这里的`directory`并没有映射到某个servlet或者XHTML网页上时，Web容器就把`host:port/webapp/directory/welcome.html`返回给浏览器。

如果收到了一个部分有效请求，Web容器会检查welcome文件列表，按照配置的顺序把部分有效请求和每一个welcome文件组合起来，并检查WAR文件里是否有与这个请求URL相匹配的静态资源或者servlet。然后，Web容器会转发请求到WAR文件里第一个与之匹配的资源上去。

如果没有声明welcome文件，GlassFish服务器会将index.xhtml视作默认的welcome文件。如果没有定义welcome文件，而且index.html也不存在的话，GlassFish服务器就列出该目录的所有文件。

根据惯例，我们可以将JSF应用的welcome文件设置为`faces/file-name.xhtml`。

3.4.5 设置上下文以及初始化参数

Web模块里的Web组件共享一个对象，这个对象代表了它们的应用上下文。我们可以传递初始化参数给这个上下文，或者传递给Web组件。

▼用NetBeans IDE增加上下文参数

- (1) 确保已经打开目标项目。
- (2) 展开项目对应的结点（在Projects窗格）。
- (3) 展开Web Pages结点，接着展开WEB-INF结点。
- (4) 双击web.xml。
- (5) 单击编辑器顶端的General。
- (6) 展开Context Parameters结点。
- (7) 单击Add，Add Context Parameter对话框就会出现。
- (8) 在Parameter Name字段处输入代表着上下文对象的名字。
- (9) 在Parameter Value字段处输入参数的值（将传递给上下文对象）。
- (10) 单击OK。

▼用NetBeans IDE增加初始化参数

我们可以使用@WebServlet注解指定Web组件初始化参数，对应的是属性initParams以及注解@WebInitParam，例如：

```
@WebServlet(urlPatterns="/MyPattern", initParams=
    {@WebInitParam(name="ccc", value="333")})
```

还有一个添加初始化参数的办法，就是用NetBeans IDE修改web.xml文件，步骤如下。

- (1) 确保已经打开目标项目。
- (2) 展开项目对应的结点（在Projects窗格）。
- (3) 展开Web Pages结点，接着展开WEB-INF结点。
- (4) 双击web.xml。
- (5) 单击编辑器顶端的Servlets。
- (6) 单击Add按钮（在Initialization Parameters表格下面），Add Initialization Parameter对话框出现。
- (7) 在Add Initialization Parameter对话框里，在Parameter Name字段处输入参数名。
- (8) 在Parameter Value字段中输入参数值。
- (9) 单击OK按钮。

3.4.6 映射错误信息到出错页面

如果Web应用运行过程中出错了，可以让应用根据不同的错误类型显示特定的出错页面。具体来讲，可以把HTTP返回的状态码或者Web组件抛出的Java异常映射到特定的出错页面。

部署描述文件中可以有多个error-page配置项，每一个配置项对应了一种错误类型。多个错误类型可以对应一个出错页面。

▼ 在NetBeans IDE里设置错误映射

- 3
- (1) 确保已经打开目标项目。
 - (2) 展开项目对应的结点（在Projects窗格）。
 - (3) 展开Web Pages结点，接着展开WEB-INF结点。
 - (4) 双击web.xml。
 - (5) 单击编辑器顶端的Pages。
 - (6) 展开Error Pages结点。
 - (7) 单击Add按钮，Add Error Page对话框出现。
 - (8) 在Add Error Page对话框里，单击Browse以定位页面文件，就是错误发生时要显示的页面。
 - (9) 在Error Code字段处，输入HTTP状态码（会导致显示出错页面的状态码）。
 - (10) 在Exception Type字段处输入异常（exception）的类型（会导致显示出错页面的异常）。
 - (11) 单击OK按钮。

3.4.7 资源引用声明

如果Web组件使用了EJB对象、数据源或者Web服务，我们可以使用Java EE提供的注解功能把这些资源注入到自己的应用里。注解解救了程序员，使得他们无需像使用旧版本Java EE那样编写大量类似配置项的读取等代码。

虽然对于程序员来说，注解使得资源注入非常方便，不过对于Web应用来说还是有一些限制的。首先，只能把资源注入到容器管理的对象里。这是因为容器只有控制组件的创建才能执行对组件的注入。基于此，不能把资源注入到一般的对象（比如一个简单的JavaBeans组件）里去。不过，JSF托管的bean是由容器管理的，因此它能够接受资源注入。

可以接受资源注入的组件见表3-1。

表3-1 能接受资源注入的Web组件

组 件	接口/类
servlet	javax.servlet.Servlet
servlet过滤器	javax.servlet.ServletFilter
事件监听器	javax.servlet.ServletContextListener
	javax.servlet.ServletContextAttributeListener
	javax.servlet. ServletRequestListener
	javax.servlet. ServletRequestAttributeListener
	javax.servlet.http.HttpSessionListener
	javax.servlet.http.HttpSessionAttibuteListener
	javax.servlet.http.HttpSessionBindingListener
标签库监听器（taglib listener）	同上
标签库标签处理程序（Taglib tag handler）	javax.servlet.jsp.tagext.JspTag
托管bean（managed bean）	原有的简单Java对象

本节内容会讲述如何使用Servlet容器支持的注解功能来注入资源。第20章“运行Persistence示例”会介绍Web应用如何使用Java持久化API支持的注解功能。第24章“Web应用安全化入门”会讲解如何使用注解来指定安全Web应用的信息。

1. 声明资源引用

`@Resource`注解用来声明对资源的引用，比如数据源，EJB或者环境变量。

`@Resource`注解可以指定类、类的方法或者类的成员变量。容器负责注入所引用的资源，并把它映射到合适的JNDI（Java Naming and Directory Interface，Java命名和目录接口）资源上。

在下面的例子里，`@Resource`注解用来把数据源注入到（需要连接数据源的）组件里去，就像使用JDBC技术来访问关系型数据库那样。

```
@Resource javax.sql.DataSource catalogDS;
public getProductsByCategory() {
    // get a connection and execute the query
    Connection conn = catalogDS.getConnection();
    ..
}
```

容器要在应用使用组件之前注入这个数据源。数据源JNDI映射从字段的名称`catalogDS`以及类型`javax.sql.DataSource`的名称就可以推导出来。

如果有多个资源需要注入到组件里去，就使用`@Resources`注解来包含它们，参考如下代码：

```
@Resources ({
    @Resource (name="myDB" type=java.sql.DataSource),
    @Resource(name="myMQ" type=javax.jms.ConnectionFactory)
})
```

本书里的Web示例应用使用Java Persistence API来访问关系型数据库。这一API不需要显式创建一个数据源的连接。因此，示例应用就不需要用`@Resource`注解来注入数据源。然而Java Persistence API支持`@PersistenceUnit`和`@PersistenceContext`注解，并分别用它们注入`EntityManagerFactory`以及`EntityManager`实例。第20章会讲述这些注解，还会讲解如何在Web应用里使用Java Persistence API。

2. 声明引用Web服务

`@WebServiceRef`注解提供对Web服务的引用，下面的例子演示如何使用`@WebServiceRef`来声明对Web服务的引用。`@WebServiceRef`使用`wsdlLocation`属性来指定要部署的Web服务的WSDL文件的URI。

```
...
import javax.xml.ws.WebServiceRef;
...
public class ResponseServlet extends HttpServlet {
    @WebServiceRef(wsdlLocation=
        "http://localhost:8080/helloservice/hello?wsdl")
    static HelloService service;
}
```

3.5 Web 应用的更多信息

有关Web应用的更多信息，可以参考如下网站。

- ❑ JSF 2.0规范:
<http://jcp.org/en/jsr/detail?id=314>
- ❑ JSF 网站:
<http://java.sun.com/javaee/javaxserverfaces/>
- ❑ Java Servlet 3.0规范:
<http://jcp.org/en/jsr/detail?id=315>
- ❑ Java Servlet网站:
<http://java.sun.com/products/servlet>

JSF 技术是服务器端的组件框架，可以用来创建基于 Java 的 Web 应用。

JSF 技术由如下部分构成。

- 一组 API，作用包括：展示组件并管理它们的状态；处理事件、服务器端的数据验证和数据转换；定义页面导航；支持国际化，提升易用性；提供所有特性的扩展。
- 一个标签库，用于向网页中添加组件及将组件连接到服务器端的对象。

JSF 技术提供良好的编程模型和丰富的标签库。这些特性通过服务器端用户界面（UI）显著地简化了构建和维护 Web 应用的过程。开发人员能够以极小的代价完成下面的任务：

- 创建网页；
- 通过增加组件标签，将组件添加到网页上；
- 将页面上的组件绑定至服务器端的数据；
- 将组件产生的事件与服务器端的应用代码关联起来；
- 保存应用的状态，并可在服务器请求结束后恢复应用的状态；
- 通过定制重用和扩展组件。

本章将对 JSF 技术进行简要介绍。在解释什么是 JSF 应用，并介绍该项技术的优势之后，本章将阐述创建一个简单 JSF 应用的过程。本章还通过描述 JSF 示例程序的各个阶段（指其生命周期的各个阶段）及其演进过程，介绍 JSF 的生命周期。

本章内容

- 什么是 JSF 应用
- JSF 技术的优势
- 创建简单的 JSF 应用
- 有关 JSF 技术的更多信息

4.1 什么是 JSF 应用

JSF 应用从功能上来说类似于其他类型的 Java Web 应用。典型的 JSF 应用包括如下部分：

- 包含组件及其布局的一组网页；

- 一组用于将组件添加至网页的标签;
- 一组后台bean, 它们是定义属性和页面上组件功能的一种JavaBeans;
- 一个Web部署描述文件 (web.xml文件);
- 作为可选项, 一个或多个应用程序配置资源文件 (如faces-config.xml文件), 用于定义页面的导航规则并配置bean和其他自定义对象, 如自定义组件;
- 作为可选项, 是一组由开发人员构建的自定义对象, 可以是自定义组件、验证器、转换器和监听器;
- 一组实现页面中自定义对象的自定义标签。

图4-1展示了典型的JSF应用中客户端与服务器间的交互过程。实现JSF技术的Web容器负责生成网页, 作为对客户端请求的应答。

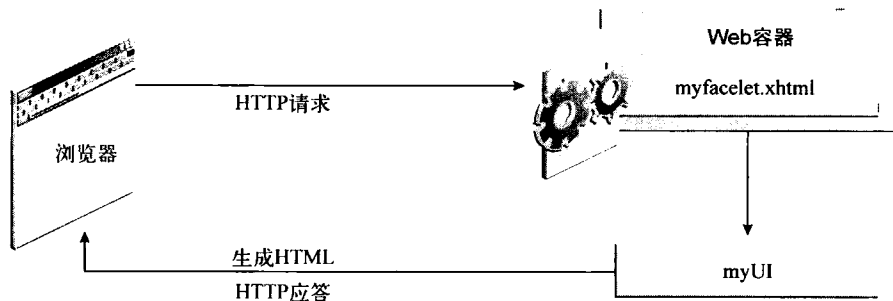


图4-1 JSF页面的请求与应答过程

网页myfacelet.xhtml是使用JSF组件标签创建的。组件标签用于将组件添加至视图 (图中以myUI表示), 视图是页面在服务器端的表示。除组件之外, 网页也可以引用对象, 例如:

- 注册至组件中的事件监听器、验证器和转换器;
- JavaBeans组件, 可以接收数据并实现特定的功能。

最终生成的页面作为对请求的应答返回至客户端。Web容器基于服务器端的视图, 将生成的结果以HTML或XHTML的形式输出给客户端 (如浏览器)。

4.2 JSF 技术的优势

使用JSF技术的一大优势在于它将Web应用程序的行为和展现有效隔离。JSF应用程序可以将HTTP请求映射为对特定组件的事件处理, 并且以有状态对象的方式在服务器上管理组件。JSF技术允许构建将页面行为与其展现有效隔离的Web应用, 而这在以前是通过传统的客户端UI架构实现的。

展现与其逻辑隔离的另一个好处是, Web应用开发团队的每一位成员能够更加专注于开发过程中的某一部分。同时, 这种隔离机制提供了一个简单的编程模型, 用于将开发过程中的每个部分有机地联系在一起。例如, 没有编程经验的页面设计人员能够使用JSF技术提供的标签, 将网

页与服务器端的对象关联起来，而无需编写任何代码。

JSF技术的另一个重要目标是使开发人员充分利用熟知的组件及Web层概念，而不受特定的脚本技术或标记语言的限制。如图4-2所示，JSF技术提供的API直接位于Servlet层的API之上。

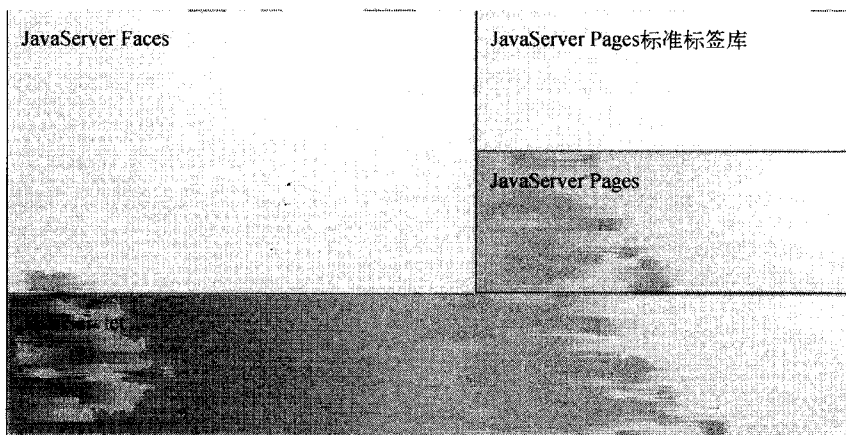


图4-2 Java Web应用技术

这种API层次结构支持多种应用开发场景，比如使用不同的表现层技术、利用组件类直接创建新的自定义组件，以及为不同的客户端设备产生特定输出。

Facelets技术，作为JSF 2.0规范的一部分，是目前推崇的一种表现层技术，用以构建基于JSF技术的Web应用程序。更多关于Facelets技术特征的信息，请参见第5章。

Facelets技术具有如下优势。

- ❑ 借助于模板化和复合组件功能，组件的代码可以被重用并扩展。
- ❑ 当使用JSF的注解功能时，可以将后台bean自动注册为JSF应用可以引用的资源。除此之外，隐式导航规则允许开发人员快速地配置页面导航。这些特点减少了应用开发过程中的手工配置环节。
- ❑ 最重要的是，JSF技术为管理组件状态、处理组件数据、验证用户输入和处理事件提供了灵活的架构支持。

4.3 创建简单的 JSF 应用

JSF技术提供了简单的且用户友好的Web应用创建过程。开发简单的JSF应用程序通常需要完成如下任务：

- ❑ 开发后台bean；
- ❑ 添加托管bean的声明；
- ❑ 使用组件标签创建网页；
- ❑ 映射FacesServlet实例。

本节将展示简单的JSF Facelets应用创建过程，以此介绍上述这些任务。

示例程序Hello包括一个后台bean和一个网页。当客户端访问该应用时，网页将输出一条HelloWorld消息。这个示例应用位于*tut-install/examples/web/hello*目录中。通过阅读应用程序组件的代码，我们可以了解这一应用的开发过程。

4.3.1 开发后台bean

如本章前文所述，后台bean作为一种托管bean，是通过JSF技术管理的一种JavaBeans组件。页面中的组件与实现应用逻辑的后台bean相关联。后台bean示例Hello.java包括下面的代码：

```
package hello;

import javax.faces.bean.ManagedBean;

@ManagedBean
public class Hello {

    final String world = "Hello World!";

    public String getworld() {
        return world;
    }
}
```

后台bean示例程序将字符串"Hello World"赋值给变量world。`@ManagedBean`注解将后台bean作为一种资源注册给JSF实例。关于托管bean和注解的更多内容，参见第9章。

4.3.2 创建网页

在典型的Facelets应用中，网页是用XHTML编写的。beanhello.xhtml文件表示的示例网页是一个简单的XHTML页面。它的内容如下：

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Facelets Hello World</title>
  </h:head>
  <h:body>
    #{hello.world}
  </h:body>
</html>
```

Facelets XHTML页面还可以包括其他元素，详见本书后面的介绍。

这个网页以EL（Expression Language，表达式语言）的形式，通过表达式`#{hello.world}`关联后台bean，从后台bean的Hello实例中读取属性world的值。注意，hello指向后台bean Hello。如果在注解`@ManagedBean`中没指定名称，则总是通过与类名相同但首字母小写的方式访问后台bean。

关于使用EL的更多内容，参见第6章。关于Facelets技术的更多内容，参见第5章。关于JSF编程模型及构建基于JSF技术的网页，参见第7章。

4.3.3 映射FacesServlet实例

最后一个步骤需要通过Web部署描述文件(web.xml)映射FacesServlet。典型的FacesServlet映射方式如下:

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

上述文件内容是典型的JSF程序中 Web部署描述文件的一部分。Web部署描述文件还可以包含其他与JSF应用程序配置相关的内容,但本例未涉及。

如果使用集成开发环境(如NetBeans IDE), FacesServlet的映射工作将自动完成。

4.3.4 hello应用程序的生命周期

每一个Web应用程序都有生命周期。通常的任务都是在Web应用程序的生命周期内执行,如处理请求、解析参数、修改和保存状态以及为浏览器生成网页。一些Web应用框架隐藏了生命周期的细节,而有些应用框架只能采用手工管理的方式。

默认情况下,JSF会自动处理绝大多数生命周期内的事务。然而,它也将生命周期内的状态信息提供给开发人员。因此,只要应用程序需要,我们可以修改事务的逻辑或执行不同的操作。

初级用户不必了解JSF应用的生命周期,但这对于较复杂的应用会有帮助。

JSF应用的生命周期从客户端发起页面请求开始,到服务器将页面作为应答返回结束。JSF应用的生命周期包括两个主要阶段,即执行和生成。

执行阶段有如下动作:

- 创建或恢复应用程序视图;
- 传递请求参数值;
- 组件值的转换和验证;
- 用组件的值更新后台bean;
- 调用应用逻辑。

第一次请求时,仅创建视图。对于随后(或回传)的请求,执行其他部分或全部动作。

在生成阶段,作为对客户端的应答生成请求的视图。生成通常是产生输出(如HTML或XHTML)的过程,输出将被客户端(通常是浏览器)读取。

下面关于JSF示例程序生命周期的简短描述概括了发生在幕后的事情。

当将hello应用部署至GlassFish服务器上,它将经历下面这些阶段:

- (1) 当构建hello应用并将其部署至GlassFish服务器上时,应用处于未初始化状态;
- (2) 当客户端发起对beanhello.xhtml页面的初始请求, Facelets应用hello被编译;

(3) 经过编译的 Facelets 应用执行，为 hello 应用创建新组件树，并把该组件树置于 FacesContext 中；

(4) 用 `hello.world`（EL 表达式）表示的组件及与之关联的后台 bean 属性填充组件树；

(5) 基于新的组件树生成视图；

(6) 为客户端生成应答页面；

(7) 组件树被自动销毁；

(8) 在后续（回传）的请求中，重建组件树并恢复其状态。

获取更多关于 JSF 生命周期的信息，请参考 JSF 规范 2.0 版。

▼ 在 NetBeans IDE 中编译、打包、部署和运行应用

(1) 在 NetBeans IDE 中单击 **File** → **Open Project**。

(2) 在 **Open Project** 对话框中，选择：

`tut-install/examples/web`

(3) 选择 **hello** 目录。

(4) 勾选 **Open as Main Project**。

(5) 单击 **Open Project**。

(6) 在 **Projects** 标签中，右键单击 **hello** 项目并选择 **Run**。

这个步骤将编译、组装和部署应用，并启动一个 Web 浏览器窗口显示下面这个 URL：

`http://localhost:8080/hello`

输出如下所示：

Hello World!

4.4 有关 JSF 技术的更多信息

有关 JSF 技术的更多信息，可以参考如下网站。

❑ **JavaServer Faces 规范 2.0 版：**

<http://jcp.org/en/jsr/detail?id=314>

❑ **JavaServer Faces 技术网站：**

<http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>

❑ **JavaServer Faces 2.0 版下载站点：**

<http://www.oracle.com/technetwork/java/javaee/download-139288.html>

❑ **Mojarra（JavaServer Faces 2.0）版本说明：**

<https://jaserverfaces.java.net/nonav/rlnotes/2.0.0/index.html>

Facelets一词指的是JSF技术所用的视图定义语言。JSP技术早先用作JSF的表现技术，它并不支持JSF 2.0的所有新特性。在JSF 2.0里，JSP技术作为表现技术已经过时了。Facelets是JSF规范的一部分，同时也是现今构建JSF应用时备受推崇的表现技术。

本章内容

- 什么是Facelets
- 开发简单的Facelets应用
- 模板化
- 复合组件
- 资源

5.1 什么是 Facelets

Facelets是一种强大，但轻量级的页面定义语言，它使用HTML风格的模板创建JSF视图，还可以创建组件树。Facelets有着如下特性：

- 使用XHTML创建Web页面；
- 除了支持JSF和JSLT外，还支持Facelets标签库；
- 支持表达式语言；
- 组件和页面的模板化。

对于大规模的开发项目来说，使用Facelets有着如下优势：

- 借助模板技术以及复合组件技术实现代码复用；
- 通过定制提供对组件和其他服务器端对象的功能扩展；
- 编译速度更快；
- 在编译阶段使用表达式语言执行验证逻辑；
- 网页高性能渲染技术。

简单来说，使用Facelets技术可以减少花在开发和部署上的时间和工作量。

Facelets视图通常为XHTML页面。JSF支持遵照XHTML Transitional DTD (Document Type

Definition，文档类型定义，参见http://www.w3.org/TR/xhtml1/#a_dtd_XHTML-1.0-Transitional）创建的XHTML文档。依照惯例，XHTML页面文件的扩展名为.xhtml。

JSF技术支持一些标签库，通过这些标签库可以把组件添加到网页里去。为了支持JSF的标签库机制，Facelets使用XML命名空间声明。表5-1列举了Facelets支持的标签库。

表5-1 Facelets支持的标签库

标 签 库	URI	前 缀	例 子	内 容
JSF Facelets标签库	http://java.sun.com/jsf/facelets	ui:	ui:component ui:insert	模板标签
JSF HTML标签库	http://java.sun.com/jsf/html	h:	h:head h:body h:outputText h:inputText	所有UI组件的JSF组件标签
JSF核心标签库	http://java.sun.com/jsf/core	f:	f:actionListener f:attribute	JSF自定义行为的标签，独立于任何渲染器
JSTL核心标签库	http://java.sun.com/jsp/jstl/core	c:	c:forEach c:catch	JSTL 1.1核心标签
JSTL功能标签库	http://java.sun.com/jsp/jstl/functions	fn:	fn:toUpperCase fn:toLowerCase	JSTL 1.1功能标签

除了这些内容，Facelets还支持复合组件的标签，我们可以为这种标签声明自定义的前缀。关于复合组件，详见5.4节。

由于JSF支持JSP 2.1中定义的表达式语言语法，Facelets可以使用EL表达式引用后台bean的属性和方法。EL表达式可以用来绑定组件对象（或者值）到托管bean的方法，也可以绑定托管bean的属性。有关EL的详细内容，参见9.1.2节。

5.2 开发简单的 Facelets 应用

这部分内容讲解开发JSF应用的一般性步骤。这一开发过程通常需要完成如下任务：

- ❑ 开发后台bean；
- ❑ 使用组件标签创建页面；
- ❑ 定义页面导航；
- ❑ 映射FacesServlet实例；
- ❑ 添加托管bean声明。

5.2.1 创建Facelets应用

本处用到的示例为guessnumber应用。这个应用的第一个页面要求用户猜测并输入一个0到10之间的数字，系统会产生一个随机数，然后比较用户的输入和这个随机数，并在之后出现的页面中告诉用户猜测结果是否正确。

1. 开发后台bean

在典型的JSF应用里，应用的每一个页面都对应着一个后台bean（后台bean是托管bean的一

种)。后台bean定义了与组件相关的方法和属性。

如下的UserNumberBean就是一个托管bean，它实现的功能是生成一个0到10之间的随机数。

```
package guessNumber;

import java.util.Random;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class UserNumberBean {

    Integer randomInt = null;
    Integer userNumber = null;
    String response = null;
    private long maximum=10;
    private long minimum=0;

    public UserNumberBean() {
        Random randomGR = new Random();
        randomInt = new Integer(randomGR.nextInt(10));
        System.out.println("Duke's number: " + randomInt);
    }

    public void setUserNumber(Integer user_number) {
        userNumber = user_number;
    }

    public Integer getUserNumber() {
        return userNumber;
    }

    public String getResponse() {
        if ((userNumber != null) && (userNumber.compareTo(randomInt) == 0)) {
            return "Yay! You got it!";
        } else {
            return "Sorry, " + userNumber + " is incorrect.";
        }
    }

    public long getMaximum() {
        return (this.maximum);
    }

    public void setMaximum(long maximum) {
        this.maximum = maximum;
    }

    public long getMinimum() {
        return (this.minimum);
    }

    public void setMinimum(long minimum) {
        this.minimum = minimum;
    }
}
```

请注意,该应用使用了@ManagedBean注解,它把后台bean注册为JSF应用的资源。@SessionScoped

注解把这个后台bean的作用域注册为会话级（session）。

2. 创建Facelets视图

创建页面或者视图，这是网页制作者的职责。这项工作涉及在页面上放置组件、把组件和后台bean的值以及属性关联起来，还要为组件注册转换器、验证器以及监听器。

具体到示例应用，XHTML Web页面充当前端。第一个页面叫做greeting.xhtml，下面让我们仔细看看这个页面的组成。

这个页面的第一部分声明了页面的类型，这里是XHTML类型：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

接下来的内容声明了页面中用到的标签库的XML命名空间：

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core">
```

接下来的内容使用相应的标签把组件添加到Web页面里去：

```
h:head>
  <title>Guess Number Facelets Application</title>
</h:head>
<h:body>
  <h:form>
    <h:graphicImage value="#{resource['images:wave.med.gif']}" />
    <h2>
      Hi, my name is Duke. I am thinking of a number from
      #{userNumberBean.minimum} to #{userNumberBean.maximum}.
      Can you guess it?
    <p></p>
    <h:inputText
      id="userNo"
      value="#{userNumberBean.userNumber}">
      <f:validateLongRange
        minimum="#{userNumberBean.minimum}"
        maximum="#{userNumberBean.maximum}" />
    </h:inputText>

    <h:commandButton id="submit" value="Submit"
      action="response.xhtml" />
    <h:message showSummary="true" showDetail="false"
      style="color: red;
      font-family: 'New Century Schoolbook', serif;
      font-style: oblique;
      text-decoration: overline"
      id="errors1"
      for="userNo" />
    </h2>
  </h:form>
</h:body>
```

注意如下标签的用法：

- Facelets HTML标签（以h:开头）用来添加组件；
- Facelets核心标签f:validateLongRange用于验证用户输入的有效性。

input Text组件接收用户的输入，并为后台bean的属性userNumber设置值，这是通过EL表达式#{userNumberBean.userNumber}实现的。JSF标准验证器f:validateLongRange用来验证用户输入的数值是否有效。

图片文件wave.med.gif是作为资源添加到页面里的。有关资源的详细内容，参见5.5节。

页面里有一个commandButton组件，该组件的ID为submit。当按下其对应的按钮时，这一组件会执行验证输入数据有效性的逻辑并把浏览器重定向到另外一个页面（即response.xhtml），用它显示对用户输入的应答。

现在可以创建第二个页面了，也就是response.xhtml。具体内容如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

  <h:head>
    <title>Guess Number Facelets Application</title>
  </h:head>
  <h:body>
    <h:form>
      <h:graphicImage value="#{resource['images:wave.med.gif']}" />
      <h2>
        <h:outputText id="result" value="#{userNumberBean.response}" />
      </h2>
      <h:commandButton id="back" value="Back" action="greeting.xhtml" />
    </h:form>
  </h:body>
</html>
```

5.2.2 配置应用

配置JSF应用涉及几项不同的配置任务，如在Web部署描述文件（如Web.xml）中映射Faces Servlet、添加托管bean声明、制定页面导航规则以及为应用中配置文件（比如faces-config.xml）的资源包添加声明。

如果使用NetBeans IDE，Web部署描述文件会自动生成，其文件名为web.xml。打开该文件，并把默认的欢迎页面文件由index.xhtml改成greeting.xhtml。请看Web.xml代码，修改的部分已用粗体标注：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
```

```

</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<session-config>
  <session-timeout>
    30
  </session-timeout>
</session-config>
<welcome-file-list>
  <welcome-file>faces/greeting.xhtml</welcome-file>
</welcome-file-list>
</web-app>

```

请注意参数PROJECT_STAGE的用法。这是一个上下文参数，用来表明JSF应用处于软件开发周期的哪个阶段。

应用所处的生命周期阶段将影响应用的行为。举例来说，如果项目阶段是Development，则调试信息会自动产生，用户调试会方便一些。如果没有定义该参数，其PROJECT_SFAGE的默认值为Production。

5

5.2.3 构建、打包、部署以及运行guessnumber Facelets示例

我们可以使用NetBeans IDE或Ant工具构建、打包、部署以及运行guessnumber Facelets示例。该示例的源代码位于目录*tut-install/examples/web/guessnumber*下。

▼ 用NetBeans IDE构建、打包以及部署guessnumber示例应用

- (1) 在NetBeans IDE里，选择菜单File→Open Project。
 - (2) 在Open Project对话框里，定位到 *tut-install/examples/web/*。
 - (3) 选择guessnumber目录。
 - (4) 勾选 Open as Main Project复选框。
 - (5) 单击Open Project。
 - (6) 在Projects标签下，右击guessnumber项目，然后选择Deploy。
- 这样示例应用就可以构建并部署到GlassFish服务器实例上了。

▼ 用Ant工具构建、打包以及部署guessnumber示例应用

- (1) 打开一个终端窗口，并切换目录到：
tut-install/examples/web/guessnumber/
- (2) 执行如下命令：

ant

该命令会调用默认的任务（target），也就是构建和打包应用到一个WAR文件（guessnumber.war），该文件位于dist目录下。

- (3) 确保GlassFish已经启动。

(4) 要部署应用，输入如下命令：

```
ant deploy
```

▼ 运行guessnumber示例应用

(1) 打开Web浏览器。

(2) 在Web浏览器地址栏里输入以下网址：

<http://localhost:8080/guessnumber>

网页如图5-1所示。

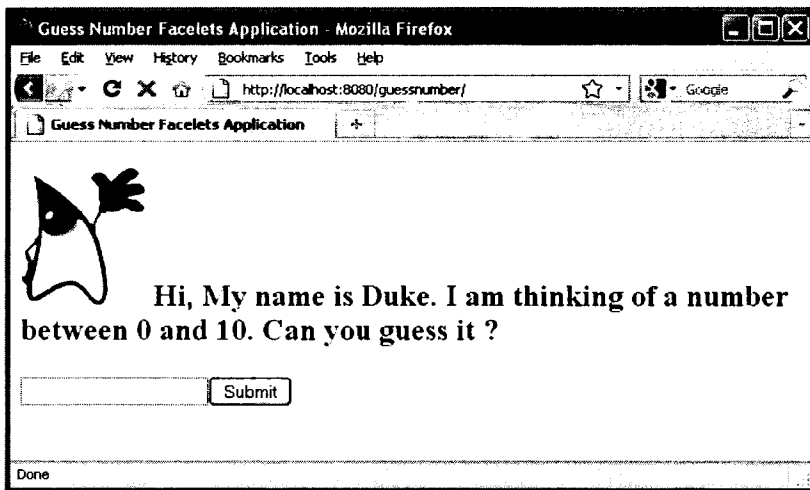


图5-1 运行guessnumber应用

(3) 在文本框里，输入一个0到10之间的数字，然后单击Submit按钮。此时出现一个新的页面，提示用户所猜的数字是否正确。

(4) 如果数字猜错了，可以单击Back按钮回退到主页面。

我们可以接着猜数字，直到猜对了为止。

5.3 模板化

JSF技术为实现可扩展、可复用的用户界面提供了工具。模板化是一个有用的Facelets特性，允许把事先创建好的页面，作为应用里其他页面的模板。通过模板化技术，我们可以复用代码，避免重复创建结构相似的页面。如果应用有很多的页面，则模板化还可以帮助我们维护一个统一的界面外观。

表5-2列举了可以用于模板化的Facelets标签以及它们的相应功能。

表5-2 Facelets模板标签

标 签	功 能
ui:component	定义组件，创建实例，并把它添加到组件树
ui:composition	定义页面组合，有可能使用模板，标签以外的内容将被忽略
ui:debug	定义一个调试组件，创建组件实例，并把它添加到组件树
ui:decorate	类似复合标签，只是不忽略这个标签以外的内容
ui:define	定义内容，由模板把内容添加到页面
ui:fragment	类似组件标签，只是不忽略这个标签以外的内容
ui:include	引用一个外部文件，该文件封装了可被其他页面复用的内容
ui:insert	把内容添加到模板里
ui:param	用来传递参数到一个包含文件里去
ui:repeat	循环标签（比如c:forEach 或者 h:dataTable）的替代方案
ui:remove	从页面里移除内容

有关Facelets模板标签的更多信息，可以参阅如下网址处的文档。

<http://docs.oracle.com/javaee/6/jaserverfaces/2.0/docs/pdldocs/facelets/>

Facelets标签库包括一个主模板标签ui:insert。用这一标签创建的模板页面允许我们定义网页的默认结构。模板页面可以作为模板供其他页面复用，这些根据模板创建出的新页面常称为派生页面。

这里有一个模板示例，对应的文件名是template.xhtml：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html">

    <h:head>
        <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8" />
        <link href="./resources/css/default.css"
            rel="stylesheet" type="text/css" />
        <link href="./resources/css/cssLayout.css"
            rel="stylesheet" type="text/css" />
        <title>Facelets Template</title>
    </h:head>

    <h:body>
        <div id="top" class="top">
            <ui:insert name="top">Top Section</ui:insert>
        </div>
        <div>
            <div id="left">
                <ui:insert name="left">Left Section</ui:insert>
            </div>
        </div>
    </h:body>
</html>
```

```

    </div>
    <div id="content" class="left_content">
      <ui:insert name="content">Main Content</ui:insert>
    </div>
  </div>
</h:body>
</html>

```

该示例定义了一个XHTML页面，该页面被划分成3个区域：头部区域、左侧区域以及右下的主要区域。这些区域有着自己的样式表。应用里的其他页面可以复用这个页面的布局。

派生页面使用`ui:composition`标签来调用这一模板。看下面的例子，这是一个文件名为`templateclient.xhtml`的派生页面，它调用了先前的那个模板文件`template.xhtml`。派生页面允许我们通过`ui:define`标签添加新内容。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">

  <h:body>
    <ui:composition template="./template.xhtml">
      <ui:define name="top">
        Welcome to Template Client Page
      </ui:define>

      <ui:define name="left">
        <h:outputLabel value="You are in the Left Section"/>
      </ui:define>

      <ui:define name="content">
        <h:graphicImage value="#{resource['images:wave.med.gif']}" />
        <h:outputText value="You are in the Main Content Section"/>
      </ui:define>
    </ui:composition>
  </h:body>
</html>

```

我们可以使用NetBeans IDE创建Facelets模板以及派生页面。有关创建页面的更多内容，参见<http://netbeans.org/kb/docs/web/jsf20-intro.html>。

5.4 复合组件

JSF技术通过Facelets提供了复合组件的概念。我们可以将复合组件看做一种特殊类型的模板，可以把它当做组件来用。

任何组件从本质上来说都是一段实现了特定功能、可以复用的代码。举例来说，`inputText`组件可以接收用户的输入。组件也可以注册验证器、转换器以及监听器，以实现特定的预定义的功能。

复合组件就是由一组标记标签和其他已经存在的组件组合而成的新组件。复合组件也可以复用，由于它是用户自己创建的组件，因此可以有客户自己定制的功能，可以有自己的验证器、转换器以及监听器，这一点和其他JSF组件一样。

通过Facelets技术，任何包含了标记标签以及其他组件的XHTML页面都可以转换成复合组件。使用相关的资源工具，复合组件可以保存在一个库里，放到定义好的资源目录中，然后就能够被应用访问。

表5-3列举了最为常用的复合标签以及对应的功能。

表5-3 复合组件的标签

标 签	功 能
composite:interface	声明复合组件的调用接口。复合组件可以用作单个组件，其特性就是接口里声明的特性的集合
composite:implementation	定义复合组件的实现。如果出现了composite:interface元素，则必须有一个对应的composite:implementation元素
composite:attribute	定义一个属性，该属性可赋予本标签声明所在的复合组件实例
composite:insertChildren	在调用者页面中，复合组件的子组件或模板文本会被放置到新的位置，该位置位于composition:implementation标签内，composite:insertChildren标签所在的位置
composite:valueHolder	用composite:interface声明复合组件，在声明里定义一个ValueHolder（适合作为调用者页面里的对象的绑定目标）
composite:editableValueHolder	用composite:interface声明复合组件，在声明里定义一个EditableValueHolder，（适合作为调用者页面里的对象的绑定目标）
composite:actionSource	用composite:interface声明复合组件，在声明里定义一个ActionSource2（适合作为调用者页面里的对象的绑定目标）

有关Facelets组件标签的完整列表以及详细信息，可以查看如下网址处的文档：

<http://docs.oracle.com/javaee/6/javaxserverfaces/2.0/docs/pdldocs/facelets/>

下面的例子展示了一个复合组件，可以接收页面上输入的电子邮件地址：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:composite="http://java.sun.com/jsf/composite"
xmlns:h="http://java.sun.com/jsf/html">

  <h:head>
    <title>This content will not be displayed</title>
  </h:head>
  <h:body>
    <composite:interface>
      <composite:attribute name="value" required="false"/>
    </composite:interface>

    <composite:implementation>
      <h:outputLabel value="Email id: "></h:outputLabel>
      <h:inputText value="#{cc.attrs.value}"></h:inputText>
    </composite:implementation>
  </h:body>
</html>
```

请注意，当定义组件值的时候，我们使用了cc.attrs.value。cc是复合组件在JSF里专用的保留字。表达式#{cc.attrs.attribute-name}用来访问在复合组件接口里定义的属性，本例中为value属性。

前述例子中的内容保存在名为email.xhtml的文件中，这一文件位于应用的根目录下的resources/emcomp子目录中。这个目录被JSF视作库文件，我们可以从这个库里访问UI组件。更多关于资源的内容，参见5.5节。

使用这个复合组件的页面，通常叫做调用者页面（using page）。调用者页面在xml命名空间声明里包含了对复合组件的引用：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:em="http://java.sun.com/jsf/composite/emcomp/">

  <h:head>
    <title>Using a sample composite component</title>
  </h:head>

  <body>
    <h:form>
      <em:email value="Enter your email id" />
    </h:form>
  </body>
</html>
```

本地的复合组件库定义在xml命名空间里，使用的声明是xmlns:em="http://java.sun.com/jsf/composite/emcomp/"。访问组件可以使用标签em:email。前面的例子内容可以保存在一个叫emuserpage.xhtml的文件里，保存在Web根目录下。当编译并部署到服务器上以后，可以通过如下网址访问：

<http://localhost:8080/application-name/faces/emuserpage.xhtml>

5.5 资源

Web资源指的是为了让Web应用正常显示，应用所需要的文件和数据，包括图片文件、脚本文件以及用户创建的一些组件库。资源必须集中放置到一个标准的位置，必须满足如下要求之一：

- 打包至Web应用的资源必须位于Web应用根目录下resources的子目录中，即resources/resource-identifier；
- 通过classpath引用的资源必须位于Web应用包的META-INF/resources目录下的一个子目录中，即META-INF/resources/resource-identifier。

JSF运行时会上述目录中按顺序搜索资源。

资源标识符是不能重复的字符串，其格式要求如下：

```
[locale-prefix/] [library-name/] [library-version/] resource-name [/resource-version]
```


位于中括号[]中的内容都是可选的，这表明只有资源的名称（*resource-name*）才是必需的。通常情况下，资源的名称通常就是文件名。

我们可以把资源所在的目录视作库目录，这样一来，应用里的其他组件就可以访问这些保存在resources目录下的文件（比如复合组件或者模板），并可以用它创建新的资源实例。

本章介绍EL（Expression Language，表达式语言）。EL提供了一种重要的机制，它使得表现层（网页）可以与应用逻辑（后台bean）通信。EL可以用在JSP技术里，也可以用在JSF技术里。

本章内容

- EL综述
- 即时求值和延后求值语法
- 值表达式和方法表达式
- 定义标签属性类型
- 文本表达式
- 运算符
- 保留字
- EL表达式的例子

6.1 EL 综述

有了EL，页面设计师可以用简单的表达式动态地访问JavaBeans组件里的数据。例如，如下条件判定标签里的test属性就用了EL表达式，其目的是比较作用域为session，名为cart的bean中numberOfItems属性值是否大于0：

```
<c:if test="${sessionScope.cart.numberOfItems > 0}">
...
</c:if>
```

JSF技术使用EL的如下功能：

- 表达式的即时求值以及延后求值；
- 读取或者设置数据；
- 调用方法。

大家可以参阅9.1.2节，了解一下如何在JSF应用里使用EL。

概括地说，EL提供了一种机制，使得使用简单的表达式就可以实现如下任务：

- 动态地读取保存在应用的JavaBeans组件中的数据，以及各种数据结构或隐式对象中的数据；

- 动态地写数据到JavaBeans组件里去，比如用户在表单里的输入；
- 调用静态或者公有的方法；
- 动态执行数学运算。

自定义标签的属性能够接收的表达式类型，也可以用EL指定，具体的类型如下所述。

- 表达式的即时求值以及延后求值 即时求值表达式是指实现技术（比如JSF）立即计算出表达式的值，而延后求值表达式则是指使用EL的底层技术延后计算表达式的值。
- 值表达式及方法表达式 值表达式引用数据，而方法表达式则调用方法。
- 右值表达式及左值表达式 右值表达式只能用于读取数据，而左值表达式则可以读写外部对象的值。

除此之外，EL提供解析表达式的扩展API。用户可以借此自定义解析器以处理EL当前尚不支持的一些表达式。

6.2 即时求值和延后求值语法

EL支持表达式的两种求值方式，即即时求值和延后求值。即时求值是指页面第一次被调用时就计算表达式的值，并返回结果，而对于延后求值来说，则是推迟到页面生命周期中某个合适的阶段才求值。

即时求值表达式的语法是`${}`，延后求值表达式的语法是`#{}`。

由于页面的生命周期有多个阶段，JSF技术大多使用延后求值表达式。在生命周期内，组件事件的处理、数据有效性的验证以及其他事项，通常会以一定的顺序来执行。因此，JSF实现必须支持延后求值，以将表达式的求值推迟至生命周期里某个合适的时间点。

其他使用EL的技术可能会出于其他原因使用延迟计算表达式。

6.2.1 即时求值

所有使用`${}`语法的表达式都是即时求值表达式。这些表达式只能用作模板里的文本或者标签属性的值，前提是该标签属性可以在运行期间接收值。

下面的例子演示了一个标签，其`value`属性引用了一个即时求值表达式，用于计算出购物车里物品的总价款。其对应bean的名字叫`cart`，是一个作用域为会话的bean。

```
<fmt:formatNumber value="${sessionScope.cart.total}"/>
```

JSF立即对表达式`${sessionScope.cart.total}`进行计算，并把值转化为合适的类型，最后把转换后的值传递给标签处理程序。

立即求值表达式只能是只读型的。前面这个例子只能得到购物车里物品的总价款的值，而不能设定它。

6.2.2 延后求值

延后求值表达式使用`#{expr}`语法，其值是在页面生命周期的某个阶段计算出来的，而何时

计算是由使用表达式的具体技术决定的。对于JSF技术来说，它的控制器（controller）可以在页面生命周期的各个阶段计算表达式的值，其时间点取决于表达式在页面里是被如何使用的。

下面的例子演示了一个JSF `inputText` 标签。它表示一个文本框组件，用户可以在这个文本框里输入一个值。`inputText` 的属性 `value` 引用了一个延后求值表达式，它指向 `customer` 这个bean的 `name` 属性。

```
<h:inputText id="name" value="#{customer.name}" />
```

第一次访问包含该标签的页面时，在显示应答页面的时候，JSF实现会计算表达式 `#{customer.name}` 的值。在这个阶段里，此表达式仅仅是从 `customer` 那里取得 `name` 属性的值，这和即时求值时一样。

对于后续的请求，JSF实现会在页面生命周期的不同阶段对表达式求值，如从请求参数里取出数值的时候、验证数值的时候以及把值传给bean的时候。

正如本例所示，延后求值表达式可以用作：

- 既可以用于读取数据，也可以用于写数据的值表达式；
- 方法表达式。

值表达式（包括即时求值表达式和延后求值表达式）和方法表达式详见6.3节。

6.3 值表达式和方法表达式

EL定义了两类表达式：值表达式和方法表达式。值表达式可以用于取值，也可以用于设置值。方法表达式可以调用类的方法，并在调用完成后返回一个值。

6.3.1 值表达式

值表达式还可以分为右值表达式和左值表达式两种。右值表达式是指只能读取数据，但是不能写数据的表达式，而左值表达式既可以读数据又可以写数据。

使用 `${}` 的表达式都是立即求值的，而且都是右值表达式。使用 `#{}` 的表达式，都是延后求值的，可以用作左值表达式，也可以用作右值表达式。考虑如下两个值表达式：

```
${customer.name}
```

```
#{customer.name}
```

前一个值表达式使用立即求值的语法，后一个值表达式使用延后求值的语法。第一个表达式访问的是 `name` 这个属性，取得其值后放到应答消息里，并最终显示在页面上。第二个表达式可以达到和第一个表达式一样的效果，但在所使用的标签技术允许的前提下，标签处理程序可以延后表达式的求值，使其发生在页面生命周期里靠后的一个合适时间点。

对于JSF技术来说，第一次请求该页面时，第二个标签的表达式也是马上计算的。对于这种情况，表达式用作右值表达式。不过对于后续的请求，这一表达式可以把用户在页面上输入的值转给后台bean的 `name` 属性。对于这种情况，表达式用作左值表达式。

1. 使用值表达式引用对象

左值表达式和右值表达式都可以引用如下的对象以及它们的属性：

- JavaBeans 组件；
- 集合类型（collection）；
- Java SE 枚举类型；
- 隐式对象。

为了引用这些对象，可以写一个表达式。该表达式从形式上来看只有一个变量，该变量就是对象的名字。如下表达式就是对一个后台bean(JavaBeans 组件)的引用，该bean的名字是customer。

```
${customer}
```

通过PageContext.findAttribute(String)，Web容器可以找到并求出出现在表达式里的变量的值，这里传入的参数String就是变量的名字。举例来说，当对`${customer}`表达式进行求值时，这一容器会查找页面、请求、会话以及应用作用域里是否存在customer，找到就返回其值。如果没找到customer，就返回null。

如果实现了自定义的EL解析器，则可以改变解析变量的方式。例如，我们可以提供一个EL解析器，该解析器拦截名为customer的对象，因此`${customer}`就返回自定义EL解析器所提供的值。

用表达式引用enum常量，可以使用String字面量。举个例子，考虑下面的Enum类：

```
public enum Suit {hearts, spades, diamonds, clubs}
```

表达式要引用Suit里的常量，比如Suit.hearts，就使用String字面量hearts。依据所在的上文，String字面量会被自动转换为enum常量。举例来说，在如下表达式里，mySuit是类Suit的一个实例。hearts在和mySuit做比较之前先转换为Suit.hearts。

```
${mySuit == "hearts"}
```

2. 用值表达式引用对象的属性

要引用bean的属性或者某个enum类的实例、集合里的元素或者隐式对象的属性，可以使用.或者[]符号来实现。

如果想引用bean的属性，比如customer这个bean的name属性，可以有两种写法，即`${customer.name}`和`${customer["name"]}`。中括号里的部分是String，也就是要引用的属性名。

可以使用双引号或者单引号引用字符串常量，还可以使用中括号和.的组合，例如：

```
${customer.address["street"]}
```

enum常量的属性也可以以同样的方式引用。不过，与JavaBeans组件的属性相比，Enum类的属性必须遵循JavaBeans 组件的规范，这也意味着属性必须至少有一个名为getProperty的方法。Property就是属性的名字，也就是表达式引用时所用的字符串。

例如，假定Enum类封装了我们所在星系中所有行星的名字，还提供了一个获取行星质量的方法。可以使用如下表达式引用Enum类Planet的getMass方法：

```
${myPlanet.mass}
```

如果要访问数组或者列表里的元素，可以使用能够转换为int类型的变量，或者使用[]符号并以int类型的数字作为下标（且无引号）。下面的例子可以解析成列表或者数组里的同一个元素，这里假定socks能够转换成int类型。

- `${customer.orders[1]}`
- `${customer.orders.socks}`

对于Map的情况，需以字符串键值（key）的方式访问Map里的元素，无需使用强制类型转换。

`${customer.orders["socks"]}`

右值表达式也可以直接引用非对象类型的值，比如数学运算的结果或者字符串常量，下面是几个例子：

- `${"literal"}`
- `${customer.age + 20}`
- `${true}`
- `${57}`

EL定义了如下常量类型：

- 布尔型——true或者false；
- 整型——同Java里的整型；
- 浮点型——同Java里的浮点型；
- 字符串——用单引号或者双引号引起来，"转义为\"，'转义为\'，\转义为\\；
- Null——null。

可以编写表达式以执行enum常量的有关运算。看下面的例子，考虑如下的Enum类：

```
public enum Suit {club, diamond, heart, spade}
```

在声明一个叫做mySuit的enum常量之后，可以编写如下表达式测试mySuit是不是等于spade：

```
${mySuit == "spade"}
```

当EL解析这个表达式的时候，EL解析机制会调Enum类的valueOf方法，传入参数分别为Suit类和spade的类型，如下所示：

```
mySuit.valueOf(Suit.class, "spade")
```

3. 值表达式的适用场合

值表达式使用\${}符号，可以应用在如下两种情况中：

- 静态文本；
- 任何可以接收表达式的标准标签或者自定义标签。

在静态文本里，表达式的值会在计算出结果之后，插入到当前输出。下面是一个嵌入在静态文本里的表达式示例。

```
<some:tag>
  some text ${expr} some text
</some:tag>
```

请注意，如果静态文本出现在标签体内，且标签体声明为tagdependent，表达式不会被求值。左值表达式只能用在那些能接收左值表达式的标签属性中。

借助于左值表达式和右值表达式，设置标签的属性值有如下3种方式：

❑ 只使用单个表达式：

```
<some:tag value="${expr}"/>
<another:tag value="#{expr}"/>
```

这两个表达式的值计算出来后，结果会转换成属性期望的类型。

❑ 使用一个或多个由文本分隔或包围的表达式：

```
<some:tag value="some${expr}${expr}text${expr}"/>
<another:tag value="some#{expr}#{expr}text#{expr}"/>
```

这种类型的表达式叫做复合表达式，按照从左到右的顺序求值。每一个嵌入在复合表达式里的表达式都要转换为String类型，然后和邻近的字符串拼接起来继续解析。结果字符串最终再转换为属性期望的类型。

❑ 只使用文本：

```
<some:tag value="sometext"/>
```

这种表达式叫做文本表达式。本例中，属性的String值转换为属性期望的类型。文本值表达式有着特别的语法规则，详见6.5节。当一个标签的属性是enum类型时，属性所用到的表达式必须是文本表达式。举例来说，标签属性可以使用表达式"hearts"来代表Suit.hearts。字符串"hearts"会转换成Suit类型，然后属性得到的值为Suit.hearts。

所有用来设置属性值的表达式都会被处理为上下文期望的类型。如果表达式计算的结果不匹配期望的数据类型，则其结果会被转换成期望的类型。例如，表达式\${1.2E4}提供了属性的值，类型为float，最后会做如下转换：

```
Float.valueOf("1.2E4").floatValue()
```

参考JSP 2.2表达式语言规范（可以在<http://jcp.org/aboutJava/communityprocess/final/jsr245/>处获得）的1.18节，那里面列举了所有的类型转换规则。

6.3.2 方法表达式

表达式语言的另外一个特性是支持延后的方法表达式。方法表达式可以调用bean的任何公有方法，并把方法的返回值作为表达式的求值结果。

在JSF技术里，组件标签代表页面里的一个组件。组件标签可以使用方法表达式调用方法来执行一些组件的处理流程。当处理组件产生的事件以及验证组件数据的有效性时，这些方法很有必要。看一个例子：

```
<h:form>
  <h:inputText
    id="name"
    value="#{customer.name}"
```

```

        validator="#{customer.validateName}"/>
<h:commandButton
    id="submit"
    action="#{customer.submit}" />
</h:form>

```

inputText标签显示为页面里的一个文本框，inputText标签有个属性叫validator，它引用了一个名为validateName的方法，该方法属于一个名为customer的bean。

因为方法的调用可以发生在页面生命周期的各个阶段，所以方法表达式必须使用#符号，以使其延后求值。

和左值表达式一样，方法表达式可以使用.和[]操作符。举个例子，#{object.method}和#{object["method"]}是等价的。中括号里的字符串转换为Java里的String类型，然后用于匹配类中方法的名字。一旦找到了这个方法，就执行它，或者返回该方法的相关信息。

方法表达式只能用于标签属性，且只能以如下两种方式使用。

□ 使用单个表达式，bean指的是JavaBeans组件，method指的是JavaBeans组件的方法：

```
<some:tag value="#{bean.method}"/>
```

这个表达式是一个方法表达式，会被传递给标签处理程序。对应bean里的方法会被延后调用。

□ 只使用文本：

```
<some:tag value="sometext"/>
```

方法表达式支持字符串，主要是为了支持JSF技术里的action属性。当这一方法表达式引用的方法被调用后，该方法会返回String类型的值，然后再转换成期望的返回值类型（该类型是定义在标签的标签库描述文件里的）。

参数化方法调用

EL支持在调用方法时传递参数。调用方法可以使用参数，而无需使用静态的EL函数。

.和[]操作符都可以用于调用带参数的方法，其语法如下所示：

□ *expr-a[expr-b](parameters)*

□ *expr-a.identifier-b(parameters)*

第一个表达式里，*expr-a*代表一个bean对象。*expr-b*计算后转换成一个字符串，代表*expr-a*对应的bean中的一个方法。第二个表达式里，*expr-a*代表一个bean对象，而*identifier-b*是一个字符串，代表这个bean中的一个方法。小括号里的*parameters*就是调用方法时传入的参数列表。*parameters*可以是0个或者多个值，也可以是表达式，参数之间以逗号隔开。

值表达式和方法表达式都支持参数。下面的这个例子修改自guessnumber应用，调用方法时传递了一个随机数，而不是像原来那样从界面上得到用户输入。

```
<h:inputText value="#{userNumberBean.userNumber('5')}">
```

上面的例子使用了值表达式。

下面的例子是一个JSF组件的标签，使用了方法表达式：

```
<h:commandButton action="#{trader.buy}" value="buy"/>
```


EL表达式`trader.buy`调用`trader`这个bean的`buy`方法。可以修改标签来传递一个参数。下面就是一个传递了参数的标签示例：

```
<h:commandButton action="#{trader.buy('SOMESTOCK')}}" value="buy"/>
```

上面的例子在调用`buy`方法时，传递了一个参数，参数的值为字符串`'SOMESTOCK'`（股票代码）。

有关EL新版本的更多信息，可以参阅<https://uel.dev.java.net>。

6.4 定义标签属性类型

正如前文所述，所有类型的表达式都可以用做标签属性的值。至于是哪种类型的表达式，以及表达式具体是如何求值的，是立即求值还是延后求值，这是由标签定义的属性类型决定的。这些标签都定义在PDL（Page Description Language，页面描述语言）文件里。

如果打算创建自定义标签，对于PDL文件里的每一个标签，都需要指明标签能接收哪种类型的表达式。表6-1列举了能够接收EL表达式的3种标签属性，并给出了它们所能接收的表达式示例，还列出了必须加到PDL文件中的属性类型定义。对于动态属性不能使用`#{}语法`。也就是说，这样的属性能在运行时接收动态计算的值。类似地，`${}`语法不能用于延后的属性。

6

表6-1 可以接收EL表达式的标签属性的定义

属性类型	例 子	属性的类型定义
动态属性	"literal"	<rtexprvalue>true</rtexprvalue>
	\${literal}	<rtexprvalue>true</rtexprvalue>
延后的值属性	"literal"	<deferred-value>
		<type>java.lang.String</type>
		</deferred-value>
	#{customer.age}	<deferred-value>
		<type>int</type>
		</deferred-value>
延后的方法属性	"literal"	<deferred-method>
		<method-signature>
		java.lang.String submit()
		</method-signature>
		</deferred-method>
	#{customer.calcTotal}	<deferred-method>
		<method-signature>
		double calcTotal(int, double)
		</method-signature>
		</deferred-method>

除了表6-1列举的标签属性，开发人员可以定义能同时接收动态表达式以及延后求值表达式的属性。在这种情况下，标签属性类型的定义既包括`rtexprvalue`定义（其值为`true`），也包括`deferred-value`或`deferred-method`（两者取其一）定义。

6.5 文本表达式

文本表达式求值后的结果要转换为文本，类型为String。文本表达式不使用\${}或者#{ }符号。如果文本表达式里包含\${}或#{ }，那就需要以如下方式进行转义。

❑ 创建复合表达式，如下所示：

```
${'${'}exprA}  
#{'#{'}exprB}
```

表达式的最终结果为字符串\${exprA}和#{exprB}。

❑ 使用转义字符\ \$和\ #来转义那些不转义便会被认作表达式的字符串：

```
\${exprA}  
\\#{exprB}
```

这一表达式的最终结果还是字符串\${exprA}和#{exprB}。

当对文本表达式求值时，其结果可以转换成另外一种类型。表6-2列举了不同的文本表达式，以及它们期望的类型和求值的结果。

表6-2 文本表达式

表 达 式	期望类型	求值结果
Hi	String	Hi
True	Boolean	Boolean.TRUE
42	int	42

文本表达式可以立即求值，也可以延后求值，既可以用作值表达式，也可以用作方法表达式。文本表达式的求值时间点取决于被调用的地方。如果使用了文本表达式的标签属性接收了延后求值的值表达式，那么当文本表达式引用一个值的时候，它的求值时间点在其生命周期的具体阶段取决于其他因素。这里所说的其他因素包括表达式的使用位置以及引用的对象。

当用作方法表达式时，引用的方法会被调用，并返回一个特定的字符串。举例来说，guessnumber应用里的commandButton 标签使用了一个文本方法表达式，求值得到的逻辑结果会通知JSF导航系统下一个应该显示的页面是什么。

6.6 运算符

除了1.3节里讨论过的. 和[]之外，表达式语言还提供了如下运算符，它们只能用在右值表达式里。

- ❑ 算术运算符 二元运算符（如+、-、*、 / ）和一元运算符（如div、 % 、mod、 - ）。
- ❑ 逻辑运算符 and、&&、or、||、not、!。
- ❑ 关系运算符 ==、eq、 != 、ne、 <、lt、 >、gt、 <=、ge、 >=、le。可以比较其他类型的值，也可以比较布尔型、字符串型、整形或者浮点常量等类型的值。

- 空值运算符 `empty`运算符属于前置运算，可用于判断某值是否为`null`或者空值。
 - 条件运算符 `A ? B : C`表达式的结果为`B`或者`C`，这取决于`A`的求值结果。
- 运算的优先级按照从高到低、从左到右的顺序列举如下：
- `[]`;
 - `()`，用于改变运算符的优先级；
 - `-`、`not`、`!`、`empty`，这几个都是一元运算符；
 - `*`、`/`、`div`、`%`、`mod`；
 - 二元运算符 `+`、`-`；
 - `<`、`>`、`<=`、`>=`、`lt`、`gt`、`le`、`ge`；
 - `==`、`!=`、`eq`、`ne`；
 - `&&`、`and`；
 - `||`、`or`；
 - `?`、`:`。

6.7 保留字

如下列举的是表达式语言的保留字，我们的代码中不能将它们用作变量名、类名等标识符。

<code>and</code>	<code>or</code>	<code>not</code>	<code>eq</code>
<code>ne</code>	<code>lt</code>	<code>gt</code>	<code>le</code>
<code>ge</code>	<code>true</code>	<code>false</code>	<code>null</code>
<code>instanceof</code>	<code>empty</code>	<code>div</code>	<code>mod</code>

6.8 EL 表达式的例子

表6-3举了几个EL表达式的例子以及表达式的求值结果。

表6-3 几个表达式的例子

EL 表达式	求值结果
<code>\${1 > (4/2)}</code>	<code>false</code>
<code>\${4.0 >= 3}</code>	<code>true</code>
<code>\${100.0 == 100}</code>	<code>true</code>
<code>\${(10*10) ne 100}</code>	<code>false</code>
<code>\${'a' < 'b'}</code>	<code>true</code>
<code>\${'hip' gt 'hit'}</code>	<code>false</code>
<code>\${4 > 3}</code>	<code>true</code>
<code>\${1.2E4 + 1.4}</code>	<code>12001.4</code>
<code>\${3 div 4}</code>	<code>0.75</code>
<code>\${10 mod 4}</code>	<code>2</code>

(续)

EL 表达式	求值结果
<code>\${!empty param.Add}</code>	如果请求里名为Add的参数为null或者为空字符串, 则求值结果为false
<code>\${pageContext.request.contextPath}</code>	请求的上下文路径
<code>\${sessionScope.cart.numberOfItems}</code>	作用域为session的那个名为cart的bean里的属性值 (属性名为numberOfItems)
<code>\${param['mycom.productId']}</code>	请求里名为mycom.productId的参数的值
<code>\${header["host"]}</code>	请求头部里包含的host信息值
<code>\${departments[deptName]}</code>	departments是个映射, 当键值等于deptName时对应的值
<code>\${requestScope['javax.servlet.forward.servlet_path']}</code>	作用域为request的名为javax.servlet.forward.servlet_path的属性的值
<code>#{customer.lName}</code>	首次请求时, 则是获得customer这个bean的lName属性值; 后续请求时, 则是设置lName的值
<code>#{customer.calcTotal}</code>	方法表达式, 求值结果为customer这个bean调用calcTotal方法的返回值

网页是Web应用程序的表现层。创建JSF应用的网页的过程包括为页面添加组件，并将其关联至后台bean、验证器、转换器以及其他与页面相关的服务器端对象。

本章将讲述如何使用不同类型的组件和核心标签创建网页。下一章介绍如何将转换器、验证器和监听器添加至组件标签，以便为组件提供更多的功能。

本章内容

- 设置页面
- 使用HTML标签为页面添加组件
- 核心标签

7.1 设置页面

典型的JSF页面包括如下元素：

- 一组JSF标签库的命名空间声明；
- 新的HTML头部（h:head）和主体（h:body）标签，这些项是可选的；
- 一个表单标签（h:form），代表用户输入组件。

为了将JSF组件添加至网页，需要为两个标准标签库提供访问页面的权限，这两个库即JSF标准HTML标签库和JSF核心标签库。JSF标准HTML标签库定义了通用的HTML用户界面组件标签。这个库将关联至HTML生成工具，参见<http://docs.oracle.com/javaee/6/jaserverfaces/2.0/docs/renderkitdocs/>。JSF核心标签库定义了执行核心操作的标签。

完整的JSF Facelets标签及它们的属性，请参考在线文档，地址为<http://docs.oracle.com/javaee/6/jaserverfaces/2.0/docs/pdldocs/facelets/>。

为了使用JSF中的标签，需要在每个页面的顶部加入声明，以指定该页面使用的标签库。

对于Facelets应用程序，XML命名空间声明唯一指定了标签库的URI以及标签前缀。

例如，创建Facelets的XHTML网页，需要包括下述声明：

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
```

XML命名空间的URI指定了标签库的位置，其前缀用于区分标签所归属的不同标签库。除了h和f之外，还可以使用其他标签前缀。然而，要在页面中使用标签时，必须使用你所选择的标签库所限定的前缀。例如，在下面的页面中，form标签必须以h为前缀引用。这是因为在前面的标签库声明中，h为本页面使用的标签前缀，以此区别HTML标签库中定义的标签。

```
<h:form ...>
```

7.2节以及7.3节将介绍如何使用JSF标准HTML标签库中的组件标签以及JSF核心标签库中提供的核心标签。

7.2 使用 HTML 标签为页面添加组件

JSF标准HTML标签库中定义的标签代表了HTML表单组件以及其他基本HTML元素。这些组件用于展示数据或接收来自用户的输入。通常在用户单击按钮后，数据将和表单一起提交至服务器。本节将详细介绍表7-1中每一个组件标签的用法。

表7-1 组件标签

标 签	功 能	对应的HTML元素	外 观
column	代表数据组件中的一列	HTML中表格的一列	表格中的一列
commandButton	将表单提交给应用	HTML中的<input type = type>元素, type的值可以为submit、reset或image	按钮
commandLink	跳转到另外一个页面或本页中其他地方的链接	HTML中的<a href>元素	超链接
dataTable	代表数据封装器	HTML中的<table>元素	内容可以动态更新的表格
form	代表输入表单 (表单中的内嵌标签接收数据并与表单一起提交)	HTML中的<form>元素	不可见
graphicImage	显示图片	HTML中的元素	图片
inputHidden	允许页面设计人员在页面中使用隐藏变量	HTML中的<input type= hidden>元素	不可见
inputSecret	允许用户在文本框中输入的数据不显示为实际输入的字符	HTML中的<input type= password>元素	文本框, 输入的字符串显示为其他字符
inputText	允许用户输入一个字符串	HTML中的<input type= text>元素	文本框
inputTextarea	允许用户输入多行字符串	HTML中的<textarea>元素	多行文本框
message	以预设语言显示消息	如果使用样式表, 则对应HTML中的标签	文本字符串
messages	以预设语言显示一组消息	如果使用样式表, 则对应HTML中的一组标签	文本字符串
outputFormat	以预设语言显示消息	普通文本	普通文本
outputLabel	为指定的文本框添加一个标签	HTML中的<label>元素	普通文本

(续)

标 签	功 能	对应的HTML元素	外 观
outputLink	在不产生动作事件的前提下, 链接至另一个页面或本页面中的其他位置	HTML中的<a>元素	超链接
outputText	显示一行文本	普通文本	普通文本
panelGrid	显示一个表格	HTML中的<table>元素及<tr>、<td>元素	表格
panelGroup	将一组组件组合在一起	HTML中的<div>或元素	表格中的一行
selectBooleanCheckbox	允许用户在两个选项中选择其一	HTML中的<input type=checkbox>元素	复选框
selectItem	代表选项列表中的一个条目, 用户必须选择其中一项	HTML中的<option>元素	不可见
selectItems	代表一个选项列表, 用户必须选择其中的一项	HTML中的一组<option>元素	不可见
selectManyCheckbox	代表用户可以进行多项选择的一组复选框	HTML中一组类型为checkbox的<input>元素	一组复选框
selectManyListbox	允许用户一次从选项中选择多个, 多个选项可同时显示	HTML中的<select>元素	列表框
selectManyMenu	允许用户从选项中选择多个条目	HTML中的<select>元素	可滚动的组合框
selectOneListbox	允许用户从一组选项中选择 一个, 多个选项可同时显示	HTML中的<select>元素	列表框
selectOneMenu	允许用户从一组选项中选择 一个条目	HTML中的<select>元素	可滚动的组合框
selectOneRadio	允许用户从一组选项中选择 一个条目	HTML中的<input type=radio>元素	一组单选按钮

下节将介绍适用于大多数组件标签的重要标签属性。对于随后介绍的每一个组件, 9.2节将讲述如何把组件(或者组件的值)与bean属性绑定起来。

7.2.1 组件标签的通用属性

大多数组件标签均支持表7-2中的属性。

表7-2 组件标签的通用属性

属 性	描 述
binding	标记一个bean属性并将组件实例与之绑定
id	组件的唯一标记
immediate	如果设置为true, 则表明与组件关联的任何事件、验证和转换都将在应用请求参数值后立即执行

(续)

属 性	描 述
rendered	指定生成组件的一个特定条件。如果条件不满足, 则不生成组件
style	为标签指定CSS (Cascading Style Sheet, 层叠样式表)
styleClass	指定包含样式定义的CSS类
value	标记一个外部的数据源并将组件的值与其绑定

所有的标签属性(除id外)均支持EL表达式。关于EL的详细介绍, 参见第6章。

1. id属性

对于组件来说, id不是一个必需的属性, 但当另一个组件或服务端的类需要引用这个组件时, 需要使用id。如果没有设置id属性, JSF实现会自动产生一个组件ID。不同于其他JSF标签属性, id属性只支持接下来在“immediate属性”中介绍的赋值语法, 即使用#{ }。更多关于表达式语法的内容, 参见6.3.1节。

2. immediate属性

可以将输入型组件和命令型组件(实现ActionSource接口, 如按钮和超链接)的immediate属性设置为true, 以强制应用在用户提交输入数据后进行事件处理、验证和转换。

需要认真考虑一个输入型组件的immediate属性值与一个命令型组件的immediate属性值的组合设置方式, 因为它决定了命令型组件被触发(用户提交数据)后应用的工作方式。

假定有一个页面, 它包括一个按钮和一个在购物车中填写购书数量的输入框。如果按钮与输入框的immediate属性都设置为true, 那么与单击按钮所产生的事件相关联的任何处理动作, 都可以访问输入框中新输入的数据。也就是说, 当应用请求参数值后, 与按钮关联的事件以及与输入框关联的验证和转换事件都将被触发。

如果按钮的immediate属性设置为true, 而输入框的immediate属性设置为false, 那么与按钮关联的事件将被处理, 而不会将输入框中的数值更新至模型层。其原因是, 与输入框相关的任何事件处理以及转换或验证都是在应用请求参数值后发生的, 而当输入框的immediate属性为false时, 请求参数值不会应用到服务器端。

3. rendered属性

组件标签使用布尔型EL表达式以及rendered属性, 以决定组件是否出现在页面中。例如, 在下面的页面代码中, 如果购物车中没有物品, 那么commandLink组件就不会出现。

```
<h:commandLink id="check"
...
rendered="#{cart.numberofItems > 0}">
<h:outputText
value="#{bundle.CartCheck}"/>
</h:commandLink>
```

几乎不同于其他任何JSF标签属性, rendered属性仅限于使用右值表达式。正如6.3节所述, 右值表达式仅能读取数据, 而不能将数据写回数据源。因此, 用于rendered属性的是那些可使用算术运算符和文本字符串的右值表达式, 而不是左值表达式。例如, 前面例子中的>运算符。

4. style和styleClass属性

style和styleClass属性允许开发人员为标签的输出指定CSS样式。7.2.12节给出了一个使用style属性直接指定样式的例子。组件标签可以引用CSS类。

下面这个例子在dataTable标签中引用了一个名为list-background的样式类。

```
<h:dataTable id="books"
    ...
    styleClass="list-background"
    value="#{bookDBAO.books}"
    var="book">
```

定义这个类的样式表来自文件stylesheet.css，它将被打包至应用程序中。更多关于定义样式的信息，参见CSS规范（Cascading Style Sheets，地址为<http://www.w3.org/Style/CSS>）。

5. value和binding属性

输出型组件的标签使用value和binding属性将组件的值或实例绑定至外部数据源。

7.2.2 添加HTML的head和body标签

HTML中的head（h:head）和body（h:body）标签将HTML的页面结构添加至JSF页面中。

□ h:head代表HTML页面中的head元素。

□ h:body代表HTML页面中的body元素。

下面的代码片段表示一个XHTML页面，使用了常见的head和body标记：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Add a title</title>
</head>
<body>
  Add Content
</body>
```

下面的代码片段表示一个XHTML页面，使用了h:head和h:body标签。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
  Add a title
</h:head>
<h:body>
  Add Content
</h:body>
```

上述两段代码均生成相同的HTML元素。head和body标签在解决资源定位时更有用。更多关于资源定位的信息，参见7.2.15节。

7.2.3 添加表单组件

`h:form`代表一个输入表单，它的子组件可以包含展现给用户的数据，以及与表单一起提交给服务器的数据。

图7-1展示了一个典型的登录表单。用户可以输入用户名（`user name`）和密码（`password`），并通过单击Login（登录）按钮提交表单。

图7-1 一个典型的表单

`h:form`标签代表页面中的表单。它将用于展现数据或接收用户输入的数据所需的全部组件括在其中，如下所示：

```
<h:form>
...其他JSF标签和内容...
</h:form>
```

`h:form`标签还包含了HTML标记，用于实现组件在页面中的布局。需要注意的是，`h:form`本身无法实现任何布局功能。它的用途是收集数据，并为表单中的其他组件声明属性。

一个页面可以含有多个`h:form`标签，但是只有用户提交的表单内的值会被最终提交至服务器端。

7.2.4 文本组件

文本组件允许用户在Web应用程序中查看和修改文本。文本组件的基本类型如下。

- 标识（`label`）——显示只读的文本。
- 文本框（`Text field`）——允许用户输入文本，通常作为表单的一部分提交。
- 文本域（`Text area`）——是文本框的一种，允许用户输入多行文本。
- 密码输入框——文本框的一种，将用户输入的文本显示为其他符号，如星号。

图7-2展示了文本组件的例子。

图7-2 文本组件示例

文本组件可以归为输入型组件或输出型组件。JSF输出型组件显示成只读文本，例如标识。JSF输入型组件显示为可编辑的文本，例如文本字段。

输入和输出型组件都可以以多种方式显示特定文本。

表7-3罗列了代表输入型组件的标签。

表7-3 输入型组件标签

标 签	功 能
h:inputHidden	允许页面设计人员在页面中添加一个隐藏变量
h:inputSecret	标准密码字段，接收一行无空格文本，并在其输入时显示为一组星号
h:inputText	标准文本字段，接收一行文本字符串
h:inputTextarea	标准文本域，接收多行文本

除了7.2.1节所列的属性，输入型组件标签还支持表7-4中罗列的标签属性。需要注意的是，这个表中没有包含输入型组件标签所支持的全部属性，只是列举了一些最常用的。欲获得完整的属性列表，请访问<http://download.oracle.com/javase/6/javaxserverfaces/2.0/docs/pdldocs/facelets/>。

表7-4 输入型组件标签属性

属 性	描 述
converter	指定一个转换器以转换组件中的本地数据。关于如何使用这个属性的更多信息，参见8.1节。
converterMessage	指定当注册在该组件上的转换器执行失败时显示的出错消息
dir	指定该组件显示文本的方向，可接受的值是LTR（从左至右）和RTL（从右至左）
label	指定一个用于在出错信息中代表该组件的名称
lang	指定已呈现标记所使用的语言编码，如en_us
required	通过一个boolean值指定用户是否必须向组件中输入数据
requiredMessage	指定当用户没有输入数据时显示的出错信息
validator	指定一个指向后台bean方法的方法表达式，以执行对组件中数据的验证，参见8.4.3节中使用f:validator标签的例子
f:validatorMessage	指定当注册到组件上的验证器验证组件的本地数据失败时显示的出错消息
valueChangeListener	指定一个指向后台bean方法的方法表达式，以处理用户向组件中输入数据的事件。参见8.4.4节中使用valueChangeListener的例子

表7-5列举了代表输出型组件的标签。

表7-5 输出型组件标签

标 签	功 能
h:outputFormat	显示一条本地化信息
h:outputLabel	标准只读标识，将指定的输入框组件显示为标识
h:outputLink	显示为<a href>标签，以链接至另一个页面，但不产生动作事件
h:outputText	显示单行文本字符串

除了7.2.1节所列的属性，输出型组件标签还支持converter标签的属性。

本节其余部分将介绍如何使用表7-3和表7-5中所列的一些标签（其他标签的使用方法与其类似）。

1. 用h:inputText生成文本框

h:inputText标签用以显示一个文本框。与之相似，h:outputText标签显示一个只读的单行文本。本节将展示如何使用h:inputText标签。h:outputText标签的使用方式与之类似。

下面是一个使用h:inputText标签的例子：

```
<h:inputText id="name" label="Customer Name" size="50"
    value="#{cashier.name}"
    required="true"
    requiredMessage="#{customMessages.CustomerName}">
    <f:valueChangeListener
        type="com.sun.bookstore6.listeners.NameChanged" />
</h:inputText>
```

label属性为组件指定了一个可读性强的名字，它将用于组件出错信息中的替代参数中，以显示组件名称。

value属性指向一个名为CashierBean的后台bean中的name属性。这个属性记录name参数中的数据。当用户提交表单之后，CashierBean中name属性的值将被设成用户在与该标签对应的文本框中输入的数据。

如果用户没有在name对应的文本框中输入数据，required属性将导致重新加载页面，并显示一个出错信息。JSF实现检查组件值是否为null或空字符串。

如果组件必须有一个不为null的值或String类型的值（至少一个字符），那就需要在标签中使用required属性，并将其值设置为true。如果标签有这个属性且被设置为true，那么当组件的值是null或长度为0的字符串时，其他注册到该组件上的验证器不会执行。如果标签的required属性没有设置为true，其他注册到该组件的验证器会执行，但是这些验证器必须处理可能出现null值或长度为0的字符串的情况。更多信息，参见9.4节的“验证null以及空字符串”。

2. 用h:inputSecret显示密码输入框

h:inputSecret标签用于显示HTML标记中的<input type="password">。当用户在这种输入框框中输入一个字符串时，用户实际输入的字符串会被一行星号代替。下面是一个例子：

```
<h:inputSecret reDisplay="false"
    value="#{LoginBean.password}" />
```

3. 用h:outputLabel生成label

h:outputLabel用于为特定的输入型组件赋予一个标签（label），使该组件有一个可读性更强的名字。下面这个页面使用h:outputLabel标签显示复选框的标签：

```
<h:selectBooleanCheckbox
    id="fanClub"
    binding="#{cashier.specialOffer}" />
<h:outputLabel for="fanClub"
    binding="#{cashier.specialOfferText}">
    <h:outputText id="fanClubLabel"
        value="#{bundle.DukeFanClub}" />
```

```
</h:outputLabel>
...
```

`h:outputLabel` 的 `for` 属性将输入型组件的 `id` 映射至标签上。`h:outputText` 内嵌至 `h:outputLabel` 中，为输入型组件增加标签。`h:outputText` 中的 `value` 属性值用于显示输入型组件旁的文字描述。

可以使用 `h:outputLabel` 的 `value` 属性，达到与 `h:outputText` 标签相同的效果，即显示文字描述。下面的这段代码展示了如何用 `h:outputLabel` 设置标签的文字说明：

```
<h:selectBooleanCheckbox
    id="fanClub"
    binding="#{cashier.specialOffer}" />
<h:outputLabel for="fanClub"
    binding="#{cashier.specialOfferText}"
    value="#{bundle.DukeFanClub}" />
</h:outputLabel>
...
```

4. 用 `h:outputLink` 生成超链接

`h:outputLink` 标签用于显示超链接，即当用户单击它时，浏览器加载另一个页面，但不产生动作事件。如果希望通过单击链接总是打开一个由 `h:outputLink` 标签的 `value` 属性指定的页面，但不想触发后续处理，请使用这个标签，而不是 `h:commandLink` 标签。请看下面的例子：

```
<h:outputLink value="javadocs">
    Documentation for this demo
</h:outputLink>
```

`outputLink` 标签中的文字显示为到新页面的链接。

5. 使用 `h:outputFormat` 显示格式化的消息

`h:outputFormat` 标签以 `MessageFormat` 的模式动态地显示消息文字，其形式见 Java API 文档中对 `java.text.MessageFormat` 的描述。下面是使用 `outputFormat` 标签的代码示例：

```
<h:outputFormat value="Hello, {0}!">
    <f:param value="#{hello.name}" />
</h:outputFormat>
```

`value` 属性指定了 `MessageFormat` 模式。`param` 标签指定了消息的替代参数。参数的值将替换文字中的 `{0}`。如果 `"#{hello.name}"` 的值是 `Bill`，那么最终的消息将显示为下面的内容：

```
Hello, Bill!
```

一个 `h:outputFormat` 标签可以为消息定义不止一个 `param` 标签。如果消息包含不止一个替换参数，请确保 `param` 标签按照正确的顺序排列，以保证数据能够按照正确的顺序替换。下面的例子是对前一例子的修改，新增加了一个参数：

```
<h:outputFormat value="Hello, {0}! You are visitor number {1} to the page.">
    <f:param value="#{hello.name}" />
    <f:param value="#{bean.numVisitor}" />
</h:outputFormat>
```

`{1}` 的值将被第二个参数的值替换。参数采用了 EL 表达式的方式 `bean.numVisitor`，其中 `numVisitor` 是后台 `bean` 的属性，用以记录访问该页面的用户。这是一个可以接收值表达式的标签

属性使用EL表达式的例子。这条消息将显示如下：

```
Hello, Bill! You are visitor number 10 to the page.
```

7.2.5 使用命令型组件标签执行动作和导航

在JSF应用程序中，按钮和超链接组件标签用于执行动作，如提交表单以及打开新的页面。这些标签之所以称为命令型组件标签，是因为它们在被触发后执行一个动作。

`h:commandButton`标签用于生成一个按钮。`h:commandLink`用于生成一个超链接。

除了7.2.1节介绍的标签属性外，`h:commandButton`和`h:commandLink`还可以使用下列属性。

- `action`，输出为String类型的结果，或指向一个返回String类型结果的bean方法的方法表达式。无论哪种方式，输出的String类型结果都用于决定当命令型组件标签被触发后应转向的页面。

- `actionListener`，指向一个bean方法的方法表达式，它处理由命令型组件标签触发的动作。

参见8.4.1节以获得更多关于使用`action`属性的信息，参见8.4.2节获得关于使用`actionListener`属性的信息。

1. 使用`h:commandButton`标签生成按钮

如果使用`h:commandButton`组件标签，当用户单击按钮时，当前页面中的数据将被处理，下一页随之打开。下面是`h:commandButton`的一个例子：

```
<h:commandButton value="Submit"
  action="#{cashier.submit}"/>
```

由于`action`属性指向了`CashierBean`的`submit`方法，因此单击按钮将触发此方法。`submit`方法完成一系列处理动作并返回处理结果。

本例中`commandButton`标签的`value`属性指定了按钮的标识。关于如何使用`action`属性的更多信息，请参考8.4.1节。

2. 使用`h:commandLink`生成超链接

`h:commandLink`标签代表一个HTML超链接，并显示成HTML中的`<a>`元素。这个标签类似于表单中的提交按钮，用于将一个动作事件提交给应用程序。

`h:commandLink`标签中必须内嵌一个`h:outputText`标签，以代表用户可单击的文本，从而产生一个事件。下面是一个例子：

```
<h:commandLink id="NAmerica" action="bookstore"
  actionListener="#{localeBean.chooseLocaleFromLink}">
  <h:outputText value="#{bundle.English}" />
</h:commandLink>
```

这个标签将产生如下HTML代码：

```
<a id="_id3:NAmerica" href="#"
  onclick="document.forms['_id3']['_id3:NAmerica'].
  value='_id3:NAmerica';
  document.forms['_id3'].submit();
  return false;">English</a>
```

注意 `h:commandLink`将产生JavaScript脚本。如果你使用这个标签，请确保在浏览器选项中打开对JavaScript的支持。

7.2.6 用

在JSF应用程序中，可以使用

```
<h:graphicImage id="mapImage" url="/template/world.jpg"/>
```

`url`属性指定了图像的路径。本例中的URL以/开头，将Web应用程序的相对路径添加至图像路径的开头。

另一种做法是使用5.5节中介绍的方法指定图像的路径。下面是一个例子：

```
<h:graphicImage value="#{resource['images:wave.med.gif']}" />
```

7.2.7 用

在JSF应用程序中，可以将面板(`panel`)作为一系列其他组件的布局容器，面板将显示成HTML中的表格。表7-6列举了用于创建面板的标签。

表7-6 面板组件标签

标 签	属 性	功 能
<code>h:panelGrid</code>	<code>Columns</code> 、 <code>columnClasses</code> 、 <code>footerClass</code> 、 <code>headerClass</code> 、 <code>panelClass</code> 、 <code>rowClasses</code>	显示一个表格
<code>h:panelGroup</code>	<code>layout</code>	将一组组件组合至一个父组件中

`h:panelGrid`标签用于生成整个表格，`h:panelGroup`标签用于生成表中的一行，其他标签用于生成行中的单元格。

`columns`属性定义如何将数据布局至表格中。因此当表格数据超过一列时，必须使用这个属性。`h:panelGrid`标签也有一系列可选属性用来指定CSS的类，如`columnClasses`、`footerClass`、`headerClass`、`panelClass`和`rowClasses`。

如果设置了`headerClass`属性，`panelGrid`必须有一个`header`作为其第一个子元素。类似地，如果设置了`footerClass`，`panelGrid`必须有一个`footer`作为其最后一个子元素。

下面是一个例子：

```
<h:panelGrid columns="3" headerClass="list-header"
  rowClasses="list-row-even, list-row-odd"
  styleClass="list-background"
  title="#{bundle.Checkout}">
  <f:facet name="header">
    <h:outputText value="#{bundle.Checkout}"/>
  </f:facet>
  <h:outputText value="#{bundle.Name}" />
```

```

<h:inputText id="name" size="50"
    value="#{cashier.name}"
    required="true">
    <f:valueChangeListener
        type="listeners.NameChanged" />
</h:inputText>
<h:message styleClass="validationMessage" for="name"/>
<h:outputText value="#{bundle.CCNumber}"/>
<h:inputText id="ccno" size="19"
    converter="CreditCardConverter" required="true">
    <bookstore:formatValidator
        formatPatterns="9999999999999999|
            9999 9999 9999 9999|9999-9999-9999-9999"/>
</h:inputText>
<h:message styleClass="validationMessage" for="ccno"/>
...
</h:panelGrid>

```

前述的h:panelGrid标签用于生成一个表格，它包含了接收用户输入的个人信息的组件。

h:panelGrid标签使用样式表类来格式化表格。下面的代码展示了list-header的定义：

```

.list-header {
    background-color: #ffffff;
    color: #000000;
    text-align: center;
}

```

由于h:panelGrid标签定义了一个headerClass，panelGrid必须包括一个header。示例中的panelGrid标签使用facet标签定义header。facet仅能有一个子元素，因此如果将多个组件组合于一个facet中，必须使用h:panelGroup标签。示例中的h:panelGrid标签仅有一个数据单元格，因此无需h:panelGroup标签。

h:panelGroup标签有一个称为layout的属性。除此之外，它还包括7.2.1节中所列举的属性。如果layout属性的值为block，则生成HTML中的div元素以显示表格中的行，否则生成HTML中的span元素用于行显示。如果为h:panelGroup标签指定样式，必须将layout属性的值设置为block，以在h:panelGroup标签中应用组件的样式。这样做的原因是诸如高度和宽度等样式属性无法以内嵌元素的方式进行设定，这就是要把有些内容内嵌进span元素的原因。

h:panelGroup标签可以用于封装一个内嵌的组件树，从而使组件树以独一组件的形式位于父组件中。

根据h:panelGrid标签columns属性的值，数据以内嵌标签的方式组合至表格的行中。本例中的columns属性值为3，这就意味着表格有3列。组件所在的列由其在组件列表中的顺序模3计算得到。因此，如果一个组件位于组件列表中的第五位，那么这个组件将位于第二列，即 $5 \bmod 3 = 2$ 。

7.2.8 显示选项组件

另一个常用的组件允许用户从列表中进行单项选择，无论只有一个还是多个选项。本类组件中常见的标签如下：

□ h:selectBooleanCheckbox标签，显示为一个复选框，表示一个布尔状态；

- `h:selectOneRadio` 标签，显示为一组单选按钮；
- `h:selectOneMenu` 标签，显示为一个有滚动条的下拉菜单；
- `h:selectOneListbox` 标签，显示为一个无滚动条的列表框。

图7-3显示了对应组件的示例。

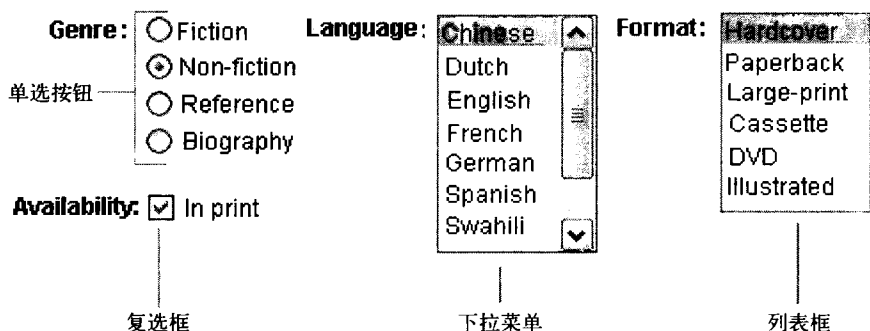


图7-3 单选组件示例

1. 使用 `h:selectBooleanCheckbox` 标签显示复选框

`h:selectBooleanCheckbox` 是 JSF 技术提供的唯一一个用于设置布尔状态的标签。

下面是一个使用 `h:selectBooleanCheckbox` 标签的例子：

```
<h:selectBooleanCheckbox
    id="fanClub"
    rendered="false"
    binding="#{cashier.specialOffer}" />
<h:outputLabel
    for="fanClub"
    rendered="false"
    binding="#{cashier.specialOfferText}">
    <h:outputText
        id="fanClubLabel"
        value="#{bundle.DukeFanClub}" />
</h:outputLabel>
```

本例中的复选框允许用户选择是否愿意加入 Duke Fan Club。复选框的标识由 `outputLabel` 标签生成。说明文字由内嵌的 `outputText` 标签生成。

2. 使用 `h:selectOneMenu` 标签显示菜单

列表框、菜单和一系列单选按钮均允许用户从一组选项选择一个。本节将介绍 `h:selectOneMenu` 标签。`h:selectOneRadio` 和 `h:selectOneListbox` 的用法类似。`h:selectOneMenu` 和 `h:selectOneListbox` 这两个标签相似，其差别在于 `h:selectOneListbox` 定义了一个 `size` 属性以决定一次显示的选项数目。

`h:selectOneMenu` 标签允许用户从一组选项选择一个。菜单组件通常也被认为是一个下拉列表或组合框。下面的代码展示了如何使用 `h:selectOneMenu` 标签，使用户可选择一种寄送方法：

```

<h:selectOneMenu id="shippingOption"
    required="true"
    value="#{cashier.shippingOption}">
    <f:selectItem
        itemValue="2"
        itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem
        itemValue="5"
        itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem
        itemValue="7"
        itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>

```

h:selectOneMenu标签的value属性保存了用户当前的选择。不一定非要为当前选择提供一个初始值。如果没有提供初始值，那么列表中的第一个选项为默认选项。

如同**h:selectOneRadio**标签，**h:selectOneMenu**标签必须包含一个**f:selectItems**或一组**f:selectItem**标签，以代表列表中的选项。7.2.10节将详细介绍这两个标签。

7.2.9 显示多项选择组件

在某些情况下，用户需要从选项列表中选择多个值。可以使用下面的组件标签实现多项选择：

- ❑ **h:selectManyCheckbox**标签，显示一组复选框；
- ❑ **h:selectManyMenu**标签，显示一个下拉菜单；
- ❑ **h:selectManyListbox**标签，显示一个列表框。

图7-4展示了上述这些组件的示例。

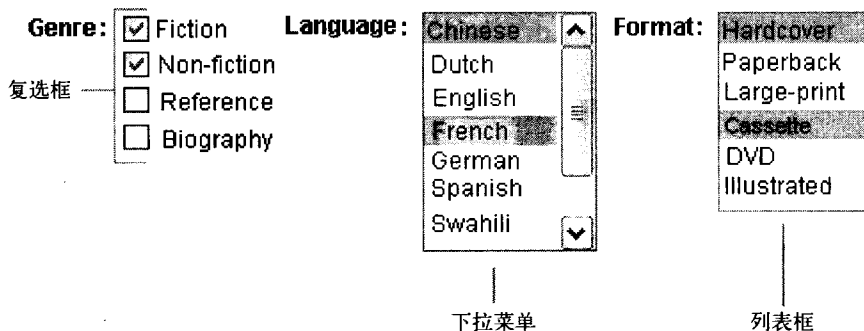


图7-4 多项选择组件示例

这些标签允许用户从一组选项中选择零个或多个值。本节将介绍**h:selectManyCheckbox**标签。**h:selectManyListbox**和**h:selectManyMenu**标签与**h:selectManyCheckbox**的用法相似。

不同于菜单，列表框仅在其框体中显示所有选项的一部分；而当用户没有在菜单中做选择时，菜单一次仅显示一个选项。**h:selectManyListbox**的size属性决定了列表框一次显示的选项数。列表框包括一个滚动条，以显示列表中剩余的选项。

`h:selectManyCheckbox` 标签绘制一组复选框，其中每个复选框代表一个可以选择的选项：

```
<h:selectManyCheckbox
    id="newsletters"
    layout="pageDirection"
    value="#{cashier.newsletters}">
    <f:selectItems
        value="#{newsletters}" />
</h:selectManyCheckbox>
```

`h:selectManyCheckbox` 标签的 `value` 属性指定了 `Cashier` 后台 bean 的 `newsletters` 属性。这个属性记录从一组列表框中选择的选项值。无需为当前选择的选项提供一个值。如果不提供该值，列表中的第一个选项为默认选项。

`layout` 属性指明一组复选框在页面中的布局方式。由于 `layout` 设置为 `pageDirection`，这组复选框将纵向排列。该属性的默认值为 `lineDirection`，即将复选框水平排列。

`h:selectManyCheckbox` 标签必须包括一个或一组代表复选框的标签。可以使用 `f:selectItems` 标签代表这组选项，可以使用 `f:selectItem` 代表每个单选按钮。下面将介绍关于这些标签的更多内容。

7.2.10 使用 `f:selectItem` 和 `f:selectItems` 标签

`f:selectItem` 和 `f:selectItems` 代表的组件可以嵌入到另一个组件中，以实现单选或多选。`f:selectItem` 标签包含一个选项的值、标识及描述信息。`f:selectItems` 标签包含整个选项列表中所有选项的值、标识及描述信息。

可以在选择组件中使用一组 `f:selectItem` 标签或一个单独的 `f:selectItems` 标签。

使用 `f:selectItems` 标签的优势在于：

- 选项可以用不同的数据结构表示，包括 `Array`、`Map` 和 `Collection`。`f:selectItems` 标签的值甚至可以代表泛化的 `POJO` 集合；
- 不同的列表可以合并至一个组件中，并组合在一起；
- 值可以在程序运行时动态产生。

使用 `f:selectItem` 标签的优势在于：

- 可以在页面中定义列表中的选项；
- 无需在 bean 中编写过多的代码，即可实现 `selectItem` 属性的设定。

本节的后续内容将介绍如何使用 `f:selectItems` 和 `f:selectItem` 标签。

1. 使用 `f:selectItems` 标签

下面这个例子来自 7.2.9 节，展示如何使用 `f:selectManyCheckbox` 标签：

```
<h:selectManyCheckbox
    id="newsletters"
    layout="pageDirection"
    value="#{cashier.newsletters}">
    <f:selectItems
        value="#{newsletters}" />
</h:selectManyCheckbox>
```

f:selectItems标签的value属性绑定至名为newsletters的后台bean。

也可以在后台bean中通过编程创建一个选项列表。参9.2节以了解如何为标签编写后台bean的属性。

2. 使用f:selectItem标签

f:selectItem标签代表选择列表中的一个选项。下面的例子再次引用7.2.8节中“使用h:selectOneMenu标签显示菜单”中用到的例子：

```
<h:selectOneMenu
    id="shippingOption" required="true"
    value="#{cashier.shippingOption}">
    <f:selectItem
        itemValue="2"
        itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem
        itemValue="5"
        itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem
        itemValue="7"
        itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>
```

itemValue属性代表了selectItem标签的默认值。itemLabel属性代表该选项在下拉菜单中显示的内容。

itemValue和itemLabel属性均支持值绑定（value-binding-enabled）模式，即它们均可以使用值绑定表达式引用外部对象的值。这两个属性也可以定义字面值，如同在介绍h:selectOneMenu标签时使用的示例一样。

7.2.11 使用数据绑定表组件

数据绑定表以表格的方式显示关系型数据。在JSF应用程序中，h:dataTable组件标签支持对一组数据对象的集合的绑定，并以HTML表格的方式显示数据。h:column标签代表表中数据的一列，通过不断遍历数据源中的每条记录，以行的方式显示完整数据。下面是一个例子：

```
<h:dataTable id="items"
    captionClass="list-caption"
    columnClasses="list-column-center, list-column-left,
        list-column-right, list-column-center"
    footerClass="list-footer"
    headerClass="list-header"
    rowClasses="list-row-even, list-row-odd"
    styleClass="list-background">
    <h:column headerClass="list-header-left">
        <f:facet name="header">
            <h:outputText value="Quantity" />
        </f:facet>
        <h:inputText id="quantity" size="4"
            value="#{item.quantity}" />
        ...
    </h:inputText>
    ...
</h:dataTable>
```

```

</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Title"/>
  </f:facet>
  <h:commandLink>
    <h:outputText value="#{item.title}"/>
  </h:commandLink>
</h:column>
...
<f:facet name="footer">
  <h:panelGroup>
    <h:outputText value="Total"/>
    <h:outputText value="#{cart.total}" />
    <f:convertNumber type="currency" />
  </h:panelGroup>
</f:facet>
</h:dataTable>

```

图7-5显示了通过h:dataTable显示的数据表格。

Quantity	Title	Price
<input type="text" value="1"/>	Web Servers for Fun and Profit	\$40.75 <input type="button" value="Remove Item"/>
<input type="text" value="3"/>	Web Components for Web Developers	\$27.75 <input type="button" value="Remove Item"/>
<input type="text" value="1"/>	From Oak to Java: The Revolution of a Language	\$10.75 <input type="button" value="Remove Item"/>
<input type="text" value="2"/>	My Early Years: Growing up on '7	\$30.75 <input type="button" value="Remove Item"/>
<input type="text" value="1"/>	Java Intermediate Bytecodes	\$30.95 <input type="button" value="Remove Item"/>
<input type="text" value="3"/>	Duke: A Biography of the Java Evangelist	\$45.00 <input type="button" value="Remove Item"/>
Subtotal: \$52.20		
Update Quantities		

图7-5 网页中的表格

示例中的h:dataTable标签显示了购物车中的书籍，以及每本书在购物车中的数量、价格和一个用于将其从购物车中移除的按钮。

h:column标签代表数据组件中数据的列。数据组件对表中要显示的数据信息进行逐行轮询，并处理与h:column标签关联的每一个列组件。

前述示例代码中的h:dataTable遍历购物车中的图书列表（cart.items）并显示其标题、作者和价格。h:dataTable标签每一次遍历图书列表时，都将绘制并显示列中的一个单元格。

h:dataTable和h:column标签使用facet显示表中不重复和不变化的部分。这些部分包括表头、脚注和文字说明。

在前述代码中，h:column标签包括f:facet标签，用于代表列表头和脚注。h:column标签通过支持headerClass和footerClass属性控制表头和脚注的样式。这些属性接收空格分隔的CSS类列表，并将它们应用于表中相应列的表头和脚注单元格中。

facet只能有一个子元素，因此当需要将更多的组件置于f:facet之中时，需要使用

`h:panelGroup` 标签。在本例中，由于显示脚注的 `facet` 包含多个标签，因此需要使用 `panelGroup` 对其进行组合。在示例的最后，`h:dataTable` 标签包括了一个 `f:facet` 标签及它的 `name` 属性，用以设置说明文字。这些说明将显示在表的下方。

本例是数据组件的一种最常见用法，因为对于应用程序开发人员和页面设计人员来说，在进行程序开发时并不知道书的数量。这一数据组件可以动态地调整表的行数以显示全部的数据。

`h:dataTable` 标签的 `value` 属性代表表中的数据。源数据可以以下列任何一种方式提供：

- ☐ bean 列表；
- ☐ bean 数组；
- ☐ 单个 bean；
- ☐ `javax.faces.model.DataModel` 对象；
- ☐ `java.sql.ResultSet` 对象；
- ☐ `javax.servlet.jsp.jstl.sql.Result` 对象；
- ☐ `javax.sql.RowSet` 对象。

所有数据组件的数据源均有一个 `DataModel` 封装器。除非显式地构造一个 `DataModel` 封装器，否则 JSF 在实现的过程中会自动创建一个封装其他可接受类型的数据的封装器。参见 9.2 节以了解如何编写数据组件的属性信息。

`var` 属性指定了在 `h:dataTable` 标签中组件使用的名字，它是 `dataTable` 的 `value` 属性中引用数据的一个别名。

在示例程序的 `h:dataTable` 标签中，`value` 属性指向一个图书列表。`var` 属性指向列表中的一本书。由于 `h:dataTable` 遍历整个图书列表，因而 `var` 属性将指向列表中当前的一本书。

`h:dataTable` 标签具备显示数据子集的能力。这个特性在前面的例子中没有体现。可以使用 `first` 和 `rows` 属性显示数据子集。

`first` 属性指定显示数据的第一行。`rows` 指定从第一行开始应显示数据的行数。例如，仅想显示第二行到第十行数据时，可以将 `first` 设置为 2，`rows` 设置为 9。当在页面中仅显示数据的一个子集时，可能需要考虑增加一个链接或按钮，当用户单击它们时，可以显示剩余数据。在默认情况下，`first` 和 `rows` 属性设为 0，其结果是显示全部数据。

表 7-7 展示了 `h:dataTable` 其他可选的属性。

表 7-7 `h:dataTable` 的可选属性

属 性	样式定义的对象
<code>captionClass</code>	表格的说明文字
<code>columnClasses</code>	所有列
<code>footerClass</code>	脚注
<code>headerClass</code>	表头
<code>rowClasses</code>	行
<code>styleClass</code>	整个表

表7-7中的属性均可指定不止一种样式。如果columnClasses或rowClasses指定不止一种样式，应用于列或行中的样式将按照样式定义在属性中的顺序依次应用。例如，如果columnClasses指定了样式list-column-center和list-column-right，且表有两列，则第一列将使用样式list-column-center，而第二列使用样式list-column-right。

如果样式属性指定的样式数超过行或列的数量，则剩余的样式将从第一行或者第一列开始应用。与之相似，如果样式属性指定的样式数少于行或列的数量，则剩余的行或列将从定义的第一个样式开始重新应用。

7.2.12 使用h:message和h:messages显示出错信息

h:message和h:messages标签用于在数据转换或验证出错后显示出错信息。h:message标签显示与特定输入型组件相关联的出错信息，而h:messages标签显示整个页面的出错信息。

下面是一个使用h:message标签的guessnumber（猜数游戏）程序：

```
<h:inputText id="userNo" value="#{UserNumberBean.userNumber}">
  <f:validateLongRange minimum="0" maximum="10" />
</h:inputText>
<h:commandButton id="submit"
  action="success" value="Submit" /><p>
<h:message
  style="color: red;
  font-family: 'New Century Schoolbook', serif;
  font-style: oblique;
  text-decoration: overline" id="errors1" for="userNo"/>
```

for属性指向产生出错信息的组件ID。出错信息的显示位置与h:message标签在页面中的位置相同。在本例中，出错信息将显示在Submit按钮之后。

style属性允许定义消息的显示样式。在本节的示例中，文本的样式为红色、New Century Schoolbook字体、serif字体族、斜体风格，且文字上方有顶盖线。message和messages标签支持很多其他定义样式的属性。请参考<http://docs.oracle.com/javaee/6/javadoc/2.0/docs/pdldocs/facelets/>处的在线文档，以获取更多关于这些属性的信息。

h:messages标签支持layout属性。它的默认值是list，即用HTML中的ul和li元素以项目列表的方式显示出错信息。如果将属性的值设为table，出错信息将通过HTML中的table元素以表格方式显示。

上述的示例显示了一个标准的注册于输入型组件的验证器。当这一验证器验证输入型组件的值失败时，与该验证器关联的message标签将显示出错信息。通常来说，当将一个转换器或验证器注册于一个组件时，你也将一系列与转换器或验证器相关的出错信息关联于该组件上。当验证器与转换器验证或转换组件数据失败时，h:message和h:messages标签将显示关联于其上的特定出错信息。

标准转换器和验证器提供标准的出错信息。应用程序架构师可以覆写这些标准信息，并为应用程序定制的转换器和验证器提供特定的，方法是为相关应用注册定制的出错信息。

7.2.13 使用h:button和h:link标签创建可加入收藏夹的URL

创建可加入收藏夹的URL是指基于页面导航结果和组件参数生成超链接。

在HTTP请求中，大多数浏览器在默认情况下发送GET请求以获取URL，发送POST请求以请求数据处理。GET请求可带有查询参数并可以缓存，这种做法不建议用于向外部服务器发送数据的POST请求。其他一些能够生成超链接的JSF标签通常使用简单的GET请求（如h:outputLink）或POST请求（如h:commandLink或h:commandButton标签）。带有查询参数的GET请求提供更好（更细粒度）的URL地址信息，产生的URL中除了有简单的地址信息外，还包括在?后面的一系列name=value参数，并通过&或者&的方式实现参数间的分隔。

使用h:link或h:button标签可以创建一个可加入收藏夹的URL。这两个标签均可以通过组件的outcome属性产生一个超链接地址。例如：

```
<h:link outcome="response" value="Message">
  <f:param name="Result" value="#{sampleBean.result}"/>
</h:link>
```

h:link标签将产生一个指向同一服务器上response.xhtml文件的URL链接，链接中还包括一个通过f:param标签创建的查询参数。当被处理时，后台bean方法#{sampleBean.result}的执行结果被分配给Result参数。下面的HTML示例源自前述标签的处理结果，假定参数的值是success：

```
<a href="http://localhost:8080/guessnumber/response.xhtml?Result=success">Response</a>
```

这是一个简单的GET请求。要创建复杂的GET请求并利用h:link的完整功能，可以使用视图参数。

7.2.14 使用视图参数配置可加入收藏夹的URL

核心标签f:metadata和f:viewparam用作配置URL的参数源。如下面的例子所示，视图参数作为页面中f:metadata的一部分进行声明。

```
<h:body>
<f:metadata>
  <f:viewParam id="name" name="Name" value="#{sampleBean.username}"/>
  <f:viewParam id="ID" name="uid" value="#{sampleBean.useridentity}"/>
</f:metadata>
<h:link outcome="response" value="Message" includeViewParams="true">
</h:link>
</h:body>
```

视图参数用f:viewparam标签进行声明，并置于f:metadata标签中。如果在组件上设置includeViewParams属性，视图参数将被添加至超链接中。

返回的URL地址如下：

```
http://localhost:8080/guessnumber/response.xhtml?Name=Duke&uid=2001
```

由于URL可能是处理多个参数值的结果，创建URL的顺序需要预先定义。不同参数值的读取顺序如下：

- (1) 组件;
- (2) 导航参数;
- (3) 视图参数。

7.2.15 使用h:output标签实现资源再定位

资源再定位是指JSF应用程序能够指定资源产生的位置。可以使用下面的HTML标签实现资源再定位:

□ h:outputScript

□ h:outputStylesheet

上述标签有name和target属性, 可用于定义资源产生的位置。这些标签的完整属性列表, 请参考在线文档, 地址为<http://docs.oracle.com/javaee/6/javaxserverfaces/2.0/docs/pdldocs/facelets/>。

对于h:outputScript标签, name和target属性定义输出资源的位置。下面是一个例子:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head id="head">
    <title>Resource Relocation</title>
  </h:head>
  <h:body id="body">
    <h:form id="form">
      <h:outputScript name="hello.js"/>
      <h:outputStylesheet name="hello.css"/>
    </h:form>
  </h:body>
</html>
```

由于标签中没有定义target属性, 样式表hello.css的位置声明来自head部分, 而hello.js脚本的位置声明来自页面中的body部分并由h:head标签定义。

下面是由上述代码产生的HTML代码:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Resource Relocation</title>
    <link type="text/css" rel="stylesheet"
          href="/ctx/faces/javafx.faces.resource/hello.css"/>
  </head>
  <body>
    <form id="form" name="form" method="post" action="..." enctype="...">
      <script type="text/javascript"
            src="/ctx/faces/javafx.faces.resource/hello.js">
      </script>
    </form>
  </body>
</html>
```

通过为h:outputScript标签设置target属性可以实现原始页面的重建, 这种方式允许GET请求提供位置参数。下面是一个例子:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head id="head">
```

```
<title>Resource Relocation</title>
</h:head>
<h:body id="body">
  <h:form id="form">
    <h:outputScript name="hello.js" target="#{param.location}"/>
    <h:outputStylesheet name="hello.css"/>
  </h:form>
</h:body>
</html>
```

在这个例子中，如果请求不提供位置参数，默认位置依旧可用：样式表来自head部分，而脚本内嵌于其中。然而，如果请求的head中提供位置参数，那么样式表和脚本的声明都将来自head。

下面的HTML来自上述的代码：

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Resource Relocation</title>
    <link type="text/css" rel="stylesheet"
      href="/ctx/faces/javax.faces.resource/hello.css"/>
    <script type="text/javascript"
      src="/ctx/faces/javax.faces.resource/hello.js">
    </script>
  </head>
  <body>
    <form id="form" name="form" method="post" action="..." enctype="...">
    </form>
  </body>
</html>
```

与之相似，如果请求的body中提供位置参数，脚本的声明将来自body。

前述代码给出了资源定位的简单应用。该特性可以为组件和页面增加更多功能。页面设计人员无需知晓资源的位置。

通过@ResourceDependency注解，组件设计人员可以为组件定义资源，如样式表和脚本。这使得页面设计人员无需考虑资源定位问题。

7.3 核心标签

JSF核心标签库中包括的标签可以执行HTML标签所无法完成的核心动作。表7-8列举了常用的核心标签及其功能。

表7-8 核心标签

标签分类	标 签	功 能
事件处理	f:actionListener	为父组件增加事件监听器
	f:phaseListener	为页面增加PhaseListener
	f:setPropertyActionListener	注册一个特殊事件监听器，当表单提交时，它的唯一目标是将值推送至后台bean
	f:valueChangeListener	为父组件增加值变更监听器
属性配置	f:attribute	为父组件增加可配置的属性

(续)

标签分类	标 签	功 能
数据转换	f:converter	为父组件增加强制数据转换器
	f:convertDateTime	为父组件增加DateTimeConverter实例
	f:convertNumber	为父组件增加NumberConverter实例
facet	f:facet	表示一个被（父组件）嵌套的组件
	f:metadata	将facet注册至父组件
多语言支持（本地化）	f:loadBundle	指定一个ResourceBundle以实现多语言支持
参数替换	f:param	将参数替换为MessageFormat实例，并将name-value格式的查询字符串添加至URL中
选项列举	f:selectItem	代表选项列表中的一个选项
	f:selectItems	代表一组选项
验证器	f:validateDoubleRange	为组件增加DoubleRangeValidator
	f:validateLength	为组件增加LengthValidator
	f:validateLongRange	为组件增加LongRangeValidator
	f:validator	为组件增加定制的验证器
	f:validateRegEx	为组件增加RegExValidator
	f:validateBean	将本地值的验证委托给BeanValidator
	f:validateRequired	强制组件必须有值
Ajax	f:ajax	将Ajax动作与一个或一组组件（基于位置）关联
事件	f:event	允许为组件添加ComponentSystemEventListener

7

这些标签（与组件标签结合使用）将在本书的其他章节进行详细介绍。表7-9列出了介绍如何使用这些核心标签的章节。

表7-9 介绍核心标签的章节

标 签	章 节 号
事件处理标签	8.2节
数据转换标签	8.1节
facet	7.2.11节和7.2.7节
loadBundle	7.2.9节
param	7.2.4节的“使用h:outputFormat显示格式化的消息”部分
selectItem和selectItems	7.2.10节
验证器标签	8.3节

前面几章介绍了一些组件,并讲述了如何在页面里使用它们。本章讲述如何通过使用转换器、监听器以及验证器为组件增加更多的功能。

- 转换器用来转换从输入型组件上接收来的数据。
- 监听器负责监听页面里发生的事件并执行事先定义好的响应方法。
- 验证器用来验证从输入型组件上取得的数据。

本章内容

- 标准转换器
- 为组件注册监听器
- 标准验证器
- 引用后台bean的方法

8.1 标准转换器

JSF实现了一系列的转换器,可以用来转换组件的数据。标准的转换器实现位于`javax.faces.convert`这个包里,包括如下类型的转换器:

- `BigDecimalConverter`
- `BigIntegerConverter`
- `BooleanConverter`
- `ByteConverter`
- `CharacterConverter`
- `DateTimeConverter`
- `DoubleConverter`
- `EnumConverter`
- `FloatConverter`
- `IntegerConverter`
- `LongConverter`

❑ NumberConverter

❑ ShortConverter

每一种转换器都对应了一个错误提示信息。如果页面里的某个组件注册了对应的转换器，并且转换器未能成功转换组件的值，则该转换器对应的错误提示信息会显示在页面上。例如，如果 `BigIntegerConverter` 转换器在转换数据时失败，则会显示如下错误提示信息：

```
{0} must be a number consisting of one or more digits
```

本例中，`{0}` 是一个参数，实际运行时会替换成对应的注册了该转换器的那个输入型组件的名字。

`DateTimeConverter` 和 `NumberConverter` 这两个标准转换器都有自己的标签，借助于标签属性，我们可以设置组件数据的格式。有关 `DateTimeConverter` 的详细信息，参见 8.1.2 节。有关 `NumberConverter` 的详细信息，参见 8.1.3 节。接下来的内容会讲述如何转换组件的值，也会讲述如何为组件注册其他类型的标准转换器。

8.1.1 转换组件的值

在使用某个特定类型的转换器转换组件的值之前，必须把它注册到组件上。注册标准的转换器有如下4种方法可以选用。

- ❑ 把标准转换器的标签嵌套到组件的标签里。支持这种方法的标签有 `convertDateTime` 和 `convertNumber`，详情分别参见 8.1.2 节和 8.1.3 节。
- ❑ 把组件的值和一个后台 bean 的属性绑定起来，转换器的类型需要和 bean 的属性相匹配。
- ❑ 在组件标签的 `converter` 属性中引用转换器。
- ❑ 把 `converter` 标签嵌套到组件的标签里，使用 `converter` 标签的 `converterId` 属性或者 `binding` 属性来引用转换器。

看第二种方法的一个例子。如果想把组件的数据转换为 `Integer` 类型，只要把组件的值绑定到后台 bean 的属性就可以了。请看：

```
Integer age = 0;
public Integer getAge(){ return age;}
public void setAge(Integer age) {this.age = age;}
```

如果组件没和 bean 的属性绑定，可以采用第三种方法，直接在组件标签中使用 `converter` 属性就可以了：

```
<h:inputText
    converter="javax.faces.convert.IntegerConverter" />
```

本例展示的 `converter` 属性引用的是转换器对应的类的完全限定名。`converter` 属性还可以设置为组件的 ID。

本例中 `inputText` 标签的数据，会被转换为一个 `java.lang.Integer` 类型的值。`NumberConverter` 转换器支持 `Integer` 类型。如果在使用 `convertNumber` 标签的属性时不需要指定具体格式，且有标准的转换器可以满足要求，那么引用转换器的时候，使用组件标签的 `converter` 属性就可以了。

最后，可以把converter标签嵌套到组件的标签里面，使用标签的converterId属性或者binding属性就可以引用转换器了。

converterId属性必须引用转换器的ID。例如：

```
<h:inputText value="#{LoginBean.Age}" />
  <f:converter converterId="Integer" />
</h:inputText>
```

如果不使用converterId属性，converter标签则可以使用binding属性。binding属性必须解析为bean的属性，且可以接收以及返回一个合适的Converter实例。

8.1.2 DateTimeConverter

把convertDateTime标签嵌套到组件的标签里，就可以转换组件的数据为java.util.Date类型。convertDateTime标签有多个属性，使得开发人员可以指定数据的格式和类型，表8-1列出了这些属性。

表8-1 convertDateTime标签的属性

属 性	属性类型	描述信息
binding	DateTimeConverter	用来将验证器绑定到一个后台bean的属性
dateStyle	String	定义由java.text.DateFormat指定的一个日期或date字符串的日期部分的格式。仅用于type为date或both，而pattern未定义的情况。有效值为default、short、medium、long以及full。如果未指定，默认为default
for	String	与复合组件一起使用，引用的是复合组件里的某个对象。for标签在复合组件标签的内部
locale	String 或者 Locale	在格式化或解析日期和时间时，会使用某个地区的预定义的格式。如果没有指定，则使用FacesContext.getLocale返回的Locale
pattern	String	自定义日期或时间以何种格式解析及显示。如果指定了这个属性，dateStyle、timeStyle和type属性都会被忽略
timeStyle	String	定义由java.text.DateFormat指定的一个时间或date字符串的时间部分的格式。仅用于type为time而pattern未定义的情况。有效值为default、short、medium、long以及full，如果未指定值，使用default
timeZone	String或者TimeZone	用于解析date字符串中的时间信息的时区
type	String	指定字符串值中是否包含日期、时间或者两者都有。有效值为date、time以及both，如果未指定值，使用date

下面是一个使用了convertDateTime标签的简单例子：

```
<h:outputText id="shipDate" value="#{cashier.shipDate}">
  <f:convertDateTime dateStyle="full" />
</h:outputText>
```

当把DateTimeConverter绑定到一个组件的时候，要保证后台bean的属性（转换器要绑定的那个属性）的数据类型确实是java.util.Date。对于刚才的例子来说，cashier.shipDate的数据类型必须是java.util.Date。

例子的标签会输出如下内容：

Saturday, September 25, 2010

如果以如下的标签指定日期格式，则显示的日期和时间 and 刚才的一样：

```
<h:outputText value="#{cashier.shipDate}">
  <f:convertDateTime
    pattern="EEEEEEEE, MMM dd, yyyy" />
</h:outputText>
```

如果想以西班牙语显示日期信息，可以使用`locale`属性：

```
<h:inputText value="#{cashier.shipDate}">
  <f:convertDateTime dateStyle="full"
    locale="Locale.SPAIN"
    timeStyle="long" type="both" />
</h:inputText>
```

本例中的标签将显示如下内容：

sabado 25 de septiembre de 2010

访问网址<http://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html>，查看Java教程中“Customizing Formats”一节的内容，可以了解如何使用`convertDateTime`标签的`pattern`属性实现格式化输出。

8.1.3 NumberConverter

把`convertNumber`标签嵌套到组件的标签里，就可以将组件的数据转换为`java.lang.Number`类型。`convertNumber`标签有多个属性，使得开发人员可以指定数据的格式和类型，表8-2列出了这些属性。

表8-2 `convertNumber`标签的属性

属 性	属性类型	描述信息
<code>binding</code>	<code>NumberConverter</code>	用于将一个转换器绑定到后台bean属性上
<code>currencyCode</code>	<code>String</code>	ISO 4217货币代码，仅当格式化货币时使用
<code>currencySymbol</code>	<code>String</code>	货币符号，仅当格式化货币时使用
<code>for</code>	<code>String</code>	与复合组件一起使用，引用的是复合组件里的某个对象。 <code>for</code> 标签嵌在复合组件标签的内部
<code>groupingUsed</code>	<code>Boolean</code>	指定是否在格式化输出中包含分组的分隔符
<code>integerOnly</code>	<code>Boolean</code>	指定是否只解析数值的整数部分
<code>locale</code>	<code>String</code> 或者 <code>Locale</code>	指定地区以确定数字以何种格式解析和显示
<code>maxFractionDigits</code>	<code>int</code>	小数部分输出的最多位数
<code>maxIntegerDigits</code>	<code>int</code>	整数部分输出的最多位数
<code>minFractionDigits</code>	<code>int</code>	小数部分输出的最少位数
<code>minIntegerDigits</code>	<code>int</code>	整数部分输出的最少位数
<code>pattern</code>	<code>String</code>	自定义格式化模式，决定如何解析和显示数字字符串
<code>type</code>	<code>String</code>	指定是否将字符串值解析和格式化为数字、货币或百分比。如果没有指定，则使用数字

下面是一个使用了convertNumber标签的例子，显示购物车里书籍的总价。

```
<h:outputText value="#{cart.total}" >
  <f:convertNumber type="currency"/>
</h:outputText>
```

当把Number Converter类型的转换器绑定到一个组件的时候，要保证后台bean的属性（即转换器要绑定的那个属性）的数据类型是基本类型或者是java.lang.Number类型。在刚才的例子中，cart.total的数据类型就是java.lang.Number。

看一个例子，购物车总价可以显示为如下形式：

\$934

如果使用如下的标签，且设定了货币的pattern属性，则可以得到同样的输出结果：

```
<h:outputText id="cartTotal"
  value="#{cart.Total}" >
  <f:convertNumber pattern="$###"
</h:outputText>
```

访问网址<http://docs.oracle.com/javase/tutorial/i18n/format/decimalFormat.html>，查看Java教程中“Customizing Formats”一节的内容，以了解如何使用convertNumber标签的pattern属性实现数据的格式化输出。

8.2 为组件注册监听器

应用开发人员实现的监听器有两种形式，即类或者后台bean方法。如果是后者，页面设计人员可以通过组件的valueChangeListener或actionListener引用这个方法。如果是前者（即类），页面设计人员可以通过valueChangeListener标签或actionListener标签引用这个监听器，同时为了给组件注册这一监听器，还要把该标签嵌套到组件的标签里去。

8.4.2节和8.4.4节讲述了页面设计人员如何使用valueChangeListener和actionListener属性引用处理事件的后台bean方法。

本节将讲述如何注册一个名为NameChanged的值变更监听器以及一个名为LocaleChange的動作监听器实现。

8.2.1 为组件注册一个值变更监听器

为组件注册值变更监听器ValueChangeListener，组件需要实现EditableValueHolder接口，并把valueChangeListener标签嵌套到页面上的组件标签里。valueChangeListener标签支持两个属性，如表8-3所示，必须使用两者之一。

表8-3 valueChangeListener标签的属性

属 性	描述信息
type	引用ValueChangeListener监听器的完全限定类名。可以接收文本表达式或者值表达式
binding	引用实现ValueChangeListener的一个对象。只能接收值表达式，且该表达式必须指向一个后台bean的属性，而该属性必须接收并返回一个ValueChangeListener实现

下面的例子展示了值变更监听器是如何注册到组件的：

```
<h:inputText id="name" size="50" value="#{cashier.name}"
    required="true">
    <f:valueChangeListener type="listeners.NameChanged" />
</h:inputText>
```

在这个例子里，核心标签的type属性指定了一个定制的监听器NameChanged作为ValueChangeListener的实现，这个组件的ID是name。

在组件的标签得到处理且本地数据也得到验证以后，对应的组件（关联到指定的ValueChangeListener的那个组件）实例会为这些ValueChangeEvent事件排队。

binding属性用于将ValueChangeListener实现绑定到后台bean的属性。这个工作方式和绑定属性到标准转换器标签类似。

8.2.2 为组件注册动作监听器

对于命令型的组件，页面设计人员可以通过嵌套actionListener标签到组件的标签里去，从而为组件注册一个ActionListener实现。与valueChangeListener标签类似，actionListener标签支持type和binding两个属性——必须使用两者之一来引用动作监听器。

在下面的例子中，commandLink标签引用了一个ActionListener实现，而不是后台bean的方法：

```
<h:commandLink id="NAmerica" action="bookstore">
    <f:actionListener type="listeners.LocaleChange" />
</h:commandLink>
```

actionListener标签的type属性指定了ActionListener实现对应的完全限定类名。和valueChangeListener标签类似，actionListener标签也支持binding属性。



8.3 标准验证器

JSF技术提供了一系列标准类和相关的标签，页面设计人员和开发人员都可以使用它们验证组件的数据是否有效。表8-4列举了所有标准验证器类以及可以在页面里验证数据有效性的标签。

表8-4 验证器类

验证器类	标 签	功 能
BeanValidator	validateBean	用于将一个bean验证器注册到组件上
DoubleRangeValidator	validateDoubleRange	检查组件的本地值是否在特定的范围内。这个值必须是浮点数或者可以转换成浮点数
LengthValidator	validateLength	检查组件的本地值的长度是否在特定的范围内。这个值必须是java.lang.String
LongRangeValidator	validateLongRange	检查组件的本地值的大小是否在特定的范围内。这个值必须是数值类型或者可以转换成long的String
RegexValidator	validateRegEx	检查组件的本地值是否匹配java.util.regex类型的正则表达式
RequiredValidator	validateRequired	验证类标签功能（class tag function），保证实现了EditableValueHolder接口的组件对应的本地值非空

和标准转换器类似，这里每一个验证器都有着一个或者多个标准的错误提示信息。如果页面里的组件注册了这里的某个验证器，且验证器无法验证组件的值，则验证器的错误提示信息会显示在页面上。举例来说，当组件的值超过了LongRangeValidator所能接受的最大值，错误提示信息则显示为：

```
{1}: Validation Error: Value is greater than allowable maximum of "{0}"
```

本例中，{1}会被替换成组件的标识或者id，{0}会被替换成验证器所设定的最大数值。

除了标准验证器之外，我们还可以通过Bean 验证来验证数据。更多信息，请参阅9.4节。

8.3.1 验证组件的值

要使用特定类型的验证器验证组件的值，需要把验证器注册到组件上。有如下3种方法：

- 把验证器相应的标签（见表8-4）嵌套到组件的标签里。8.3.2节讲述如何使用validateLongRange标签。可用同样的方法来使用其他的标准标签。
- 从组件标签的validator属性引用方法，该方法执行验证逻辑。
- 把validator标签嵌套到组件标签里去，并使用验证器标签的validatorId属性或binding属性引用这个验证器。

参见8.4.3节，可获得使用validator属性的更多信息。

validatorId属性的工作机制和converter标签的converterId属性类似，参见8.1.1节。

务必牢记，只能在实现了EditableValueHolder的组件里进行验证，因为只有这些组件接收的值才可以被验证。

8.3.2 LongRange Validator

下面的例子展示了如何在一个输入型组件里使用validateLongRange验证器，该组件的id为quantity。

```
<h:inputText id="quantity" size="4"
  value="#{item.quantity}" >
  <f:validateLongRange minimum="1"/>
</h:inputText>
<h:message for="quantity"/>
```

这个标签要求用户输入的数字最小为1，size属性则说明该数字最多为4位数，validateLongRange标签还有一个属性maximum，这个属性可以设置输入的最大值。

所有标准验证器标签都可以接收EL值表达式。这意味着属性可以引用后台bean的属性，而不仅是设置为字符串常量值。例如，前例中的validateLongRange标签可以引用后台bean的minimum属性，以获得验证器实现可以接受的最小值，代码如下所示：

```
<f:validateLongRange minimum="#{ShowCartBean.minimum}" />
```

8.4 引用后台 bean 的方法

组件的标签有一系列属性，用于引用后台bean里的方法，这些方法能够为这个标签关联的组

件执行特定功能。这些属性参见表8-5。

表8-5 可以引用后台bean方法的组件标签属性

属 性	功 能
action	引用后台bean的方法以执行标签的导航处理逻辑，返回的是代表接下来要显示的页面地址的字符串
actionListener	引用后台bean的方法以处理动作事件
validator	引用后台bean的方法以执行组件值的验证
valueChangeListener	引用后台bean的方法以处理值变更事件

只有实现了ActionSource接口的组件才可以使用action属性和actionListener属性。只有实现了EditableValueHolder的组件才可以使用validator属性或者valueChangeListener属性。

组件标签引用后台bean的方法，是将方法表达式作为某个属性的值实现的。被属性引用的方法必须遵循特定签名（signature），参见<http://docs.oracle.com/javaee/6/javaxserverfaces/2.0/docs/pdldocs/jsp/>处文档中对标签属性的定义。例如，inputText标签validator属性的定义如下：

```
void validate(javax.faces.context.FacesContext,
              javax.faces.component.UIComponent, java.lang.Object)
```

下面的内容会举一些例子来说明如何使用这些属性。

8.4.1 引用执行页面导航的方法

如果页面包含一个组件，比如一个按钮或者一个超链接，当组件被激活时（按下按钮或者单击了链接），应用能导航到另外的页面，响应这个组件的标签必须包括action属性。action属性实现如下任务之一：

- 设定一个字符串，告诉应用接下来要显示哪个页面；
- 引用后台bean的方法，执行一些事项，然后返回表示下一页的字符串。

下面是一个展示如何引用一个导航方法的例子：

```
<h:commandButton
    value="#{bundle.Submit}"
    action="#{cashier.submit}" />
```

8.4.2 引用处理动作事件的方法

如果页面里的一个组件产生了一个动作事件（比如单击了超链接），而且该事件是由后台bean的方法来处理，那么可以通过组件的actionListener属性引用那个方法。

如下的例子展示了如何引用一个导航方法：

```
<h:commandLink id="NAmerica" action="bookstore"
    actionListener="#{localeBean.chooseLocaleFromLink}">
```

这一组件标签的actionListener属性通过方法表达式来引用chooseLocaleFromLink方法。当用

户单击了这一组件生成的超链接，chooseLocaleFromLink方法就会处理单击事件。

8.4.3 引用执行验证逻辑的方法

如果页面中对某个组件的输入值的有效性检查是由后台bean实现的，就可以使用validator属性，通过组件标签来引用方法。

如下的例子展示了如何引用后台bean方法来执行验证逻辑，待验证的对象是页面中输入型组件里的电子邮件地址（email）。

```
<h:inputText id="email" value="#{checkoutFormBean.email}"
  size="25" maxlength="125"
  validator="#{checkoutFormBean.validateEmail}"/>
```

8.4.4 引用处理值变更事件的方法

如果页面里的组件产生了值变更事件，通过使用组件的valueChangeListener属性，可以让后台bean来处理这个事件。

下面的例子展示了当用户在页面上name对应的输入框处输入了名字以后，组件如何引用valueChangeListener来处理事件：

```
<h:inputText
  id="name"
  size="50"
  value="#{cashier.name}"
  required="true">
  <f:valueChangeListener type="listeners.NameChanged" />
</h:inputText>
```

为了引用这个后台bean的方法，标签使用了valueChangeListener属性：

```
<h:inputText
  id="name"
  size="50"
  value="#{cashier.name}"
  required="true"
  valueChangeListener="#{cashier.processValueChange}" />
</h:inputText>
```

组件标签的valueChangeListener属性通过方法表达式引用了CashierBean（后台bean）的processValueChange方法。processValueChange方法处理用户在页面上本组件生成的输入框中输入姓名这个事件。

本书第7章和第8章讲述了如何把组件添加到网页里,以及如何使用组件标签和核心标签把组件和服务器端的对象关联起来,还讲述了如何借助转换器、监听器和验证器为组件添加扩展功能。开发JSF应用同样涉及为服务器端对象编写代码,这些服务器端对象包括后台bean、转换器、事件处理程序以及验证器。

本章会对后台bean做简要介绍,讲述如何为后台bean编写相关的方法和属性,以实现完整的JSF应用。本章最后还会介绍Bean验证。

本章内容

- ☐ 后台bean
- ☐ 为后台bean的属性编写代码
- ☐ 为后台bean的方法编写代码
- ☐ 使用Bean验证

9.1 后台 bean

典型的JSF应用包含一个或多个后台bean,它们都是JSF托管bean,且与页面中的组件相关联。本节将介绍创建、配置以及在应用里使用后台bean的基本概念。

9.1.1 创建后台bean

后台bean的构造方法和其他的JavaBeans组件的构造方法一样,没有参数。后台bean有一组属性和方法,这些方法为组件实现特定的功能。后台bean的属性均可以和如下元素绑定:

- ☐ 组件的值;
- ☐ 组件的实例;
- ☐ 转换器的实例;
- ☐ 监听器的实例;
- ☐ 验证器的实例。

后台bean的方法可以实现一些常用的功能,例如:

- 验证组件的数据是否有效;
- 处理组件所触发的事件;
- 执行导航逻辑, 以确定应用接下来要显示的页面。

如同所有JavaBeans组件一样, 后台bean的属性包括私有的成员变量以及一组对应的存取方法, 如下面的代码所示:

```
Integer userNumber = null;
...
public void setUserNumber(Integer user_number) {
    userNumber = user_number;
}
public Integer getUserNumber() {
    return userNumber;
}
public String getResponse() {
    ...
}
```

当bean属性绑定到组件的值时, 属性的类型可以是基本数据类型、数值型, 还可以是任何Java对象类型, 只要有合适的转换器可用就行。例如, 属性可以是Date类型, 前提是应用有合适的转换器可用且该转换器可以转换Date类型为String类型, 同时也支持String到Date类型的转换。参见9.2节以了解更多与组件标签可以接收的数据类型有关的信息。

当bean属性绑定到组件实例时, 属性的类型必须和组件对象的类型一致。例如, 如果javax.faces.component.UISelectBoolean组件绑定到了一个属性, 则该属性必须能够接收以及返回一个UISelectBoolean对象。类似地, 如果属性绑定到一个转换器、验证器或者监听器的实例, 则属性的类型必须要和绑定的对象一样, 即必须为转换器、验证器或者监听器类型。

更多有关编写后台bean以及相关属性的内容, 参见9.2节。

9.1.2 使用EL引用后台bean

页面设计人员可以使用表达式语言, 把组件的值或对象绑定到后台bean的属性上, 或者从组件标签里引用后台bean的方法。如6.1节所述, 表达式语言提供如下一些特性:

- 表达式的延后求值;
- 使用值表达式读写数据;
- 方法表达式。

表达式的延后求值很重要, 因为JSF生命周期分为几个阶段, 而在每个阶段里, 组件的事件处理、数据转换、数据验证以及数据传递到外部的对象这些都是有顺序的。JSF实现必须支持表达式的延后求值, 直到生命周期的合适时机再求值。因此, 标签属性总是使用延后求值语法(用#{ }加以区分)。

要把数据保存到外部对象里, 几乎所有的JSF标签属性都使用左值表达式。这种类型的表达式允许存取和设置外部对象的数据。

最后, 有些组件的标签属性可以使用方法表达式来调用对象的方法, 以处理组件事件或者验

证、转换组件的数据。

看个JSF标签使用EL的例子，假定某应用的标签引用了方法以执行用户输入数据的验证：

```
<h:inputText id="userNo"
    value="#{UserNumberBean.userNumber}"
    validator="#{UserNumberBean.validate}" />
```

该标签用一个左值表达式把名为userNo的组件的值绑定到UserNumberBean.useNumber这个后台bean的属性上。该标签使用了一个方法表达式来引用UserNumberBean的validate方法，它对组件的本地值进行验证。用户在页面上输入的值（本地值）保存在一个局部变量里，当对表达式求值的时候，validate方法就会被调用以验证该局部变量的值是否有效。

几乎所有的JSF标签属性都使用值表达式。除了引用bean属性，值表达式还可以引用列表（list）、映射（map）、数组（array）、隐式对象以及资源。

另外一个使用值表达式的地方是把组件实例绑定到后台bean的属性上。页面设计人员通过引用binding的属性来实现这一操作：

```
<inputText binding="#{UserNumberBean.userNoComponent}" />
```

除了将表达式与标准组件标签一同使用，我们还可以配置自己的定制组件属性以接收表达式，方法是创建javax.el.ValueExpression或者javax.el.MethodExpression实例。

有关EL的信息，请参阅第6章。

有关从组件的标签引用后台bean的方法的更多内容，请参阅8.4节。

9.2 为后台 bean 的属性编写代码

正如9.1节所述，后台bean的属性可以绑定到如下元素：

- 组件的值；
- 组件的实例；
- 转换器的实例；
- 监听器的实例；
- 验证器的实例。

这些属性遵循JavaBeans组件（也称作bean）的惯例。有关JavaBeans组件的更多信息，可以参考*Java Beans Tutorial*（JavaBeans教程），网址为<http://docs.oracle.com/javase/tutorial/javabeans/index.html>。

通过组件标签的value属性，可以把组件的值绑定到后台bean的属性；使用组件的binding属性，可以把组件的实例绑定到后台bean的属性。类似地，转换器、监听器以及验证器标签通过其binding属性将其实现绑定到后台bean的属性。

为了把组件的值绑定到后台bean的属性，属性的类型必须和它所绑定的组件的值的类型相匹配。例如，如果一个后台bean的属性绑定到了一个UISelectBoolean组件的值，属性应当接收以及返回一个boolean类型的值或者一个Boolean类型的封装器对象。

为了把组件的实例绑定到后台bean的属性，bean的属性必须和组件的类型相匹配。例如，如果一个后台bean的属性绑定到一个UISelectBoolean实例，则属性应当可以接收以及返回一个UISelectBoolean类型的值。

类似地，要绑定一个转换器、监听器或者验证器的实现到一个后台bean的属性，属性必须接收以及返回同样类型的转换器、监听器或者验证器对象。例如，如果使用convertDateTime标签来绑定一个DateTimeConverter转换器到一个属性，这个属性必须接收以及返回一个DateTimeConverter实例。

本节余下的内容将讲述如何编写属性相关的代码，使属性可以绑定到组件值、组件对象（有关组件对象的内容，参见7.2节）的实例，还可以绑定到转换器、监听器和验证器的实现。

9.2.1 为绑定到组件值的属性编写代码

为了编写后台bean属性的相关代码，属性的类型必须与组件值的类型相匹配。

表9-1列举了javax.faces.component类以及它们所能接收值的类型。

表9-1 可接收的组件值类型

组 件 类	可接收的组件值类型
UIInput、UIOutput、 UISelectItem、UISelectOne UIData	任何基本类型、数字类型或者任何Java对象，只要有合适的转换器可用 bean 数组、bean列表、单个bean、Java.sql.ResultSet、javax.servlet.jsp.jstl.sql. Result、javax.sql.RowSet
UISelectBoolean	boolean或者Boolean
UISelectItems	java.lang.String、Collection、Array、Map
UISelectMany	数组或者列表，里面的元素可以是任意标准类型

当使用组件标签的value属性把组件绑定到属性的时候，页面设计人员需要确保对应的属性与组件值的类型相匹配。

1. UIInput和UIOutput属性

在下面的例子里，h:inputText标签把name组件绑定到名为CashierBean的后台bean的name属性上：

```
<h:inputText id="name" size="50"  
    value="#{cashier.name}">  
</h:inputText>
```

下面是CashierBean这个后台bean的代码片段，它展示了bean的属性类型，也就是先前的组件标签所绑定的属性类型：

```
protected String name = null;  
  
public void setName(String name) {  
    this.name = name;  
}
```



```
public String getName() {
    return this.name;
}
```

正如8.1节讲述的那样，转换一个输入型组件或者输出型组件的值，可以使用转换器或者创建一个bean属性，然后把组件绑定到该属性，不过类型一定要匹配。此处有一个标签的例子，出自8.1.2节，它显示所购图书的发货日期。

```
<h:outputText value="#{cashier.shipDate}">
    <f:convertDateTime dateStyle="full" />
</h:outputText>
```

这个标签代表的bean属性的类型必须是java.util.Date。如下代码片段展示了CashierBean这个后台bean的shipDate属性。在先前的例子里，它绑定了标签的值：

```
protected Date shipDate;

public Date getShipDate() {
    return this.shipDate;
}
public void setShipDate(Date shipDate) {
    this.shipDate = shipDate;
}
```

2. UIData属性

数据组件必须绑定到表9-1中的某一后台bean属性类型。数据组件在7.2.11节已有讲述，下面有关dataTable起始标签的代码就摘自那一节。

```
<h:dataTable id="items"
    ...
    value="#{cart.items}"
    var="item" >
```

值表达式指向了购物车的bean（cart）的属性items。cart这个购物车bean维护了一个映射，映射里的每一个元素都是ShoppingCartItem类型的bean。

cart的getItems方法的返回值是ShoppingCartItem的所有实例的List。ShoppingCartItem实例是在客户把书添加到购物车时保存在映射里的，请看如下的代码片段：

```
public synchronized List getItems() {
    List results = new ArrayList();
    results.addAll(this.items.values());
    return results;
}
```

包含在数据组件中的所有子组件，都绑定到cart这个bean的属性，而这个cart则绑定到整个数据组件。例如，下面的代码中有一个h:outputText标签，它显示表格里每一本书的书名。

```
<h:commandLink action="#{showcart.details}">
    <h:outputText value="#{item.item.name}" />
</h:commandLink>
```

3. UISelectBoolean属性

UISelectBoolean组件的值保存在后台bean的属性里，该属性的数据类型必须是boolean或者Boolean。7.2.8节的selectBooleanCheckbox标签的那个例子，就是把组件绑定到了一个属性。下面的例子展示了把组件的值绑定到一个boolean类型属性的标签：

```
<h:selectBooleanCheckbox title="#{bundle.receiveEmails}"
    value="#{custFormBean.receiveEmails}" >
</h:selectBooleanCheckbox>
<h:outputText value="#{bundle.receiveEmails}">
```

在下面的例子中，属性绑定到了由示例标签表示的组件上：

```
protected boolean receiveEmails = false;

...
public void setReceiveEmails(boolean receiveEmails) {
    this.receiveEmails = receiveEmails;
}
public boolean getReceiveEmails() {
    return receiveEmails;
}
```

4. UISelectMany属性

因为UISelectMany组件允许用户从列表里选择一个或多个选项，该组件必须映射到类型为List或者array的后台bean的属性。这个bean属性代表一组从可用选项里选出来的当前项集合。

下面的例子是出自7.2.9节的selectManyCheckbox标签：

```
<h:selectManyCheckbox
    id="newsletters"
    layout="pageDirection"
    value="#{cashier.newsletters}">
    <f:selectItems value="#{newsletters}"/>
</h:selectManyCheckbox>
```

下面的代码展示了bean属性，该属性映射至前面例子中selectManyCheckbox标签的value：

```
protected String newsletters[] = new String[0];

public void setNewsletters(String newsletters[]) {
    this.newsletters = newsletters;
}
public String[] getNewsletters() {
    return this.newsletters;
}
```

UISelectedItem和UISelectItems组件用来表示UISelectMany组件里的所有选项。参见接下来的“UISelectedItem属性”和“UISelectItems属性”，以获得有关为UISelectedItem和UISelectItems组件编写bean属性相关代码的更多信息。

5. UISelectOne属性

UISelectOne属性接收的数据类型与UIInput和UIOutput属性接收的数据类型相同，因为UISelectOne组件代表的是一系列选项中某个选定的选项。这个选项可以是任何基本数据类型，以及任何其他可以应用转换器的类型。

下面是一个selectOneMenu标签的例子，来自于7.2.8节中的“使用h:selectOneMenu标签显示菜单”。

```
<h:selectOneMenu id="shippingOption"
    required="true"
    value="#{cashier.shippingOption}">
    <f:selectItem
        itemValue="2"
        itemLabel="#{bundle.QuickShip}"/>
```

```
<f:selectItem
    itemValue="5"
    itemLabel="#{bundle.NormalShip}"/>
<f:selectItem
    itemValue="7"
    itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>
```

下面的代码是该标签所对应的bean属性:

```
protected String shippingOption = "2";

public void setShippingOption(String shippingOption) {
    this.shippingOption = shippingOption;
}
public String getShippingOption() {
    return this.shippingOption;
}
```

注意, shippingOption代表的是UISelectOne组件列表里的当前选中项。

UISelectItem和UISelectItems组件代表UISelectOne组件里的所有值。7.2.8节中的“使用h:selectOneMenu标签显示菜单”已讲述过这部分内容了。

有关如何为UISelectItem和UISelectItems组件的后台bean属性编写相关代码的更多内容, 参见接下来的“UISelectItem属性”和“UISelectItems属性”中的内容。

6. UISelectItem属性

UISelectItem组件代表UISelectMany或UISelectOne组件包含的一系列值里的某一个选定项。UISelectItem组件必须绑定到类型为javax.faces.model.SelectItem的后台bean的属性上。SelectItem对象是由一个代表选项值的Object, 以及分别代表UISelectItem对象的标识和描述信息的String组成。selectOneMenu标签的例子, 引自7.2.8节中的“使用h:selectOneMenu标签显示菜单”。该例子包含了selectItem标签, 这一标签设置了页面内列表的数据。下面是一个有关bean属性的例子, 该例子为bean的列表设置了数据:

```
SelectItem itemOne = null;

SelectItem getItemOne(){
    return itemOne;
}
void setItemOne(SelectItem item) {
    itemOne = item;
}
```

7. UISelectItems属性

UISelectItems组件是UISelectMany和UISelectOne组件的子组件。每一个UISelectItems组件都包含了一组javax.faces.model.SelectItem实例或者对象的集合, 如数组、列表, 还有可能是一组POJO。

下面的内容讲述如何为包含了SelectItem实例的selectItems标签编写属性相关的代码。

在后台bean里, 可以以编程的方式用UISelectItem实例填充UISelectItems。

(1) 在后台bean里, 创建一个绑定到SelectItem组件的列表。

(2) 定义一系列SelectItem对象，设置其初始值，并用这些SelectItem对象填充列表。

下面这段代码取自某个后台bean，用于展示如何创建SelectItems属性。

```
import javax.faces.model.SelectItem;
...
protected ArrayList options = null;
protected SelectItem newsletter0 =
    new SelectItem("200", "Duke's Quarterly", "");
...
//in constructor, populate the list
options.add(newsletter0);
options.add(newsletter1);
options.add(newsletter2);
...
public SelectItem getNewsletter0(){
    return newsletter0;
}

void setNewsletter0(SelectItem firstNL) {
    newsletter0 = firstNL;
}
// Other SelectItem properties

public Collection[] getOptions(){
    return options;
}
public void setOptions(Collection[] options){
    this.options = new ArrayList(options);
}
}
```

这段代码首先初始化了列表里的所有选项。在定义每一个newsletter属性的时候就为其赋了初始值，然后将所有newsletter（SelectItem类型）添加到了列表里。最后，代码包含了必须有的存取方法，即setOptions和getOptions。

9.2.2 为绑定到组件实例的属性编写代码

绑定到组件实例的属性返回和接收的是组件的实例，而不是组件的值。如下组件把组件的实例绑定到后台bean的属性：

```
<h:selectBooleanCheckbox
    id="fanClub"
    rendered="false"
    binding="#{cashier.specialOffer}" />
<h:outputLabel for="fanClub"
    rendered="false"
    binding="#{cashier.specialOfferText}" >
    <h:outputText id="fanClubLabel"
        value="#{bundle.DukeFanClub}" />
</h:outputLabel>
```

selectBooleanCheckbox标签生成复选框并将fanClub这个UISelectBoolean类型的组件绑定到CashierBean这个后台bean的specialOffer属性上。outputLabel标签绑定fanClubLabel组件（表示复选框的标识）到CashierBean的specialOfferText属性。如果用户订购的书籍超过100美元，单

击Submit（提交）按钮，CashierBean的submit方法会设置这两个组件的rendered属性为true，这将使得重新生成页面时，复选框以及它的标识会显示出来。

因为与例子里标签相对应的组件绑定到后台bean的属性，这些属性必须与组件的类型相匹配。这意味着，specialOfferText属性必须是UIOutput类型，而specialOffer属性则必须是UISelectBoolean类型：

```
UIOutput specialOfferText = null;

public UIOutput getSpecialOfferText() {
    return this.specialOfferText;
}

public void setSpecialOfferText(UIOutput specialOfferText) {
    this.specialOfferText = specialOfferText;
}

UISelectBoolean specialOffer = null;

public UISelectBoolean getSpecialOffer() {
    return this.specialOffer;
}

public void setSpecialOffer(UISelectBoolean specialOffer) {
    this.specialOffer = specialOffer;
}
```

有关组件绑定的更多常规信息，可以参见9.1节。

有关按下按钮时如何引用后台bean的方法以执行导航逻辑，参见8.4.1节。

有关如何编写后台bean的方法来处理页面导航，参见9.3.1节。

9.2.3 为绑定到转换器、监听器以及验证器的属性编写代码

JSF技术里使用的所有标准转换器、监听器以及验证器标签都支持绑定属性，从而允许把转换器、监听器以及验证器的实现绑定到后台bean的属性上。

如下的例子展示了一个标准的convertDateTime标签，它使用了值表达式，其binding属性将DateTimeConverter实例绑定到LoginBean的convertDate属性：

```
<h:inputText value="#{LoginBean.birthDate}">
    <f:convertDateTime binding="#{LoginBean.convertDate}" />
</h:inputText>
```

因此，convertDate属性必须接收以及返回一个DateTimeConverter对象，如下所示：

```
private DateTimeConverter convertDate;
public DateTimeConverter getConvertDate() {
    ...
    return convertDate;
}

public void setConvertDate(DateTimeConverter convertDate) {
    convertDate.setPattern("EEEEEEEE, MMM dd, yyyy");
    this.convertDate = convertDate;
}
```

因为转换器绑定了后台bean的属性，后台bean的属性可以修改转换器的属性或者为之添加新的功能。在刚才的例子中，属性设置了日期的模式，转换器用此模式解析用户的输入，并将解析结果保存到一个Date对象里。

至于后台bean属性绑定到验证器或者监听器实现的代码编写，从方式上来说，大致和转换器的代码编写一样，用途也大致一样。

9.3 为后台 bean 的方法编写代码

后台bean的方法可以被页面里的组件调用，以执行一些应用特定的功能，这些功能包括：

- 执行与导航相关的处理逻辑；
- 处理动作事件；
- 执行验证组件的值是否合法的逻辑；
- 处理值变更事件。

使用后台bean来实现这些功能，可以不用使用Validator 接口处理验证逻辑，也避免了使用某个监听器接口处理事件。此外，使用后台bean而不是Validator实现来执行验证功能，也就没有必要为Validator实现创建定制的标签了。

通常，把这些方法放到同一个后台bean里是个不错的办法。为组件将来方便地引用这些方法，这个bean要定义相关的属性。之所以这么做，原因是这些方法可能需要访问组件的数据，以决定如何处理与组件相关的事件或者执行验证逻辑。

接下来的内容主要讲述如何编写几种不同类型的后台bean的方法。

9.3.1 编写处理导航的方法

action方法，即后台bean用来处理导航逻辑的方法，必须是公有类型的方法。该方法没有参数，返回值是一个Object，这是一个逻辑结果，被导航系统用来确定接下来要显示哪个页面。组件标签的action属性可以引用这个方法。

下面的action方法取自CashierBean这个后台bean。当用户在页面上单击了Submit按钮，该方法就会被调用。如果用户订购了超过100美元的图书，则该方法会设置fanClub和specialOffer组件的rendered属性为true。这样一来，当下一次显示该页面的时候，fanClub和specialOffer就会在页面上显示出来。

设置了组件的rendered属性为true之后，这个方法返回逻辑结果null。这样一来，JSF在重新显示该页面的时候就不用创建新的视图，而是刷新当前显示的页面，且用户的数据得到保留。如果这个方法返回的是purchase，这是用于导航至支付页面的逻辑结果，且当显示支付页面时，不会保留用户的输入。

如果用户购买的图书不足100美元，或者thankYou组件已经显示了，那么该方法会返回receipt。JSF会在方法调用结束之后显示收据页面：

```
public String submit() {  
    ...  
    if(cart().getTotal() > 100.00 &&  
        !specialOffer.isRendered())  
    {  
        specialOfferText.setRendered(true);  
        specialOffer.setRendered(true);  
        return null;  
    } else if (specialOffer.isRendered() &&  
        !thankYou.isRendered()){  
        thankYou.setRendered(true);  
        return null;  
    } else {  
        clear();  
        return ("receipt");  
    }  
}
```

通常来说，action方法返回的是一个String类型的结果，正如刚才的例子所示。此外，也可以定义一个Enum类来封装所有可能的结果字符串，然后定义action方法，使其返回enum常量，它代表着Enum类定义的某个特定的String结果。

如下的例子使用Enum类来封装所有的逻辑结果：

```
public enum Navigation {  
    main, accountHist, accountList, atm, atmAck, transferFunds,  
    transferAck, error  
}
```

当它返回一个结果时，action方法使用。来引用来自Enum类的结果：

```
public Object submit(){  
    ...  
    return Navigation.accountHist;  
}
```

8.4.1节讲述了组件标签如何引用这个方法。9.2.2节讲述了如何编写组件实例要绑定到的bean属性。

9.3.2 编写处理动作事件的方法

后台bean里处理动作事件的方法必须是公有类型的方法，用于接收动作事件，返回值类型为void。组件标签的actionListener属性就可以引用该方法。只有实现了javax.faces.component.ActionSource接口的组件才可以引用该方法。

下面的例子中，LocalBean这个后台bean有个名为chooseLocaleFromLink的方法，用来处理用户在页面上单击了超链接的事件：

```
public void chooseLocaleFromLink(ActionEvent event) {  
    String current = event.getComponent().getId();  
    FacesContext context = FacesContext.getCurrentInstance();  
    context.getViewRoot().setLocale((Locale)  
        locales.get(current));  
}
```

该方法首先从输入参数event里取得生成单击事件的组件，然后取得组件的ID，组件ID对应了某个地域。locales是应用里的所有的Locale的集合，这一方法利用获取方法从HashMap对象(locales)里找出组件ID的对应值，最后用从HashMap对象里找到的这个值设置地域。

8.4.2节讲述了组件标签是如何引用这个方法的。

9.3.3 编写执行验证的方法

这里并不实现Validator接口以实现组件的验证功能，而是为后台bean添加一个方法，用该方法来验证组件里的输入值是否合法。执行验证的后台bean的方法必须接受3个参数，分别是FacesContext、要被验证数据的组件以及要验证的数据，就像Validator接口的validate方法所做的那样。组件通过validator属性引用后台bean的方法。只有UIInput组件的值或者扩展自UIInput的组件的值，才能够被验证。

下面是一个后台bean方法的例子，该方法用于验证用户的输入：

```
public void validateEmail(FacesContext context,
    UIComponent toValidate, Object value) {

    String message = "";
    String email = (String) value;
    if (email.contains('@')) {
        ((UIInput)toValidate).setValid(false);
        message = CoffeeBreakBean.loadErrorMessage(context,
            CoffeeBreakBean.CB_RESOURCE_BUNDLE_NAME,
            "EmailError");
        context.addMessage(toValidate.getClientId(context),
            new FacesMessage(message));
    }
}
```

仔细研究一下刚才的代码片段：

- (1) 首先，validateEmail方法取得组件的本地值；
- (2) 然后，方法检查值里是否包含@字符；
- (3) 如果不包含，将组件的valid属性设置为false；
- (4) 加载错误消息，并把消息放到FacesContext实例队列中去，通过组件ID把组件和消息关联起来。

有关组件的标签如何引用这个方法，可以参考8.4.3节。

9.3.4 编写处理值变更事件的方法

后台bean里处理值变更事件的方法必须是公有类型，它能接受值变更事件，并且返回void。组件使用valueChangeListener属性引用这个方法。本节主要讲述如何编写后台bean的方法以替换掉过去的方案，即ValueChangeListener实现。

下面这个例子来自于8.2.1节，有一个ValueChangeListener实例注册到了id为name的h:inputText标签上。ValueChangeListener实例可以处理用户在这个标签对应的组件里输入数据的事件。当用户

输入了数据，就会产生值变更事件。相应地，ValueChangeListener类的processValueChange(ValueChangeEvent)方法就会被调用。

```
<h:inputText id="name" size="50" value="#{cashier.name}"
    required="true">
    <f:valueChangeListener type="listeners.NameChanged" />
</h:inputText>
```

替代ValueChangeListener实现的方案是编写一个后台bean的方法来处理这个值变更事件。其做法是，把实现了ValueChangeListener接口的类（名为NameChanged）里的processValueChange(ValueChangeEvent)代码搬到自己的后台bean里。

下面是后台bean的方法，它用于处理用户在页面上的name字段处输入数据的事件：

```
public void processValueChange(ValueChangeEvent event)
    throws AbortProcessingException {
    if (null != event.getNewValue()) {
        FacesContext.getCurrentInstance().
            getExternalContext().getSessionMap().
                put("name", event.getNewValue());
    }
}
```

为了让这个方法处理由输入型组件生成的ValueChangeEvent事件，可以利用组件标签的valueChangeListener属性引用它，详见8.4.4节。

9.4 使用 Bean 验证

验证用户输入的有效性以维护数据的完整性，是应用程序逻辑的一项重要内容。即使是最简单的应用，数据验证也可以发生在不同的层，如前面章节介绍的那个guessnumber示例应用。guessnumber这个示例应用就是在表现层验证h:inputText标签对应的输入组件里的内容是否为数值型数据，在业务层验证数据是否在一定范围之内。

Java EE 6平台引入了JavaBeans验证，即Bean验证，这是一种新的验证模型。该模型通过注解的方式添加约束来实现验证，注解可以放置在bean的某个属性、方法或者JavaBeans组件（如后台bean）的类上。

约束可以是内置的，也可以是用户自定义的。用户自定义的约束叫做“定制约束”。javax.validation.constraints包里定义了一些内置的约束。表9-2列举了所有的内置约束。

表9-2 内置的Bean验证约束

约 束	描 述	例 子
@AssertFalse	字段或属性的值必须是false	@AssertFalse Boolean isUnsupported;
@AssertTrue	字段或属性的值必须是true	@AssertTrue Boolean isActive;
@DecimalMax	字段或属性的值必须是小数，并且小于或等于约束中给出的值	@DecimalMax("30.00") BigDecimal discount;

(续)

约 束	描 述	例 子
@DecimalMin	字段或属性的值必须是小数，并且大于或等于约束中给出的值	@DecimalMin("5.00") BigDecimal discount;
@Digits	字段或属性的值必须是指定范围内的数字。元素integer指定了数字的最大整数位数，fraction元素指定了最大的小数位数	@Digits(integer=6, fraction=2) BigDecimal price;
@Future	字段或属性的值必须是未来的日期	@Future Date eventDate;
@Max	字段或属性的值必须是小于或等于指定值的整数	@Max(10) int quantity;
@Min	字段或属性的值必须是大于或等于指定值的整数	@Min(5) int quantity;
@NotNull	字段或属性的值必须非空	@NotNull String username;
@Null	字段或属性的值必须为空	@Null String unusedString;
@Past	字段或属性的值必须是过去的日期	@Past Date birthday;
@Pattern	字段或属性的值必须匹配regex定义的正则表达式	@Pattern(regex= "\\(\\d{3}\\)\\d{3}-\\d{4}") String phoneNumber;
@Size	字段或属性的size被求值且必须在指定的范围内。如果字段或属性是String类型，那么size指的是字符串的长度；如果字段或属性是Collection类型，size指的是Collection里的元素个数；如果字段或属性是Map，size指的是Map里的元素个数；如果字段或属性是数组，size指的是数组的长度。可以使用（可选的）max或min元素来设定范围	@Size(min=2, max=240) String briefMessage;

在下面的例子里，注解添加到了类的成员变量上，使用的是内置的@NotNull约束：

```
public class Name {
    @NotNull
    private String firstname;

    @NotNull
    private String lastname;
}
```

可以为JavaBeans组件添加多个约束。例如，可以在firstname和lastname字段上放置额外的约束，以验证字段长度：

```
public class Name {
    @NotNull
    @Size(min=1, max=16)
    private String firstname;

    @NotNull
    @Size(min=1, max=16)
    private String lastname;
}
```

下面的例子展示了一个方法，该方法有一个用户自定义的约束，用来检查预先定义电子邮件地址格式是否正确，如是否为公司电子邮件账号：

```
@ValidEmail
public String getEmailAddress() {
    return emailAddress;
}
```

对于内置的约束来说，有默认实现可以直接使用。用户自定义或定制的约束则需要实现类的支持。上面的例子中，`@ValidEmail`自定义约束就需要相应的实现类。

任何验证错误都可以得到妥善地处理，在页面上可以用`h:messages`来显示具体出错信息。

任何包含了Bean验证注解的后台bean，都会自动地获取JSF页面中字段的约束。

更多有关验证约束的信息，可以参考19.1.2节中“验证持久化字段和属性”的内容。

验证null以及空字符串

Java语言里，`null`和空字符串是不同的。空字符串是指字符串长度为0，而`null`字符串是指根本就没有值。

空字符串表示为`""`，本质上来说这是一个包含了0个字符的字符数组。`null`字符串表示为`null`，可以理解为这个对象没有初始化。

后台bean元素（对应JSF里的文本组件，比如），会被JSF实现初始化为空字符串。当用户输入对这种字段不必需时，这类字符串的验证是一个问题。考虑下面的例子，`testString`是bean的变量，它的值由用户输入设置。在这个例子里，该字段是可选的，即用户可以不输入数据。

```
if (testString.equals(null)) {
    doSomething();
} else {
    doAnotherThing();
}
```

在默认情况下，即使用户没有输入数据，`doAnotherThing`方法也会被调用。因为`testString`的初始值设置成了空字符串，而不是`null`。

为了让Bean验证模型按预期工作，必须在Web应用的部署描述文件`web.xml`里将上下文参数`javax.faces.INTERPRET_EMPTY_STRING_VALUES_AS_NULL`设置为`true`：

```
<context-param>
  <param-name>
    javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
  </param-name>
  <param-value>true</param-value>
</context-param>
```

这个参数使得JSF实现把空字符串作为`null`对待。

另一方面，假设在某个元素上有`@NotNull`约束，意味着该输入是必需的。在这种情况下，空字符串会被传递给验证约束。然而，如果设置了上下文参数`javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL`为`true`，则Bean验证运行的时候，传递的后台bean属性值就是`null`，这会导致`@NotNull`约束失败。

Web被用于交付服务后不久，服务提供商很快意识到在某些情况下，需要发布的是含动态内容的服务。applet（小应用程序）就是为了解决这个问题出现最早的一项技术，它主要基于客户端平台发布动态内容。与此同时，开发人员也开始研究如何在服务器端实现同样的目的。最开始，CGI（Common Gateway Interface，通用网关接口）服务器端脚本技术是用来生成动态内容的主要技术。虽然该技术得到了广泛应用，但是技术本身有一些缺点，如它的平台依赖性以及缺乏扩展性。为了解决这些问题，Java Servlet技术应运而生。这是一项跨平台的技术，可以用来生成面向用户的动态内容。

本章内容

- ☐ 什么是servlet
- ☐ servlet生命周期
- ☐ 共享信息
- ☐ 创建以及初始化servlet
- ☐ 编写service方法
- ☐ 过滤请求和应答
- ☐ 调用其他Web资源
- ☐ 访问Web上下文
- ☐ 维护客户端状态
- ☐ 结束servlet
- ☐ mood示例应用
- ☐ 有关Java Servlet技术的更多信息

10.1 什么是 servlet

servlet是Java编程语言提供的类，它通过请求-应答编程模型扩展服务器的功能，供运行其上的应用使用。虽然servlet可以响应任何类型的请求，它主要还是用于扩展运行在Web服务器上的Web应用的功能。对于这样的应用，Java Servlet技术定义了HTTP专用servlet类。

`javax.servlet`和`javax.servlet.http`这两个包提供了编写servlet所需的接口和类。所有的servlet必须实现`Servlet`接口，这个接口定义了生命周期方法。实现一般的应用可以使用或者扩展Java Servlet API提供的`GenericServlet`类。`HttpServlet`类提供了方法，比如`doGet`和`doPost`，可以处理HTTP特定的请求。

10.2 servlet 生命周期

servlet的生命周期是由容器（servlet被部署的地方，即Web服务器）控制的，当请求映射到servlet时，容器执行如下的步骤。

(1) 如果servlet的实例尚不存在，Web容器：

- (a) 加载servlet类；
- (b) 创建这个类的实例；
- (c) 初始化这个servlet实例，方法是调用`init`方法。servlet初始化的内容参见10.4节。

(2) 调用`service`方法，并传递请求对象和应答对象。`service`方法将在10.5节中讲述。

如果容器需要移除servlet，可以通过调用servlet的`destroy`方法来结束servlet。这部分内容将在10.10节中讨论。

10.2.1 处理servlet生命周期内的事件

在servlet的生命周期里，可以监视和响应事件，这可以通过定义监听器对象来实现。监听器对象的方法会在对应的生命周期事件发生时被调用。为了使用这些监听器对象，必须事先定义以及指定监听器类。

定义监听器类

可以通过实现监听器接口来定义一个监听器类。表10-1列举了可以监听的事件以及对应的必须实现的接口。当监听器方法被调用时，会接收一个事件参数（该参数包含了事件的有关信息）。例如，`HttpSessionListener`接口里的方法会接收传递过来的参数`HttpSessionEvent`，该参数里面包含了`HttpSession`。

表10-1 servlet生命周期事件

对 象	事 件	监听器接口以及事件类
Web上下文（参见10.8节）	初始化和销毁	<code>javax.servlet.ServletContextListener</code> 和 <code>ServletContextEvent</code>
	属性的添加、删除以及替换	<code>javax.servlet.ServletContextAttributeListener</code> 和 <code>ServletContextAttributeEvent</code>
会话（参见10.9节）	创建、失效、激活、钝化以及超时	<code>javax.servlet.http.HttpSessionListener</code> 、 <code>javax.servlet.http.HttpSessionActivationListener</code> 以及 <code>HttpSessionEvent</code>
	属性的添加、删除以及替换	<code>javax.servlet.http.HttpSessionAttributeListener</code> 和 <code>HttpSessionBindingEvent</code>

(续)

对 象	事 件	监听器接口以及事件类
请求	Servlet 请求正在被 Web 组件处理	javax.servlet.ServletRequestListener 和 ServletRequestEvent
	属性的添加、删除以及替换	javax.servlet.ServletRequestAttributeListener 和 ServletRequestAttributeEvent

使用@WebListener注解可以定义一个监听器，用于在特定的Web应用上下文中截获发生的各种事件。带@WebListener注解的类必须实现如下接口中的某一种：

```

javax.servlet.ServletContextListener
javax.servlet.ServletContextAttributeListener
javax.servlet.ServletRequestListener
javax.servlet.ServletRequestAttributeListener
javax.servlet.http.HttpSessionListener
javax.servlet.http.HttpSessionAttributeListener

```

例如，如下的代码片段定义了一个监听器，它实现了两个接口：

```

import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener()
public class SimpleServletListener implements ServletContextListener,
    ServletContextAttributeListener {
    ...
}

```

10.2.2 处理servlet错误

在servlet执行的过程中，有可能发生各种各样的异常。当发生异常时，Web容器会创建一个默认的面，包含如下的消息：

```
A Servlet Exception Has Occurred
```

但是，可以为特定类型的异常指定一个错误页面，这样当异常发生时，可使容器返回那个特定的错误页面。

10.3 共享信息

Web组件和大多数对象一样，通常是和其他对象配合起来使用以实现特定的任务。Web组件可以通过如下方式来完成任：

- 使用私有的辅助对象（比如 JavaBeans组件）；
- 共享公有属性所对应的对象；
- 使用数据库；
- 调用其他Web资源。关于Web组件使用Java Servlet技术调用其他Web资源的机制，将在10.7节里介绍。

10.3.1 有作用域的对象

Web组件可以通过对象的属性来维护和传递共享信息，而这些属性有着不同的作用域。可以通过代表作用域的类的`getAttribute`方法和`setAttribute`方法来访问这些属性。表10-2列举了这些有作用域的对象。

表10-2 有作用域的对象

对 象	类	从何处访问
Web上下文	<code>javax.servlet.ServletContext</code>	在Web上下文里的Web组件（参见10.8节）
会话	<code>javax.servlet.http.HttpSession</code>	处理属于会话的请求的那个Web组件（参见10.9节）
请求	<code>javax.servlet.servlet.ServletRequest</code>	处理请求的Web组件
页面	<code>javax.servlet.jsp.JspContext</code>	创建了这个对象的JSP页面

10.3.2 控制对共享资源的并发访问

对于支持多线程的服务器来说，共享资源可能会被并发访问。除了有作用域的对象属性，共享资源还包括内存中的数据（比如类的实例或变量）以及外部对象（比如文件、数据库连接和网络连接）。

并发访问可能发生在如下情况下：

- ❑ 多个Web组件访问保存在Web上下文里的对象；
- ❑ 多个Web组件访问保存在会话里的对象；
- ❑ 一个Web组件里的多个线程访问实例变量。通常来说，Web容器会为每一个请求创建一个线程。为确保一个servlet实例在同一时刻只处理一个请求，servlet可以实现`SingleThreadMode`接口。如果servlet实现了这个接口，则不会有两个线程同时在servlet的服务方法中执行。Web容器还有两个办法可以实现该目的，其一是只允许以同步方式访问servlet实例，其二是维护一个池（Web组件实例池），使得每一个新的请求都被分配一个空闲的实例。`SingleThreadMode`接口不能避免由于Web组件访问共享资源导致的同步问题，比如静态类变量或者外部对象。

当资源可以被并发访问时，可能会引发数据不一致问题。可以使用线程的同步技术解决这种问题。线程的同步技术参见<http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>，这部分内容在由Sharon Zakhour等所著的*The Java Tutorial: A Short Course on the Basics, Fourth Edition*（Addison-Wesley, 2006）一书中也有讲述。

10.4 创建以及初始化 servlet

在Web应用里，使用`@WebServlet`注解可以定义一个servlet组件。这个注解是类级别的注解，

包含将这个类声明为servlet所需的一些元数据。使用该注解的servlet必须指定至少一个URL模式。在注解里使用urlPatterns属性或者value属性就可以实现对URL模式的声明。所有其他的属性都是可选的，且有着各自的默认值。当注解只有URL模式这一个属性时，使用value属性来设定URL模式；如果还有其他的属性要设定，则使用urlPatterns属性来设定URL模式。

使用@WebServlet注解的类必须继承并扩展javax.servlet.http.HttpServlet类。例如，如下代码片段定义了一个servlet，其URL模式设定为/report：

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;

@WebServlet("/report")
public class MoodServlet extends HttpServlet {
    ...
}
```

在载入并实例化servlet类之后，在传递来自客户端的请求之前，Web容器会初始化servlet。为了定制这个过程，可以覆写Servlet接口的init方法，或指定@WebServlet注解的initParams属性，让servlet可以读取持久化的配置数据、初始化资源以及执行其他只需要做一次的事项。initParams属性包含了一个名为@WebInitParam的注解。如果服务器不能完成初始化过程，servlet就会抛出UnavailableException异常。

10.5 编写 service 方法

servlet提供的服务可以在GenericServlet的service方法中实现，也可以在HttpServlet对象的doMethod（这里的Method可以是Get、Delete、Options、Post、Put或Trace）方法中实现，还可以在任何实现了Servlet接口的类中所定义的协议相关的方法中实现。术语服务方法指的是servlet类中为客户端提供服务的任何方法。

服务方法的通常任务是从请求中提取信息，访问外部资源，然后基于这些信息组织应答信息。对于HTTP servlet，构造应答信息的过程是：

- (1) 从应答对象里中获取输出流；
- (2) 填充应答的头部；
- (3) 将消息体的内容写入到输出流。

应答的头部信息必须在提交应答之前设置。在提交应答后，Web容器会忽略所有设置或添加头部信息的尝试。下面两节讲述如何从请求中提取信息并创建应答信息。

10.5.1 从请求里提取信息

请求里包含了客户端和servlet之间传递的数据。所有的请求都要实现ServletRequest接口，这个接口定义了一些方法，可以访问如下的信息：

- 通常用于在客户端和servlet之间传递信息的参数；
- 值为对象的属性（object-valued attribute），通常用来传递servlet与容器之间的信息，或者在相互协作的servlet间传递信息；

- 通信所用协议的有关信息，以及请求里的客户端和服务器的有关信息；
- 多语言支持相关的信息。

可以从请求里获取输入流，然后手工解析数据。要读取字符型数据用BufferedReader对象，该对象可以使用请求的getReader方法获得。要读取二进制数据，可以使用ServletInputStream对象，该对象可以用getInputStream方法获得。

HTTP servlet接收容器传递的HTTP请求对象，即HttpServletRequest，它包含了请求URL、HTTP头部信息、查询字符串等。HTTP请求URL包含了如下的部分：

```
http://[host]:[port][request-path]?[query-string]
```

其中request-path是由如下部分组成的。

- 上下文路径 由/和servlet Web应用的上下文根拼接而成。
- servlet路径 由/开头，响应请求的组件别名所对应的路径。
- 路径信息 request-path里既不属于上下文路径，也不属于servlet路径的部分。

可以使用getContextPath、getServletPath以及getPathInfo方法（都是由HttpServletRequest接口提供）来访问相应的信息。请求URI使用的编码和路径信息使用的URL编码不一样，除了这一点之外，请求URI总是由上下文路径加上servlet路径，再加上路径信息组成。

查询字符串是由一组参数和对应的值组成的字符串。可以用getParameter方法从请求的查询字符串中提取出独立的参数。有两种方法可以构造查询字符串：

- 查询字符串显式地出现在网页里；
- 当表单以GET的方式提交时，查询字符串就附加在URL的后边。

10.5.2 构造应答信息

应答信息包含了在服务器和客户端之间传递的数据。所有的应答都要实现ServletResponse接口。该接口定义了一些方法，使得开发人员可以完成如下任务。

- 获取输出流并发送数据给客户端。如果发送的是字符型数据，可以使用PrintWriter对象，它是由应答对象的getWriter方法返回的。如果发送的是二进制数据（比如在MIME编码的应答信息里），可以使用ServletOutputStream对象，它是由getOutputStream返回的。如果要发送混合数据，即字符型数据和二进制数据混杂在一起，比如内容类型（Content-Type）为多个部分的应答消息，可以使用ServletOutputStream，其中字符型数据部分要手工处理。
- 指明应答要返回的内容类型（比如text/html），可以调用setContentType(String)方法来实现。该方法必须在提交应答之前调用。内容类型的名称列表可以在IANA（Internet Assigned Numbers Authority）中查询，其网址为<http://www.iana.org/assignments/media-types/>。
- 可以调用setBufferSize(int)方法来指明是否缓冲输出。默认情况下，任何要输出的内容都会立即发送到客户端。缓冲机制使得要输出的内容可以积攒一定的字节数之后再发送，这使servlet有更多的机会设置合适的状态码以及头部信息，或者把请求重定向到另外的Web资源。setBufferSize (int) 方法必须在输出任何内容之前或者提交应答之前调用。

□ 设置多语言支持相关的信息，比如locale以及字符编码。

HTTP应答对象，即`javax.servlet.http.HttpServletResponse`，有如下字段用以代表HTTP的头部信息。

□ 状态码 用来表明不满足请求的原因或者请求已经被重定向。

□ cookie 用来在客户端保存应用的特定信息，有时候cookie可以用来维护一个标识符，以跟踪用户的会话（参见10.9.4节）。

10.6 过滤请求和应答

过滤器是一个对象，可以转换请求（或者应答）的头部信息以及内容。过滤器和Web组件的区别在于，过滤器自身通常不创建应答信息。取而代之的是，过滤器可以应用到任何的Web资源上。因此，过滤器应当独立于Web资源，仅有“过滤”的功能。从而，它可以和各种类型的Web资源组合起来使用。

过滤器有如下的主要功能：

- 查询请求，并作出相应的回应；
- 阻止某些请求和应答的进一步传递；
- 修改请求的头部和数据，实现方式是提供一个定制的请求对象；
- 修改应答的头部和数据，实现方式是提供一个定制的应答对象；
- 与外部资源交互。

过滤器的应用领域包括鉴权、登录、图片转换、数据压缩、加密、字符串分解、XML转换等。

可以配置过滤器链按顺序过滤Web资源。过滤器链可以没有过滤器，也可以包括一个过滤器或者多个过滤器。当包含了组件的Web应用被部署到Web容器里，且因Web容器装载该组件被实例化时，我们可以设定过滤器链。

10.6.1 编程实现过滤器

过滤器API定义在`javax.servlet`包里，相关的接口有`Filter`、`FilterChain`以及`FilterConfig`。实现`Filter`接口就可以定义一个过滤器。

在Web应用里，使用`@WebFilter`注解可以定义一个过滤器。这个注解是一个类级别的注解，包含将这个类声明为过滤器所需的一些元数据。使用该注解的过滤器必须指定至少一个URL模式。在注解里使用`urlPatterns`属性或者`value`属性就可以声明URL模式。所有其他的属性都是可选的，有着各自的默认值。如果注解里唯一的属性是URL模式，则使用`value`属性设定URL模式。如果注解还声明了其他属性，则使用`urlPatterns`属性来设定URL模式。

使用了`@WebFilter`注解的类，必须实现`javax.servlet.Filter`接口。

为过滤器添加配置数据，需要指定`@WebFilter`注解的`initParams`属性，该属性包含了`@WebInitParam`注解。如下的代码片段定义了一个过滤器，并指定了初始化参数`initParams`：

```
import javax.servlet.Filter;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;

@WebFilter(filterName = "TimeOfDayFilter",
urlPatterns = {"/"},
initParams = {
    @WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
    ...
}
```

Filter接口里最重要的方法是doFilter，传入参数包括请求对象、应答对象以及过滤器链对象。这个方法可以执行如下的事项。

- ❑ 检查请求的头部。
- ❑ 如果希望过滤器能修改请求的头部或者数据，则定制请求对象。
- ❑ 如果希望过滤器能修改应答的头部或者数据，则定制应答对象。
- ❑ 调用过滤器链中的下一个实体。如果当前的过滤器是过滤器链里的最后一个过滤器（该过滤器链以目标Web组件或者静态资源结束），则下一个实体就是位于最后的Web资源，否则就是WAR里配置好的下一个过滤器。过滤器调用下一个实体，是通过调用过滤器链对象的doFilter方法实现的，此时需要传递的参数为请求对象和应答对象，或是对二者的封装（由过滤器链创建）。如果过滤器选择阻止请求，则需要放弃调用下一个过滤器。在这种情况下，过滤器负责填充应答内容。
- ❑ 调用过滤器链中下一个过滤器后，检查应答头部。
- ❑ 处理过程中如果发生了错误，则抛出异常。

除了doFilter方法，过滤器必须实现init方法和destroy方法。init方法会在初始化过滤器时被容器调用。如果希望传递初始化参数给过滤器，可以从init方法的参数FilterConfig对象里提取出需要的参数。

10.6.2 通过编程定制请求和应答

10

对过滤器来说，有很多种方法可以修改请求和应答。比如说，过滤器可以为请求添加属性，或者为应答添加数据。

过滤器要想修改应答内容，必须在应答被返回给客户端之前实施拦截。为了实现这个功能，可以传递一个替代流（stand-in stream）给生成应答内容的servlet。替代流阻止了servlet在完成时要关闭最初应答流的行为，从而允许过滤器修改servlet的应答内容。

为了传递这个替代流给servlet，过滤器需要创建一个应答的包装类，该包装类需要覆写getWriter或者getOutputStream方法以返回这个替代流。这个应答的包装类以参数的形式传递给过滤器链的doFilter方法。这个包装类方法默认被定制的请求或应答对象所调用。

为了覆写请求方法，可以将请求封装到一个对象里，用这个对象继承并扩展ServletRequestWrapper或者HttpServletRequestWrapper。为了覆写应答方法，需要把应答封装到

一个对象里，用该对象继承并扩展ServletResponseWrapper或者HttpServletResponseWrapper。

10.6.3 设定过滤器映射

Web容器使用过滤器映射来决定如何对Web资源应用过滤器。过滤器映射通过名称把过滤器匹配到一个Web组件，或者通过URL模式匹配到一个Web资源。调用过滤器的顺序是过滤器出现在WAR文件的过滤器映射列表里的顺序。为一个WAR文件设定过滤器映射列表，可通过编辑部署描述文件实现。对文件的修改可以通过NetBeans IDE完成，也可以通过手工编辑XML文件完成。

如果要记录Web应用收到的每一个请求，可以映射网站访问计数过滤器到URL模式/*。

可以映射一个或多个Web资源到同一个过滤器上，也可以为一个Web资源映射多个过滤器。如图10-1所示，过滤器F1映射到servlet S1、S2以及S3，而过滤器F2映射到servlet S2，过滤器F3则映射到servlet S1和S2。

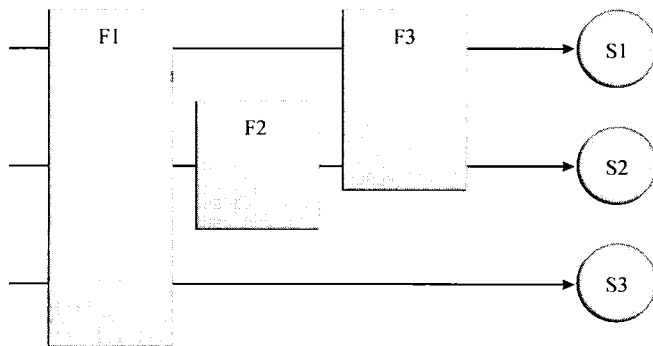


图10-1 过滤器到servlet的映射

过滤器链是一个对象，它以参数的方式传递给过滤器的doFilter方法。过滤器链是由过滤器映射间接组成的。过滤器链里过滤器的顺序和过滤器映射在Web应用部署描述文件里的顺序是一致的。

当过滤器映射到servlet S1，Web容器调用F1的doFilter方法。S1的过滤器链里每一个过滤器的doFilter方法被过滤器链中“前一个过滤器”调用，调用方式是chain.doFilter。因为S1的过滤器链包含了过滤器F1和F3，F1调用chain.doFilter会调用F3的doFilter方法。当F3的doFilter调用完毕，程序的控制将返回到F1的doFilter方法。

▼ 用NetBeans IDE设定过滤器映射

- (1) 展开Project窗格中应用的项目结点。
- (2) 展开项目结点下的Web Pages以及WEB-INF结点。
- (3) 双击web.xml。

- (4) 单击编辑器窗格顶部的Filters。
- (5) 展开编辑器窗格的Servlet Filters结点。
- (6) 单击Add Filter Element，通过名称或者URL模式把过滤器映射到Web资源。
- (7) 在Add Servlet Filter对话框中，在Filter Name处输入过滤器的名称。
- (8) 单击Browser以定位要映射的servlet类的位置。
- 使用通配符，以便可以把过滤器应用到多个servlet。
- (9) 单击OK按钮。
- (10) 为了限定如何将过滤器应用于请求之上，可以执行如下步骤。
 - (a) 展开Filter Mapping 结点。
 - (b) 从过滤器列表里选出过滤器。
 - (c) 单击Add。
 - (d) 在Add Filter Mapping对话框中，选择如下分发器（dispatcher）中的一个：
 - ☐ REQUEST——仅当请求直接来自于客户端时；
 - ☐ ASYNC——仅当收到的客户端请求是异步请求时；
 - ☐ FORWARD——仅当请求被重定向到某个组件时（参见10.7.2节）；
 - ☐ INCLUDE——仅当请求由某个被包含的组件处理时（参见10.7.1节）；
 - ☐ ERROR——仅当处理请求出错的时候，才使用错误页面处理机制（参见10.2.2节）。

通过组合多个分发器类型，可以把过滤器应用到上述所说的各种场合的组合中。如果没有指定类型，默认选项为REQUEST。

10.7 调用其他 Web 资源

Web组件可以直接或者间接调用其他Web资源。Web组件间接调用其他Web资源，是指把指向其他Web组件的URL作为应答中的内容，返回给客户端。Web组件直接调用其他资源有两种方式，要么包含其他资源的内容，要么把请求重定向到其他资源。

为了调用运行Web组件的服务器上可用的资源，必须先取到RequestDispatcher对象，对应的方法是getRequestDispatcher("URL")。可以从请求里，也可以从Web上下文中得到RequestDispatcher对象，然而这两种方式略有不同。前一种方法以参数的方式获得请求资源的路径，该路径可以是相对路径（也就是说，不是以/开头的路径），而从Web上下文中获取，则需要绝对路径。如果资源不可用或者服务器没有为那类资源实现RequestDispatcher对象，getRequestDispatcher会返回null。servlet需要具备处理这种情况的能力。

10.7.1 在应答里包含其他资源

把另外的Web资源（比如横幅广告的内容或者版权信息）包含到Web组件返回的应答里，有时候很有用。要包含另外的资源，调用RequestDispatcher对象的include方法即可：

```
include(request, response);
```

如果是静态资源，include方法会开启服务器端的包含。如果资源是Web组件，调用该方法的作用是发送请求到所包含的那个Web容器，执行该Web组件，然后把执行结果包含在应答内容里返回给最初请求的客户端。一个被包含的Web组件可以访问请求对象，但是对应答对象可执行的操作受限：

- ❑ 可以修改应答的主体信息，并提交应答信息；
- ❑ 不能设置应答的头部信息，也不能调用任何影响应答头部信息的方法（比如setCookie）。

10.7.2 转交控制权给其他Web组件

在有些应用里，可能需要某个Web组件对请求做些预处理，然后让其他的组件生成应答内容。例如，可能想要部分处理请求，然后把请求转交给另外一个组件，这取决于请求究竟想实现什么。

要转交控制权给其他的Web组件，可以调用RequestDispatcher的forward方法。当请求被转发时，请求的URL设成要转发的页面的路径。初始的URI以及它的各组成部分保存为请求的属性：

```
javax.servlet.forward.[request-uri|context-path|servlet-path|path-info|query-string]
```

forward方法可以把请求传递给下一个资源。如果已经访问过servlet里的ServletOutputStream对象或者PrintWriter对象，那就不能再使用这个方法了；如果再次使用该方法，则会抛出一个IllegalStateException异常。

10.8 访问 Web 上下文

Web 组件执行时的上下文，是一个实现了ServletContext接口的对象。可以通过getServletContext方法取得Web上下文对象。Web上下文对象提供了方法以供访问：

- ❑ 初始化参数；
- ❑ 与Web上下文相关的资源；
- ❑ 值为对象的属性；
- ❑ 日志功能。

计数器的访问方法是同步的，可以避免servlet里正在并发运行的竞争操作。过滤器可以使用上下文的方法getAttribute取到计数器对象，这个递增的数字也记录在日志里。

10.9 维护客户端状态

许多应用需要客户端发来一系列相互关联的请求。例如，Web应用可以在多个请求之间共享一个购物车。基于Web的应用有义务维护这样的状态，即会话（session），因为HTTP是无状态的。为了支持那些需要维护状态的应用，Java Servlet技术提供了API，用来管理会话，实现了几种会话机制。

10.9.1 访问会话

会话用`HttpSession`对象来表示，可以通过调用请求对象的`getSession`方法获得会话。这个方法返回关联到这个请求的当前会话；如果请求还没有对应的会话，则创建一个。

10.9.2 把对象关联到会话

可以把值为对象的属性通过名字关联到会话上。任何Web组件都可以访问这样的属性，只要这个Web组件与其属于同一个Web上下文，而且正在处理着属于同一个会话的请求。

应用可以通知Web上下文和servlet生命周期事件的会话监听器对象（参见10.2.1节）。对于会话有关的特定事件，可以发通知给特定对象，如下。

- ❑ 当对象添加到会话或者从会话里删除的时候。要想收到这样的通知信息，对象必须实现`javax.servlet.http.HttpSessionBindingListener`接口。
- ❑ 当某对象关联的会话将要钝化或者将要激活的时候。当在虚拟机之间移动，或保存到持久化存储或者从持久化存储载入的时候，会话会被钝化或者激活。要想收到这样的通知，对象必须实现`javax.servlet.http.HttpSessionActivationListener`接口。

10.9.3 会话管理

因为HTTP客户端没有办法告诉服务器端自己不再需要会话，所以每个会话都有一个超时机制，超时后会话使用的资源可被收回。超时周期可以使用会话的`getMaxInactiveInterval`方法以及`setMaxInactiveInterval`方法获取和设置。

- ❑ 为了确保一个活跃的会话不超时，应当通过调用服务方法定期地访问会话，因为这将重置会话的存活周期（time-to-live）计数器。
- ❑ 当客户端的某个特定交互过程结束时，可以使用会话的`invalidate`方法使会话在服务器端失效以及删除包含在会话里的数据。

▼ 使用NetBeans IDE设置超时期限

使用NetBeans IDE在部署描述文件里设置超时期限，可以遵照如下步骤。

- (1) 确保项目已经打开。
 - (2) 在Projects窗格里展开项目结点。
 - (3) 展开Web Pages以及WEB-INF结点。
 - (4) 双击web.xml。
 - (5) 单击编辑器上部的General标签。
 - (6) 在Session Timeout 字段处输入一个整数值。
- 这个整数值代表了以分钟为单位的超时时间间隔。

10.9.4 会话追踪

Web容器可以使用几种机制把用户和会话关联起来，所有这些机制都有一个共同点，即在客户端和服务端之间传递一个标识符。这个标识符可以通过客户端cookie来维护，或者在Web组件返回给客户端的每一个URL里都包含这个标识符。

如果应用使用会话对象，必须保证打开会话追踪功能。如果客户端关闭了cookie功能，则允许应用重写URL。要使用URL重写功能，需要对servlet返回的所有URL调用应答对象的`encodeURL(URL)`方法。只有在关掉cookie功能的情况下，这个方法才会在URL里包含会话ID，否则就返回原来的URL。

10.10 结束 servlet

servlet容器有时要决定servlet是否应当从服务中删去，如当容器想要回收内存资源或者被关闭的时候。在这种情况下，容器调用Servlet接口的`destroy`方法。在这个方法里，可以释放servlet使用的任何资源以及保存任何持久化的状态。这个`destroy`方法还可以释放在`init`方法里创建的数据库对象。

应当在servlet的服务方法全部执行完毕之后再删除servlet。服务器要尽力保证这一点，需要仅在所有的服务请求都返回后，或者超过服务器所设定的一个妥善关闭时限之后才调用`destroy`方法。如果servlet有长时操作，时间超过了妥善关闭时限，即使`destroy`方法已经被调用了，操作也可以一直运行下去。必须确保任何正在处理客户端请求的线程都要正常完成。

本节接下来的内容会讲述如何实现：

- 追踪有多少个线程正在运行service方法；
- 彻底关闭应用，让`destroy`方法通知那些长时间运行的线程即将关闭，并等待线程完成；
- 定期轮询那些长时方法，验证它们是否关闭。如果必要的话，停止工作，清理并返回。

10.10.1 追踪服务请求

为了追踪服务请求，可以在servlet类里增加一个字段，作为目前正在运行的服务方法的计数器。对该计数器的访问需要支持同步方法，用于增加、减少以及返回其值。

```
public class ShutdownExample extends HttpServlet {
    private int serviceCounter = 0;
    ...
    // Access methods for serviceCounter
    protected synchronized void enteringServiceMethod() {
        serviceCounter++;
    }
    protected synchronized void leavingServiceMethod() {
        serviceCounter--;
    }
    protected synchronized int numServices() {
        return serviceCounter;
    }
}
```


每一次进入service方法的时候,应当增加服务计数器的值。每一次service方法返回的时候,则应减少计数器的值。HttpServlet的子类一般不需要覆写service方法,这个算是少有的例外。新的方法需要调用super.service,用于保证父类service方法的功能正常实现。

```
protected void service(HttpServletRequest req,
                        HttpServletResponse resp)
    throws ServletException,IOException {
    enteringServiceMethod();
    try {
        super.service(req, resp);
    } finally {
        leavingServiceMethod();
    }
}
```

10.10.2 将关闭事件通知方法

为保证应用能彻底关闭,destroy方法需等待所有的服务请求都完成之后再释放共享资源。可以将这个过程分解为两个步骤,第一是要检查服务方法调用计数器,第二是要通知长时运行的方法应用要关闭了。为实现这个目标,需要再添加一个字段,表明当前应用是否处于“正在关闭中”的状态,这个字段要有惯常的读取和设置方法:

```
public class ShutdownExample extends HttpServlet {
    private boolean shuttingDown;
    ...
    //Access methods for shuttingDown
    protected synchronized void setShuttingDown(boolean flag) {
        shuttingDown = flag;
    }
    protected synchronized boolean isShuttingDown() {
        return shuttingDown;
    }
}
```

下面是一个destroy方法的示例,它使用这些新引入的字段,以彻底关闭应用:

```
public void destroy() {
    /* Check to see whether there are still service methods */
    /* running, and if there are, tell them to stop. */
    if (numServices() > 0) {
        setShuttingDown(true);
    }

    /* Wait for the service methods to stop. */
    while(numServices() > 0) {
        try {
            Thread.sleep(interval);
        } catch (InterruptedException e) {
        }
    }
}
```

10.10.3 妥善处理长时方法

要彻底关闭应用,最后一步是确保妥善处理所有长时方法。可能长时运行的方法应当检查接

收终止消息的字段，且在必要的情况下中断自己的工作。

```
public void doPost(...) {
    ...
    for(i = 0; ((i < lotsOfStuffToDo) &&
        !isShuttingDown()); i++) {
        try {
            partOfLongRunningOperation(i);
        } catch (InterruptedException e) {
            ...
        }
    }
}
```

10.11 mood 示例应用

Mood示例应用位于`tut-install/examples/web/mood`。这个应用用来显示Duke一天之中不同时刻的心情。这个例子展示了如何使用`@WebServlet`、`@WebFilter`以及`@WebListener`注解分别创建servlet、过滤器以及监听器，进而讲解了如何开发一个简单的应用。

10.11.1 mood示例应用里的组件

mood这个示例应用由3个组件组成，即`mood.web.MoodServlet`、`mood.web.TimeOfDayFilter`以及`mood.web.SimpleServletListener`。

`MoodServlet`是应用的表现层，它以图片的方式显示Duke的心情，具体心情因时间的不同而不同。`@WebServlet`注解则指定了URL模式：

```
@WebServlet("/report")
public class MoodServlet extends HttpServlet {
    ...
```

`TimeOfDayFilter`设置了初始化参数，以指明Duke的初始心情为`awake`：

```
@WebFilter(filterName = "TimeOfDayFilter",
    urlPatterns = {"/*"},
    initParams = {
        @WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
    ...
```

过滤器调用了`doFilter`方法，该方法包含了一个`switch`语句，基于当前的时间来确定Duke的心情。

`SimpleServletListener`记录了servlet生命周期中的变化，日志的详细内容会记录在服务器的日志文件里。

10.11.2 构建、打包、部署以及运行mood示例

可以使用NetBeans IDE或者Ant来构建、打包、部署以及运行mood示例。

▼使用NetBeans IDE构建、打包、部署以及运行mood示例

- (1) 选择File→Open Project。
 - (2) 在Open Project对话框里，定位到*tut-install/examples/web*。
 - (3) 选择mood文件夹。
 - (4) 勾选 Open as Main Project复选框。
 - (5) 单击Open Project。
 - (6) 在Projects标签中，右键单击mood项目，然后选择Build。
 - (7) 右键单击项目并选择Deploy。
 - (8) 打开一个Web浏览器，输入网址<http://localhost:8080/mood/report>。
- 这一URL指定了上下文根，紧跟着为servlet指定的URL模式。

此时出现一个标题是“Servlet MoodServlet at /mood”的网页，页面里显示一个表示Duke心情的字符串，以及一个表示心情的示意图片。

▼使用Ant构建、打包、部署以及运行mood示例

- (1) 打开一个终端窗口，切换目录到*tut-install/examples/web/mood*。
 - (2) 输入并执行如下命令ant。
- 此命令会构建WAR文件，并把它复制到*tut-install/examples/web/mood/dist*目录里。
- (3) 输入ant deploy。
- 忽略命令执行输出信息里的URL。
- (4) 打开一个Web浏览器，输入网址<http://localhost:8080/mood/report>。
- 这一URL指定了上下文根，紧跟着为servlet指定的URL模式。

出现一个标题是“Servlet MoodServlet at /mood”的网页，页面里显示一个表示Duke心情的文本字符串，以及一个表示心情的示意图片。

10.12 有关 Java Servlet 技术的更多信息

有关Java Servlet技术的更多信息，可以参考以下网站。

□ Java Servlet 3.0规范：

<http://jcp.org/en/jsr/detail?id=315>

□ Java Servlet官方网站：

<http://www.oracle.com/technetwork/java/index-jsp-135475.html>

Part 3

第三部分

Web 服务

第三部分主要讲述 Web 服务的相关内容，包含以下三章。

本 部 分 内 容

- 第 11 章 Web 服务简介
- 第 12 章 用 JAX-WS 构建 Web 服务
- 第 13 章 用 JAX-RS 构建 REST 式 Web 服务



本书第三部分主要讨论Java EE 6里的Web服务技术，这些技术在本书里是指JAX-WS（Java API for XML Web Service）和JAX-RS（Java API for RESTful Web Services）。

本章内容

- 什么是Web服务
- Web服务的类型
- Web服务类型的选用

11.1 什么是 Web 服务

Web服务是基于客户端-服务器模型的应用，在万维网上基于HTTP协议进行通信。正如W3C（World Wide Web Consortium）所描述的那样，Web服务为运行在不同平台和框架之上的软件提供了互操作的标准方式。Web服务有着良好的互操作性以及可扩展性。同时，因为Web服务的信息描述是用XML表示的，所以能够用机器来自动处理。Web服务能够以松耦合的方式组合起来实现更为复杂的功能。提供单一功能的简单应用互相配合，最终可以提供实现复杂功能的综合服务。

11.2 Web 服务的类型

从概念层面上来说，服务是指通过网络访问的端点提供的软件组件。服务的使用者以及提供者以消息交换的方式，实现请求的调用以及应答的返回。消息采用自包含文档的形式，因此无需对消息的接收者做太多技术上的假定。

从技术层面上来说，Web服务可以用多种方式实现。本书这个部分主要讨论两种Web服务类型，即“重量级”的Web服务和REST式Web服务。

11.2.1 “重量级”的Web服务

在Java EE 6里，JAX-WS提供了重量级Web服务的功能（将在第12章里讲述）。“重量级”Web服务采用了遵照SOAP（Simple Object Access Protocol，简单对象访问协议）标准的XML消息格

式 (XML 定义了消息结构和消息格式)。这样的系统通常都有描述服务所提供操作的信息, 该描述信息以 WSDL (Web Services Description Language, Web 服务描述语言) 格式编写, 可以被机器读取。WSDL 使用 XML 从语法上定义接口。

SOAP 消息格式和 WSDL 接口定义语言在业界应用广泛。有很多开发工具 (比如 NetBeans IDE) 可以降低开发 Web 服务应用程序的复杂性。

基于 SOAP 的设计必须包含以下要素。

- ❑ 必须建立一份正式的约定来描述 Web 服务所能提供的接口。WSDL 可以用来描述约定的细节, 可能包含的内容有消息、操作、绑定关系以及 Web 服务的位置。我们也可以在 JAX-WS 服务里处理 SOAP 消息而无需发布 WSDL。
- ❑ 架构必须满足复杂的非功能性需求。很多 Web 服务的规范阐述了这样的需求, 并建立了通用词汇, 例如事务 (transaction)、安全性 (security)、寻址 (addressing)、信任 (trust) 以及协调 (coordination) 等。
- ❑ 架构能够处理异步的请求和调用。这种情况下, 可以使用由标准提供的基础功能, 比如 WSRM (Web Services Reliable Messaging, Web 服务可靠消息传递) 以及相关的 API (比如 JAX-WS) 等现成的技术, 来实现对客户端异步调用的支持。

11.2.2 REST 式 Web 服务

在 Java EE 6 里, JAX-RS 提供了开发 REST (Representational State Transfer, 表述性状态转移) 式 Web 服务的功能。REST 非常适合基本的、点对点 (ad hoc) 集成的场景。REST 式 Web 服务经常结合 HTTP 来使用, 这比集成基于 SOAP 的服务要好一些, 因为不再需要 XML 格式的消息, 也不再需要 WSDL 来描述服务的 API 定义。

Jersey 项目是一个现成的 JAX-RS 参考实现。Jersey 的实现支持定义在 JAX-RS 里的注解, 使得开发人员可以很容易地使用 Java 语言和 JVM 构建 REST 式 Web 服务。

因为 REST 式 Web 服务使用广为人知的 W3C/IETF 标准 (HTTP、XML、URI 以及 MIME 等), 且有着轻量级的特点, 只需用很少的工具就可以构建服务, 所以开发 REST 式 Web 服务代价低廉, 使用起来几乎没有什么门槛。我们可以使用某种开发工具 (比如 NetBeans IDE) 进一步降低开发 REST 式 Web 服务的复杂性。

REST 风格的设计比较适合以下情形。

- ❑ Web 服务是完全无状态的。可以通过一个有效的方法测试这个场景, 即测试当 Web 服务器重启后, 原先的交互是否仍然存在。
- ❑ 缓存机制可以提升性能。如果 Web 服务返回的数据不是动态产生的, 且可以缓存起来, 那么 Web 服务器加上其他中间件提供的缓存机制, 能够改善系统性能。然而, 开发人员必须小心, 因为对于大多数 Web 服务器来说, 这样的缓存机制仅对 HTTP GET 操作有效。
- ❑ 服务创建者以及服务使用者对于应用的上下文, 有着相同的理解。因为没有正式的途径来描述 Web 服务的接口, 双方必须就交换数据的格式描述以及处理方式等达成一致意见。

在现实世界里,大多数商业应用在提供REST式Web服务时,同时也发布所谓的增值工具包,以开发人员可用的常用编程语言来描述接口。

- 带宽非常重要,需要有所限制。REST对于能力有限的设备特别有用,比如说PDA和手机。对于这类设备来说,XML有效负载之外的头部信息以及SOAP元素附加层的开销都需要严格限制。
- 采用了REST风格,开发人员可以很容易地发布Web服务或者整合Web服务到现存的网站。开发人员可以使用JAX-RS、AJAX这样的技术以及DWR (Direct Web Remoting) 这样的开发工具,在自己的Web应用里使用Web服务。无需从头做起,服务就可以通过XML发布出去,供HTML网页使用,无需大刀阔斧地重构网站架构。既有的开发人员的工作可以更有效率,因为他们只是用自己熟悉的技术添加新的东西,而不是用新技术重新开发。

REST式Web服务将在第13章详细介绍。第13章还要简单介绍如何用NetBeans IDE和Maven项目管理工具创建一个REST式Web服务的框架。

11.3 Web 服务类型的选用

一般而言,REST式Web服务适用于Web应用的整合,而在企业应用的场合下,则使用“重量级”的Web服务来整合对QoS有着更高要求的应用。

- JAX-WS 通常适用于对服务有着较高QoS要求的企业计算。与JAX-RS相比而言,JAX-WS很容易支持WS-*系列的协议。这些协议提供了安全性、可靠性以及其他方面的标准,可以与其他遵循客户端和服务端模型的WS-*协议进行互操作。
- JAX-RS 简化了编写Web应用(有着部分或完全的REST风格)的难度,并满足Web应用所期望的特性,比如松耦合(服务器端升级更为简单,无须中断客户端的服务)、伸缩性(规模由小变大)以及架构上的简化(使用套装组件,比如代理服务器和HTTP路由器)。开发人员更倾向于在Web应用里使用JAX-RS,因为各种类型的客户端调用REST式Web服务都很简单,与此同时服务器端升级和扩展也比较方便。客户端可以选择使用Web服务的部分或全部功能,并与其他基于Web的服务整合到一起。

注意 关于Web服务的类型,有篇文章进行了深度分析,这篇文章名为“RESTful Web Services vs. ‘Big’ Web Services : Making the Right Architectural Decision”,由Cesare Pautasso、Olaf Zimmermann和Frank Leymann合著,发表在2008年举行的第17届世界WWW国际会议上,文章的网址为<http://www2008.org/papers/pdf/p805-pautassoA.pdf>,对应页码为805~814。

Java提供基于XML的Web Service API (JAX-WS), 实现Web服务与客户端间的通信。利用JAX-WS技术, 开发人员可以编写面向消息及RPC (Remote Procedure Call, 远程过程调用) 的Web服务。

在JAX-WS中, Web服务的调用依赖XML协议, 如SOAP。SOAP规范定义了服务的封装结构、编码规则及Web服务调用与应答的规则。调用与应答通过HTTP协议, 以SOAP消息 (XML文件) 的形式在服务器端与客户端间传输。

尽管SOAP消息比较复杂, 但JAX-WS提供的API将这种复杂性进行了有效封装。在服务器端, 应用开发人员通过在Java语言的接口中定义方法声明Web服务所能提供的操作。同时, 开发人员还需要编写一些类, 具体实现这些方法。客户端程序的开发也很简单。客户端创建一个代理 (proxy), 并通过这个代理 (代表远端服务的本地对象) 实现对方法的调用。借助JAX-WS, 开发人员无需创建或解析SOAP消息。JAX-WS负责在系统运行时, 实现SOAP消息与调用的方法之间, SOAP消息与请求的应答之间的转换。

JAX-WS为客户端和Web服务带来的好处是充分利用了Java编程语言的平台无关性。除此之外, JAX-WS对具体使用的平台不做限定: JAX-WS客户端可以访问非Java平台运行的Web服务, 反之亦然。这种灵活性源于JAX-WS使用的是W3C制定的通用技术, 如HTTP、SOAP和WSDL。通过声明一组处理消息的端点操作, WSDL以XML的方式定义了对于服务的描述。

注意 JAX-WS示例程序中的一些文件依赖安装GlassFish服务器时指定的端口。本书中的示例假定服务器运行于默认的端口上, 即8080。这些示例无法运行在其他端口上。

本章内容

- ☐ 用JAX-WS开发简单的Web服务和客户端
- ☐ JAX-WS支持的类型
- ☐ Web服务的互操作性与JAX-WS
- ☐ 有关JAX-WS的更多信息

12.1 用JAX-WS开发简单的Web服务和客户端

本节将展示如何构建并部署一个简单的Web服务及两个客户端，分别是应用程序客户端及Web客户端。服务的源代码位于目录`tut-install/examples/jaxws/helloservice/`。客户端源代码位于目录`tut-install/examples/jaxws/appclient/`和`tut-install/examples/jaxws/webclient/`。

图12-1展示了JAX-WS技术如何管理Web服务与客户端间的通信。

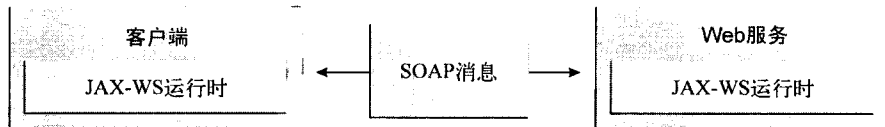


图12-1 基于JAX-WS的Web服务与客户端间的通信

开发基于JAX-WS的Web服务，首先需要开发一个用`javax.jws.WebService`注解的Java类。`@WebService`注解将类声明为Web服务的端点。

服务端点接口（service endpoint interface）和服务端点实现（service endpoint implementation，简称SEI）分别对应一种Java接口和Java类，而接口和类声明了一组客户端可以从服务中调用的方法。构建JAX-WS端点时不一定需要有接口。此时，Web服务的实现类以隐式的方式定义SEI。

可以在实现类中为`@WebService`注解添加`endpointInterface`元素，从而以显式的方式指定接口。此时，必须提供一个接口，且其中定义的公有方法在端点实现类中可用。

创建一个Web服务及客户端的基本步骤如下。

- (1) 编写实现类。
- (2) 编译实现类。
- (3) 将文件打包至WAR中。
- (4) 部署WAR文件。用于与客户端通信的Web服务实例，在将WAR部署至GlassFish服务器的过程中生成。
- (5) 编写客户端类。
- (6) 用Ant中的`wsimport`任务生成并编译连接服务的Web服务实例。
- (7) 编译客户端类。
- (8) 运行客户端。

如果使用NetBeans IDE创建服务和客户端，IDE将完成`wsimport`的工作。

随后的内容将对上述步骤进行更加详尽地阐述。

12.1.1 对JAX-WS端点的要求

JAX-WS端点必须满足如下要求。

- 实现类必须通过`javax.jws.WebService`或`javax.jws.WebServiceProvider`注解进行声明。
- 实现类可以通过`@WebService`注解中的`endpointInterface`元素显式地引用SEI，但不做强制

要求。如果endpointInterface没在@WebService中指定,则将为实现类隐式地定义一个SEI。

- 实现类的业务方法必须是公有类型且不能声明为static或final。
- 暴露给Web服务客户端的业务方法必须使用javax.jws.WebMethod来注解。
- 暴露给Web服务客户端的业务方法只能使用与JAXB兼容的参数及返回类型。更多关于JAXB默认数据类型的信息,参见<http://docs.oracle.com/javaee/5/tutorial/doc/bnazq.html#bnazs>。
- 实现类不能声明为final或abstract。
- 实现类必须有一个默认的公有构造方法。
- 实现类不能定义finalize方法。
- 实现类中的生命周期事件回调方法可以使用 javax.annotation.PostConstruct 或 javax.annotation.PreDestroy进行注解。在实现类响应Web服务客户端的请求之前,容器调用使用@PostConstruct注解的方法。在端点操作结束前,容器调用使用@PreDestroy注解的方法。

12.1.2 编写服务端点实现类

在本例中,实现类Hello通过@WebService注解成为一个Web服务端点。Hello类通过@WebMethod注解声明了唯一一个方法sayHello,并使其对Web服务客户端可见。sayHello方法返回给客户端一句问候语。问候语由方法调用时传递的姓名参数构造而成。实现类必须定义一个默认的且无参数的公有构造方法。

```
package helloservice.endpoint;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Hello {
    private String message = new String("Hello, ");

    public void Hello() {
    }

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

12.1.3 构建、打包及部署服务

可以使用NetBeans IDE或Ant构建、打包及部署helloservice应用。

▼使用NetBeans IDE构建、打包及部署服务

- (1) 在NetBeans IDE中选择File → Open Project。
- (2) 在Open Project对话框中,定位目录到tut-install/examples/jaxws。

- (3) 选择helloservice目录。
- (4) 勾选Open as Main Project复选框。
- (5) 单击Open Project。
- (6) 在Projects标签中, 右键单击helloservice项目, 选择Deploy。

这个命令将构建应用, 并将应用打包至helloservice.war中, 该文件位于目录*tut-install/examples/jaxws/helloservice/dist/*下。该命令还将这个WAR文件部署至GlassFish服务器上。

后续操作 在浏览器中访问URL <http://localhost:8080/helloservice/HelloService?wsdl>, 以查阅所部署服务的WSDL文件。现在, 我们已具备条件创建访问服务的客户端。

▼使用Ant构建、打包及部署服务

- (1) 在终端窗口中, 切换目录到*tut-install/examples/jaxws/helloservice/*。
- (2) 键入下面的命令:

```
ant
```

这个命令调用default目标, 构建并打包应用至WAR文件helloservice.war中, 这一文件位于dist目录下。

- (3) 确保GlassFish服务器已启动。
- (4) 键入如下命令:

```
ant deploy
```

后续操作 在浏览器中访问URL <http://localhost:8080/helloservice/HelloService?wsdl>, 以查阅所部署服务的WSDL文件。现在, 我们已具备条件创建访问服务的客户端。

12.1.4 测试Web服务端点中的方法

GlassFish服务器支持对Web服务端点中的方法进行测试。

▼在无客户端的情况下测试服务

遵循如下步骤, 测试HelloService中的sayHello方法。

- (1) 在Web浏览器中键入如下URL, 打开Web服务的测试界面:
<http://localhost:8080/helloservice/HelloService?Tester>
- (2) 在Methods下键入一个名字, 将其作为传递给sayHello方法的参数。
- (3) 单击sayHello按钮。

该步骤将引导用户进入sayHello的Method invocation页面。

在Method returned下, 用户将看到来自服务端点的应答。

12.1.5 简单的JAX-WS应用客户端

HelloAppClient是一个单机运行的应用客户端，可以访问HelloService中的sayHello方法。客户端调用通过一个端口和一个本地对象（作为远端服务的代理），实现对方方法的调用。这一端口是在开发阶段通过wsimport任务创建的，后者通过WSDL文件产生符合JAX-WS规范的跨平台发布信息。

1. 编写应用客户端

当通过端口调用远端方法时，客户端执行如下步骤。

(1) 使用Web服务在部署时生成的helloservice.endpoint.HelloService类,该类代表了WSDL文件的URI中定义的服务:

```
import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
        "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private static HelloService service;
```

(2) 通过对服务调用getHelloPort方法获取服务代理，也称为端口。

```
helloservice.endpoint.Hello port = service.getHelloPort();
```

这一端口实现了服务中定义的SEI。

(3) 调用端口的sayHello方法，将一个字符串传递给服务:

```
return port.sayHello(arg0);
```

下面是HelloAppClient的完整代码，位于如下目录中:

```
tut-install/examples/jaxws/appclient/src/appclient/
package appclient;

import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
        "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private static HelloService service;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println(sayHello("world"));
    }

    private static String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port = service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```

2. 构建、打包、部署及运行应用客户端

可以使用NetBeans IDE或Ant构建、打包、部署及运行appclient应用。为了构建客户端，必须有已经部署好的helloservice服务，参见12.1.3节。

▼使用NetBeans IDE 构建、打包、部署及运行应用客户端

- (1) 在NetBeans IDE中选择File → Open Project。
- (2) 在Open Project对话框中，定位到*tut-install/examples/jaxws*。
- (3) 选择appclient目录。
- (4) 勾选Open as Main Project复选框。
- (5) 单击Open Project。
- (6) 在Projects标签中，右键单击appclient项目，选择Run。
- (7) 可以在输出窗口中查看应用客户端的输出结果。

▼使用Ant构建、打包、部署及运行应用客户端

- (1) 在终端窗口中，进入目录*tut-install/examples/jaxws/appclient*。
- (2) 键入下面的命令：

```
ant
```

这个命令调用default目标来运行wsimport任务，构建并打包应用至JAR文件appclient.jar中，这一文件位于dist目录中。

- (3) 运行客户端，键入如下命令：

```
ant run
```

12.1.6 简单的JAX-WS Web客户端

HelloServlet是一个servlet，如同Java客户端一样，调用Web服务的sayHello方法。如同应用客户端一样，它也通过端口实现调用。

1. 编写servlet

为了调用端口上的方法，客户端需要执行如下步骤。

- (1) 导入HelloService端点及WebServiceRef注解：

```
import helloservice.endpoint.HelloService;
...
import javax.xml.ws.WebServiceRef;
```

- (2) 通过指定WSDL的位置，定义对Web服务的引用：

```
@WebServiceRef(wsdlLocation =
    "WEB-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
```

- (3) 声明Web服务，并定义一个私有方法，该方法通过端口调用sayHello方法：

```
private HelloService service;
...
```

```
private String sayHello(java.lang.String arg0) {
    helloservice.endpoint.Hello port = service.getHelloPort();
    return port.sayHello(arg0);
}
```

(4) 在servlet中，调用这个私有方法：

```
out.println("<p>" + sayHello("world") + "</p>");
```

下面的是HelloServlet代码中的主体部分。完整代码位于目录*tut-install/examples/jaxws/src/java/webclient*。

```
package webclient;

import helloservice.endpoint.HelloService;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;

@WebServlet(name="HelloServlet", urlPatterns={"/HelloServlet"})
public class HelloServlet extends HttpServlet {
    @WebServiceRef(wsdlLocation =
        "WEB-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private HelloService service;

    /**
     * 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 HelloServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet HelloServlet at " +
                request.getContextPath () + "</h1>");
            out.println("<p>" + sayHello("world") + "</p>");
            out.println("</body>");
            out.println("</html>");

        } finally {
            out.close();
        }
    }
}
```

```

    }

    // doGet and doPost methods, which call processRequest, and
    // getServletInfo method

    private String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port = service.getHelloPort();
        return port.sayHello(arg0);
    }
}

```

2. 构建、打包、部署及运行Web客户端

可以使用NetBeans IDE或Ant构建、打包、部署及运行webclient应用。为了构建这一客户端，必须有已经部署好的helloservice服务，参见12.1.3节。

▼使用NetBeans IDE 构建、打包、部署及运行Web客户端

- (1) 在NetBeans IDE中选择File → Open Project。
- (2) 在Open Project对话框中，定位到*tut-install/examples/jaxws*。
- (3) 选择webclient目录。
- (4) 勾选Open as Main Project复选框。
- (5) 单击Open Project。
- (6) 在Projects标签中，右键单击webclient项目，选择Deploy。

该任务运行wsimport，构建并打包应用至WAR文件webclient.war中（该文件位于dist目录中），并最终部署至服务器上。

- (7) 在浏览器中访问如下URL：

<http://localhost:8080/webclient/HelloServlet>
sayHello方法的输出结果将显示在窗口中。

▼使用Ant 构建、打包、部署及运行Web客户端

- (1) 在终端窗口中，进入目录*tut-install/examples/jaxws/webclient/*。
- (2) 键入下面的命令：

```
ant
```

这个命令调用default目标以运行wsimport任务，构建并将应用打包至WAR文件webclient.war中，这一文件位于dist目录中。

- (3) 键入如下命令：

```
ant deploy
```

该命令将WAR文件部署至服务器上。

- (4) 在Web浏览器中访问如下URL：

<http://localhost:8080/webclient/HelloServlet>
sayHello方法的输出结果将显示在窗口中。

12.2 JAX-WS 支持的类型

JAX-WS接管了Java编程语言的数据类型与XML文档定义的JAXB之间的映射工作。应用开发人员无需知晓映射的细节，但必须清楚不是所有Java语言中的类都可以作为JAX-WS中方法的参数或返回类型。了解更多JAXB支持的类型，请参阅<http://docs.oracle.com/javaee/5/tutorial/doc/bnazq.html#bnazs>中的JAXB默认数据类型列表。

12.3 Web 服务的互操作性与 JAX-WS

JAX-WS支持WS-I(Web Services Interoperability, Web服务互操作性)基础配置(Basic Profile) 1.1版。WS-I基础配置是一个文档，定义了SOAP 1.1和WSDL 1.1规范，用以提高SOAP的互操作性。关于WS-I的相关网站，参阅12.4节。

为了支持WS-I基础配置1.1版，JAX-WS运行时需支持doc/literal和rpc/literal的服务编码规则、静态端口、动态代理以及DII(Dynamic Invocation Interface, 动态调用接口)。

12.4 有关 JAX-WS 的更多信息

有关JAX-WS及相关技术的更多信息，可以参阅如下文档。

- 支持XML Web Services 2.2规范的Java API:

<http://jax-ws.java.net/index.html>

- JAX-WS主页

<http://jax-ws.java.net/>

- W3C关于SOAP(简单对象访问协议) 1.2版的说明:

<http://www.w3.org/TR/soap/>

- W3C关于WSDL(Web服务描述语言) 1.1版的说明:

<http://www.w3.org/TR/wsdl>

- WS-I基础规范1.1:

<http://www.ws-i.org>

用JAX-RS构建REST式Web服务

本章讲述REST架构、REST式Web服务以及相关的Java API，即JSR 311定义的JAX-RS。

Jersey是JAX-RS的参考实现，该实现支持由JSR 311定义的注解，这使得开发人员使用Java语言开发REST式Web服务变得容易不少。

如果使用的是GlassFish服务器，可以借助于Update Tool安装Jersey的示例和文档。Update Tool的用法可以参考2.1.3节。Jersey示例和文档是以插件的方式提供的，在Update Tool里的Available Add-ons处可以看到。

本章内容

- 什么是REST式Web服务
- 创建一个REST式根资源类
- JAX-RS的示例应用
- 有关JAX-RS的更多信息

13.1 什么是 REST 式 Web 服务

REST式Web服务最适合构建在Web上。REST (Representational State Transfer, 表述性状态转移) 其实指的是一种架构风格，它设定诸如统一接口之类的约束条件。如果将该风格应用到Web服务上，则能够带来诸如性能、可扩展性以及可修改性等方面的提升，使得服务能够更好地运行在互联网环境中。在REST架构风格里，数据和功能均被视作资源，可以用URI (Uniform Resource Identifier, 统一资源描述符) 来访问。网页里的链接是典型的URI。可以对资源进行一系列简单的、定义完善的操作。REST架构风格要求架构必须基于客户端-服务器模型，并使用无状态的通信协议，典型的例子就是HTTP。在REST架构风格里，客户端和服务器通过标准化的接口和协议来操作资源。

如下特性使得REST式应用简单、轻量化，开发速度也快。

- 通过URI识别资源 REST式Web服务公开一套资源，这些资源用于标识与客户端交互的

目标对象。资源是由URI确定的，URI为资源提供了一个全局的寻址空间，还可以用于服务发现。这个主题在13.2.3节还会讨论。

- ❑ **统一接口** 可以使用4个固定的操作来使用资源，即PUT（创建资源）、GET（读取资源）、POST（更新资源）以及DELETE（删除资源）。PUT操作用于创建一个资源，DELETE操作用于删除已经创建的资源，GET操作用于查询资源的当前状态，POST操作可用于修改资源的状态。这个主题在13.2.4节还会讨论。
- ❑ **自描述的消息** 资源和它们的展现之间是松耦合的，因此可以以不同的方式，比如使用HTML、XML、纯文本、PDF、JPEG、JSON等方式来访问资源。与资源相关的元数据可用于诸如控制缓存机制、检测传输错误、协商合适的展现格式，以及执行身份验证或者访问控制。这个主题在13.2.4节还会讨论。
- ❑ **通过超链接实现有状态的交互** 对资源的交互都是无状态的。也就是说，请求消息是自包含的（每次交互都包含完整的信息）。有状态的交互，可以实现状态信息的显式传递。有多种技术可以实现不同请求间状态信息的传递，如URI重写、cookie以及隐藏的表单字段等。状态可以嵌入到应答消息里，这样一来状态在接下来的交互中仍然有效。关于该主题，一部分在13.2.4节的“使用实体提供者映射HTTP应答和请求实体消息的主体信息”中会讨论，还有一部分内容在JAX-RS简介文档里的“Building URIs”一节里介绍。

13.2 创建一个 REST 式根资源类

根资源类（root resource class）是指使用了@Path注解或者至少有一个方法使用了@Path注解的POJO类。还有一种情况，POJO类使用请求方法指示符注解，比如@GET、@PUT、@POST或者@DELETE。资源方法（resource method）是指在资源类中使用了请求方法指示符注解的方法。本节内容讲述如何使用JAX-RS来注解Java类，以创建REST式Web服务。

13.2.1 用JAX-RS开发REST式Web服务

JAX-RS是Java所提供的API。Java引入该API的目的是为了简化开发使用REST架构的应用。

JAX-RS API使用Java语言提供的注解功能，简化了REST式Web服务的开发。开发人员使用JAX-RS注解修饰编程语言的类文件，来定义资源以及能够应用在资源上的操作。JAX-RS注解是运行时的注解，因此运行时的反射功能会为资源生成辅助类以及中间文件。Java EE应用归档文件包含了JAX-RS资源类、配置的资源、辅助类以及中间文件，还有暴露给客户端的资源（需先将归档文件部署到Java EE服务器）。

表13-1列举了一些Java 编程用到的JAX-RS定义的注解，并简要介绍了如何使用这些注解。有关JAX-RS API的更多信息，可以访问<http://docs.oracle.com/javaee/6/api/>。

表13-1 JAX-RS 注解总结

注 解	描 述
@Path	@Path注解的值是一个相对URI路径，指明了该Java类在服务器上的相对位置，例如/helloworld。可以在这个URI中嵌入变量以构造URI路径模板，例如可以获取用户名，然后把它作为URI里的变量传给应用：/helloworld/{username}
@GET	@GET注解是一种请求方法指示符，对应于名称与之类似的HTTP方法，使用了该注解的Java方法用来处理HTTP的GET请求。资源的行为由资源响应的HTTP方法决定
@POST	@POST注解是一种请求方法指示符，对应于名称与之类似的HTTP方法，使用了该注解的Java方法用来处理HTTP的POST请求。资源的行为由资源响应的HTTP方法决定
@PUT	@PUT注解是一种请求方法指示符，对应于名称与之类似的HTTP方法，使用了该注解的Java方法用来处理HTTP的PUT请求。资源的行为由资源响应的HTTP方法决定
@DELETE	@DELETE注解是一种请求方法指示符，对应于名称与之类似的HTTP方法，使用了该注解的Java方法会处理HTTP的DELETE请求。资源的行为由资源响应的HTTP方法决定
@HEAD	@HEAD注解是一种请求方法指示符，对应于名称与之类似的HTTP方法，使用了该注解的Java方法会处理HTTP的HEAD请求。资源的行为由资源响应的HTTP方法决定
@PathParam	@PathParam注解是从URI中提取出来并可以用在资源类中的一类参数。URI路径参数是从请求URI中提取的，而且参数名和URI路径模板变量名相对应，而后者是由类级别的@Path注解指定的
@QueryParam	@QueryParam注解是从URI中提取出来并可以用在资源类中的一类参数，查询参数取自请求URI查询参数
@Consumes	@Consumes注解用来指定资源能够接收的由客户端发送来的MIME媒体类型
@Produces	@Produces注解用来指定资源能够生成并发送给客户端的MIME媒体类型，例如"text/plain"
@Provider	@Provider注解用于JAX-RS运行时关注的事物，比如MessageBodyReader和MessageBodyWriter。对于HTTP请求来说，MessageBodyReader用来将HTTP请求信息主体信息映射为方法参数。对于应答来说，返回值使用MessageBodyWriter映射成HTTP应答实体的主体消息。如果应用程序需要提供其他元数据，如HTTP头部或不同的状态码，方法可以返回一个封装了实体的Response对象，该Response对象可以使用Response.ResponseBuilder创建

13.2.2 JAX-RS应用概述

如下代码展示了一个很简单的根资源类，该根资源类使用了JAX-RS的注解。

```
package com.sun.jersey.samples.helloworld.resources;

import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;

// The Java class will be hosted at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getClichedMessage() {
        // Return some cliched textual content
        return "Hello World";
    }
}
```

接下来的内容将介绍本例中用到的注解。

- ❑ `@Path`注解的值是相对URI路径。在先前的例子里，Java类运行在服务器上的相对URI路径/helloworld下。在`@Path`注解中使用固定的URI路径是最简单的使用方式。URI里还可以包含变量。URI路径模板就是包含变量的URI。
- ❑ `@GET`注解是请求方法指示符的一种，此外还有`@POST`、`@PUT`、`@DELETE`以及`@HEAD`。这些注解是由JAX-RS定义的，且与名称类似的HTTP方法相对应。例子里，使用注解的Java方法会处理HTTP的GET请求。资源的行为由响应资源的HTTP方法决定。
- ❑ `@Produces`注解用来指定资源可以生成并发送给客户端的MIME媒体类型。本例中，Java方法会生成MIME媒体类型为 "text/plain"的数据。
- ❑ `@Consumes`注解用来指定由客户端发来，且能够被资源接收的MIME媒体类型。这个例子可以改写以重设`getClichedMessage`方法的返回值，如下面的代码所示：

```
@POST
@Consumes("text/plain")
public void postClichedMessage(String message) {
    // Store the message
}
```

13.2.3 `@Path`注解和URI路径模板

`@Path`注解设定资源要响应的URI路径模板。`@Path`注解有类级别注解和方法级别注解两种。`@Path`注解的值是URI路径模板的一部分，这个路径是相对路径（相对于资源被部署在服务器上的基准路径）。`@Path`也是应用的上下文根，同时还是JAX-RS运行时要响应的URL模式。

URI路径模板是嵌入了变量的URI。嵌入的变量在运行时会被具体值替换掉，这样资源就能够根据替换后的URI响应请求。变量名需要用大括号括起来。例如，下面是一个使用`@Path`注解的例子：

```
@Path("/users/{username}")
```

这个例子会提示用户输入自己的名字，然后一个被配置成响应这个URI路径模板的JAX-RS Web服务将响应该请求。比如说，如果用户输入了自己的名字“Galileo”，Web服务响应如下的URI：

```
http://example.com/users/Galileo
```

要得到用户输入的名字，可以在请求的方法参数里使用`@PathParam`注解，如下面的代码所示：

```
@Path("/users/{username}")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

默认情况下, URI变量必须匹配正则表达式"`[^/]+?`". 这个变量也可以通过在变量名称后指定不同的正则表达式来实现定制。例如, 如果用户名只能包含大小写字母和数字, 则可在定义变量的时候覆盖默认的正则表达式:

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]}")
```

在这个例子里, `username`变量仅匹配以大写或者小写字母开头的, 后跟0个或多个字母、数字以及下划线的用户名。如果用户名不匹配这一模板, 则向客户端发送404错误 (Not Found)。

`@Path`的值并非必须以/开头或者结束。JAX-RS在运行时解析出的URI路径模板都是一样的, 无论值的前后是否有空格。

URI路径模板可以有一个或者多个变量, 每一个变量名用大括号括起来: {代表着变量名的开始, }代表着变量名的结束。在前面的例子里, `username`是一个变量名。在运行时, 配置成响应URI路径模板的资源会尝试处理URI数据, 把URI里的`{username}`替换成具体的值。

举例说明, 如果要部署一个响应URI路径模板 (`http://example.com/myContextRoot/resources/{name1}/{name2}/`) 的资源, 必须把应用部署到 Java EE 服务器上, 该应用响应对 `http://example.com/myContextRoot` 这一URI的请求, 然后以如下方式用 `@Path` 注解我们的资源:

```
@Path("/{name1}/{name2}/")
public class SomeResource {
    ...
}
```

这个例子里, JAX-RS 辅助servlet的URI模式是在web.xml里设定的, 使用的是默认值:

```
<servlet-mapping>
  <servlet-name>My JAX-RS Resource</servlet-name>
  <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

变量名可以在URI路径模板里使用多次。

如果变量值里的字符与URI的保留字符相冲突, 有冲突的字符应当替换成使用百分号编码格式的字符。比如说, 变量值里的空格应当替换成`%20`。

当定义URI路径模板时, 要仔细检查以确保替换URI后的结果是有效的URI。

表13-2列举了一些URI路径模板变量的例子, 以及替换变量后, URI是如何解析的。如下的变量名和值用在了下面的例子里:

- `name1`: james
- `name2`: gatz
- `name3`:
- `location`: Main%20Street
- `question`: why

注意 `name3`这个变量的值是空字符串。

表13-2 URI路径模板的示例

URI路径模板	替换后的URI
http://example.com/{name1}/{name2}/	http://example.com/james/gatz/
http://example.com/{question}/{question}/{question}/	http://example.com/why/why/why/
http://example.com/maps/{location}	http://example.com/maps/Main%20Street
http://example.com/{name3}/home/	http://example.com/home/

13.2.4 响应HTTP资源

资源的行为表现是由资源响应的HTTP方法决定的，这里典型的HTTP方法有GET、POST、PUT以及DELETE。

1. 请求方法指示符注解

请求方法指示符注解是由JAX-RS定义的运行时注解，它与名称类似的HTTP方法对应。在资源类文件内部，通过请求方法指示符注解可以把HTTP方法映射到Java方法。资源的行为表现决定于资源要响应的那个HTTP方法。JAX-RS为常用的HTTP方法定义了一组请求方法指示符注解，如@GET、@POST、@PUT、@DELETE以及@HEAD。开发人员也可以自定义请求方法指示符。关于如何自定义请求方法指示符不在本书讲述范围内。

下面的例子是从存储服务示例应用中节选的，展示了如何使用PUT方法创建或者更新存储容器：

```
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);

    URI uri = uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());

    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}
```

默认情况下，如果没有显式实现的话，JAX-RS运行时会自动地支持HEAD方法和OPTIONS方法。对于HEAD方法，运行时调用GET对应的方法实现（如实现存在的话），且会忽略应答体（即使设置了应答体）。对于OPTIONS方法，应答头部中的Allow会被设置为资源支持的一组HTTP方法。除此之外，JAX-RS运行时返回一个WADI（Web Application Definition Language，Web应用定义语言）文档以描述资源，详情参见<http://wadl.java.net/>。

用请求方法指示符注解的方法，其返回类型只能是void、Java编程语言中的类型，或者javax.ws.rs.core.Response对象。可以使用PathParam注解或者QueryParam注解从URI中解析出多

个参数，具体内容可以参考13.2.6节。把Java类型的数据转换为实体对象，是实体提供者（比如 `MessageBodyReader` 或者 `MessageBodyWriter`）要做的事。那些需要在应答信息里提供额外元数据的方法应当返回一个 `Response` 类的实例。`ResponseBuilder` 类使用 `builder` 模式为创建 `Response` 实例提供了一个简便方法。HTTP的PUT和POST方法需要读取HTTP请求体，因此应当在响应PUT和POST请求的方法中使用 `MessageBodyReader`。

`@PUT`和`@POST`可以用来创建或者更新资源。POST的含义很丰富，因此当使用POST的时候，由应用来定义其语义。PUT有定义良好的语义，当使用PUT进行创建的时候，客户端要声明新创建资源的URI。

PUT对于创建或者更新资源有着明确的语义。客户端发送的表述（`representation`）必须和使用GET收到的表述相同，假定媒体类型相同。PUT不允许部分更新一个资源，这是使用PUT方法时的一个很常见的错误。通常的应用模式是使用POST创建资源并通过返回201响应，将新创建资源的URI的值设置在返回信息头部中。这种模式下，Web服务声明新创建的资源的URI。

2. 使用实体提供者映射HTTP应答和请求实体消息的主体信息

实体提供者展现及与其相关联的Java类型之间提供映射服务。实体提供者有两种类型，即 `MessageBodyReader` 和 `MessageBodyWriter`。对于HTTP请求来说，`MessageBodyReader` 用来把HTTP请求实体的消息体信息映射到方法的参数。对于应答来说，使用 `MessageBodyWriter` 把返回值映射到HTTP应答实体的消息体里。如果应用需要补充附加的元数据，比如HTTP头或者不同的状态码，方法可以返回一个包装了这些实体的 `Response` 实例。`Response` 实例可以通过 `Response.ResponseBuilder` 创建。

表13-3列举了已经由实体自动支持的标准类型。只有在这些标准类型都不满足要求的情况下，才需要手工编写实体提供者。

表13-3 实体支持的类型

Java类型	支持的媒体类型
<code>byte[]</code>	所有类型 (<code>*/*</code>)
<code>java.lang.String</code>	所有文本类型 (<code>text/*</code>)
<code>java.io.InputStream</code>	所有类型 (<code>*/*</code>)
<code>java.io.Reader</code>	所有类型 (<code>*/*</code>)
<code>java.io.File</code>	所有类型 (<code>*/*</code>)
<code>javax.activation.DataSource</code>	所有类型 (<code>*/*</code>)
<code>javax.xml.transform.Source</code>	XML媒体类型 (<code>text/xml</code> , <code>application/xml</code> 和 <code>application/*+xml</code>)
<code>javax.xml.bind.JAXBElement</code> 和 应用提供的JAXB类	XML媒体类型 (<code>text/xml</code> , <code>application/xml</code> 和 <code>application/*+xml</code>)
<code>Multimap<String,String></code>	表单内容 (<code>application/x-www-form-urlencoded</code>)
<code>StreamingOutput</code>	所有的媒体类型 (<code>*/*</code>)，只对 <code>MessageBodyWriter</code> 有效

下面的代码展示了如何用`@Consumes`、`@Provider`注解配合起来使用 `MessageBodyReader`：


```
@Consumes("application/x-www-form-urlencoded")
@Provider
public class FormReader implements MessageBodyReader<NameValuePair> {
```

下面的例子展示了如何用@Produces、@Provider注解配合起来使用MessageBodyWriter:

```
@Produces("text/html")
@Provider
public class FormWriter implements
    MessageBodyWriter<Hashtable<String, String>> {
```

下面的例子展示了如何使用ResponseBuilder:

```
@GET
public Response getItem() {
    System.out.println("GET ITEM " + container + " " + item);

    Item i = MemoryStore.MS.getItem(container, item);
    if (i == null)
        throw new NotFoundException("Item not found");
    Date lastModified = i.getLastModified().getTime();
    EntityTag et = new EntityTag(i.getDigest());
    ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);

    if (rb != null)
        return rb.build();

    byte[] b = MemoryStore.MS.getItemData(container, item);
    return Response.ok(b, i.getMimeType()).
        lastModified(lastModified).tag(et).build();
}
```

13.2.5 使用@Consumes和@Produces定制请求和应答

发送给资源的信息以及传回客户端的信息都被指定了特定的MIME媒体类型, 这是在HTTP请求或者应答的头部里设定的。可以使用下面的注解指定资源能够响应或创建的MIME媒体类型:

- javax.ws.rs.Consumes
- javax.ws.rs.Produces

默认情况下, 资源类可以响应以及创建任何在HTTP请求以及HTTP应答的头部里指定的、用于展现的MIME媒体类型。

1. @Produces注解

@Produces注解用来指定一个资源能够创建并发送回客户端的MIME媒体类型。如果@Produces应用在类级别上, 则资源里的所有方法都默认可以创建指定的MIME类型; 如果应用在方法级别上, 则该注解将覆写任何应用在类级别上的@Produces注解。

如果资源里没有方法能够创建客户端请求的MIME类型, 则JAX-RS运行时会发送给客户端一个HTTP 406 (Not Acceptable) 错误信息。

@Produces的值是一个字符串数组, 里面每一项都代表了其支持的MIME类型, 例如:

```
@Produces({"image/jpeg,image/png"})
```

下面的例子展示了如何在类级别以及方法级别上应用@Produces注解:

```

@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}

```

`doGetAsPlainText()`方法默认使用类级别上定义的`@Produces`注解指定的MIME媒体类型。`doGetAsHtml`方法上的`@Produces`注解覆写了类级别的`@Produces`设置,因此该方法能够生成的是HTML类型,而不是普通的文本类型。

如果某资源类可以创建多个MIME媒体类型,对资源方法的选择取决于客户端所声明的最可被接受的媒体类型。具体来说,HTTP请求里的Accept头部声明了它最可接受的媒体类型。例如,如果Accept头部是Accept: text/plain,那么就调用`doGetAsPlainText`方法。如果Accept头部是Accept: text/plain;q=0.9,text/html,它声明客户端可以接受的媒体类型是text/plain和text/html,但是倾向于后者,因此就调用`doGetAsHtml`方法。

在一个`@Produces`注解里,可以声明多个媒体类型。下面的代码例子展示了怎样做到这一点:

```

@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}

```

如果可接受的媒体类型是application/xml或application/json, `doGetAsXmlOrJson`方法将会被调用。如果两个媒体类型是同等可接受的,那么会选择声明在前面的那个方法。上面的例子使用的是显式的MIME媒体类型声明,就是为了明确这一点。也可以使用相应的常量值,这样可以减少拼写错误。相关的信息参见Media Type类的常量字段值,网址为<http://jsr311.java.net/nonav/releases/1.0/javax/ws/rs/core/MediaType.html>。

2. @Consumes 注解

`@Consumes`注解用来指定资源能够接受或处理的来自客户端的MIME媒体类型。如果`@Consumes`应用在类级别,该类的所有响应方法都默认接受指定的MIME类型。如果`@Consumes`应用在方法级别,那么方法级别上的`@Consumes`注解会覆写应用在类级别上的`@Consumes`注解。

如果资源不能够处理客户端请求中的MIME类型, JAX-RS 运行时会返回HTTP 415 (Unsupported Media Type) 错误给客户端。

`@Consumes`注解的值是一个可接受的MIME类型的字符串数组。例如:

```
@Consumes ({"text/plain",text/html"})
```

如下的例子展示了如何在类级别以及方法级别上使用`@Consumes`注解:

```

@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
    @POST
    public String doPost(MimeMultipart mimeMultipartData) {
        ...
    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public String doPost2(FormURLEncodedProperties formData) {
        ...
    }
}

```

doPost方法默认的MIME媒体类型是@Consumes注解在类级别上指定的。doPost2方法的@Consumes注解覆写了类级别的@Consumes注解，它指定自己可以接受以URL编码的表单数据。

如果没有对应的资源方法可以响应请求的MIME类型，则HTTP 415错误（Unsupported Media Type）就会返回给客户端。

本节前面讨论的HelloWorld例子可以使用@Consumes注解修改一下以设置消息，如下面的代码所示：

```

@POST
@Consumes("text/plain")
public void postClickedMessage(String message) {
    // Store the message
}

```

本例中，Java方法会处理由MIME媒体类型text/plain标识的展现方式。注意资源类的方法返回的是void类型，这意味着不会返回展现内容，而实际上返回给客户端的是状态码为HTTP 204（No Content）的信息。

13.2.6 从请求里提取参数

资源方法的参数可以使用带参数的注解，这样一来就可以从请求里提取出信息供方法使用。前面有个例子展示了对于匹配了@Path里声明的路径的请求URL，使用@PathParam参数可以提取出路径参数。

可以提取出如下类型的参数供资源类使用：

- ☐ 查询参数；
- ☐ URI 路径参数；
- ☐ 表单参数；
- ☐ cookie参数；
- ☐ 头部参数；
- ☐ 矩阵参数。

查询参数是从请求的URI查询参数里提取出来的，在方法的参数里，它由javax.ws.rs.QueryParam注解来指定。下面的例子来自sparklines示例应用，它演示了如何使用@QueryParam注解从请求URL

的Query组件中提取出查询参数:

```
@Path("smooth")
@GET
public Response smooth(
    @DefaultValue("2") @QueryParam("step") int step,
    @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
    @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
    @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
    @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
    @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
    @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor
) { ... }
```

如果查询参数step存在于请求URI的Query组件里,则step的值将被提取出来,解析成一个32位的有符号整数,然后再赋给方法的step参数。如果step不存在,则@DefaultValue注解所声明的默认值2将会赋给方法的step参数。如果step的值不能解析成32位有符号整数,则返回状态码为HTTP 400 (Client Error) 的应答消息。

用户自定义的Java类型也可以用做查询参数。下面的代码展示了ColorParam类(先前的查询参数例子里有用到它):

```
public class ColorParam extends Color {
    public ColorParam(String s) {
        super(getRGB(s));
    }

    private static int getRGB(String s) {
        if (s.charAt(0) == '#') {
            try {
                Color c = Color.decode("0x" + s.substring(1));
                return c.getRGB();
            } catch (NumberFormatException e) {
                throw new WebApplicationException(400);
            }
        } else {
            try {
                Field f = Color.class.getField(s);
                return ((Color)f.get(null)).getRGB();
            } catch (Exception e) {
                throw new WebApplicationException(400);
            }
        }
    }
}
```

ColorParam构造方法接受一个String类型的参数。

@QueryParam和@PathParam只能供如下Java类型使用。

- 所有的基本类型, char除外。
- 所有基本类型的包装类, Character除外。
- 任何有构造函数的类, 只要该构造函数仅接受一个String类型的参数。
- 有valueOf(String)静态方法, 且接受一个String参数的任何类。
- 任何有构造函数的类, 只要该构造函数仅接受一个String类型的参数。

□ `List<T>`、`Set<T>`或`SortedSet<T>`、这里 *T* 匹配的是上面列举的类型。有时候，同一个名字的参数可能包含不止一个值，如果是这种情况，可以使用这些类型获取所有的值。

如果`@DefaultValue`未和`@QueryParam`一起使用，且查询参数在请求里不存在，那么：对于`List`、`Set`和`SortedSet`类型的参数来说，其值将会是空集合，即一个元素都没有；如果是一般对象类型，则值为`null`；如果是基本数据类型的话，则返回各自的默认值。

URI 路径参数是从请求URI里提取出来的，且参数名对应URI路径模板的变量名（是由注解在类级别的`@Path`指定的）。在方法的参数里，URI参数是由`javax.ws.rs.PathParam`注解指定的。下面的代码展示了如何使用`@Path`变量以及在方法里的`@PathParam`注解：

```
@Path("/{username}")
public class MyResourceBean {
    ...
    @GET
    public String printUsername(@PathParam("username") String userId) {
        ...
    }
}
```

在先前的代码片段里，URI路径模板里的变量名是`username`，该变量被指定为`printUsername`方法的一个参数。`@PathParam`注解用到了变量名`username`。在运行的时候，且在`printUsername`被调用之前，`username`的值被从URI提取出来，然后转换为`String`类型。转换后的结果才可以作为`userId`变量，传递给`printUsername`方法。

如果不能将URI路径模板变量转换为指定的类型，JAX-RS会在运行时向客户端返回出错信息HTTP 400（Bad Request）。如果不能将`@PathParam`注解转换为指定的类型，JAX-RS会在运行时向客户端返回出错信息HTTP 404（Not Found）。

`@PathParam`参数和其他带参数的注解（`@MatrixParam`、`@HeaderParam`、`@CookieParam`以及`@FormParam`）遵循与`@QueryParam`一样的规则。

`cookie`参数使用`javax.ws.rs.CookieParam`注解，能提取出HTTP头部中与`cookie`相关的信息。使用`javax.ws.rs.HeaderParam`注解的头部参数可以从HTTP头中提取信息。使用`javax.ws.rs.MatrixParam`注解的矩阵参数可以从URL路径中提取信息。

使用`javax.ws.rs.FormParam`注解的表单参数可以从MIME媒体类型为`application/x-www-form-urlencoded`的请求中提取出信息，并遵从HTML表单指定的编码，详见<http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1>。这种参数非常有用，可以提取HTML表单里POST操作发送出去的信息。

下面的代码片段从以POST方式提交的表单数据里提取出`name`参数：

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) {
    // Store the message
}
```

要获取请求里（以及路径参数）的参数名和值组成的映射，可以使用如下的代码：

```
@GET
public String get(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

如下方法提取了请求的头部，以及cookie中的参数名及相应的值，并将这些数据保存在一个映射里：

```
@GET
public String get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = ui.getRequestHeaders();
    Map<String, Cookie> pathParams = ui.getCookies();
}
```

通常来说，`@Context`可以用来获取与请求或者应答相关的Java数据类型。

对于表单参数来说，可以这么做：

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    // Store the message
}
```

13.3 JAX-RS 的示例应用

本节讲述如何创建、部署以及运行一个JAX-RS应用，并演示创建、构建、部署以及测试一个简单Web应用（这个Web应用使用了JAX-RS注解）的基本步骤。

13.3.1 REST式Web服务

本节讲述如何使用NetBeans IDE创建一个REST式Web服务。NetBeans IDE生成一个应用的基本框架，我们只需写代码去实现应用的方法。如果不使用IDE的话，可以将Jersey里的某个示例应用作为模板，适当修改后再使用。

▼用NetBeans IDE创建REST式Web服务

(1) 在Netbeans IDE里，创建一个简单的Web应用。这个例子创建一个非常简单的“Hello, World” Web应用。

(a) 在NetBeans IDE里，选择File→New Project。

(b) 在Categories里选择Java Web。在Projects里选择Web Application，单击Next。

注意 在这一步中，也可以在Maven Web项目中创建一个REST式Web服务：只需在Categories中选择Maven，并在Projects中选择Maven Web Project。其余步骤是一样的。

(c) 输入项目的名字HelloWorldApplication，单击Next。

(d) 确保Server是GlassFish Server (或类似的服务器)。

(e) 单击Finish。

项目创建完成, index.jsp文件显示在Source窗格里。

(2) 右击项目, 选择New, 然后从Patterns里选择RESTful Web Services。

(a) 选择Simple Root Resource, 单击New。

(b) 输入Resource Package的名字, 比如helloworld。

(c) 在Path字段输入helloworld。在Class Name字段输入HelloWorld。至于MIMeType (类型), 则选择text/html。

(d) 单击Finish。

此时REST Resources Configuration 页面出现。

(e) 单击OK。

名为HelloWorld.java的新资源文件会被添加到项目中, 并出现在Source窗格里。这个文件提供了创建一个REST式Web服务的模板。

(3) 在HelloWorld.java文件里, 找到getHtml()方法, 用以下的代码替换掉//TODO注释以及异常处理, 从而完成如下的方法。

注意 因为本例中MIME类型是HTML, 可以在return语句里使用HTML标签。

```
/**
 * Retrieves representation of an instance of helloWorld.HelloWorld
 * @return an instance of java.lang.String
 */
@GET
@Produces("text/html")
public String getHtml() {
    return "<html><body><h1>Hello, World!!</body></h1></html>";
}
```

(4) 测试这个Web服务。右键单击项目结点, 然后单击Test RESTful Web Services。

这一步会部署应用, 并且在浏览器里启动一个测试客户端。

(5) 当测试客户端出现以后, 在左边的窗格里选择helloworld资源, 然后单击右边窗格里的Test按钮。

Hello,World! !就会显示在下面的Response窗口里。

(6) 设置Run Properties。

(a) 右击项目结点, 选择Properties。

(b) 在对话框里选择Run category。

(c) 设置Relative URL来指定REST式Web服务相对于Context Path (上下文路径) 的位置, 对于本例来说, 就是resources/helloworld。

小贴士 可以在Test RESTful Web Services浏览器窗口里找出相对URL的值。在右边窗格的上方，Resource后面，就是要测试的REST式Web服务的URL。Context Path后面的部分（<http://localhost:8080/HelloWorldApp>）是需要在这里输入的相对URL。

如果不设置这个属性，当应用运行时，文件index.jsp会作为默认页面。由于这个文件的默认返回值也是HelloWorld，我们可能注意不到REST式Web服务没有运行。所以要注意这个默认值，有必要设置这个属性，或者修改index.jsp以提供一个指向REST式Web服务的链接。

(7) 右击这个项目，选择Deploy。

(8) 右击这个项目，选择Run。

此时打开一个浏览器窗口，其中显示返回值Hello, World!!。

参考内容 使用NetBeans IDE部署以及运行JAX-RS应用的其他有关示例应用，参见13.3.2节，以及一篇名为“Your First Cup: An Introduction to the Java EE Platform”的文章（网址为<http://docs.oracle.com/javaee/6/firstcup/doc/>）。还可以参阅NetBeans IDE官方网站上的教程，比如有一篇名为“Getting Started with RESTful Web Services”的文章（网址为<http://www.netbeans.org/kb/docs/websvc/rest.html>）。该教程有一节讲述了如何创建一个CRUD型的数据库应用。Create（创建）、Read（读）、Update（更新）以及Delete（删除）简称CRUD，它们是在关系型数据库中存储持久化数据的4个基本操作。

13.3.2 rsvp示例应用

rsvp示例应用位于*tut-install/examples/jaxrs/rsvp*目录下，该示例允许事件的“受邀者”表态是否要“出席”。示例应用的事件、“受邀者”以及对于邀请的应答，都通过Java Persistence API保存在Java DB数据库里。rsvp里的JAX-RS资源是以一个无状态会话企业bean的形式暴露出来的。

1. rsvp示例应用的组件

rsvp示例应用里有3个企业bean，它们分别是rsvp.ejb.ConfigBean、rsvp.ejb.StatusBean以及rsvp.ejb.ResponseBean。

ConfigBean是一个单件会话bean，用来初始化数据库里的数据。

StatusBean公开了一个JAX-RS资源，用来显示某事件的所有“受邀者”的当前状态。该资源的URI路径模板声明如下：

```
@Path("/status/{eventId}/")
```

URI路径变量eventId在getResponse方法里被用作@PathParam变量，getResponse方法使用了@GET注解，表明它是用来处理HTTP GET请求的。变量eventId用于在数据库里查找特定事件的所有当前应答。

ResponseBean公开了一个JAX-RS资源，用来设置“受邀者”对于某个特定事件的应答。ResponseBean的URI路径模板声明如下：

```
@Path("/{eventId}/{inviteId}")
```


路径模板里声明了两个URI路径变量：`eventId`和`inviteId`。和`StatusBean`里一样，`eventId`是某特定事件的唯一标识。该事件的每一个“受邀者”都有唯一的ID，即`inviteId`。这两个变量用于两个JAX-RS方法，即`ResponseBean`的`getResponse`方法和`putResponse`方法。`getResponse`方法用于响应HTTP GET请求，并显示“受邀者”的应答，以及修改应答的表单。

某位“受邀者”如想修改自己的应答，可以选择新的应答并提交表单数据，这会被当做HTTP PUT请求来处理，对应的方法是`putResponse`。`putResponse`方法有一个字符串类型的参数`userResponse`用了注解，即`@FormParam("attendeeResponse")`。由`getResponse`生成的HTML表单保存了修改后的应答，并将这一应答对应的ID保存在`attendeeResponse`选择列表里。注解`@FormParam("attendeeResponse")`表明，从HTTP PUT请求里提取出来的应答的ID会保存在`userResponse`字符串里。`putResponse`方法使用`userResponse`、`eventId`以及`inviteId`来更新数据库里“受邀者”的应答。

`rsvp`里的事件、“受邀者”以及应答都封装在Java Persistence API实体里。实体类`rsvp.entity.Event`、`rsvp.entity.Person`以及`rsvp.entity.Response`分别代表着事件、“受邀者”以及应答这3个实体。

`rsvp.util.ResponseEnum`类声明了一个枚举类型，它代表所有可能的应答状态。

2. 运行rsvp示例应用

可以用NetBeans IDE或者Ant来部署和运行rsvp示例应用。

▼使用NetBeans IDE部署和运行rsvp示例应用

- (1) 在NetBeans IDE里，选择File→Open Project。
- (2) 在Open Project对话框里，定位到`tut-install/examples/jaxrs/`。
- (3) 选择rsvp文件夹。
- (4) 勾选Open as Main Project复选框。
- (5) 单击Open Project。
- (6) 在左边的窗格里，右键单击rsvp项目，选择Run。

项目会被编译、装配以及部署到GlassFish服务器上。Web浏览器会被启动，并打开<http://localhost:8080/rsvp>。

- (7) 在Web浏览器窗口中单击Event Status链接，以选择Duke's Birthday事件。

此时可以看到当前“受邀者”及其应答状态。

- (8) 单击某个“受邀者”的名字，选择一个应答，并单击Submit response，然后单击Back回到事件页面。

现在显示在“受邀者”表格里应该是所有受邀者的新状态和应答状态。

▼使用Ant部署和运行rsvp示例应用

在运行rsvp之前，必须先启动Java DB数据库。

- (1) 打开一个终端窗口，并切换目录到`tut-install/examples/jaxrs/rsvp`。

(2) 输入如下的命令并执行：

```
ant all
```

这个命令会构建、装配以及部署rsvp应用到GlassFish服务器上。

(3) 打开Web浏览器，并访问<http://localhost:8080/rsvp>。

(4) 在Web浏览器窗口中单击Event Status链接，选择Duke's Birthday事件。可以看到当前“受邀”此时其应答状态。

(5) 单击某个“受邀者”的名字，选择一个应答，单击Submit response，然后单击Back回到事件页面。

现在显示在“受邀者”表格里的应该是这位受邀者的新状态和应答状态。

13.3.3 真实示例

大多数博客网站使用了REST式Web服务。这些网站都有下载XML文件的功能(RSS或者Atom格式)。这些XML文件包括到其他资源的链接列表。其他使用了与REST式类似的开发接口的网站或者Web应用包括Twitter和Amazon的S3(Simple Storage Service)。利用Amazon的S3，用户可以使用REST式HTTP接口或者SOAP接口创建、列举以及获取存储对象。Jersey有一个使用了REST式接口的、实现存储服务的例子。网址<http://netbeans.org/kb/docs/websvc/twitter-swing.html>处有一个教程，讲述如何使用NetBeans IDE创建一个简单的、图形化的基于REST风格的客户端，以显示Twitter中公开的按照时间排序的消息，同时允许用户查看以及更新自己的Twitter状态。

13.4 有关 JAX-RS 的更多信息

有关REST式Web服务以及JAX-RS的更多信息，可以参阅如下内容。

- “RESTful Web Service vs. ‘Big’ Web Service: Making the Right Architectural Decision”，参见：

<http://www2008.org/papers/pdf/p805-pautassoA.pdf>

- 项目Jersey的线上社区给出了JAX-RS参考实现，参见：

<https://wikis.oracle.com/display/Jersey/Main>

- Roy Thomas Fielding 的博士论文“Architectural Styles and the Design of Network-based Software Architectures”中的第5章，即“Representational State Transfer (REST)”，参见：

http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

- Leonard Richardson和Sam Ruby的*RESTful Web Services*，参见O'Reilly的网站：

<http://oreilly.com/catalog/9780596529260/>

- 构建REST式Web服务的Java API: JSR-311，参见：

<http://jcp.org/en/jsr/detail?id=311>

□ JAX-RS 项目，参见：

<http://jsr311.java.net/>

□ Jersey项目，参见：

<https://jersey.java.net/>

□ JAX-RS 概述文档，参见：

<https://wikis.oracle.com/display/Jersey/Overview+of+JAX-RS+1.0+Features>

Part 4

第四部分

企业 bean

本部分介绍 EJB 组件。

本 部 分 内 容

- 第 14 章 企业 bean
- 第 15 章 企业 bean 应用初步
- 第 16 章 运行企业 bean 示例

企业bean是实现EJB (Enterprise JavaBeans) 技术的Java EE组件。企业bean运行于EJB容器内，而EJB容器是应用服务器（如GlassFish）中企业应用的运行环境（参见1.4.2节）。对应用开发人员来说，EJB容器是透明的，它为企业bean提供了一系列的系统级服务，如事务和安全。这些服务使得开发人员能够快速构建和部署企业bean，而企业bean正是事务型Java EE应用的核心。

本章内容

- 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的命名规范
- 企业bean的生命周期
- 有关企业bean的更多信息

14.1 什么是企业 bean

企业bean是用Java语言编写的，封装了应用的业务逻辑，是运行于服务器端的组件。业务逻辑是满足业务需求的代码片段。例如，在一个库存管理应用中，企业bean可以通过方法checkInventoryLevel和orderProduct实现业务逻辑。通过调用这些32位的方法，客户端可以访问应用提供的库存管理服务。

14.1.1 使用企业bean的好处

基于下述原因，企业bean可以简化大型分布式应用的开发过程。首先，EJB容器为企业bean提供了系统级的服务，bean开发人员可以将精力集中到解决业务问题上。EJB容器负责提供系统级的服务，如事务管理和安全认证，企业bean开发人员则无需关注。

其次，bean代替客户端，实现了应用的业务逻辑，从而使客户端开发人员可专注于客户端的

应用展现。他们无需编写繁琐的代码以实现业务逻辑或数据库访问。客户端的简化，对于运行于小型设备上的客户端尤为重要。

再次，由于企业bean是可移植的组件，应用装配人员可以在已有bean的基础上构建出新的应用。由于应用使用了标准的API，这类应用可以运行于任何与Java EE兼容的服务器上。

14.1.2 何时使用企业bean

如果应用有如下需求，可以考虑使用企业bean。

- ❑ 应用必须具有伸缩性。满足未来不断增长的用户量，应用组件需要分布式部署至多个设备上。应用中的企业bean可以运行于不同的设备上，而客户端无需知晓企业bean的具体位置便可访问它。
- ❑ 具备事务管理能力，确保数据的完整性。企业bean支持事务机制，以管理对共享对象的并发访问。
- ❑ 应用通常有大量的客户端。仅需几行代码，远程客户端可以轻松地定位企业bean。企业bean实现了对简化的、类型多样且数量众多的客户端的支持。

14.1.3 企业bean的类型

表14-1总结了企业bean的两种类型。后面将对每一种类型进行详细介绍。

表14-1 企业bean的类型

企业bean类型	目 的
会话型	执行客户端提交的任务；可以实现Web服务
消息驱动型	作为特定类型消息的监听器，如JMS（Java Message Service）API

14.2 什么是会话 bean

会话bean封装了业务逻辑，客户端可以以编程的方式，通过本地、远程或Web服务调用业务逻辑。客户端通过调用会话bean中的方法，访问部署于服务器上的应用。会话bean为客户端提供服务，并通过在服务器上执行服务的实现逻辑隐藏其复杂性。

会话bean不具有持久性，即它的数据不保存至数据库中。

示例代码参见第16章。

14.2.1 会话bean的类型

会话bean有3种类型：有状态、无状态和单件。

1. 有状态会话bean

对象的状态由其实例变量的值构成。在有状态会话bean中，实例变量代表了唯一的客户端或

会话bean的状态。由于客户端与其对应的bean进行交互，此时bean的状态通常称为会话状态 (conversational state)。

正如它的名字一样，会话bean类似于交互型会话。会话bean不可共享，它仅有一个客户端，如同交互型会话仅有一个用户一样。当客户端终止，会话bean也将终止，且不再与客户端相关联。

状态的保存仅限于客户端或会话bean的延续期内。如果客户端删除了bean，会话将终止，同时状态消失。由于客户端与bean之间直接的会话终止了，也就无需再保留其状态，因此状态的这种短暂特性并不是问题。

2. 无状态会话bean

无状态会话bean不维护其与客户端间会话的状态。当客户端调用无状态bean的方法时，bean的实例变量可包含一个面向此客户端的状态，但仅限于调用的期间内。当方法执行结束，面向客户端的状态不应保留。然而，不同的客户端都会修改可复用对象池中无状态bean内的实例变量的状态，且这种状态会延续至对可复用无状态bean的下一次调用。除非在方法调用的过程中，否则一个无状态bean的所有实例都是平等的，进而EJB容器可将实例分配给任意客户端。也就是说，无状态会话bean的状态可以应用于所有客户端。

由于无状态bean可以支持多个客户端，因而为需要大量客户端访问的应用提供了更好的伸缩性。在典型的应用场景中，应用在满足相同数量客户端需求的前提下，所需的无状态会话bean的数量要少于有状态会话bean。

无状态会话bean可以实现Web服务，而有状态会话bean则不行。

3. 单件会话bean

每个应用中的单件会话bean仅实例化一次，便可用于应用的整个生命周期。单件会话bean为特定的情景而设计，在这种情景下，客户端可以以共享且并发的模式访问这个唯一的企业bean实例。

单件会话bean提供的功能与无状态会话bean相似，其区别在于单件会话bean在应用中仅有一个实例bean。与之相反，无状态会话bean可以有多个，且任意一个无状态会话bean都可以用来响应客户端的请求。与无状态会话bean相同，单件会话bean可以实现Web服务的端点。

单件会话bean即使被不同的客户端调用，也能维护自身状态的一致，但并不要求其具备在服务器异常终止或正常停机后对状态的维护能力。

使用单件会话bean的应用可以指定应用启动时执行单件实例化，即初始化，从而使其完成应用相关的初始化任务。单件bean也可以实现应用终止前的清理工作，这种能力正是源于单件会话bean可运行于应用的整个生命周期这一特性。

14.2.2 何时使用会话bean

有状态的会话bean可在如下场合中应用。

- ❑ bean的状态记录bean与特定客户端间交互的信息。
- ❑ bean需要在跨方法调用的过程中保存客户端的信息。

- bean实现客户端与应用的其他组件间的数据转换，为客户端呈现简化的数据和访问视图。
 - 在后台，bean管理由多个企业bean组成的工作流。
- 为了提升性能，可以在如下场景中使用无状态会话bean。
- bean的状态中不记录特定客户端的信息；
 - bean的某个方法可能被所有客户端调用，以执行某个通用任务。例如，可以使用无状态会话bean发送电子邮件以确认在线订单。
 - 用bean实现Web服务。
- 单件会话bean适合如下场景。
- 其状态需要在整个应用内共享。
 - 某个企业bean需要被多个线程并发访问。
 - 应用需要通过企业bean在启动和终止时执行特定任务。
 - 用bean实现Web服务。

14.3 什么是消息驱动 bean

消息驱动bean是一种企业bean，使得Java EE应用可以处理异步消息。这种bean通常用作JMS消息监听器。它与事件监听器相似，但接收的是JMS消息而非事件。消息可以通过任意Java EE组件（可以是应用客户端、另一个企业bean或Web组件）发送，或通过JMS应用发送，也可以通过一个非Java EE实现的系统发送。消息驱动bean可以处理JMS或其他类型的消息。

14.3.1 消息驱动bean与会话bean的区别

消息驱动bean与会话bean之间最显著的区别在于，客户端不会通过接口访问消息驱动bean。接口的介绍参见14.4节。不同于会话bean，消息驱动bean仅有一个bean类。

在一些方面，消息驱动bean与无状态会话bean有类似之处。

- 消息驱动bean的实例不保存特定客户端的数据或会话状态。
- 消息驱动bean的所有实例均“平等”，这使得EJB容器可以将消息分配给任意消息驱动bean的实例。容器可以以可复用对象池的方式并发处理消息流。
- 一个消息驱动bean可以处理来自不同客户端的消息。

消息驱动bean实例中的实例变量，可以在处理不同客户端消息的过程中记录一些状态，如JMS API连接、数据库连接或对某个企业bean对象的引用。

客户端组件无需定位消息驱动bean便可直接调用其中的方法。例如，客户端通过JMS访问消息驱动bean，此时消息接收方的消息驱动bean类是MessageListener。利用GlassFish服务器提供的资源，可以在部署应用时指定用于接收消息的消息驱动bean。

消息驱动bean有如下特征：

- 收到客户端消息后立即执行；
- 以异步方式调用；

- 生命周期相对较短;
- 不直接关联数据库中的共享数据,但可以访问并更新这些数据;
- 对事务敏感;
- 无状态。

当消息到达后,容器调用消息驱动bean的onMessage方法处理消息。onMessage方法通常将收到的消息转换为JMS 5种消息类型中的一种,并根据应用的业务逻辑进行处理。onMessage可以调用辅助方法,也可调用一个会话bean,以处理消息中的信息或将其存储至数据库。

消息可以在特定的事务环境中被发送至一个消息驱动bean,因此onMessage中的所有操作都可以在一个事务中。如果消息处理是回滚,则消息将被重新发送。更多信息参见第27章。

14.3.2 何时使用消息驱动bean

会话bean支持发送JMS消息并以同步的方式接收,但不支持异步接收。为了避免对服务器资源的无谓消耗,不要在服务器端的组件中使用阻塞的同步机制接收消息。通常JMS消息不应使用同步的方式收发。以异步的方式接收消息,请使用消息驱动bean。

14.4 访问企业 bean

注意 本节中讲述的内容仅适用于会话bean,不适用于消息驱动bean。由于消息驱动bean的编程模型不同,因而消息驱动bean没有接口,也没有定义客户端访问所需要的无接口视图。

客户端访问企业bean有两种方式,分别是利用无接口视图和业务接口。无接口视图对外暴露了企业bean的实现类对客户端提供的公有方法。使用企业bean无接口视图的客户端可以调用企业bean的实现类或其父类中的任何公有方法。业务接口是包含企业bean中业务方法的标准Java语言接口。

客户端只能通过bean的业务接口中定义的方法,或通过拥有无接口视图的企业bean中的公有方法访问会话bean。业务接口或无接口视图定义了企业bean的客户端视图。企业bean的其他特征(如方法实现、部署设置)对客户端来说是透明的。

设计良好的接口和无接口视图将简化Java EE应用的开发与维护。整洁的接口或无接口视图不仅可以对客户端隐藏EJB层的复杂性,而且可以实现影响隔离,从而使企业bean内部的变化对访问它的客户端不产生影响。例如,如果修改一个会话bean的业务方法的实现逻辑,无需随之修改客户端的代码。但如果修改了方法在接口中的定义,则必须随之调整客户端的代码。由此可知,精心的接口和无接口视图设计,对于降低企业bean的变更对客户端的影响是非常重要的。

会话bean可以有不止一个业务接口。会话bean应该(非必须)实现这些接口。

14.4.1 在客户端中使用企业bean

企业bean的客户端通过依赖注入（使用Java语言的注解功能）或JNDI查询（使用Java命名与目录接口语法查找企业bean的实例）的方式获得对企业bean实例的引用。

依赖注入是获得企业bean引用的最简便方法。运行在Java EE服务器管理环境中的客户端、JavaServer Faces的Web应用客户端、JAX-RS Web服务的客户端、其他企业bean的客户端或Java EE应用客户端，均支持基于javax.ejb.EJB注解的依赖注入。

运行于Java EE服务器管理环境之外的应用，如Java SE应用，必须通过显式的方式查找企业bean。JNDI支持定位Java EE组件的全局语义环境，以支持企业bean的显式查找。

可移植的JNDI语法

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

- JNDI命名空间之`java:global`，通过配置方式实现在JNDI中查找远程企业bean。JNDI的地址格式如下：

```
java:global[/application name]/module name/enterprise bean name[/interface name]
```

*application name*和*module name*在默认情况下是应用或模块文件去掉文件扩展名后的部分。*application name*仅当应用打包至EAR文件中时才必须要提供。*interface name*仅当企业bean实现了多个业务接口时才必须要提供。

- JNDI命名空间之`java:module`，在同一模块内部查找本地企业bean。使用`java:module`命名空间的JNDI地址格式如下：

```
java:module/enterprise bean name[/interface name]
```

*interface name*仅当企业bean实现多个业务接口时才必须要提供。

- JNDI命名空间之`java:app`，在同一应用中查找打包至其中的本地企业bean。也就是说，企业bean打包至包含多个Java EE模块的EAR文件中。使用`java:app`命名空间的JNDI地址格式如下：

```
java:app[/module name]/enterprise bean name[/interface name]
```

*module name*是可选的。*interface name*仅当企业bean实现多个业务接口时才必须要提供。

例如，将一个名为MyBean的企业bean打包至Web应用归档文件myApp.war中，模块名称是myApp。可以用`java:module/MyBean`方式配置其地址。使用`java:global`命名空间配置JNDI名称，即`java:global/myApp/MyBean`，可以达到相同效果。

14.4.2 远程还是本地访问

设计Java EE应用时，首先需要考虑并决定客户端以何种方式访问企业bean：本地、远程或者通过Web服务访问。

是否支持本地或远程的访问方式依赖于如下因素。

- 相关bean之间耦合性的松紧程度 紧耦合的bean之间相互依赖。例如，处理销售订单的

会话bean调用给客户发送订单确认电子邮件的会话bean，这两个bean之间紧密耦合。紧耦合bean适用于本地访问。由于它们构成一个逻辑单元，相互之间调用频繁，本地访问的方式将有利于提升性能。

- ❑ **客户端类型** 如果是一个应用客户端访问企业bean，应该支持远程访问。在生产环境中，这些客户端通常运行于与应用服务器（如GlassFish）不同的物理服务器上。如果企业bean的客户端是Web组件或其他企业bean，则访问类型依赖于组件的部署方式。
- ❑ **组件的分布性** Java EE应用的伸缩性源于其服务器端的组件可以分布式部署在不同的物理设备上。例如，在分布式应用中，运行Web组件的服务器可能与组件要访问的bean部署到的服务器不是同一个。在这种分布式场景中，企业bean需要支持远程访问。
- ❑ **性能** 由于网络延迟等因素，远程调用可能慢于本地调用。另一方面，如果将组件部署在不同服务器上，可以提升应用的整体性能。当然，这两种说法都比较宽泛。性能问题在不同的运行环境中会有差别。尽管如此，必须牢记的一点是，应用的设计会影响性能。

如果不确定选用哪种类型的企业bean访问方式，那就使用远程方式。这种方式将带来更大的灵活性。未来，可以将组件部署至不同的环境中，以满足不断增长的应用需求。

对于一个企业bean来说，既支持本地访问又支持远程访问的情况虽然不常见，但还是有可能的。在这种情况下，要么bean的业务接口必须通过@Remote或@Local注解显式地指定为一个业务接口，要么bean类必须通过@Remote或@Local注解显式地指定业务接口。同一个业务接口不能既是远程又是本地的。

14.4.3 本地客户端

本地客户端有如下特征：

- ❑ 必须与要访问的企业bean在同一个应用中；
- ❑ 可以是Web组件或其他企业bean；
- ❑ 对于本地客户端来说，需要知道企业bean的具体位置。

企业bean的无接口视图为本地视图。企业bean实现类中的公有方法将公开给访问企业bean的无接口视图的本地客户端。使用无接口视图的企业bean不需要实现业务接口。

本地业务接口定义了bean的业务方法和生命周期方法。如果bean的业务接口不用@Local或@Remote进行注解，且bean的类不通过@Local或@Remote指定接口，则业务接口默认为本地接口。

创建仅支持本地访问的企业bean，可以通过下述的一种方式实现（非必须）。

- ❑ 创建一个企业bean的实现类，但不实现业务接口。这意味着给客户端公开一个无接口视图。例如：

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

- ❑ 用@Local将接口注解为一个企业bean的业务接口。例如：

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

- 通过为bean类增加@Local注解指定接口并给出接口名称。例如：

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

1. 使用无接口视图访问本地企业bean

客户端通过依赖注入或JNDI查找的方式，访问一个公布了本地无接口视图的企业bean。

- 要通过依赖注入的方式获得对企业bean的无接口视图的引用，使用javax.ejb.EJB注解并指定企业bean的实现类：

```
@EJB
ExampleBean exampleBean;
```

- 要通过JNDI查找的方式获得对企业bean的无接口视图的引用，使用javax.naming.InitialContext接口的lookup方法：

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

对于使用无接口视图的企业bean，客户端不使用new操作符来获得企业bean的新实例。

2. 访问实现业务接口的本地企业bean

客户端通过依赖注入或JNDI查找的方式，访问实现了本地业务接口的企业bean。

- 要通过依赖注入的方式获得对企业bean的本地业务接口的引用，使用javax.ejb.EJB注解并指定企业bean的本地业务接口名称：

```
@EJB
Example example;
```

- 要通过JNDI查找的方式获得对企业bean的本地业务接口的引用，使用javax.naming.InitialContext接口的lookup方法：

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

14.4.4 远程客户端

企业bean的远程客户端有如下特征。

- 与它要访问的企业bean运行于不同的主机或Java虚拟机上。（并非必须要运行于不同的Java虚拟机上。）
- 可以是Web组件、应用客户端或其他企业bean。
- 对于远程客户端来说，无需知道企业bean的具体位置。
- 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无接口视图访问企业bean。

创建一个支持远程访问的企业bean，必须选择下述方式中的一种。

- 为企业bean的业务接口添加@Remote注解：

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

- 为bean的类添加@Remote注解，并指定一个或多个业务接口：

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

远程接口定义了特定bean的业务方法和生命周期方法。例如，名为BankAccountBean的远程接口可以有名为deposit和credit的业务方法。图14-1展示了接口如何控制企业bean的客户端视图。

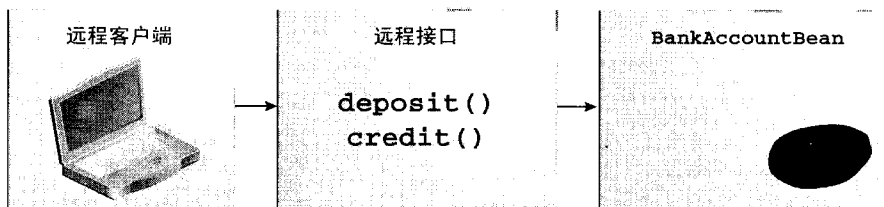


图14-1 支持远程访问的企业bean的接口

客户端通过依赖注入或JNDI查找的方式访问实现了远程业务接口的企业bean。

- 要通过依赖注入的方式获得对企业bean的远程业务接口的引用，使用javax.ejb.EJB注解并指定企业bean的远程业务接口名称：

```
@EJB
Example example;
```

- 要通过JNDI查找的方式获得对企业bean的远程业务接口的引用，使用javax.naming.InitialContext接口的lookup方法：

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

14.4.5 Web服务客户端

Web服务客户端可以通过两种方式访问Java EE应用。第一种方式，客户端可以访问通过JAX-WS创建的Web服务。（更多信息参见第12章。）第二种方式，Web服务客户端可以调用无状态会话bean的业务方法。消息驱动bean不能被Web服务客户端访问。

只要使用正确的协议（SOAP、HTTP、WSDL），任何Web服务客户端均可以访问无状态会话bean，无论客户端是不是用Java语言编写的。客户端甚至不知道服务是由何种技术实现的：无状态会话bean、JAX-WS或其他技术。除此之外，企业bean和Web组件也可以是Web服务的客户端。这种灵活性确保了Java EE应用与Web服务的集成。

Web服务客户端通过bean的Web服务端点实现类来访问无状态会话bean。在默认情况下，所有bean类中的公有方法均可以被Web服务客户端访问。@WebMethod注解可用于定制Web服务中方法的行为。如果@WebMethod用于注解bean类中的方法，那么仅有那些用@WebMethod注解的方法可

以被Web服务客户端访问。

代码示例参见16.3节。

14.4.6 方法的参数和方法的访问

访问方式会影响客户端调用bean方法的参数。接下来的内容不仅适用于方法参数，也适用于返回值。

1. 隔离

与本地调用方式相比较，远程调用时的参数更加具有隔离性。在远程调用中，客户端和bean操作的是参数对象的不同副本。如果客户端修改了对象的值，bean一侧副本的值不会修改。在客户端无意中修改数据时，这层隔离机制可以起到保护bean中数据的作用。

对于本地调用，客户端和bean可以修改同一个参数对象。通常来说，开发人员不应该依赖本地调用的这种特征来实现数据的传递。也许有一天，组件可能以分布式的方式部署至其他环境中，此时本地调用将被远程调用所代替。

对于远程客户端来说，Web服务客户端所操作的数据，仅是实现Web服务的bean所操作数据的副本。

2. 访问数据的粒度

由于远程调用通常慢于本地调用，因此远程方法的参数粒度相对较粗。粗粒度对象相较于细粒度对象有更大的数据量，因此访问的次数需要随之减少。同理，Web服务客户端调用的方法中的参数也应该是粗粒度的。

14.5 企业 bean 的内容

企业bean的开发过程需要提供如下文件。

- 企业bean类 实现企业bean的业务方法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暴露本地的无接口视图，业务接口可以不定义。
 - 辅助类 企业bean类需要的其他类，如异常和工具类。
- 将程序文件打包至EJB的JAR文件或Web应用的WAR文件中。

14.5.1 在EJB的JAR模块中打包企业bean

EJB的JAR文件是一种跨平台的程序包，可用于不同的应用中。

装配一个Java EE应用，需要打包一个或多个模块（如EJB的JAR文件）至一个封装应用的EAR文件中。当部署EAR（包含企业bean的EJB JAR文件）文件时，同时也就完成了将企业bean部署至应用服务器（如GlassFish）的过程。也可以不以EAR文件的形式部署EJB的JAR文件。图14-2展示了EJB的JAR文件结构。

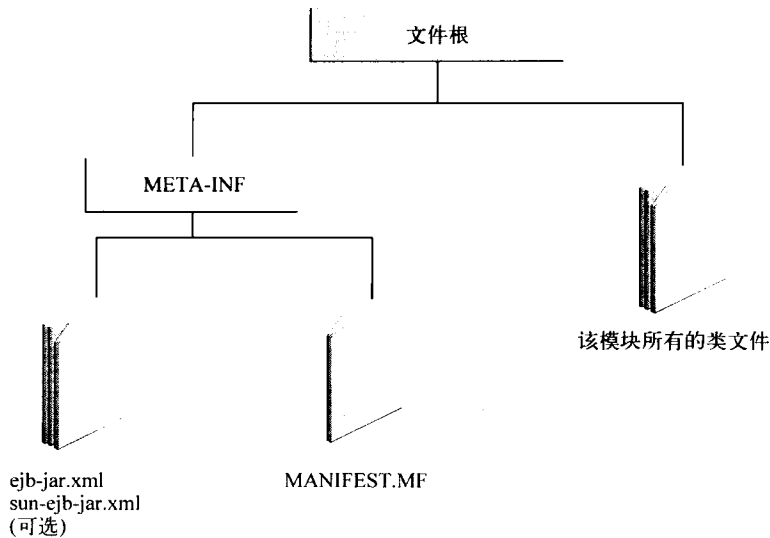


图14-2 企业bean的JAR文件结构

14.5.2 在WAR模块中打包企业bean

企业bean通常用于实现Web应用中的业务逻辑。在这种情形下,将Web应用的企业bean在WAR模块中打包可以简化部署过程和应用的集成。实现企业bean的Java类文件或JAR文件均可以打包至WAR模块中。

为了能够调用WAR模块中企业bean的类文件,需将这些类文件放置于WEB-INF/classes目录下。

为了能够调用WAR模块中包含企业bean的JAR文件,需将JAR文件放置于WAR模块的WEB-INF/lib目录下。

包含企业bean的WAR模块无需使用ejb-jar.xml部署描述文件。如果应用使用ejb-jar.xml,它必须位于WAR模块的WEB-INF目录下。

打包在WAR模块中,包含企业bean类的JAR文件不被视作EJB的JAR文件,即便是JAR文件遵从EJB的JAR文件格式。JAR文件中的企业bean,在语义上等同于位于WAR模块的WEB-INF/classes目录下的企业bean,且所有企业bean的环境命名空间均限定在WAR模块之内。

例如,假定一个Web应用由一个实现购物车功能的企业bean、一个实现信用卡处理功能的企业bean和一个Java servlet前端构成。实现购物车的bean公开一个本地的无接口视图,其代码如下:

```
package com.example.cart;

@Stateless
public class CartBean { ... }
```

实现信用卡处理功能的bean被打包至其JAR文件cc.jar中,并公开一个本地的无接口视图,其

代码如下：

```
package com.example.cc;

@Stateless
public class CreditCardBean { ... }
```

servlet（即com.example.web.StoreServlet）处理Web前端，并使用CartBean和CreditCardBean。

这个应用的WAR模块构成如下：

```
WEB-INF/classes/com/example/cart/CartBean.class
WEB-INF/classes/com/example/web/StoreServlet
WEB-INF/lib/cc.jar
WEB-INF/ejb-jar.xml
WEB-INF/web.xml
```

14.6 企业 bean 的命名规范

由于企业bean是由多个部分构成，遵从一个命名规范对应用来说非常有用。表14-2总结了本教程中示例bean的命名规范。

表14-2 企业bean命名规范

元 素	语 法	示 例
企业bean名	<i>nameBean</i>	AccountBean
企业bean类	<i>nameBean</i>	AccountBean
业务接口	<i>name</i>	Account

14.7 企业 bean 的生命周期

企业bean在其“一生”（或称生命周期）内经历多个阶段。每一种企业bean（有状态会话bean、无状态会话bean、单件会话bean或消息驱动bean）有着不同的生命周期。

以下对企业bean方法的介绍，将和随后两章里的代码保持一致。对于企业bean的初学者，可以跳过本节，先运行示例代码初步了解一下。

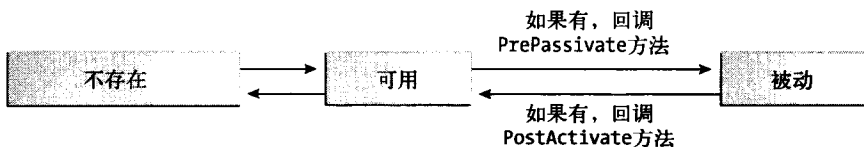
14.7.1 有状态会话bean的生命周期

图14-3展示了一个有状态会话bean在其生命周期内所经历的阶段。客户端通过获得对有状态会话bean的引用开始生命周期。如果有用@PostConstruct注解的方法，容器执行依赖注入，并调用这个方法。此时，bean中的业务方法便可被客户端调用。

在可用阶段，EJB容器通过将bean从内存转移至外存储器，来钝化bean。通常情况下，EJB容器使用最近最少使用（Least-Recently-Used）算法选择需要挂起的bean。如果bean有通过@PrePassivate注解的方法，EJB容器将在钝化该bean之前，立即调用这个方法。如果客户端调用

一个已钝化的bean中的业务方法，EJB容器将激活该bean，并且如果该bean有用@PostActivate注解的方法，EJB容器将调用这个方法，并将bean恢复至可用阶段。

- ① 创建
- ② 如果有，执行依赖注入
- ③ 如果有，回调PostConstruct方法
- ④ 如果有，执行init方法或ejbCreate<METHOD>方法



- ① 移除
- ② 如果有，回调PreDestroy方法

图14-3 有状态会话bean的生命周期

在生命周期结束时，客户端调用一个通过@Remove注解的方法，并且如果bean有通过@PreDestroy注解的方法，则EJB容器调用这个方法。bean的实例届时可被垃圾收集器回收。

只有一个生命周期方法的调用能够通过编码进行控制，即用@Remove注解的方法。其他所有在图14-3中介绍的方法都是由EJB容器调用。更多信息参见第28章。

14.7.2 无状态会话bean的生命周期

由于无状态会话bean从不被钝化，其生命周期仅有两个阶段：业务方法调用不存在阶段和可用阶段。图14-4展示了无状态会话bean的不同阶段。

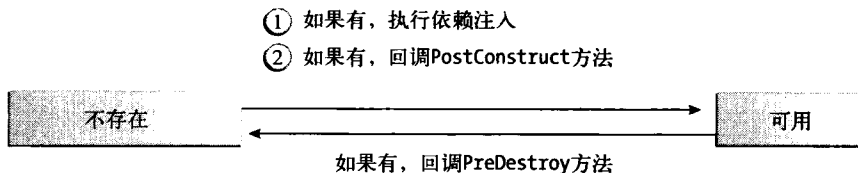


图14-4 无状态会话bean的生命周期

EJB容器通常会创建并维护一个无状态会话bean的池，并以此开始无状态会话bean的生命周期。容器执行所有依赖注入，并且调用使用@PostConstruct注解的方法（如果有的话）。此时，bean处于可用状态，其业务方法可以被客户端调用。

在生命周期结束时，如果有使用@PreDestroy注解的方法，EJB容器将调用这个方法。bean的实例届时可被垃圾收集器回收。

14.7.3 单件会话bean的生命周期

与无状态会话bean类似，单件会话bean从不被钝化，且其生命周期仅有两个阶段，即业务方法调用不存在阶段和可用阶段，如图14-5所示。

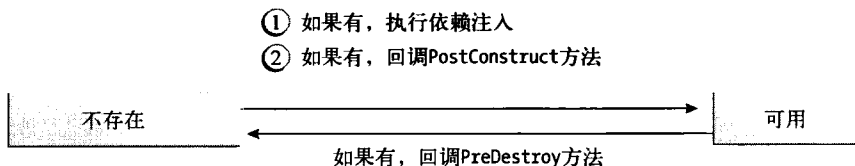


图14-5 单件会话bean的生命周期

EJB容器通过创建单件会话bean的实例初始化单件会话bean的生命周期。如果单件会话bean使用@Startup注解，则初始化的过程在应用部署时即发生。容器执行所有依赖注入，并且调用使用@PostConstruct注解的方法（如果有的话）。此时，单件会话bean处于可用状态，其业务方法可以被客户端调用。

在生命周期结束时，如果有使用@PreDestroy注解的方法，EJB容器将调用这个方法。单件会话bean的实例届时可被垃圾收集器回收。

14.7.4 消息驱动bean的生命周期

图14-6展示了消息驱动bean的生命周期的各个阶段。

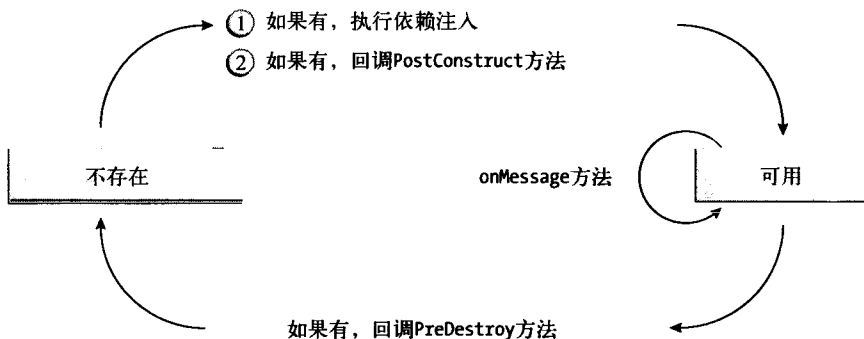


图14-6 消息驱动bean的生命周期

EJB容器通常创建一个消息驱动bean的实例池。对于每一个实例，EJB容器执行如下任务。

- (1) 如果消息驱动bean使用依赖注入，则容器在初始化实例前注入对象引用。
- (2) 如果有使用@PostConstruct注解的方法，容器调用该方法。

与无状态会话bean类似，消息驱动bean从不被钝化，且其生命周期仅有两个阶段：不存在和可用。

在生命周期结束的时候，如果有使用@PreDestroy注解的方法，容器会调用该方法。之后，bean的实例就处于可以被回收的状态。

14.8 有关企业 bean 的更多信息

有关EJB技术的更多信息，可以参考如下内容。

□ EJB 3.1规范：

<http://jcp.org/en/jsr/summary?id=318>

□ EJB官方网站：

<http://www.oracle.com/technetwork/java/ejb-141389.html>

第 15 章

企业bean应用初步

本章将介绍如何开发、部署、运行名为converter的Java EE应用。converter应用的功能是实现日元和欧元之间的汇率转换。converter应用由一个完成汇率转换的企业bean和两个客户端（应用客户端和Web客户端）构成。

下面是对本章所述操作过程的概述：

- (1) 创建企业bean——ConverterBean；
- (2) 创建Web客户端；
- (3) 将converter部署至服务器；
- (4) 用浏览器运行Web客户端。

在开始前，请确保已完成如下工作：

- ☐ 阅读第1章的内容；
- ☐ 熟悉企业bean（参考第14章）；
- ☐ 启动服务器（参考2.2节）。

本章内容

- ☐ 创建企业bean
- ☐ 修改Java EE应用

15.1 创建企业 bean

本示例中的企业bean是一个名为ConverterBean的无状态会话bean。ConverterBean的源代码位于 `tut-install/examples/ejb/converter/src/java/` 目录下。

创建ConverterBean需要执行如下操作：

- (1) 编写bean的实现类（源代码已提供）；
- (2) 编译源代码。

15.1.1 编写企业bean的类

本示例中企业bean的类名为ConverterBean。这个类实现了两个业务方法：`dollarToYen`和

yenToDollar。企业bean的类不实现业务接口，因而这个企业bean仅公开一个本地的无接口视图。这个企业bean类中的公有方法对客户端可见，客户端从而可以获得对ConverterBean的引用。ConverterBean类的源代码如下：

```
package com.sun.tutorial.javaee.ejb;

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

@Stateless
public class ConverterBean {
    private BigDecimal yenRate = new BigDecimal("115.3100");
    private BigDecimal euroRate = new BigDecimal("0.0071");

    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);
    }
}
```

请注意，在这个企业bean的类中使用了@Stateless注解。该注解将告知容器：ConverterBean是一个无状态会话bean。

15.1.2 创建converter的Web客户端

Web客户端在如下servlet类中定义：

tut-install/examples/ejb/converter/src/java/converter/web/ConverterServlet.java

Java servlet是响应HTTP请求的Web组件。

ConverterServlet类使用依赖注入获得对ConverterBean的引用。私有成员变量converterBean用javax.ejb.EJB进行注解，其类型为ConverterBean。ConverterBean公开一个本地的无接口视图，因此企业bean实现类就是成员变量的类型：

```
@WebServlet
public class ConverterServlet extends HttpServlet {
    @EJB
    ConverterBean converterBean;
    ...
}
```

当用户输入一个金额以转换成为日元和欧元时，应用可以从请求参数中获取该金额，随后调用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);
...
}
...
}
```

用户可以在随后的页面中查看转换后的结果。

15.1.3 构建、打包、部署及运行converter示例

现在可以编译企业bean的类文件(ConverterBean.java)以及servlet类文件(ConverterServlet.java), 并将编译后的类文件打包至WAR文件中。

▼ 在NetBeans IDE中构建、打包及部署converter示例

- (1) 在NetBeans IDE中, 选择File → Open Project。
 - (2) 在Open Project对话框中, 选择路径:
tut-install/examples/ejb/
 - (3) 选择converter目录。
 - (4) 勾选Open as Main Project和Open Required Projects复选框。
 - (5) 单击Open Project。
 - (6) 在Projects标签中, 右键单击converter项目, 并选择Deploy。
- 打开Web浏览器, 输入URL地址<http://localhost:8080/converter>。

▼ 用Ant构建、打包及部署converter示例

- (1) 在终端窗口中, 切换目录到:
tut-install/examples/ejb/converter/
- (2) 键入如下命令:

```
ant all
```

这个命令调用default任务, 编译企业bean和servlet的源文件, 并将编译出的类文件放置到项目的build目录(不是src目录)下。默认任务将项目打包至WAR模块(即converter.war)中。关于Ant工具的更多信息, 参见2.5节。

注意 当编译代码时, ant任务将引用位于classpath中的包含Java EE API的JAR文件。JAR文件位于GlassFish服务器安装目录的modules目录下。如果使用其他工具编译Java EE组件的源代码, 请确保在classpath中可以找到包含Java EE API的JAR文件。

▼ 运行converter示例

(1) 打开Web浏览器，输入如下URL：

`http://localhost:8080/converter`

图15-1显示了浏览器输出结果。

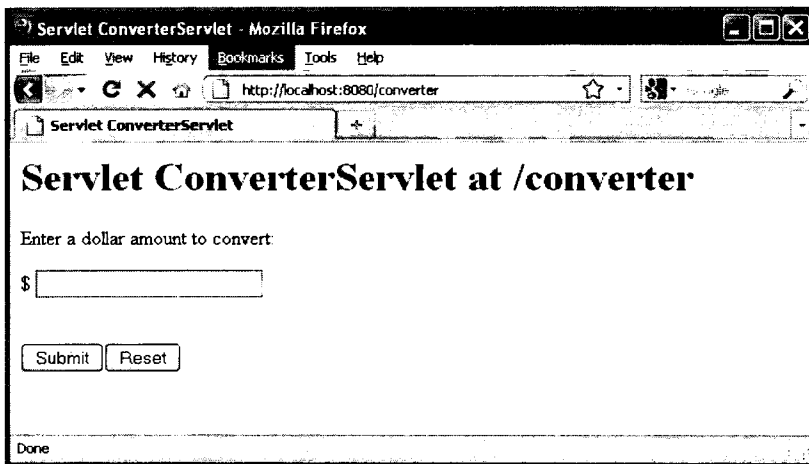


图15-1 converter的Web客户端

(2) 在输入框中输入100并单击Submit。

浏览器将显示第二个页面，展示转换后的值。

15.2 修改 Java EE 应用

GlassFish服务器支持迭代式开发流程。无论何时修改Java EE应用，都必须重新部署应用。

▼ 修改类文件

如果要修改企业bean中的类文件，开发人员需要更新类的代码，重新编译，并重新部署应用。例如，如果修改ConverterBean这个类中的DollarToYen方法，以更新汇率，需要执行如下操作。

修改ConverterServlet的过程与之相同。

(1) 修改ConverterBean.java并保存文件。

(2) 重新编译源文件。

❑ 在NetBeans IDE中重新编译ConverterBean.java，右键单击converter项目并选择Run。

这个过程将重新编译ConverterBean.java文件，在build目录中替换老的class文件，并将应用重新部署至GlassFish服务器。

□ 使用Ant重新编译ConverterBean.java。

(a) 在终端窗口中，切换目录到`tut-install/examples/ejb/converter/`。

(b) 键入如下命令：

```
ant all
```

这个命令将重新打包、部署及运行应用。

会话bean提供了一种简单而强大的方式用于将业务逻辑封装在应用中。它们可以被远程Java客户端、Web服务客户端以及运行在同一服务器上的组件访问。

第15章讲述了如何构建一个名为ConverterBean的无状态会话bean。本章将研究另外4个会话bean的代码。

- CartBean——可被远程客户端访问的有状态会话bean。
- CounterBean——单件会话bean。
- HelloServiceBean——实现Web服务的无状态会话bean。
- TimerSessionBean——设置定时器的无状态会话bean。

本章内容

- cart示例
- 单件会话bean示例counter
- Web服务示例helloservice
- 使用定时器服务
- 处理异常

16.1 cart 示例

Cart示例实现了在线书店的购物车,使用一个有状态会话bean实现对购物车的管理。这个bean的客户端可以将一本书添加至购物车、从购物车中移除一本书,或查看购物车中的内容。Cart示例需要如下代码:

- 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都需要一个远程业务接口。为了满足特定应用的需求,企业bean还需要一些辅助类。CartBean这个会话bean使用两个辅助类: BookException和IdVerifier (关于它们的内容,参见16.1.4节)。

本示例的源代码位于*tut-install/examples/ejb/cart*目录下。

16.1.1 业务接口

业务接口Cart是一个普通的Java接口，定义了bean类中实现的所有业务方法。如果bean类仅实现一个接口，则该接口被认作是业务接口。除非业务接口用javax.ejb.Remote注解，否则它就是一种本地接口。在这种情况下，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，业务接口必须通过@Local或@Remote以显式的方式注解，或者在bean类中通过@Local或@Remote指定。然而，当决定bean类是否要实现多个接口时，下列接口不算在其中：

- ❑ java.io.Serializable;
- ❑ java.io.Externalizable;
- ❑ 所有javax.ejb包中定义的接口。

业务接口Cart的源代码如下：

```
package com.sun.tutorial.javaee.ejb;

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();
}
```

16.1.2 会话 bean 类

本示例中的会话bean类名为CartBean。与其他有状态会话bean类似，CartBean必须满足下列要求：

- ❑ 类用@Stateful注解；
 - ❑ 类实现业务接口中定义的业务方法。
- 有状态会话bean还可以完成下列操作。
- ❑ 实现业务接口（普通Java接口）。实现bean的业务接口是一种良好的习惯。
 - ❑ 实现任意可选的生命周期回调方法，即那些用@PostConstruct、@PreDestroy、@PostActivate和@PrePassivate注解的方法。
 - ❑ 实现任意可选的用@Remote注解的方法。

CartBean的源代码如下：

```
package com.sun.tutorial.javaee.ejb;

import java.util.ArrayList;
```

```
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;

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

    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<String>();
    }

    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<String>();
    }

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

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

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

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

1. 生命周期回调方法

bean类中的方法可以通过如下注解声明为生命周期回调方法。

- ❑ **javax.annotation.PostConstruct** 容器完成bean的实例化，待所有依赖注入完成之后，且在企业bean中的第一个方法被调用之前，调用以@PostConstruct注解的方法。
- ❑ **javax.annotation.PreDestroy** 在@Remote注解的方法执行完成之后，且在容器清除企业bean实例之前，调用以@PreDestroy注解的方法。
- ❑ **javax.ejb.PostActivate** 容器在将bean从外部存储器移至内存，并使其处于活动状态后，调用以@PostActivate注解的方法。
- ❑ **javax.ejb.PrePassivate** 容器在把企业bean钝化之前，调用以@PrePassivate注解的方法。钝化意味着容器临时将bean从运行环境中移除并保存至外部存储器中。

生命周期回调方法不能有返回值及传入参数。

2. 业务方法

会话bean的主要任务是为客户端执行业务逻辑。客户端通过依赖注入或JNDI查找的方式获得对象的引用，并调用其中的方法。从客户端的角度来看，业务方法表面上是在本地运行，而实际上是在远端的会话bean中执行。下面的代码片段展示了CartClient程序如何调用业务方法：

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

通过下面的代码，CartBean实现了业务方法：

```
public void addBook(String title) {
    contents.addElement(title);
}

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

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

业务方法的签名必须遵循如下规则。

- ❑ 方法名称不能以ejb开头，以避免与EJB架构中的生命周期回调方法冲突。例如，业务方法不能命名为ejbCreate或ejbActivate。
- ❑ 访问控制修饰符必须设定为public。
- ❑ 如果bean允许通过远程业务接口访问，则方法的参数和返回类型必须是Java RMI (Remote Method Invocation, 远程方法调用) API中支持的类型。

□ 如果bean是Web服务端点，以@WebMethod注解的方法的参数和返回类型必须是JAX-WS支持的类型。

□ 方法的修饰符不能是static或final。

throws语句可以抛出为应用定义的异常。例如，如果书不在购物车中，removeBook方法将抛出一个BookException异常。

为了指出系统级问题，如不能连接数据库，某个业务方法应该抛出一个javax.ejb.EJBException。容器不会封装应用的异常，如BookException。由于EJBException是RuntimeException的子类，因此业务方法的throws语句无需抛出EJBException。

16.1.3 @Remove 方法

企业bean的客户端，通过调用有状态会话bean中以javax.ejb.Remove注解的方法移除bean的实例。容器将在@Remove方法执行结束后移除企业bean，无论方法执行是否成功。

在CartBean中，remove方法由@Remove注解：

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

16.1.4 辅助类

CartBean有两个辅助类：BookException和IdVerifier。removeBook方法抛出BookException类，而IdVerifier在create方法中验证customerId。辅助类可以位于企业bean类所在的EJB JAR文件，或打包企业bean的WAR文件中，也可以在包含企业bean的EJB JAR和WAR文件的EAR文件中。

16.1.5 构建、打包、部署及运行 cart 示例

现在可以编译远程接口（Cart.java）、本地接口（CartHome.java）、企业bean的类（CartBean.java）、客户端的类（CartClient.java）以及辅助类（BookException.java和IdVerifier.java）。步骤如下。

构建、打包、部署及运行cart应用可以用NetBeans IDE或Ant工具。

▼使用NetBeans IDE构建、打包、部署及运行cart示例

(1) 在NetBeans IDE中，选择File → Open Project。

(2) 在Open Project对话框中，选择路径：

tut-install/examples/ejb/

(3) 选择cart目录。

(4) 勾选Open as Main Project和Open Required Projects复选框。

(5) 单击Open Project。

(6) 在Projects标签中, 右键单击cart项目, 并选择Deploy。

这个步骤将构建并将应用打包至cart.ear中(位于*tut-install/examples/ejb/cart/dist/*目录下), 并将EAR文件部署至GlassFish服务器实例上。

(7) 运行cart应用客户端, 应选择Run → Run Main Project。

在Output窗格中, 可以看到应用客户端的输出:

```
...
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.
Java Result: 1
run-cart-app-client:
run-nb:
BUILD SUCCESSFUL (total time: 14 seconds)
```

▼使用Ant构建、打包、部署及运行cart示例

(1) 在终端窗口中, 切换目录至:

tut-install/examples/ejb/cart/

(2) 键入如下命令:

```
ant
```

这个命令调用default任务, 构建并打包应用至EAR文件中。cart.ear位于dist目录下。

(3) 键入如下命令

```
ant deploy
```

此处将cart.ear部署至GlassFish服务器上。

(4) 键入如下命令:

```
ant run
```

这个任务将获取应用客户端的JAR文件(即cartClient.jar), 并运行应用客户端。客户端的JAR文件(即cartClient.jar)包含了应用客户端的类、辅助类BookException以及业务接口Cart。

这个任务等同于执行如下命令:

```
appclient -client cartClient.jar
```

当运行客户端时, 应用客户端容器将注入应用客户端的类中声明的对组件的引用, 此处即对企业bean Cart的引用。

all任务

为了简化起见, 引入all任务以实现构建、打包、部署并运行应用这一系列任务。键入如下命令, 执行all任务即可:

```
ant all
```

16.2 单件会话 bean 示例 counter

counter 示例展示了如何创建一个单件会话 bean。

16.2.1 创建单件会话 bean

javax.ejb.Singleton 注解用于指定一个企业 bean 的实现类是单件会话 bean:

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

1. 初始化单件会话 bean

EJB 容器负责决定初始化单件会话 bean 实例的时机，除非单件会话 bean 的实现类使用了 javax.ejb.Startup 注解。在这种情况下（有时称为立即初始化，即 eager initialization），EJB 容器必须在应用启动时立即初始化单件会话 bean。在 EJB 容器转发来自客户端的对应用中任何一个企业 bean 的请求之前，需要首先初始化单件会话 bean。例如，这种机制允许单件会话 bean 为应用启动执行特定的任务。

下面的单件会话 bean 实现了对应用状态的保存，是一种立即初始化场景：

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

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

有时会使用多个单件会话 bean 初始化应用的数据，因此必须按照特定的顺序对这些 bean 进行初始化。在这种情况下，使用 javax.ejb.DependsOn 注解来声明单件会话 bean 的启动依赖关系。注解 @DependsOn 的 value 属性是一个或多个字符串，指定了目标单件会话 bean 的名字。如果在 @DependsOn 注解中指定多个依赖的单件会话 bean，其排列的顺序并不一定要和 EJB 容器初始化它们的顺序保持一致。

下面的单件会话 bean PrimaryBean，应该首先被初始化：

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

SecondaryBean 依赖 PrimaryBean:

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

这种声明确保 EJB 容器在初始化 SecondaryBean 之前，首先初始化 PrimaryBean。

下面的单件会话 bean TertiaryBean，依赖 PrimaryBean 和 SecondaryBean。


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

SecondaryBean 通过 @DependsOn 注解，显式地声明了其对 PrimaryBean 的依赖，因此在 PrimaryBean 之后初始化。在这种情况下，EJB 容器首先初始化 PrimaryBean，然后是 SecondaryBean，最后是 TertiaryBean。

然而，如果 SecondaryBean 没有显式地声明对 PrimaryBean 的依赖，EJB 容器可能先初始化 PrimaryBean，也可能先初始化 SecondaryBean。也就是说，EJB 容器可以按照 SecondaryBean → PrimaryBean → TertiaryBean 的顺序来初始化单件会话 bean。

2. 管理单件会话 bean 的并发访问

单件会话 bean 的设计目标是实现并发访问，即多个客户端可以同时访问会话 bean 的同一实例。客户端仅需要一个对单件会话 bean 的引用，即可调用其公开的任何业务方法，且无需担心其他客户端是否同时在访问相同单件实例的业务方法。

当创建一个单件会话 bean 时，可以以两种方式控制对单件会话 bean 的业务方法的并发访问：容器管理并发性和 bean 管理并发性。

javax.ejb.ConcurrencyManagement 注解用于指定单件 bean 的并发控制方法，即容器管理或 bean 管理。@ConcurrencyManagement 注解中的 type 属性必须设置为 javax.ejb.ConcurrencyManagementType.CONTAINER 或者 javax.ejb.ConcurrencyManagementType.BEAN。如果在单件会话 bean 的实现类中没有 @ConcurrencyManagement 注解，则 EJB 容器将默认使用容器管理的方式。

3. 容器管理并发性

如果单件 bean 设置为由容器管理并发性，则 EJB 容器管理客户端对单件 bean 中业务方法的访问。javax.ejb.Lock 注解和 javax.ejb.LockType 类型用于指定单件 bean 的业务方法或 @Timeout 方法的访问级别。

如果方法可以被并发访问或在多个客户端间共享，则使用 @Lock(READ) 注解业务方法或超时 (timeout) 方法。如果单件会话 bean 在被客户端访问时需要加锁（此时，其他客户端不可访问该 bean），则使用 @Lock(WRITE) 进行注解。通常情况下，客户端修改单件会话 bean 状态的方法需要用 @Lock(WRITE) 注解。

用 @Lock 注解单件类，说明所有业务方法和超时方法都将使用指定的锁定类型，除非用 @Lock 显式地声明方法级的锁定机制。如果单件类的声明没有用 @Lock 进行注解，则默认的锁定类型 @Lock(WRITE) 将应用于所有的业务方法和超时方法。

下面的示例展示了如何在单件类中使用 @ConcurrencyManagement、@Lock(READ) 和 @Lock(WRITE) 注解，实现容器管理并发性。

虽然在默认情况下，单件会话 bean 使用容器管理并发性，但可以在单件类定义中增加类级别的注解，即 @ConcurrencyManagement(CONTAINER)，以显式设置并发管理类型。

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

```

@Lock(READ)
public String getState() {
    return state;
}

@Lock(WRITE)
public void setState(String newState) {
    state = newState;
}
}

```

getState方法用@Lock(READ)进行注解，因而可以被多个客户端并发访问。然而，当setState方法被调用时，由于setState方法用@Lock(WRITE)进行注解，ExampleSingletonBean中的所有方法都将锁定到其他客户端。这可以防止两个不同的客户端同时修改ExampleSingletonBean中state变量的值。

下面的单件会话bean中的getData和getStatus方法的锁定类型为READ，而setStatus方法的锁定类型为WRITE。

```

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

    public String getData() {
        return data;
    }

    public String getStatus() {
        return status;
    }

    @Lock(WRITE)
    public void setStatus(String newStatus) {
        status = newStatus;
    }
}

```

如果方法的锁定类型为WRITE，客户端访问单件会话bean中方法的请求都将被阻塞，直至当前客户端完成方法的调用，或者访问超时。当访问超时出现时，EJB容器抛出javax.ejb.ConcurrentAccessTimeoutException异常。javax.ejb.AccessTimeout注解指定了访问超时的毫秒数。类级别的@AccessTimeout指定单件bean中所有方法的超时值，除非方法声明中用@AccessTimeout注解显式地覆写对默认超时的定义。

@AccessTimeout可以用于使用了@Lock(READ)和@Lock(WRITE)注解的方法。@AccessTimeout仅有一个必选属性value和一个可选属性unit。在默认情况下，value以毫秒为单位。要修改value的单位，使用java.util.concurrent.TimeUnit支持的常量类型：NANOSECONDS、MICROSECONDS、MILLISECONDS、SECONDS。

在下面的单件会话bean代码中，定义了默认的访问超时时长为120 000 ms，即2分钟。doTedious-Operation方法覆写了默认定义，将超时时间设为360 000 ms，即6分钟。

```

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

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

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

```

下面的单件会话bean的访问超时时长为60 s，在其声明时使用了TimeUnit.SECONDS常量。

```

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

```

4. bean管理并发性

使用bean管理并发性单件会话bean，支持对bean中所有业务方法和超时方法的并发访问。单件会话bean的开发人员负责确保单件会话bean的状态在所有客户端间的同步。对于使用bean管理并发性单件会话bean，开发人员在开发此类bean时，可以使用Java语言的同步原语，如synchronization和volatile，以避免并发访问时发生错误。

在单件会话bean的定义中增加类级别的@ConcurrencyManagement注解，实现bean管理并发性：

```

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

```

5. 处理单件会话bean中的错误

当单件会话bean在被EJB容器初始化时遇到错误，该单件实例将被销毁。

不同于其他企业bean，单件会话bean的实例一旦初始化，即使其业务方法和生命周期方法导致系统级异常，实例也不会被销毁。这种机制确保了应用在其整个生命周期内使用的是同一个单件实例。

16.2.2 counter 示例的架构

counter示例由一个名为CounterBean的单件会话bean和一个JSF Facelets的Web前端构成。

CounterBean是一个简单的单件会话bean，仅有一个名为getHits的方法，返回一个代表Web页面访问次数的整数。下面是CounterBean的代码：

```

package 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将CounterBean声明为一个单件会话bean。CounterBean使用本地、无接口视图。

CounterBean使用EJB容器默认的元数据值以简化单件实现类的编码。由于类定义中没有**@ConcurrencyManagement**注解，因此使用默认的容器管理并发访问。类和业务方法的声明中没有**@Lock**注解，因此仅有getHits这个业务方法使用默认的**@Lock(WRITE)**。

下面这个版本的CounterBean在功能上等同于前一个版本：

```

package 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前端由JSF托管bean构成，且Count.java文件中使用了Facelets的XHTML文件模板template.xhtml和template-client.xhtml。由JSF托管的Count类通过依赖注入的方式获得对CounterBean的引用。Count定义了名为hitCount的JavaBeans属性。当getHitCount方法被XHTML调用时，CounterBean的getHits方法被调用，并返回页面当前的点击量。

下面是托管Bean Count的类定义：

```

@ManagedBean
@SessionScoped
public class Count {
    @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template-client.xhtml文件用于生成Facelets视图，展示当前页面的点击量。template-client.xhtml使用表达式语言中的语句#{count.hitCount}访问Count这个托管bean的hitCount属性。下面是template-client.xhtml文件的内容：

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html">
    <body>

        This text above will not be displayed.

        <ui:composition template="/template.xhtml">

            This text will not be displayed.

            <ui:define name="title">
                This page has been accessed #{count.hitCount} time(s).
            </ui:define>

            This text will also not be displayed.

            <ui:define name="body">
                Hooray!
            </ui:define>

            This text will not be displayed.

        </ui:composition>

        This text below will also not be displayed.

    </body>
</html>

```

16.2.3 构建、打包、部署及运行 counter 示例

可以使用NetBeans IDE或Ant构建、打包、部署及运行counter示例。

▼使用NetBeans IDE构建、打包、部署及运行counter示例

- (1) 在NetBeans IDE中, 选择File → Open Project。
- (2) 在Open Project对话框中, 选择路径:
`tut-install/examples/ejb/`
- (3) 选择counter目录。
- (4) 勾选Open as Main Project复选框。
- (5) 单击Open Project。
- (6) 在Projects标签中, 右键单击counter项目, 并选择Run。
在浏览器中打开URL `http://localhost:8080/counter`, 显示点击量。
- (7) 单击浏览器的刷新按钮, 观察页面点击量的增加情况。

▼使用Ant构建、打包、部署及运行counter示例

- (1) 在终端窗口中, 切换目录至:
`tut-install/examples/ejb/counter`
- (2) 键入如下命令:

`ant all`
这个命令将构建并部署counter至GlassFish服务器。
- (3) 在Web浏览器中输入如下URL:
`http://localhost:8080/counter`
- (4) 单击浏览器的刷新按钮, 观察页面点击量的增加情况。

16.3 Web 服务示例 hello service

这个示例展示了一个简单的Web服务, 基于客户端发来的请求创建并返回应答信息。HelloServiceBean是一个无状态会话bean, 实现了方法sayHello。这个方法与12.1.5节中介绍的sayHello方法一样。

16.3.1 Web 服务端点的实现类

HelloServiceBean是端点实现类。端点实现类通常是开发基于企业bean的Web服务的主要部分。Web服务的端点实现类通常需要满足如下要求。

- ❑ 类需要用`javax.jws.WebService`或`javax.jws.WebServiceProvider`进行注解。
- ❑ 实现类可以显式地通过`@WebService`注解的`endpointInterface`元素引用一个SEI, 但这不是必须的。如果在`@WebService`中没有指定`endpointInterface`, 则实现类隐式引用SEI。
- ❑ 实现类的业务方法必须是公有的, 且不能是`static`或`final`类型。
- ❑ 暴露给Web服务客户端的业务方法必须使用`javax.jws.WebMethod`注解。

- ❑ 暴露给 Web 服务客户端的业务方法的参数和返回值必须与 JAXB 兼容，参见 <http://docs.oracle.com/javaee/5/tutorial/doc/bnazq.html#bnazs> 中的 JAXB 默认数据类型列表。
- ❑ 实现类不能声明为 `abstract` 或 `final`。
- ❑ 实现类必须有一个默认的公有构造函数。
- ❑ 端点类必须使用 `@Stateless` 注解。
- ❑ 实现类不能定义 `finalize` 方法。
- ❑ 实现类可以使用 `javax.annotation.PostConstruct` 或 `javax.annotation.PreDestroy` 注解来定义生命周期事件回调方法。

在实现类开始响应 Web 服务客户端之前，容器会调用 `@Post Construct` 方法。

在端点被从操作中移除之前，容器会调用 `@Pre Destroy` 方法。

16.3.2 无状态会话 bean 的实现类

`HelloServiceBean` 实现了用 `@WebMethod` 注解的 `sayHello` 方法。`HelloServiceBean` 的源代码如下：

```
package com.sun.tutorial.javaee.ejb;

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

@Stateless
@WebService
public class HelloServiceBean {
    private String message = "Hello, ";

    public void HelloServiceBean() {}

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

16.3.3 构建、打包、部署及测试 hello-service 示例

可以使用 NetBeans IDE 或 Ant 构建、打包及部署 hello-service 示例，然后使用管理控制台测试 Web 服务端点方法。

▼ 使用 NetBeans IDE 构建、打包及部署 hello-service 示例

- (1) 在 NetBeans IDE 中，选择 `File` → `Open Project`。
- (2) 在 `Open Project` 对话框中，选择路径：

`tut-install/examples/ejb/`

- (3) 选择helloservice目录。
- (4) 勾选Open as Main Project和Open Required Projects复选框。
- (5) 单击Open Project。
- (6) 在Projects标签中, 右键单击helloservice项目, 并选择Deploy。

这个步骤将构建并将应用打包至helloservice.ear中(这一文件位于*tut-install/examples/ejb/helloservice/dist*目录下), 并将这一EAR文件部署至GlassFish服务器上。

▼使用Ant构建、打包及部署helloservice示例

- (1) 在终端窗口中, 切换目录至:
tut-install/examples/ejb/helloservice/
- (2) 键入如下命令:

```
ant
```

这个命令调用default任务, 编译源文件并打包应用至JAR文件中。JAR文件位于*tut-install/examples/ejb/helloservice/dist*目录下, 文件名为helloservice.jar。

- (3) 键入如下命令, 部署helloservice:

```
ant deploy
```

一经部署, GlassFish服务器将自动生成用于Web服务调用的辅助文件, 包括WSDL文件。

▼在无客户端情况下测试Web服务

GlassFish服务器的管理控制台可以用来测试Web服务端点中的方法。参考如下过程, 测试HelloServiceBean中的sayHello方法。

- (1) 在Web浏览器中输入如下URL, 进入管理控制台:
http://localhost:4848/
- (2) 在管理控制台的左侧窗格中选择Applications结点。
- (3) 在Applications表中选择helloservice。
- (4) 在Modules and Components列表中, 单击View Endpoint。
- (5) 在Web Service Endpoint Information页面单击Tester链接:
/HelloServiceBeanService/HelloServiceBean?Tester
- (6) tester页面在浏览器窗口或标签中打开。
- (7) 在Methods下, 输入一个名字, 作为sayHello方法的参数。
- (8) 单击sayHello按钮。

sayHello的方法调用页面打开。在此页面也可查看端点返回的应答信息。

16.4 使用定时器服务

实现工作流的应用通常需要定时的消息通知。企业bean容器的定时器服务可以为除有状态会

话bean以外的各种企业bean设置定时通知。可以根据日历、特定的时间点、一个时间段或一定的时间间隔设定一个定时通知。例如，可以设定一个定时器，使之在5月23日10:30触发，或30天之后触发，或每隔12小时触发。

企业bean定时器有两种形式，即可编程定时器和自动定时器。可编程定时器通过显式调用一个TimerService接口中的定时器创建方法进行设置。当一个包含java.ejb.Schedule或java.ejb.Schedules注解的企业bean部署成功之后，自动定时器会被自动创建。

16.4.1 创建日历型定时器表达式

定时器可以根据日历设定，其表达式语法与UNIX的cron命令类似。可编程定时器和自动定时器均可使用日历型定时器表达式。表16-1展示了日历型定时器的属性。

表16-1 日历型定时器的属性

属 性	描 述	可接受的值	默 认 值	举 例
second	秒	0~59	0	second="30"
minute	分钟	0~59	0	minute="15"
hour	小时	0~23	0	hour="13"
dayOfWeek	星期	0~7（0和7均指周日） Sun、Mon、Tue、Wed、Thu、Fri、Sat	*	dayOfWeek="3" dayOfWeek="Mon"
dayOfMonth	日期	1~31 -7~-1（负数代表距离月末的第几天） Last [1st, 2nd, 3rd, 4th, 5th, Last][Sun, Mon, Tue, Wed, Thu, Fri, Sat]	*	dayOfMonth="15" dayOfMonth="-3" dayOfMonth="Last" dayOfMonth="2nd Fri"
month	月	1~12 Jan、Feb、Mar、Apr、May、Jun、Jul、Aug、Sep、Oct、Nov、Dev	*	month="7" Month="July"
year	年	4位日历年	*	year="2010"

1. 在日历表达式中指定多个值
- 在日历表达式中可以指定多个值，下面将详细介绍。
2. 在日历表达式中使用星号
- 将属性的值设定为星号，意味着属性可接受的所有值。
- 下面的表达式代表每分钟：

minute="*"

下面的表达式代表一周的每一天：

dayOfWeek="*"

3. 指定值的列表

可以使用逗号(,)分隔的方式,为属性指定多个值。可以在列表中指定值的范围。在这种情况下不支持星号和间隔。

列表中的重复值将被忽略。

下面的表达式设定周二和周四:

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

下面的表达式代表4:00 am、9:00 am ~ 5:00 pm的每个小时,以及10:00 pm:

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

4. 指定值的范围

使用破折号(-)为属性指定一个取值范围。取值的范围不能出现星号、列表或间隔。**x-x**形式的范围设定等同于单值表达式**x**。用**x-y**进行范围设定,当**x**大于**y**时,其结果等同于“**x**减去最大值,最小值减去**y**”。也就是说,表达式以**x**为开始,至可接受值的最大值,再从最小值开始,以**y**为结束。

下面的表达式代表9:00 am ~ 5:00 pm:

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

下面的表达式代表周五至周一:

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

下面的表达式代表从一个月的第25天开始至月末,并从下个月的第1天至第5天:

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

它等同于下面的表达式:

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

5. 指定间隔

正斜杠(/)将属性指定为从某个时间点开始,以及**N**秒、**N**分钟或**N**小时的时间间隔。对于**x/y**形式的表达式,**x**代表开始时间点,**y**代表间隔。星号可以用在**x**的所在位置,其含义等同于将**x**设为0。

间隔仅用于设定**second**、**minute**和**hour**属性。

下面的表达式代表一个小时内每隔10分钟:

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

它等同于:

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

下面的表达式代表从中午开始每隔两小时:

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

16.4.2 可编程定时器

当可编程定时器超时（触发）时，容器调用bean的实现类中用@Timeout注解的方法。@Timeout注解的方法包含业务逻辑，可以处理定时任务。

1. @Timeout方法

企业bean中用@Timeout注解的方法不能有返回值（即一定要声明为void类型），但可以将javax.ejb.Timer类型的对象作为唯一的参数（参数非必须）。这类方法可以不抛出应用异常。

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

2. 创建可编程定时器

bean通过调用TimerService接口中的create方法创建定时器。这种方法可以创建单次型、间隔型和日历型定时器。

对于单次型、间隔型定时器，定时器的超时时间可以通过时长或绝对时间的方式设置。时长以毫秒为单位，到期后，超时事件将被触发。为了指定一个绝对时间，创建一个java.util.Date对象，并将其传递给TimerService.createSingleActionTimer方法或TimerService.createTimer方法。

下面的代码设置一个在1分钟（6000 ms）后超时的可编程定时器：

```
long duration = 6000;
Timer timer = timerService.createSingleActionTimer(duration, new TimerConfig());
```

下面的代码设定一个在2010年5月1日12:05 pm超时的可编程定时器，其超时时间通过java.util.Date设定：

```
SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy 'at' HH:mm");
Date date = formatter.parse("05/01/2010 at 12:05");
Timer timer = timerService.createSingleActionTimer(date, new TimerConfig());
```

对于日历型定时器，定时器在超时时生成一个javax.ejb.ScheduleExpression对象，并作为参数传递给TimerService.createCalendarTimer方法。ScheduleExpression类代表日历型定时器的表达式，且有对应16.4.1节中所述属性的方法。

下面的代码使用ScheduleExpression辅助类，创建一个日历型可编程定时器：

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

关于方法签名的更多信息，参考TimerService的API文档，地址为<http://docs.oracle.com/javase/6/api/javax/ejb/TimerService.html>。

16.4.7节中介绍的bean，通过如下方式创建一个定时器：

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

在timersession示例中，客户端调用一个业务方法，而此方法调用createTimer。

定时器在默认情况下具有持久性。如果服务器停机或崩溃，持久性的定时器将会被保存，并在服务器重启后恢复。如果持久性的定时器在服务器停机时超时，容器将在服务器重启后调用@Timeout注解的方法。

通过调用TimerConfig.setPersistent(false)方法，并将TimerConfig对象传递给定时器的创建方法，可将可编程定时器设定为非持久性。

createTimer的Date和long型参数以毫秒为时间单位。由于定时器服务不是为实时应用准备的，因此@Timeout方法的调用无法精确到毫秒级。定时器服务通常用于以小时、天或更长时间单位的业务应用。

16.4.3 自动定时器

当企业bean包含用@Schedule或@Schedules注解的方法，在其部署时，EJB容器创建自动定时器。企业bean可以有多个自动超时的方法，而可编程定时器则不同，它在企业bean的类中，仅允许一个用@Timeout注解的方法。

自动定时器可以通过注解的方式进行配置，也可以通过ejb-jar.xml部署描述文件进行配置。

根据@Schedule的属性中设定的时间计划，为企业bean中的方法增加@Schedule注解，可使其成为一个超时方法。

@Schedule注解有与日历表达式相对应的元素(详见16.4.1节)以及persistent、info、timezone元素。

persistent元素是可选的，可以接收一个布尔类型的值，以设定在服务器重启或崩溃时是否保留自动定时器。默认情况下，所有自动定时器都具有持久性。

timezone元素是可选的，它指定自动定时器关联的时区。如果设置该元素，所有的自动定时器将按照设定的时区工作，而无视EJB容器所在的时区。默认情况下，所有的自动定时器设定为服务器所在的默认时区。

info元素是可选的，用于设定定时器的描述信息。定时器的描述信息可以通过Timer.getInfo方法获得。

下面的超时方法使用@Schedule，将定时器设定为每周日的午夜触发：

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

@Schedules注解用于为给定的超时方法设定多个日历型定时器表达式。

下面的超时方法使用@Schedules注解设定多个日历型定时器表达式。第一个表达式代表定时器在每月的最后一天超时。第二个表达式表示定时器在每周五的11:00 pm超时。

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

16.4.4 取消及保存定时器

下列事件可以取消定时器：

- 当单次型定时器超时的时候，EJB容器调用与之关联的超时方法，并取消定时器；
- 当bean调用Timer接口的cancel方法时，容器取消定时器。

当调用一个已取消的定时器中的方法时，容器抛出`javax.ejb.NoSuchObjectLocalException`异常。

保存Timer对象以备将来使用，可以通过调用它的`getHandle`方法，将TimerHandle对象保存至数据库中来实现。（TimerHandle对象可以序列化。）从数据库中获得定时器句柄，并调用句柄中的`getTimer`方法，可以重新实例化Timer对象。TimerHandle对象不能作为远程或Web服务接口中所定义方法的参数进行传递。换言之，远程客户端或Web服务客户端不能访问bean的TimerHandle对象。然而，本地客户端不受此限制。

16.4.5 获得定时器的信息

除了定义cancel和getHandle方法，Timer接口定义了如下方法，以获得定时器的信息：

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

`getInfo`方法返回`createTimer`方法的最后一个参数传递的对象。例如，在前述示例程序的`createTimer`代码中，传入的参数是内容为created timer的String对象。

调用TimerService接口的`getTimers`方法可以获得bean的所有活动定时器。`getTimers`方法返回一个Timer对象的集合。

16.4.6 事务和定时器

企业bean通常在事务中创建定时器。如果这个事务回滚，则定时器的创建也将回滚。与之相似的是，如果bean在回滚的事务中取消一个定时器，则定时器的取消也要回滚。在这种情况下，定时器的时长将重置，就如同“取消”这一动作从未发生。

在使用容器管理事务的bean中，@Timeout方法通常有Required或RequiredNew事务属性以保持事务的完整性。依赖这些属性，EJB容器在调用@Timeout方法前开始一个新的事务。如果事务回滚，容器将至少再重新调用@Timeout方法一次。

16.4.7 timersession 示例

本示例的源代码位于`tut-install/examples/ejb/timersession/src/java/`目录。

TimerSessionBean是一个单件会话bean，展示了如何设置可编程定时器和自动定时器。在随后展示的TimerSessionBean的代码中，`setTimer`和`@Timeout`方法用于设定一个可编程定时器。当容器创建一个bean时，会为之注入一个TimerService实例。由于它是业务方法，`setTimer`对

TimerSessionBean的本地无接口视图可见，且可被客户端调用。在本例中，客户端将30 000 ms的间隔作为参数，调用setTimer方法。setTimer方法通过调用TimerService的createTimer方法创建新的定时器。现在，定时器已设置，EJB容器将在定时器超时时（大约在30 s之后）调用TimerSessionBean的programmaticTimeout方法。

```
...
    public void setTimer(long intervalDuration) {
        logger.info("Setting a programmatic timeout for " +
            intervalDuration + " milliseconds from now.");
        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，即automaticTimeout。自动定时器设为每3分钟触发一次，并在@Schedule中使用日历型定时器表达式。

```
...
    @Schedule(minute="*/3", hour="*")
    public void automaticTimeout() {
        this.setLastAutomaticTimeout(new Date());
        logger.info("Automatic timeout occured");
    }
...

```

TimerSessionBean有两个业务方法：getLastProgrammaticTimeout和getLastAutomaticTimeout。客户端调用这些方法可以分别得到可编程定时器和自动定时器最近一次被激活的日期和时间。

下面是TimerSessionBean类的源代码：

```
package timersession.ejb;

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

@Singleton
public class TimerSessionBean {
    @Resource
    TimerService timerService;

    private Date lastProgrammaticTimeout;
    private Date lastAutomaticTimeout;

    private Logger logger = Logger.getLogger(
        "com.sun.tutorial.javaee.ejb.timersession.TimerSessionBean");

    public void setTimer(long intervalDuration) {

```

```

        logger.info("Setting a programmatic timeout for "
            + intervalDuration + " milliseconds from now.");
        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="*/3", hour="*")
    public void automaticTimeout() {
        this.setLastAutomaticTimeout(new Date());
        logger.info("Automatic timeout occurred");
    }

    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;
    }
}

```

注意 GlassFish服务器有一个默认的最小超时时间，为1000 ms (1 s)。如果需要将超时时间设置为小于1000 ms，修改位于文件`domain-dir/config/domain.xml`中`minimum-delivery interval-in-millis`元素的值。受到虚拟机的限制，`minimum-delivery interval-in-millis`可以设置的实际最小值为10 ms。

16.4.8 构建、打包、部署及运行 timersession 示例

可以使用NetBeans IDE或Ant构建、打包、部署及运行timersession示例。

▼使用NetBeans IDE构建、打包、部署及运行timersession示例

(1) 在NetBeans IDE中, 选择File → Open Project。

(2) 在Open Project对话框中, 选择路径:

tut-install/examples/ejb/

(3) 选择timersession目录。

(4) 勾选Open as Main Project复选框。

(5) 单击Open Project。

(6) 选择Run → Run Main Project。

这个步骤将构建并打包应用至timersession.war中(该文件位于*tut-install/examples/ejb/timersession/dist/*目录中), 将这个WAR文件部署至GlassFish服务器实例上, 并运行Web客户端。

▼使用Ant构建、打包、部署及运行timersession示例

(1) 在终端窗口中, 切换目录至:

tut-install/examples/ejb/timersession/

(2) 键入如下命令:

```
ant build
```

这个命令运行default任务, 编译源文件, 将应用打包成timersession.war, 并放置于*tut-install/examples/ejb/timersession/dist/*目录下。

(3) 键入如下命令来部署应用:

```
ant deploy
```

▼运行Web客户端

(1) 在Web浏览器中打开<http://localhost:8080/timersession>。

(2) 单击Set Timer按钮设置一个可编程定时器。

(3) 等待一段时间后, 单击浏览器的刷新按钮。

可以看到可编程定时器和自动定时器最近一次超时的日期和时间。

可以打开位于*domain-dir/server/logs/*目录下的server.log文件以查看超时发生时记录的日志。

16.5 处理异常

企业bean抛出的异常分为两类: 系统级异常和应用级异常。

系统级异常表明支撑应用的服务发生了错误。例如, 无法建立到外部资源的连接, 或无法找到注入的资源。如果遇到系统级错误, 企业bean将抛出javax.ejb.EJBException。由于EJBException是RuntimeException的子类, 因而无需在方法声明中使用throws语句。如果系统级异常被抛出, EJB容器可能销毁bean的实例。因此, 系统级异常不能被bean的客户端程序处理, 此时需要系统

管理员进行干预。

应用级异常表明企业bean的业务逻辑在运行中遇到了错误。应用级异常通常是自定义的异常，如CartBean示例的业务方法抛出的BookException。当企业bean抛出一个应用级异常，容器不会把它封装至其他异常中。客户端应该能够处理它所接收到的任何一种应用异常。

如果系统级异常发生在事务内，EJB容器会回滚该事务。然而，如果应用级异常是在事务内抛出的，容器并不回滚该事务。

Part 5

第五部分

Java EE 平台的上下文与依赖注入

第五部分介绍 Java EE 平台的上下文与依赖注入。

本 部 分 内 容

- 第 17 章 Java EE 平台的上下文与依赖注入入门
- 第 18 章 运行简单的上下文与依赖注入示例

Java EE平台的上下文 与依赖注入入门

CDI (Context and Dependency Injection, 上下文与依赖注入) 是Java EE 6平台的特征之一, 它将Java EE 6平台的事务层与Web层衔接在一起。CDI是一组可以组合使用的服务, 简化了开发人员在Web应用中使用企业bean及JSF技术的过程。CDI以使用有状态的对象为设计目标, 具有更为广泛的应用领域, 使得开发人员可以以更大的灵活性集成不同类型的组件, 并且保持组件间的松耦合性和类型安全。

在被JSR 299定义之前, CDI曾被称为Web Beans。CDI使用的相关规范包括:

- JSR 330, Java依赖注入;
- 托管bean规范, 是Java EE 6平台规范 (JSR 316) 的一个分支。

本章内容

- CDI概述
- 关于bean
- 关于托管bean
- 可注入对象bean
- 使用限定词
- 注入bean
- 使用作用域
- 为bean设定EL名称
- 增加存取方法
- 在Facelets页面中使用托管bean
- 使用Producer方法注入对象
- 配置CDI应用
- 有关CDI的更多信息

17.1 CDI 概述

CDI提供的基础服务包括：

- 上下文 具备绑定功能，即将有状态组件的生命周期和交互行为绑定至预定义且可扩展的生命周期上下文中。
- 依赖注入 能够以类型安全的方式将组件注入至应用中，可以在部署阶段选择要注入特定接口的那个实现。

除此之外，CDI还提供下列服务：

- 与EL集成，允许在JSF或JSP页面中直接使用任何组件；
- 可以修饰已注入的组件；
- 使用类型安全的拦截器绑定将拦截器与组件相关联；
- 事件通知模型；
- 除了Java Servlet规范定义的3种标准作用域（请求、会话和应用），另外增加Web会话域；
- 完整的SPI（Service Provider Interface，服务提供者接口），允许第三方框架无缝集成至Java EE 6平台中。

CDI主要应用于松耦合的场景中。在这种场景中，CDI实现以下任务。

- 通过预定义类型和限定词解耦服务器与客户端，从而支持服务器端实现的多样化。
- 通过如下方式实现相互协作的组件生命周期期间的解耦：
 - 借助于自动生命周期管理，使组件运行于特定的上下文中；
 - 允许有状态组件通过单纯的消息传递像服务一样交互。
- 通过事件将消息的产生和使用完全解耦。
- 通过Java EE的拦截器实现正交解耦。

除了松耦合特性之外，CDI通过如下方式实现强类型：

- 淘汰通过字符名称查找变量的方式，使编译器能够检测到录入错误；
- 支持以注解的方式声明应用运行的一切要素，最大程度地降低了对XML部署描述文件的依赖，从而可以更容易地在开发阶段使用工具检查代码并轻松理解依赖关系。

17.2 关于 bean

CDI在其他Java技术（如JavaBeans、EJB技术）的基础上重新定义了bean的概念。在CDI中，bean是定义应用的状态和业务逻辑的上下文对象的来源。如果Java EE组件实例的生命周期由容器管理，且该容器符合CDI规范中定义的生命周期上下文模型，则组件可视为bean。

更确切地说，bean有如下属性：

- 一组（非空）bean类型；
- 一组（非空）限定词（参见17.5节）；
- 作用域（参见17.7节）；

- bean的EL名称（可选，参见17.8节）；
- 一组拦截器绑定；
- bean的实现。

bean类型对客户端可见。几乎任何一种Java类型均可成为bean类型。

- bean类型可以是接口、具体类或抽象类，而且可以声明为final或拥有final方法。
- bean类型可以参数化，即类型参数和类型变量。
- bean类型可以是数组。当数组中的元素类型相同时，两个数组的类型才可认为相同。
- bean类型可以是基本数据类型。基本数据类型可视为与其在java.lang中定义的对包装类类型相同。
- bean类型可以是raw。

17.3 关于托管 bean

托管bean通过Java类来实现，这个类称为bean的类。若一个顶级（top-level）Java类被其他任何一种Java EE技术规范（如JSF技术规范）定义为托管bean，或满足如下所有条件，则这个Java类可称为托管bean。

- 不是非静态内嵌类。
- 是一个具体类或者是一个使用了@Decorator注解的类。
- 没有使用定义EJB组件的注解，或者在ejb-jar.xml文件中没有声明为EJB bean的类。
- 有合适的构造方法。也就是说，满足如下条件之一：
 - 这个类有一个无参数的构造方法；
 - 这个类有一个使用了@Inject注解的构造方法。

定义一个托管bean无需特别声明（比如无需为类定义增加注解）。

17.4 可注入对象 bean

注入的概念早已成为Java技术的一部分。从Java EE 5平台开始，利用注解就可以将资源或其他类型的对象注入容器管理的bean中。有了CDI，人们则可以注入更多类型的对象，并可将这些对象注入至非容器托管的对象中。

可以注入下列类型的对象。

- （几乎）任何Java类。
- 会话bean。
- Java EE资源：数据源、Java消息服务（Java Message Service）主题、队列、连接工厂等。
- 持久化上下文（JPA中的EntityManager对象）。
- Producer字段。
- Producer方法返回的对象。

- Web 服务的引用。
- 远程企业bean的引用。

例如，假定创建一个简单的Java类，该类有一个返回字符串的方法：

```
package greetings;

public class Greeting {
    public String greet(String name) {
        return "Hello, " + name + ".";
    }
}
```

这个类可以视作一个bean，我们可以把它注入到其他类中。这个bean目前还没有公开给EL。17.8节将介绍如何使一个bean可以被EL访问。

17.5 使用限定词

可以使用限定词（qualifier）为特定的bean类型提供不同的实现。限定词是可以应用于bean中的注解。限定词类型是通过@Target({METHOD, FIELD, PARAMETER, TYPE})和@Retention(RUNTIME)定义的Java注解。

例如，可以声明一个@Informal的限定词类型，并将其应用到扩展了Greeting的类上。声明限定词的类型需要使用下面的代码：

```
package greetings;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}
```

接下来可以定义一个bean类，用以扩展Greeting类并使用这个限定词：

```
package greetings;

@Informal
public class InformalGreeting extends Greeting {
    public String greet(String name) {
        return "Hi, " + name + "!";
    }
}
```

bean的上述两种实现均可以在应用中使用。

如果在定义bean的时候不使用限定词，则bean会自动有一个限定词@Default。未使用注解的

Greeting类可以以如下方式声明：

```
package greetings;

import javax.enterprise.inject.Default;

@Default
public class Greeting {
    public String greet(String name) {
        return "Hello, " + name + ".";
    }
}
```

17.6 注入 bean

为了使用创建的bean，可以将其注入到其他bean中，从而使之可以被应用（如JSF应用）使用。例如，创建一个名为Printer的bean，并将前面介绍的Greeting bean注入其中：

```
import javax.inject.Inject;

public class Printer {

    @Inject Greeting greeting;
    ...
}
```

这段代码将用@Default注解的Greeting实现注入到bean中。下面的代码将用@Informal注解的类注入其中：

```
import javax.inject.Inject;

public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

本例中bean的内容尚不完整。除了需要理解作用域的使用，还要注意：对于JSF应用，bean还需要支持EL访问。

17.7 使用作用域

对于需要将bean注入另一个bean的Web应用来说，在用户与应用交互的过程中，bean需要保存状态。我们可以通过为bean指定一个作用域来定义其状态。可以为对象指定表17-1中所列的任何一种作用域。当然，作用域的选取取决于如何使用它。

表17-1 作用域

作用域	注解	持续时间
请求	@RequestScoped	在一个HTTP请求中用户与Web应用的交互
会话（session）	@SessionScoped	在多个HTTP请求中用户与Web应用的交互
应用	@ApplicationScoped	在Web应用中的所有用户交互间的共享状态

(续)

作用域	注解	持续时间
依赖	@Dependent	在未指明作用域时的默认值。意味着对象仅为一个客户端 (bean) 服务, 并且与客户端 (bean) 有着相同的生命周期
对话 (conversation)	@ConversationScoped	对于用户与JSF应用之间的交互, 在开发人员可控制的清晰范围之内, 作用域扩展至JSF生命周期的多次调用。所有长时间运行的对话仅限于特定的HTTP servlet会话中, 不能跨越会话的边界

17

前3个作用域在JSR 299和JSF API中均有定义。后两个作用域仅在JSR 299中定义。

定义并实现定制化的作用域是一个高阶主题。定制的作用域可用于实现并扩展CDI规范的场合。

作用域为对象提供了一个预定义的生命周期上下文。此类对象可以在需要的时候自动创建, 也可以在创建它的上下文终止时自动销毁。除此之外, 此类对象的状态可以在任何运行于相同上下文中的客户端间共享。

有些Java EE组件, 如servlet、企业bean以及JavaBeans组件, 没有默认的预定义作用域。下面这些组件也是如此。

- ❑ 单件bean, 如EJB单件bean, 其状态在所有客户端间共享。
- ❑ 无状态对象, 如servlet和无状态会话bean, 不包含客户端可见的状态。
- ❑ 必须被客户端显式创建和销毁的对象, 如JavaBeans组件和有状态会话bean, 其状态只能以显式的方式在客户端间传递。

然而, 如果创建一个托管的Java EE组件 (bean), 它将成为一个有作用域的对象, 并在预定义的生命周期上下文内存在。

Web应用Printer使用简单的请求应答机制, 因此这一托管的bean可以通过如下方式注解:

```
import javax.inject.Inject;
import javax.enterprise.context.RequestScoped;

@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

对于作用域为会话、应用或对话的bean, 必须定义为可序列化, 但对于作用域为请求的bean, 则无此要求。

17.8 为 bean 设定 EL 名称

为了使bean可以通过EL访问, 可以使用@Named这个内建限定词:

```
import javax.inject.Inject;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
```

```

@Named
@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;
    ...

```

`@Named`限定词支持通过bean的名称(其首字母小写)访问bean。例如,Facelets页面通过printer引用bean。

可以为`@Named`指定一个参数,以使用一个非默认名称:

```
@Named("MyPrinter")
```

利用这个注解,Facelets可以通过MyPrinter引用bean。

17.9 增加存取方法

为了可以访问托管bean的状态,需要为状态增加存取方法。`createSalutation`方法调用bean的`greet`方法,而`getSalutation`方法获取结果。

bean有了存取方法才算完整。最终的代码如下:

```

package greetings;

import javax.inject.Inject;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;

    private String name;
    private String salutation;

    public void createSalutation() {
        this.salutation = greeting.greet(name);
    }

    public String getSalutation() {
        return salutation;
    }

    public String setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

```

17.10 在 Facelets 页面中使用托管 bean

在Facelets页面中使用托管bean,需要创建一个使用用户界面元素的表单,以调用其中的方法

并显示结果。下面的例子在表单里提供一个按钮，当用户输入名字之后按下按钮以提交表单，会看到显示在按钮下方的问候语：

```
<h:form id="greetme">
  <p><h:outputLabel value="Enter your name: " for="name"/>
    <h:inputText id="name" value="#{printer.name}"/></p>
  <p><h:commandButton value="Say Hello"
    action="#{printer.createSalutation}"/></p>
  <p><h:outputText value="#{printer.salutation}"/></p>
</h:form>
```

17.11 使用 Producer 方法注入对象

利用Producer方法可以注入非bean对象、在运行时值可能发生变化的对象，或需要定制实现的对象。例如，若要初始化一个通过限定词@MaxNumber定义的数值，可以在托管bean中定义这个值，并为其定义一个名为getMaxNumber的producer方法：

```
private int maxNumber = 100;
...
@Produces @MaxNumber int getMaxNumber() {
    return maxNumber;
}
```

当注入另一个托管bean中的对象时，容器自动调用producer方法，将其值初始化为100：

```
@Inject @MaxNumber private int maxNumber;
```

如果其值在运行时会发生变化，那么实现方式略有不同。例如，下面的代码定义了一个producer方法，用以产生一个通过@Random定义的随机数：

```
private java.util.Random random =
    new java.util.Random( System.currentTimeMillis() );

java.util.Random getRandom() {
    return random;
}

@Produces @Random int next() {
    return getRandom().nextInt(maxNumber);
}
```

当在另一个托管bean中注入这个对象时，需要声明对象的上下文实例：

```
@Inject @Random Instance<Integer> randomInt;
```

此时可以调用实例的get方法：

```
this.number = randomInt.get();
```

17.12 配置 CDI 应用

使用CDI的应用必须有一个名为beans.xml的文件，文件内容可以为空（在特定情况下才需要有内容），但文件必须存在。对于Web应用，beans.xml文件可以位于WEB-INF目录或

WEB-INF/classes/META-INF目录下。对于EJB模块或JAR文件, beans.xml文件必须位于META-INF目录下。

17.13 有关 CDI 的更多信息

有关Java EE平台上CDI的更多信息, 参见如下资料。

❑ Java EE平台上下文与依赖注入规范:

<http://jcp.org/en/jsr/detail?id=299>

❑ Java EE平台上下文与依赖注入入门:

<http://docs.jboss.org/weld/reference/latest/en-US/html/>

❑ Java依赖注入规范:

<http://jcp.org/en/jsr/detail?id=330>

运行简单的上下文与 依赖注入示例

本章将详细介绍如何构建并运行简单的CDI示例。示例位于如下目录：

tut-install/examples/cdi/

若要构建并运行示例，需要完成如下工作：

- (1) 使用NetBeans IDE或Ant工具编译并打包示例；
- (2) 使用NetBeans IDE或Ant工具部署示例；
- (3) 在Web浏览器中运行示例。

每个示例都有一个对应的build.xml文件，位于如下目录：

tut-install/examples/bp-project/

请参考第2章了解安装、构建以及运行示例的基础知识。

本章内容

- CDI示例simplegreeting
- CDI示例guessnumber

18.1 CDI 示例 simplegreeting

simplegreeting示例展示了CDI的一些最基础特性：作用域、限定词、bean注入以及在JSF应用中访问托管bean。当运行该示例时，单击代表正式或非正式问候的按钮，其显示结果依赖于调用的类。本示例包括4个源文件、1个Facelets页面和模板以及多个配置文件。

18.1.1 simplegreeting的源文件

simplegreeting示例的4个源文件包括：

- 默认的Greeting类（参见17.4节）；
- 用@Informal注解的限定词接口定义，以及实现该接口的InformalGreeting类（参见17.5节）；
- 托管bean的类Printer，它注入两个接口中的一个（参见17.9节）。

上述代码的源文件位于如下目录：

`tut-install/examples/cdi/simplegreeting/src/java/greetings`

18.1.2 Facelets模板和页面

在Facelets应用中使用托管bean，可以利用一个简单的模板文件和index.xhtml页面。模板文件template.xhtml的内容如下所示：

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8"/>
    <link href="resources/css/default.css"
      rel="stylesheet" type="text/css"/>
    <title>
      <ui:insert name="title">Default Title</ui:insert>
    </title>
  </h:head>

  <body>
    <div id="container">
      <div id="header">
        <h2><ui:insert name="head">Head</ui:insert></h2>
      </div>

      <div id="space">
        <p></p>
      </div>

      <div id="content">
        <ui:insert name="content"/>
      </div>
    </div>
  </body>
</html>
```

在创建Facelets页面时，可以重新定义模板文件的标题和头部，也可以为其增加一个表单：

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <ui:composition template="/template.xhtml">

    <ui:define name="title">Simple Greeting</ui:define>
    <ui:define name="head">Simple Greeting</ui:define>
    <ui:define name="content">
      <h:form id="greetme">
```

```

        <p><h:outputLabel value="Enter your name: " for="name"/>
          <h:inputText id="name" value="#{printer.name}"/></p>
        <p><h:commandButton value="Say Hello"
          action="#{printer.createSalutation}"/></p>
        <p><h:outputText value="#{printer.salutation}"/> </p>
      </h:form>
    </ui:define>
  </ui:composition>
</html>

```

表单用于接收用户输入的名称。按钮用Say Hello标识，为其定义的动作是调用Printer这个托管bean中的createSalutation方法——该方法随后调用Greeting类中的greet方法。

该页面将返回设置方法执行后预先设定的问候语。下面显示的是运行该页面后用户可能得到的问候语，具体的返回结果取决于注入的是默认类，还是通过@Inject进行注解的类。*name*将显示为用户提交的输入。

Hello, *name*.

或者

Hi, *name*!

Facelets页面和模板位于如下目录：

tut-install/examples/cdi/simplegreeting/web

在Facelets页面中用到的一个简单CSS文件位于如下目录：

tut-install/examples/cdi/simplegreeting/web/resources/css/default.css

18.1.3 配置文件

必须创建一个空的beans.xml文件，以使GlassFish服务器知道该应用是一个CDI应用。该文件在某些场景下需要有内容，但因为本应用比较简单，所以使用空文件就行了。

本示例也需要基本的Web应用部署描述文件web.xml和sun-web.xml。这些配置文件位于如下目录：

tut-install/examples/cdi/simplegreeting/web/WEB-INF

18.1.4 构建、打包、部署及运行CDI示例simplegreeting

可以使用NetBeans IDE或Ant工具构建、打包、部署及运行simplegreeting应用。

▼ 使用NetBeans IDE构建、打包及部署simplegreeting示例

该过程将构建后生成的应用复制到如下目录：

tut-install/examples/cdi/simplegreeting/build/web

该目录下的内容将部署至GlassFish服务器。

(1) 在NetBeans IDE中，选择File → Open Project。

(2) 在Open Project对话框中，选择路径：

tut-install/examples/cdi/

- (3) 选择simplegreeting目录。
- (4) 勾选Open as Main Project复选框。
- (5) 单击Open Project。
- (6) (可选) 执行如下步骤以修改Printer.java文件:
 - (a) 展开Source Packages结点;
 - (b) 展开greeting结点;
 - (c) 双击Printer.java文件;
 - (d) 在编辑窗格中, 将注解@Informal注释掉:

```
//@Informal
@Inject
Greeting greeting;
```

- (e) 保存文件。

- (7) 在Projects标签中, 右键单击simplegreeting项目, 并选择Deploy。

▼ 使用Ant构建、打包并部署simplegreeting示例

- (1) 在wed终端窗口中切换目录至:

tut-install/examples/cdi/simplegreeting/

- (2) 键入如下命令:

```
ant
```

该命令执行default任务, 构建并打包应用至WAR文件simplegreeting.war中。该文件位于dist目录下。

- (3) 键入如下命令:

```
ant deploy
```

键入上述命令可将simplegreeting.war部署至GlassFish服务器。

▼ 运行simplegreeting示例

- (1) 在Web浏览器中输入如下URL:

<http://localhost:8080/simplegreeting>

Simple Greeting页面将被打开。

- (2) 在文本输入框中输入一个名字。

例如Duke。

- (3) 单击Say Hello按钮。

如果没有修改Printer.java文件, 则下面的字符串将显示在按钮下方:

Hi, Duke!

如果将Printer.java文件中的@Informal注解注释掉，则下面的字符串将显示在按钮下方：
Hello, Duke.

图18-1展示了在不修改Printer.java文件的情况下页面呈现的内容。

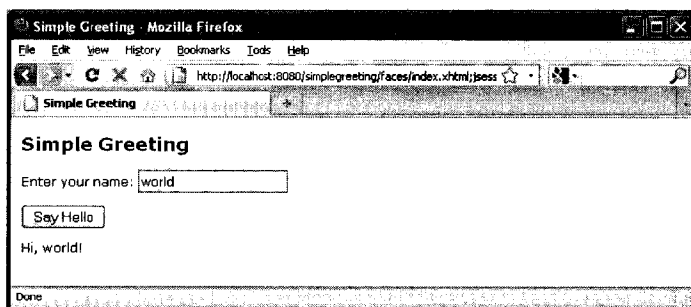


图18-1 Simple Greeting应用

18.2 CDI 示例 guessnumber

guessnumber示例比simplegreeting示例略微复杂些。它展示了@Produces注解的使用，以及会话和应用的作用域。该示例实现了一个游戏，其玩家最多有10次机会猜测一个数字。它与第5章中介绍的示例guessnumber有相似之处。区别在于玩家可以持续猜测数字，直至猜测正确或用尽10次机会。

该示例包括4个源文件、1个Facelets页面和模板，以及一些配置文件。配置文件和模板与simplegreeting示例中用到的相同。

18.2.1 guessnumber的源文件

guessnumber示例中用到的4个源文件包括：

- ❑ 用@MaxNumber注解的限定词接口；
- ❑ 用@Random注解的限定词接口；
- ❑ 名为Generator的托管bean，定义了用@Produces注解的方法；
- ❑ 名为UserNumberBean的托管bean。

源文件位于如下目录：

`tut-install/examples/cdi/guessnumber/src/java/guessnumber`

1. 用@MaxNumber和@Random注解的限定词接口

用@MaxNumber注解的限定词接口定义如下：

```
package guessnumber;
```

```
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
```

```
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface MaxNumber {

}
```

用@Random注解的限定词接口定义如下:

```
package guessnumber;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface Random {

}
```

2. 托管bean Generator

托管bean Generator为应用定义了两个用@Produces注解的方法。用@ApplicationScoped注解的bean，其上下文的作用域将扩展至用户与应用交互的整个过程:

```
package guessnumber;

import java.io.Serializable;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;

@ApplicationScoped
public class Generator implements Serializable {

    private static final long serialVersionUID = -7213673465118041882L;

    private java.util.Random random =
        new java.util.Random( System.currentTimeMillis() );
```

```

    private int maxNumber = 100;

    java.util.Random getRandom() {
        return random;
    }

    @Produces @Random int next() {
        return getRandom().nextInt(maxNumber);
    }

    @Produces @MaxNumber int getMaxNumber() {
        return maxNumber;
    }
}

```

3. 托管bean UserNumberBean

托管bean UserNumberBean是JSF应用的后台bean，实现了游戏的基本逻辑。这个bean实现如下功能：

- ❑ 实现bean中属性的存取方法；
- ❑ 注入两个限定词对象；
- ❑ 提供reset方法，允许玩家完成游戏后开始新游戏；
- ❑ 提供check方法，判断玩家猜测的数字是否正确；
- ❑ 提供validateNumberRange方法，判断玩家的输入是否有效。

bean的定义如下：

```

package guessnumber;

import java.io.Serializable;

import javax.annotation.PostConstruct;
import javax.enterprise.context.SessionScoped;
import javax.enterprise.inject.Instance;
import javax.inject.Inject;
import javax.inject.Named;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;

@Named
@SessionScoped
public class UserNumberBean implements Serializable {

    private static final long serialVersionUID = 1L;
    private int number;
    private Integer userNumber;
    private int minimum;
    private int remainingGuesses;

    @MaxNumber
    @Inject
    private int maxNumber;

    private int maximum;

```

```
@Random
@Inject
Instance<Integer> randomInt;

public UserNumberBean() {
}

public int getNumber() {
    return number;
}

public void setUserNumber(Integer user_number) {
    userNumber = user_number;
}

public Integer getUserNumber() {
    return userNumber;
}

public int getMaximum() {
    return (this.maximum);
}

public void setMaximum(int maximum) {
    this.maximum = maximum;
}

public int getMinimum() {
    return (this.minimum);
}

public void setMinimum(int minimum) {
    this.minimum = minimum;
}

public int getRemainingGuesses() {
    return remainingGuesses;
}

public String check() throws InterruptedException {
    if (userNumber > number) {
        maximum = userNumber - 1;
    }
    if (userNumber < number) {
        minimum = userNumber + 1;
    }
    if (userNumber == number) {
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage("Correct!"));
    }
    remainingGuesses--;
    return null;
}

@PostConstruct
public void reset() {
    this.minimum = 0;
    this.userNumber = 0;
    this.remainingGuesses = 10;
    this.maximum = maxNumber;
    this.number = randomInt.get();
}
```

```

public void validateNumberRange(FacesContext context,
                                UIComponent toValidate,
                                Object value) {
    if (remainingGuesses <= 0) {
        FacesMessage message = new FacesMessage("No guesses left!");
        context.addMessage(toValidate.getClientId(context), message);
        ((UIInput) toValidate).setValid(false);
        return;
    }
    int input = (Integer) value;

    if (input < minimum || input > maximum) {
        ((UIInput) toValidate).setValid(false);

        FacesMessage message = new FacesMessage("Invalid guess");
        context.addMessage(toValidate.getClientId(context), message);
    }
}
}

```

18.2.2 Facelets页面

本示例与simplegreeting示例使用相同的模板。相比较而言，index.xhtml文件要复杂一些。

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
    <ui:composition template="/template.xhtml">

        <ui:define name="title">Guess My Number</ui:define>
        <ui:define name="head">Guess My Number</ui:define>
        <ui:define name="content">
            <h:form id="GuessMain">
                <div style="color: black; font-size: 24px;">
                    <p>I'm thinking of a number between
                        <span style="color: blue">#{userNumberBean.minimum}</span> and
                        <span style="color: blue">#{userNumberBean.maximum}</span>. You have
                        <span style="color: blue">#{userNumberBean.remainingGuesses}</span>
                        guesses.</p>
                </div>
                <h:panelGrid border="0" columns="5" style="font-size: 18px;">
                    Number:
                    <h:inputText id="inputGuess"
                        value="#{userNumberBean.userNumber}"
                        required="true" size="3"
                        disabled="#{userNumberBean.number eq userNumberBean.userNumber}"
                        validator="#{userNumberBean.validateNumberRange}">
                    </h:inputText>
                    <h:commandButton id="GuessButton" value="Guess"
                        action="#{userNumberBean.check}"
                        disabled="#{userNumberBean.number eq userNumberBean.userNumber}" />
                    <h:commandButton id="RestartButton" value="Reset"
                        action="#{userNumberBean.reset}"
                        immediate="true" />
                    <h:outputText id="Higher" value="Higher!"
                        rendered="#{userNumberBean.number gt userNumberBean.userNumber and userNumberBean.userNumber ne 0}"
                        style="color: red" />
                    <h:outputText id="Lower" value="Lower!"
                        rendered="#{userNumberBean.number lt userNumberBean.userNumber and userNumberBean.userNumber ne 0}"

```

```

        style="color: red"/>
    </h:panelGrid>
    <div style="color: red; font-size: 14px;">
        <h:messages id="messages" globalOnly="false"/>
    </div>
</h:form>
</ui:define>

</ui:composition>
</html>

```

Facelets页面为玩家展示数字的范围（最大值和最小值）以及剩余的猜测次数。玩家与游戏的交互在panelGrid表中进行。该表包括一个输入框、Guess和Reset按钮，以及一个文本框（显示猜测的数值是大于还是小于正确值）。玩家每次单击Guess按钮，userNumberBean.check方法都将被调用，以重置最大值或最小值或者在猜测正确时产生一个FacesMessage消息。userNumberBean.validateNumberRange方法判断每次的猜测结果是否有效。

18.2.3 构建、打包、部署及运行CDI示例guessnumber

可以使用NetBeans IDE或Ant工具构建、打包、部署及运行guessnumber应用。

▼使用NetBeans IDE构建、打包及部署guessnumber示例

该过程将构建后生成的应用复制到如下目录：

tut-install/examples/cdi/guessnumber/build/web

该目录下的内容将部署至GlassFish服务器。

(1) 在NetBeans IDE中，选择File → Open Project。

(2) 在Open Project对话框中选择路径：

tut-install/examples/cdi/

(3) 选择guessnumber目录。

(4) 勾选Open as Main Project复选框。

(5) 单击Open Project。

(6) 在Projects标签中，右键单击guessnumber项目，并选择Deploy。

▼使用Ant构建、打包并部署simplegreeting示例

(1) 在终端窗口中，切换目录至：

tut-install/examples/cdi/guessnumber/

(2) 键入如下命令：

```
ant
```

该命令执行default任务，构建并打包应用至WAR文件guessnumber.war中。该文件位于dist目录下。

(3) 键入如下命令

```
ant deploy
```

键入上述命令，将guessnumber.war部署至GlassFish服务器。

▼ 运行guessnumber示例

(1) 在Web浏览器中输入如下URL：

<http://localhost:8080/guessnumber>

Guess My Number页面将被打开，如图18-2所示。

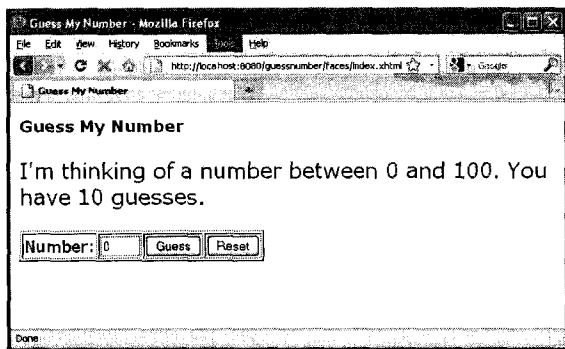


图18-2 Guess My Number示例

(2) 在Number输入框中输入一个数字，单击Guess按钮。

数字的范围最大值和最小值将被修改，并显示猜测的剩余次数。

(3) 不断尝试猜测，直至猜到正确数字或用尽10次尝试机会。

如果猜中正确答案，则输入框和Guess按钮将不可用，如图18-3所示。

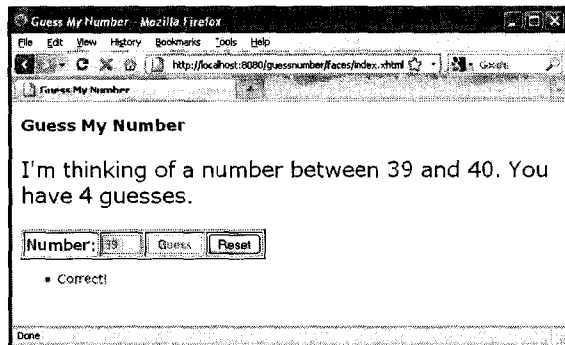


图18-3 Guess My Number游戏结束

(4) 单击Reset按钮，重新开始一次新的猜数字游戏。

Part 6

第六部分

持久化

第六部分主要讲述 Java Persistence API 相关的内容。

本 部 分 内 容

- 第 19 章 Java Persistence API 简介
- 第 20 章 运行 Persistence 示例
- 第 21 章 Java Persistence 查询语言
- 第 22 章 使用 Criteria API 构造查询

为了能在Java应用里方便地管理关系型数据，Java Persistence API为Java开发人员提供了对象/关系映射能力。Java Persistence包括如下方面的内容：

- Java Persistence API;
- 查询语言;
- Java Persistence Criteria API;
- 对象/关系映射元数据。

本章内容

- 实体
- 实体继承
- 管理实体
- 查询实体
- 有关Persistence的更多信息

19.1 实体

实体是一种用在持久化领域的轻量级对象。通常来说，实体代表了关系型数据库里的一个表，每个实体的实例对应了表里的一条记录。实体编程的主要工作是编写实体类，不过也会涉及辅助类的编写。

实体的持久化状态是通过持久化的字段或者持久化的属性表示的。这些字段或者属性使用对象/关系映射注解，把实体以及实体关系映射到底层数据存储里的关系型数据上。

19.1.1 实体类的需求

实体类必须满足如下的需求。

- 类必须使用`javax.persistence.Entity`注解。
- 类必须有一个公有类型或保护类型的构造方法，且该构造方法没有传入参数。类可以有其他的构造方法。

- ❑ 类不能声明为final，且其中的方法或者持久化实例变量均不能声明为final。
- ❑ 如果实体实例要传给远程对象，例如通过会话bean的远程业务接口进行传递，那么这个类必须实现Serializable接口。
- ❑ 实体类可以扩展实体类或者非实体类，非实体类可以扩展实体类。
- ❑ 持久化实例的变量必须声明为私有类型、保护类型或包级私有类型，且仅能通过该实体的方法实现直接访问。客户端要访问实体的状态，必须使用存取方法或者业务方法。

19.1.2 实体类的持久化字段和属性

实体的持久化状态，可以通过实体实例变量或者属性来访问。字段或者属性必须是如下Java语言里的数据类型。

- ❑ Java基本类型。
- ❑ java.lang.String。
- ❑ 其他可序列化的类型，包括：
 - Java基本类型的封装类；
 - java.math.BigInteger；
 - java.math.BigDecimal；
 - java.util.Date；
 - java.util.Calendar；
 - java.sql.Date；
 - java.sql.Time；
 - java.sql.Timestamp；
 - 用户自定义的可序列化的类型；
 - byte[]；
 - Byte[]；
 - char[]；
 - Character[]。
- ❑ 枚举类型。
- ❑ 其他实体或实体的集合。
- ❑ 可嵌套的（embeddable）类。

实体可以使用持久化的字段、持久化的属性或者它们的组合。如果实体的成员变量使用了映射注解，则实体使用持久化的字段。如果实体的获取方法使用了映射注解，而这个获取方法对应了JavaBeans风格的属性，则实体使用持久化的属性。

1. 持久化的字段

如果实体类使用了持久化的字段，则Persistence运行时将直接访问实体类的实例变量。所有没有使用javax.persistence.Transient注解或者没被标记为Java transient的字段都会被持久化

到数据存储里去。实例变量必须使用对象/关系映射注解。

2. 持久化的属性

如果实体使用持久化的属性，实体必须遵照JavaBeans组件定义方法的惯例。JavaBeans风格的属性存取方法一般在实体的变量声明之后进行声明。实体的每一个持久化的属性（比如属性名为`property`，类型为`Type`），都有一个获取方法`getProperty`以及一个设置方法`setProperty`。如果属性是布尔类型，可以使用`isProperty`来代替`getProperty`。例如，如果`Customer`实体使用持久化的属性，且有一个名为`firstName`的私有实例变量，则实体类可以定义`getFirstName`方法和`setFirstName`方法以读取和设置`firstName`的值。

对于一个单值持久化的属性，其方法签名为如下：

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

对于持久化的属性，必须将对象/关系型映射注解应用到其获取方法上。映射注解不能应用到使用了`@Transient`注解或被标注为`transient`的字段或者属性上。

3. 在实体字段和属性里使用集合

使用集合保存数据的持久化字段和属性，必须使用Java支持的集合类型接口，无论实体使用的是持久化的字段还是持久化的属性。可以使用如下的集合接口类型：

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

如果实体类使用了持久化的字段，前例中的方法签名类型必须是这些集合类型之一。这些集合类型的变种也可以使用。例如，如果`Customer`实体有一个持久化的属性，它包含了一组电话号码，则该实体可以有如下方法：

```
Set<PhoneNumber> getPhoneNumbers() { ... }  
void setPhoneNumbers(Set<PhoneNumber>) { ... }
```

如果实体的字段或者属性包括了基本类型的集合或者可嵌入类，则在字段或者属性上使用`javax.persistence.ElementCollection`注解。

`@ElementCollection`注解有两个属性，分别为`targetClass`和`fetch`。`targetClass`属性指定了基本类或者可嵌入类的类名。如果字段或者属性的类型是Java的基本数据类型，则该属性是可选的。`fetch`属性是可选的，使用`javax.persistence.FetchType`常量，可以用来指定集合使用懒加载还是急加载的方式，分别对应的常量值为`LAZY`和`EAGER`。默认情况下集合使用的是懒加载方式。

下面要介绍的实体`Person`有一个持久化的字段，即`nicknames`。这是一个`String`类的集合，且使用急加载方式。`targetClass`属性不是必需的，因为它的类型`String`是Java的基本数据类型。

```

@Entity
public class Person {
    ...
    @ElementCollection(fetch=EAGER)
    protected Set<String> nickname = new HashSet();
    ...
}

```

实体元素的集合以及实体关系的集合可以使用Java的java.util.Map这种数据结构。Map包含了一个键（key）和一个值（value）。

使用Map元素或者关系时，必须遵照如下规则：

- Map的键或者值可以是Java的基本数据类型或可嵌入类，也可以是一个实体；
- 当Map的值是可嵌入类或者基本数据类型的时候，需使用@ElementCollection注解；
- 当Map的值是实体的时候，需使用@OneToMany或者@ManyToMany注解；
- 仅在双向关系的某一个方向上使用Map类型。

如果Map里键的类型是Java的基本数据类型，则使用javax.persistence.MapKeyColumn注解，以设置值（列）到键的映射。默认情况下，@MapKeyColumn的name属性的格式是RELATIONSHIP-FIELD/PROPERTY-NAME_KEY。例如，如果引用的关系字段名是image，则默认的名称属性为IMAGE_KEY。

如果Map的键的类型为实体，则使用javax.persistence.MapKeyJoinColumn注解。如果映射需要多个列，则使用javax.persistence.MapKeyJoinColumns注解来包含多个@MapKeyJoinColumn注解。如果没有使用@MapKeyJoinColumn，映射的列的名字则默认设置成RELATIONSHIP-FIELD/PROPERTY-NAME_KEY。例如，如果关系字段名为employee，则默认的名称属性为EMPLOYEE_KEY。

如果关系字段或者属性没有使用Java编程语言里的基本数据类型，则键对应的类必须显式地使用javax.persistence.MapKeyClass注解。

如果Map的键是主键，或者是实体（作为Map的值）的持久化字段或属性，则使用javax.persistence.MapKey注解。@MapKeyClass以及@MapKey注解不能用在同一个字段或者属性上。

如果Map值是Java编程语言里的基本数据类型或者可嵌入类，在底层的数据库里，它将会映射成一个集合表（collection table）。如果使用的不是基本数据类型，则@ElementCollection注解的targetClass属性必须设置成Map值对应的类型。

如果Map值是一个实体，且为多对多关系或者一对多关系中单向关系的一部分，在底层的数据库里，它将会映射成一个联结表。使用了Map的单向的一对多关系，也可以使用@JoinColumn注解。

如果实体是一对多/多对一的双向关系的一部分，它将会被映射到实体的数据表里，这个数据表保存着所有Map的值。如果没有使用基本数据类型，则@OneToMany和@ManyToMany的targetEntity属性必须设置成Map值对应的类型。

4. 验证持久化字段和属性

Java API提供了JavaBeans 验证机制，即Bean验证，它可以用于验证应用里数据的有效性。Bean验证已经集成在Java EE容器里，支持在企业应用里任何不同的层上使用同样的验证逻辑。

Bean验证约束可以应用到持久化的实体类、可嵌入类以及映射的超类。默认情况下，Persistence提供者对于那些使用了Bean验证注解的持久化字段或者属性，在PrePersist、PreUpdate以及PreRemove生命周期事件之后，将会立即自动执行实体上的验证逻辑。

Bean验证约束是通过为Java类的字段或者属性添加注解实现的。Bean验证为自定义约束提供了一组内置约束和一组API。自定义约束可以是内置约束的特定组合，或者是没使用内置约束的新约束。每一个约束都至少和一个验证类相关联，验证类会验证受约束字段或者属性的值是否有效。对于自定义约束，开发人员必须为约束提供一个验证类。

Bean验证约束应用到持久化类的持久化字段或者属性上。当添加Bean验证约束时，需使用和持久化类同样的访问策略。也就是说，如果直接访问持久化类的持久化字段，则在类的字段上应用Bean验证约束注解。如果通过获取方法访问持久化类的持久化属性，则将约束应用到对应的获取方法上。

表9-2列举了Bean验证内置的约束，这些约束定义在javax.validation.constraints包里。

表9-2里的所有内置约束都有对应的注解，ConstraintName.List把同一个字段或者属性上的同一类型的多个约束进行分组。例如，如下的持久化字段有两个@Pattern约束：

```
@Pattern.List({
    @Pattern(regexp="..."),
    @Pattern(regexp="...")
})
```

下面的实体类Contact有应用到其持久化字段上的Bean验证约束。

```
@Entity
public class Contact implements Serializable {
    private static final long serialVersionUID = 1L;
    @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{3}]?[- ]?\\d{3}[- ]?\\d{4}$",
        message="{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp="^[\\d{3}\\\\\\d{3}\\\\\\d{3}]?[- ]?\\d{3}[- ]?\\d{4}$",
        message="{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(javax.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

firstName和lastName两个字段使用了@NotNull注解，说明这两个字段是必需的。如果创建了一个新的Contact实例，但是它的firstName字段或者lastName字段没被初始化，Bean验证机制会

抛出一个验证错误。类似地，如果修改了刚才创建出的那个Contact实例，且修改后的firstName或者lastName字段值为空，则也会抛出验证错误的异常。

email字段使用了@Pattern约束注解，声明了一个复杂的正则表达式，可以判断大多数电子邮件地址的有效性。如果email字段的值不匹配这个正则表达式，则会抛出验证错误的异常。

homePhone和mobilePhone字段有相同的@Pattern约束注解。这个正则表达式会检查号码是否匹配10位长的美国以及加拿大电话号码格式，且格式为 (xxx) xxx-xxxx。

birthday字段使用了@Past约束注解，它要求birthday的值必须早于当前日期。

19.1.3 实体里的主键

每一个实体都有唯一的对象标识符。例如，对于一个客户实体来说，可以用客户编号来识别它。这个唯一的标识符，或者说主键，使得客户端可以定位到某个特定的实体实例。每一个实体都必须有一个主键，实体的主键可能很简单，也可能很复杂。

对于简单的主键，可以在属性或字段上使用javax.persistence.Id注解，以指示这是用作主键的属性或者字段。

当一个主键包括不止一个属性时，可以使用复合主键。复合主键对应了一组持久化的属性或者字段，且必须定义在一个主键类里。复合主键使用 javax.persistence.EmbeddedId 和 javax.persistence.IdClass 注解。

主键或者复合主键的属性或者字段，必须是如下Java编程数据类型之一：

- ☐ Java基本类型；
- ☐ Java基本类型的包装类；
- ☐ java.lang.String；
- ☐ java.util.Date（时间类型是DATE）；
- ☐ java.sql.Date；
- ☐ java.math.BigDecimal；
- ☐ java.math.BigInteger。

永远不要将浮点类型作为主键的类型。如果使用自生成主键（generated primary key），只有整型的数据是可以移植的。

主键类必须满足如下需求：

- ☐ 类的访问控制描述符必须是public；
- ☐ 如果使用基于属性的访问机制，主键类的属性必须声明为public或protected；
- ☐ 类必须有一个公有的默认构造方法；
- ☐ 类必须实现hashCode()方法以及equals(Object other)方法；
- ☐ 类必须是可序列化的；
- ☐ 复合主键必须表现并映射为实体类的多个字段或者属性，或者表现并映射为一个可嵌入类；

- 如果类映射到了实体类的多个字段或者多个属性，用作主键的那些字段或者属性的名字和类型必须与实体类中对应的字段或属性相匹配。

如下的主键类就是一个复合主键，orderId和itemId一起唯一确定一个实体：

```
public final class LineItemKey implements Serializable {
    public Integer orderId;
    public int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer orderId, int itemId) {
        this.orderId = orderId;
        this.itemId = itemId;
    }

    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return (
            (orderId==null?other.orderId==null:orderId.equals
            (other.orderId)
            )
            &&
            (itemId == other.itemId)
        );
    }

    public int hashCode() {
        return (
            (orderId==null?0:orderId.hashCode())
            ^
            ((int) itemId)
        );
    }

    public String toString() {
        return "" + orderId + "-" + itemId;
    }
}
```

19.1.4 实体关系的多样性

实体关系的多样性是指实体关系有多种类型：一对一、一对多、多对一以及多对多。

- 一对一 每一个实体的实例对应另外一个实体的实例。例如，为仓库建模时，每一个存储箱StorageBin有一把锁Widget。存储箱和锁之间是一对一的关系。声明一对一关系，需要在对应的持久化属性或者字段上使用javax.persistence.OneToOne注解。
- 一对多 某实体实例对应了其他实体的多个实例。例如，一份销售订单可以有多条订单项。在订单管理应用里，订单Order和订单项LineItem之间是一对多的关系。声明一对多

关系，需要在对应的持久化属性或者字段上使用`javax.persistence.OneToOne`注解。

- **多对一** 某实体的多个实例对应另外一个实体的一个实例。这种关系是“一对多”的反例。还是上面的例子，从订单项`LineItem`的角度来看，它与订单`Order`之间是多对一的关系。声明多对一关系，需要在对应的持久化属性或者字段上使用`javax.persistence.ManyToOne`注解。
- **多对多** 实体的一个实例可以与其他实体的多个实例相对应，本实体的多个实例也可以同时与那个实体的一个实例相对应。例如，每门大学课程都有很多学生选修，同时，每一个学生要学很多门课程。所以，在选课的应用中，课程`Course`和学生`Student`之间就是多对多的关系。声明多对多关系，需要在对应的持久化属性或者字段上使用`javax.persistence.ManyToMany`注解。

19.1.5 实体关系的方向性

19

关系的方向有两种：双向关系和单向关系。双向关系有关系拥有者一方以及相反的一方，而单向关系只有关系拥有者一方。关系拥有者一方决定了Persistence运行时如何对数据库里的关系进行数据更新。

1. 双向关系

在双向的实体关系中，每一个实体都有一个关系字段或者属性引用了其他的实体。通过实体字段或者属性，实体类的代码可以访问相关的对象。如果实体有一个相关字段，则我们可以说该实体“知道”这个相关的对象。例如，如果订单`Order`知道自己拥有的那个订单项实例`LineItem`，同时订单项`LineItem`也知道自己所属的订单`Order`，则可以说`Order`和`LineItem`就有着所谓的双向关系。

双向关系必须遵照如下的规则。

- 双向关系里的相反方向，必须使用`@OneToOne`、`@OneToMany`、`@ManyToMany`、的`mappedBy`元素，来实现对关系拥有者一方的引用。`mappedBy`元素指明实体的哪个属性或者字段是关系的拥有者。
- 多对一的双向关系里的“多”的一方，不能定义`mappedBy`元素。“多”的一方永远是关系的拥有者。
- 对于一对一的双向关系来说，拥有者一方就是包含了外键的一方。
- 对于多对多的双向关系来说，任何一方都可以是拥有者。

2. 单向关系

在单向关系中，只有一个实体有关系字段或者属性来引用其他的实体。例如，`LineItem`可能有一个关系字段来标识产品`Product`，但是`Product`并没有关系字段或者属性来保存`LineItem`的信息。换句话说，`LineItem`知道`Product`，但是`Product`并不知道哪个`LineItem`实例引用了自己。

3. 查询以及关系方向

Java Persistence查询语言以及基于Criteria API的查询，通常是沿着关系进行的。关系的方向

决定了查询是否能够从一个实体切换到另外一个实体。例如，可以从LineItem查询到Product，但是相反的方向却不行。对Order和LineItem来说，查询可以在两个方向上进行，因为这两个实体间存在双向关系。

4. 级联操作和关系

使用关系的实体通常对于关系中另外一个实体的存在有依赖性。例如，订单项是订单的一部分，如果删除了订单，也应当删除相应的订单项。这种关系叫做级联式删除关系。

javax.persistence.CascadeType枚举了级联操作定义的所有类型，它们可以应用到关系注解的cascade元素上。表19-1列举了实体可用的级联式操作。

表19-1 实体可用的级联操作

级联操作	描述信息
ALL	所有级联操作都应用到与父实体相关的实体上。ALL 等效于设定 cascade={ DETACH , MERGE , PESIST , REFRESH , REMOVE }
DETACH	如果父实体从持久化上下文分离了，则相关的实体也会分离出去
MERGE	如果父实体合并到持久化上下文里了，则也会合并相关的实体
PERSIST	如果父实体持久化到持久化上下文里了，则也会持久化相关的实体
REFRESH	如果父实体在当前持久化上下文中刷新了，则相关的实体也会被刷新
REMOVE	如果父实体从当前的持久化上下文里删除了，则相关的实体也会被删除

级联式删除关系可以通过为@OneToOne和@OneToMany关系设置cascade=REMOVE实现，例如：

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<Order> getOrders() { return orders; }
```

5. 关系中孤儿实体的删除

当从关系中删除一对一关系或者一对多关系的目标实体时，通常我们也期望对那个目标实体实行级联式删除操作。这样的目标实体叫做“孤儿实体”，其orphanRemoval属性可以用来设定那些孤儿实体应当相应地删除掉。例如，如果一个订单有多个订单项且移除了订单中的一个订单项，那么移除的这个订单项就是孤儿实体。如果orphanRemoval属性设置为true，则当从订单里移除某个订单项时，该订单项实体也会被删除。

@OneToMany和@OneToOne注解里的orphanRemoval属性是一个布尔值，并且默认值为false。

下面的例子展示了当从关系中删除一个实体时，如何把级联式删除操作应用到那个已经成为孤儿实体的customer上去：

```
@OneToMany(mappedBy="customer", orphanRemoval="true")
public List<Order> getOrders() { ... }
```

19.1.6 实体里的可嵌入类

可嵌入类用来表示某个实体的状态，和实体类不同的是，它没有持久化的标识符。可嵌入类

的实例与自己所属的实体共享标识符。可嵌入类仅能作为其他实体的状态存在。实体可以有单值或集合类型的可嵌入类属性。

可嵌入类有着和实体类一样的规则。但是，它使用的是`javax.persistence.Embeddable`注解，而不是实体类所使用的`@Entity`注解。

下面展示的`ZipCode`是一个可嵌入类，它有`zip`和`plusFour`字段：

```
@Embeddable
public class ZipCode {
    String zip;
    String plusFour;
    ...
}
```

这个可嵌入类是由`Address`实体调用的：

```
@Entity
public class Address {
    @Id
    protected long id
    String street1;
    String street2;
    String city;
    String province;
    @Embedded
    ZipCode zipCode;
    String country;
    ...
}
```

如果实体有可嵌入类类型的字段或者属性（这些字段或属性表示的是实体类的持久化状态），则对应的字段或者属性可使用`javax.persistence.Embedded`注解，但这个注解不是必须的。

可嵌入类自身也可以使用其他可嵌入类表示自己的状态，而且可能使用包含Java基本数据类型或者其他可嵌入类的集合表示自身的状态。除此之外，可嵌入类也可以包含与其他实体或者实体集合的关系。如果可嵌入类拥有这样的关系，则关系的方向是从目标实体或者实体的集合指向拥有这个可嵌入类的实体。

19.2 实体继承

实体支持类的继承性、多态关联以及多态查询。实体类可以扩展非实体类，非实体也可以扩展实体类。实体类既可以是抽象类，也可以是具体类。

`roster`示例演示了实体的继承性，详见20.2.2节。

19.2.1 抽象实体

使用`@Entity`注解可以把一个抽象类声明为实体类。抽象实体有点像具体实体类，只是不能实例化。

抽象实体像具体实体那样，可以被查询。如果查询的目标是一个抽象实体，那么查询操作将作用在抽象实体的所有具体子类上：

```

@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;
}

```

19.2.2 映射超类

实体可以继承超类，超类包含了持久化的状态以及映射信息，但超类不是实体。也就是说，超类不使用@Entity注解，因而不会被Java Persistence提供者映射为一个实体。这些通常用于多个实体类具有某些公共持久状态与映射信息的情况。

映射的超类可以用`javax.persistence.MappedSuperclass`注解来设定。

```

@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;
    ...
}

```

映射的超类不能被查询，也不能用于EntityManager接口或者Query接口所定义的方法中。要想在EntityManager里使用或者执行Query操作，必须使用映射超类的实体子类。映射的超类不能作为关系的目标实体。映射的超类可以是抽象类或者具体类。

映射的超类在底层数据库里没有任何对应的数据表。从映射的超类继承而来的实体定义了数据表的映射。比如说，在刚才的例子中，底层的数据表是FULLTIMEEMPLOYEE和PARTTIMEEMPLOYEE，但不存在EMPLOYEE表。

19.2.3 非实体超类

实体可能有非实体超类，这些超类可以是抽象类，也可以是具体类。非实体超类的状态是非持久化的，任何从非实体超类继承而来的实体状态都是非持久化的。非实体超类不能用于

EntityManager接口或者Query接口所定义的方法中。任何非实体超类里的映射或者关系的注解，都将被忽略。

19.2.4 单表映射继承体系策略

开发人员可以通过配置决定Java Persistence提供者如何把继承的实体映射到底层的数据存储上，方法是在继承体系的根类上使用注解javax.persistence.Inheritance。下面的映射策略用来把实体数据映射到底层数据库里。

- 每一个继承体系的类对应一个数据表。
- 每一个具体的实体类对应一个数据表。
- “联结”策略。对于子类来说，特有的字段或者属性映射到了另外一个数据表，父类对应的数据表则包含了公用的字段或者属性。

策略是通过设置@Inheritance的strategy元素配置的，只能从javax.persistence.InheritanceType所定义的枚举值中选取：

```
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};
```

如果实体继承体系的根类没使用@Inheritance注解，则使用默认的策略，即InheritanceType.SINGLE_TABLE。

1. 单表映射继承体系策略

在这个策略中（对应枚举值InheritanceType.SINGLE_TABLE），整个继承体系里的所有类都映射到数据库中的同一个表。该表有一个识别字段，也叫识别列（discriminator column），该字段的值可以区分出不同子类的实体。

识别字段的元素参见表19-2。通过在实体类继承体系的根类上使用javax.persistence.DiscriminatorColumn注解，可以为它们设置值。

表19-2 @DiscriminatorColumn的元素

类 型	名 称	描述信息
String	name	用作识别字段的列名称，默认值为DTYPE。这个元素是可选的
DiscriminatorType	discriminatorType	用作识别字段的列类型，默认值为DiscriminatorType.STRING。这个元素是可选的
String	columnDefinition	用于创建识别字段的SQL语句，默认值是由Persistence提供者生成的，和具体厂商有关。这个元素是可选的
String	length	用于识别字段类型为String的列长。如果不是String类型，则length元素会被忽略。这个元素的默认值是31，且本身也是可选的

在 `@DiscriminatorColumn` 注解中，只能使用下面的枚举值（从 `javax.persistence.DiscriminatorType` 中选取）来设置 `discriminatorType` 元素的值：

```
public enum DiscriminatorType {  
    STRING,  
    CHAR,  
    INTEGER  
};
```

如果在实体继承体系的根类上没有指定 `@DiscriminatorColumn`，而又必须要有识别字段，则 `Persistence` 提供者假定有一个默认的列，列名称为 `DTYPE`，列类型为 `DiscriminatorType.STRING`。

`javax.persistence.DiscriminatorValue` 注解可以为类继承体系里的每一个实体类设置识别字段的值。`@DiscriminatorValue` 注解只能修饰具体的实体类。

如果在类继承体系上使用了识别字段的实体没有使用 `@DiscriminatorValue` 注解，`Persistence` 提供者会提供一个默认值。不过这个默认值和具体实现有关，不同厂商有可能不一样。如果 `@DiscriminatorColumn` 的 `discriminatorType` 元素类型为 `DiscriminatorType.STRING`，那么默认值就是实体类的名字。

这个策略对实体和查询间的多态关系提供了很好的支持，从而使查询可以覆盖整个实体类的继承体系。然而，这个策略要求映射到子类持久属性的数据库字段（列）必须可以为空。

2. 单表映射实体类策略

在这个策略中（对应枚举值 `InheritanceType.TABLE_PER_CLASS`），继承体系里的每一个具体类都映射到数据库里的一个表。类里的所有字段或者属性，包括从父类继承的字段或者属性，都映射到对应表的字段（列）上。

这个策略对多态关系的支持不够好。如果查询覆盖了整个的实体类继承体系，通常需要 SQL 的 UNION 查询或者对于每一个子类进行单独的 SQL 查询。

由于对这个策略的支持是可选的，所以不是所有的 Java Persistence API 提供者都支持此策略。GlassFish 服务器里默认的 Java Persistence API 是不支持这个策略的。

3. 联结子类策略

在这个策略中（对应枚举值 `InheritanceType.JOINED`），类继承体系的根类对应一个单独的表，每一个子类有一个单独的表且仅包含子类独有的字段。也就是说，子类对应的表不包含那些继承的字段或者属性。子类表有一个或者多个字段（列）作为该表的主键，这个主键同时为连接到超类表的外键。

这项策略对多态关系提供了很好的支持，但是在实例化实体子类的时候，需要执行一个或多个联结操作。如果继承层级较深的话，联结策略将使性能降低。类似地，覆盖了整个类继承体系的查询在各个子类的表之间需要联结操作，同样会导致性能下降。

有些 Java Persistence API 提供者（包括 GlassFish 服务器使用的默认 Java 持久化提供者）在使用联结子类策略时，需要根实体有一个识别字段。如果应用里没使用自动建表功能，需要手工设置表里识别字段的值，或者使用 `@DiscriminatorColumn` 注解匹配自己的数据库表定义。有关识别字段的更多内容，可以参阅 19.2.4 节的“单表映射继承体系策略”。

19.3 管理实体

实体是由实体管理器管理的。实体管理器是一个`javax.persistence.EntityManager`实例。每一个`EntityManager`实例都和一个持久化上下文相关联。持久化上下文包含了一组存在于特定数据存储里的托管实体的实例。持久化上下文定义了一个范围，特定实体实例的创建、持久化以及删除都在这个上下文里完成。`EntityManager`接口定义了一组方法，用来与持久化上下文交互。

19.3.1 EntityManager接口

`EntityManager` API可以创建以及删除持久化实体的实例，可以通过实体的主键查找实体，还可以在实体上执行查询操作。

1. 容器托管的实体管理器

使用容器托管的实体管理器时，`EntityManager`实例关联的持久化上下文自动由容器注入到所有应用组件中，只要这些组件是在同一个JTA（Java Transaction API）事务里使用这个`EntityManager`实例。

JTA事务通常涉及多个应用组件之间的调用。为了完成一个JTA事务，这些组件通常需要访问同一个持久化上下文。当`EntityManager`通过`javax.persistence.PersistenceContext`注解注入到应用的组件里时，这些组件即可使用实体管理器。持久化上下文自动在当前JTA事务里传递。映射到相同持久化单元的`EntityManager`的引用，提供了在事务内部对持久化上下文的访问能力。通过自动注入持久化上下文，多个应用组件之间不必互相传递`EntityManager`实例即可在事务内部完成对数据的操作。容器托管的实体管理器的生命周期是由Java EE容器管理的。

为了获得`EntityManager`实例，把实体管理器注入到应用组件里即可：

```
@PersistenceContext
EntityManager em;
```

2. 应用托管的实体管理器

从另一方面来说，在应用托管的实体管理器里，持久化上下文不会传递给应用组件，`EntityManager`实例的生命周期是由应用管理的。

当应用需要访问一个不会随JTA事务在多个`EntityManager`实例中传递的持久上下文时（在一个特定的持久化单元里，存在多个`EntityManager`实例），应用托管的实体管理器就派上用场了。在这种情况下，每一个`EntityManager`创建各自的持久化上下文。`EntityManager`以及和它相关联的持久化上下文由应用显式地创建和销毁。`EntityManager`实例不是线程安全的，所以应用托管的实体管理器还可以用于不能直接注入`EntityManager`实例的情况。`EntityManagerFactory`实例是线程安全的。

在这种情况下，应用要创建`EntityManager`实例，可以通过`javax.persistence.EntityManagerFactory`提供的`createEntityManager`方法完成。

为了获得`EntityManager`实例，必须首先获得`EntityManagerFactory`实例。获取的方法是使用`javax.persistence.PersistenceUnit`注解，以将`EntityManagerFactory`注入到应用的组件里：

```
@PersistenceUnit
EntityManagerFactory emf;
```

然后从EntityManagerFactory实例那里得到EntityManager:

```
EntityManager em = emf.createEntityManager();
```

应用托管的实体管理器不会自动传递JTA事务的上下文。这样的应用需要手动地获取到JTA事务管理器的访问权，然后在执行实体的相关操作时，增加事务边界有关的信息。javax.transaction.UserTransaction接口定义了开始、提交以及回滚事务的方法。通过创建一个使用了@Resource注解的实例变量，可以注入UserTransaction实例:

```
@Resource
UserTransaction utx;
```

可以调用UserTransaction.begin方法开始一个事务。当所有的实体操作都完成了，调用UserTransaction.commit方法来提交事务。UserTransaction.rollback方法则可以用来回滚当前事务。

下面的例子展示了如何管理应用（使用了应用托管的实体管理器）里的事务:

```
@PersistenceContext
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();
}
```

3. 通过EntityManager查找实体

EntityManager.find方法用来通过实体的主键在数据存储里查找实体:

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, Order newOrder) {
    Customer cust = em.find(Customer.class, custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```

4. 管理实体实例的生命周期

可以通过EntityManager实例调用对实体的操作来管理实体实例。实体实例有4种状态：new、managed、detached或者removed。

- ☐ 状态为new的实体没有持久化的标识符，没有与任何持久化上下文关联。
- ☐ 状态为managed的实体实例有持久化的标识符，并关联到某个持久化上下文。
- ☐ 状态为detached的实体实例有持久化的标识符，不过没有关联到某个持久化上下文。

- 状态为removed的实体实例有持久化的标识符，并关联到某个持久化上下文。不过，它即将被从数据存储中删除。

5. 持久化实体实例

处于new状态的实体实例，可以通过两种方式把状态变成managed，一是对这个实例调用persist方法，二是通过与该实体关联的实体传递过来的级联式persist操作。第二种方式要求实体所关联的另一实体在关系注解中使用cascade=PERSIST或cascade=ALL设置。这意味着，当persist操作关联的事务完成后，实体的数据会保存在数据库里。如果实体已经处于managed状态，则persist操作将被忽略，不过persist操作仍会级联到关联的实体上，此时要求关联实体在关系注解中使用cascade=PERSIST或cascade=ALL设置。如果调用一个状态为removed的实体实例中的persist方法，实体将变成managed状态。如果实体状态是detached，则persist调用将抛出IllegalArgumentException异常，或者导致事务的提交失败。

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(Order order, Product product,
    int quantity) {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}
```

如果在关系注解里，该实体的cascade元素值为PERSIST或者ALL，则persist操作会级联传递到与之关联的所有实体上：

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

6. 删除实体实例

处于managed状态的实体实例，可以通过两种方式把状态变成removed，一是对这个实例调用remove方法，二是通过与该实体关联的实体传递过来的级联式remove操作。第二种方式要求实体所关联的另一实体在关系注解中使用cascade=REMOVE或cascade=ALL设置。如果remove方法是被状态为new的实体所调用，remove操作会被忽略，不过remove操作会级联到相关联的实体上去，此时要求关联实体在关系注解中使用cascade=REMOVE或cascade=ALL设置。如果remove方法是被一个处于detached状态的实体调用，则remove操作将抛出IllegalArgumentException异常，或者事务的提交将失败。如果在一个状态为removed的实体上调用remove方法，则remove方法会被忽略。当事务完成之后或者作为flush操作的结果，实体数据都会从数据存储里删除。

```
public void removeOrder(Integer orderId) {
    try {
        Order order = em.find(Order.class, orderId);
        em.remove(order);
    }...
}
```

在本例中，所有与order相关联的LineItem实体也会被删掉，因为Order的getLineItems方法在关系注解里把cascade设置成ALL了。

7. 同步实体数据到数据库

当提交实体关联的事务时，持久化实体的状态会与数据库里的数据进行同步。如果状态为managed的实体与另外一个状态为managed的实体是双向关系的话，数据就会基于关系的拥有者一方的数据进行持久化。

调用EntityManager实例的flush方法，可以强制状态为managed的实体的数据与数据库里的数据进行同步。如果实体与另外的实体有关联，而且这个关联实体的关系注解里cascade元素设置成了PERSIST或者ALL，当flush方法被调用的时候，这个相关实体的数据也会和数据库里的数据进行同步。

如果实体的状态为removed，调用flush方法会将实体数据从数据存储里删除。

19.3.2 持久化单元

持久化单元定义了由应用里的EntityManager实例托管的所有实体类的集合。这一组实体类代表了包含在一个数据存储里的数据。

持久化单元是通过persistence.xml配置文件定义的。下面是persistence.xml配置文件的一个例子：

```
<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.Order</class>
    <class>com.widgets.Customer</class>
  </persistence-unit>
</persistence>
```

这个文件定义了名为OrderManagement的持久化单元，它使用支持JTA的数据源：jdbc/MyOrderDB。jar-file和class元素指定了托管的持久化类：实体类、可嵌入类以及映射超类等。jar-file元素指定了JAR文件，后者对打包进来的持久化单元是可见的（持久化单元包含了托管的持久化类），而class元素显式地指定了要托管的持久化类。

jta-data-source（用于支持JTA的数据源）元素以及non-jta-data-source（用于不支持JTA的数据源）元素指定了容器所使用数据源的全局JNDI名称。

JAR文件或者META-INF（包含了persistence.xml文件）的上级目录，叫做持久化单元的根。持久化单元的作用域，是由持久化单元的根决定的。同一持久化作用域中的不同持久化单元的名称必须不同。

持久化单元可以直接打包到WAR或者EJB JAR文件里面去，也可以先打包到一个JAR文件，

然后再把这个JAR文件包含到一个WAR或者EAR文件里去。

- 如果是把持久化单元作为一组类打包到EJB JAR文件里去, 则persistence.xml文件应当放置在EJB JAR的META-INF目录下。
- 如果是把持久化单元作为一组类打包到WAR文件里去, 则persistence.xml文件应当放置在WAR文件里的WEB-INF/classes/META-INF目录下。
- 如果是先打包到一个JAR文件, 然后再把这个JAR文件包含到一个WAR或者EAR文件里去, 则JAR文件应当位于以下目录中的某一个里:
 - WAR的WEB-INF/lib目录;
 - EAR文件的库目录。

注意 在Java Persistence API 1.0里, JAR文件可以使用EAR文件的根目录作为持久化单元的根。这项特性在新版本里不再被支持。可移植的应用应当使用EAR文件的库目录作为持久化单元的根。

19.4 查询实体

Java Persistence API为查询实体提供了如下方法。

- JPQL (Java持久化查询语言) 是一种简单的、基于字符串的语言, 类似于SQL, 可以用来查询实体以及实体之间的关系。参考第21章以获取更多信息。
- Criteria API使用Java编程语言提供的API, 用来创建类型安全的查询, 以查询实体以及实体之间的关系。参考第22章以获取更多信息。

JPQL和Criteria API都有一定的优势, 但也有一定的不足。

JPQL查询通常仅仅几行语句, 比Criteria 查询更精简, 且可读性要好很多。熟悉SQL的开发人员会发现学习JPQL的语法很容易。JPQL命名查询可以定义在实体类的Java注解中, 也可以定义在应用的部署描述文件里。然而, JPQL查询不是类型安全的, 当从实体管理器里提取查询结果时, 需要做类型转换。这意味着在编译的时候可能捕获不到这种类型转换错误。JPQL查询不支持可扩充参数。

Criteria查询允许开发人员在应用的业务逻辑层定义查询。虽然应用也有可能使用JPQL动态查询, Criteria查询提供了比JPQL动态查询更好的性能, 这是因为JPQL动态查询每一次被调用的时候都必须动态解析。Criteria查询是类型安全的, 所以不需要做类型转换, 而JPQL查询需要做类型转换。Criteria API 只是一种Java API, 不需要开发人员学习新的语法。典型的Criteria查询要比JPQL冗长, 在把查询提交给实体管理器之前, 需要开发人员创建若干个对象, 以执行作用在这些对象上的操作。

19.5 有关 Persistence 的更多信息

有关Java Persistence API的更多信息，可以参阅如下内容。

- ❑ Java Persistence 2.0 API规范：

<http://jcp.org/en/jsr/detail?id=317>

- ❑ EclipseLink——GlassFish服务器里的Java PersistenceAPI实现：

<http://www.eclipse.org/eclipselink/jpa.php>

- ❑ EclipseLink团队的博客：

<http://eclipselink.blogspot.com/>

- ❑ EclipseLink的wiki文档：

<http://wiki.eclipse.org/Eclipselink>

本章讲述如何使用Java Persistence API, 主要讲解3个示例的源代码以及示例应用的构建运行步骤。第一个示例order是一个使用有状态会话bean来管理订单系统中相关实体的应用。第二个示例roster是一个管理联赛运动系统的应用。第三个应用address-book是一个保存联系人信息的Web应用。本章假设读者已经熟悉第19章里介绍的概念和内容。

本章内容

- order应用
- roster应用
- address-book应用

20.1 order 应用

order应用是一个简单的库存及订单管理应用, 用于维护部件的目录, 以及管理部件与订单的关系。这一应用有分别表示部件、供货商、订单以及订单项的实体。可以使用有状态的会话bean访问这些实体, 而这个会话bean包含了应用的业务逻辑。在应用的部署阶段, 一个简单的单件会话bean会创建并初始化相关的实体。一个Facelets Web应用用来操作以及展示这些数据。

一个订单包含多项要素。订单号是什么? 这个订单里有哪些部件? 这些部件又是由哪些部件组成的? 部件是谁生产的? 部件的规格是什么? 部件有没有示意图? 完整的订单管理系统需要管理所有这些要素, 而order应用仅是一个简化的版本, 只管理部分要素。

order应用由一个单独的WAR模块组成, WAR模块包含了企业bean类、实体类、辅助类、Facelets的 XHTML文件以及相关的类文件。

20.1.1 order示例中的实体关系

order示例应用演示了实体关系的几种类型: 自引用关系、一对一关系、一对多关系、多对一关系以及单向关系。

1. 自引用关系

自引用关系 (self-referential) 出现在同一个实体的相关字段之间。Part类有一个字段bomPart,

它有一个一对多的关系，对应的字段为parts。parts同时也是Part类的字段。也就是说，部件可以由多个部件组成，而每一个部件仅对应一个物料清单。

Parts的主键是一个组合主键，是由partNumber和revision两个字段组合而成。在EJB_ORDER_PART表里，这个主键映射为PARTNUMBER和REVISION字段：

```
...
@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;
}
...
```

2. 一对一关系

Part有一个字段vendorPart，它和VendorPart的part字段构成一对一的关系。也就是说，每一个部件都唯一对应厂商生产的某个部件，反之亦然。

下面是Part里的关系映射：

```
@OneToOne(mappedBy="part")
public VendorPart getVendorPart() {
    return vendorPart;
}
```

下面是VendorPart里的关系映射：

```
@OneToOne
@JoinColumns({
    @JoinColumn(name="PARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="PARTREVISION",
        referencedColumnName="REVISION")
})
public Part getPart() {
    return part;
}
```

需要注意，因为Part使用的是组合主键，@JoinColumn注解用于把PERSISTENCE_ORDER_VENDOR_PART表里的字段（列）映射到PERSISTENCE_ORDER_PART表中的字段（列）。PERSISTENCE_ORDER_VENDOR_PART表的PARTREVISION字段对应PERSISTENCE_ORDER_PART里的REVISION字段。

3. 映射到重叠的主键和外键上的一对多关系

Order类有一个字段lineItems，它和LineItem类的order字段之间构成一对多的关系。也就是说，每一个订单有一个或者多个订单项。

LineItem使用的是由orderId和itemId组成的组合主键。这个组合主键映射到PERSISTENCE_

ORDER_LINEITEM表里的ORDERID和ITEMID字段上。ORDERID是外键,对应的是PERSISTENCE_ORDER_ORDER表中的ORDERID。这意味着ORDERID映射了两次:一次作为主键字段orderId;另一次是作为关系字段order。

下面是Order里的映射关系:

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

下面是LineItem里的映射关系:

```
@ManyToOne
public Order getOrder() {
    return order;
}
```

4. 单向关系

LineItem有一个字段vendorPart,它和VendorPart之间构成单向的多对一关系。也就是说,在这种关系中,目标实体没有字段与之对应:

```
@ManyToOne
public VendorPart getVendorPart() {
    return vendorPart;
}
```

20.1.2 order应用里的主键

order应用里使用的主键类型包括如下几种:单值主键、组合主键以及自生成的主键。

1. 自生成的主键

VendorPart使用自生成的主键值。也就是说,应用不为实体分配主键的值,但是依赖于持久化提供者来创建一个主键的值。@GeneratedValue注解可以用来设定实体将要使用自生成的主键。

在VendorPart里,下面的代码展示了如何设置,以自动生成主键值:

```
@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注解要与@GeneratedValue的元素strategy=TABLE一起使用。也就是说,用来生成主键的策略是使用数据库里的一个表。@TableGenerator注解用来配置主键生成表的相关设置。name元素设置了主键生成表的名称,它就是VendorPart里的vendorPartGen。

EJB_ORDER_SEQUENCE_GENERATOR表的两列GEN_KEY和GEN_VALUE, 用来保存生成的主键的值。这个表可以用来创建其他实体的主键, 因此pkColumnValue元素设置成VENDOR_PART_ID, 从而把不同实体自生成的主键区分开来。allocationSize元素指定注解在分配主键值的时候, 一次增长多少。在这个例子里, 每一个VendorPart的主键将以10为步长增长。

主键字段vendorPartNumber类型是Long, 因为自生成主键字段的类型必须为整型。

2. 组合主键

组合主键是指主键由多个字段组成, 并遵循19.1.3节所述的需求。要使用一个组合主键值, 必须创建一个包装类。

在order应用里, 有两个使用组合主键的实体: Part和LineItem。

□ Part使用PartKey包装类。Part的主键是部件编号和修订号的组合。PartKey封装了这个主键。

□ LineItem使用LineItemKey包装类。LineItem的主键是订单号和物品编号的组合。

LineItemKey封装了这个主键。

下面是LineItemKey组合主键的包装类代码:

```
package order.entity;

public final class LineItemKey implements
    java.io.Serializable {

    private Integer orderId;
    private int itemId;

    public int hashCode() {
        return ((this.getOrderId()==null
            ?0:this.getOrderId().hashCode())
            ^ ((int) this.getItemId()));
    }

    public boolean equals(Object otherObj) {
        if (this == otherObj) {
            return true;
        }
        if (!(otherObj instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherObj;
        return ((this.getOrderId()==null
            ?other.orderId==null:this.getOrderId().equals
            (other.orderId)) && (this.getItemId ==
            other.itemId));
    }

    public String toString() {
        return "" + orderId + "-" + itemId;
    }
}
```

@IdClass注解用来设定实体类使用的主键类。对于LineItem类来说, @IdClass以如下方式使用:

```
@IdClass(order.entity.LineItemKey.class)
@Entity
...
```



```
public class LineItem {
    ...
}
```

LineItem的两个字段使用了@Id注解，以标记这些字段是组合主键的一部分：

```
@Id
public int getItemId() {
    return itemId;
}
...
@Id
@Column(name="ORDERID", nullable=false,
        insertable=false, updatable=false)
public Integer getOrderId() {
    return orderId;
}
```

对于orderId来说，可以使用@Column注解来设定字段在表里对应的字段名称。@Column还声明这个字段作为一个重叠外键，不能执行插入或者更新操作，因为它是关联PERSISTENCE_ORDER_ORDER表的ORDERID字段（参见20.1.1节的“映射到重叠的主键和外键上的一对多关系”）。也就是说，orderId可以通过Order实体设置。

在LineItem的构造方法里，订单项的标号（LineItem.itemId）使用Order.getNextId方法来设置：

```
public LineItem(Order order, int quantity, VendorPart
    vendorPart) {
    this.order = order;
    this.itemId = order.getNextId();
    this.orderId = order.getOrderId();
    this.quantity = quantity;
    this.vendorPart = vendorPart;
}
```

Order.getNextId统计当前订单项的数量，返回加一后的数值：

```
public int getNextId() {
    return this.lineItems.size() + 1;
}
```

Part不需要在构成组合主键Part的那两个字段上使用@Column注解，因为Part的组合主键不是重叠的主键或外键：

```
@IdClass(order.entity.PartKey.class)
@Entity
...
public class Part {
    ...
    @Id
    public String getPartNumber() {
        return partNumber;
    }
    ...
    @Id
    public int getRevision() {
        return revision;
    }
    ...
}
```

20.1.3 映射多个数据库表的实体

Part 的字段映射到了两个数据库表: PERSISTENCE_ORDER_PART 和 PERSISTENCE_ORDER_PART_DETAIL。后者保存了部件的规格信息以及有关的图表信息。@SecondaryTable 注解可以用来指定从表 (secondary table)。

```
...
@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 {
    ...
}
```

PERSISTENCE_ORDER_PART_DETAIL 和 PERSISTENCE_ORDER_PART 共用同样的主键值。@SecondaryTable 注解的 pkJoinColumns 元素用来指定 PERSISTENCE_ORDER_PART_DETAIL 表的主键是 PERSISTENCE_ORDER_PART 的外键。@PrimaryKeyJoinColumn 注解设置了主键字段的名称, 还设定它对应的是主表里的哪个字段。本例中, PERSISTENCE_ORDER_PART_DETAIL 和 PERSISTENCE_ORDER_PART 的主键字段的名称相同, 均是 PARTNUMBER 和 REVISION。

20.1.4 order 应用里的级联操作

与其他实体有关系的实体, 通常对关系中的其他实体的存在有着依赖性。例如, 订单项是订单的一部分, 如果删除了订单, 也应当相应地删掉订单项。这称为级联式删除关系。

order 应用里, 实体的关系里有两个级联式删除依赖。如果删除了一个订单项 (Line Items) 对应的订单 (Order), 也应当删除订单项。如果删除掉了一个 VendorPart 对应的 Vendor, 则也应当删掉 VendorPart。

通过在关系的相反方向设置 cascade 元素, 可以为实体关系设定级联操作。cascade 元素在 Order.LineItems 里设置成 ALL。这意味着从订单到订单项方向上的所有持久化操作 (删除、更新等) 都是级联的。

下面就是 Order 里的关系映射:

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

下面就是 LineItem 里的关系映射:

```
@ManyToOne
public Order getOrder() {
    return order;
}
```

20.1.5 order应用里的BLOB和CLOB类型

数据库里的PARTDETAIL表有一个名为DRAWING的字段，其类型为BLOB。BLOB代表二进制的大型对象，它可以用来保存二进制数据，比如图片。DRAWING字段映射到类型为 `java.io.Serializable` 的 `Part.drawing` 字段上。`@Lob`注解可以用来表示某个字段是大型对象。

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public Serializable getDrawing() {
    return drawing;
}
```

PERSISTENCE_ORDER_PART_DETAIL也有一个字段，即SPECIFICATION，类型为CLOB。CLOB表示字符类型的大型对象，它可以用来保存由于太大不能保存在VARCHAR里的字符串数据。SPECIFICATION映射到 `Part.specification` 字段（对应的类型为 `java.lang.String`）。`@Lob`注解用在此处表示这个字段是一个大型对象。

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public String getSpecification() {
    return specification;
}
```

这两个字段都使用`@Column`注解，都把`table`元素设置为从表。

20.1.6 order应用里的时间类型

`Order.lastUpdate`是一个持久化属性，它在Java中的类型为`java.util.Date`。该属性将映射到PERSISTENCE_ORDER_ORDER.LASTUPDATE数据库字段，后者的类型为TIMESTAMP。为确保成功映射这两种数据类型，必须使用`@Temporal`注解，并在`@Temporal`的元素里指定合适的时间类型。`@Temporal`的元素的类型为`javax.persistence.TemporalType`。可能的类型值有如下几种：

- ❑ DATE——映射成 `java.sql.Date`;
- ❑ TIME——映射成 `java.sql.Time`;
- ❑ TIMESTAMP——映射成 `java.sql.Timestamp`。

下面是Order里相应的代码：

```
@Temporal(TIMESTAMP)
public Date getLastUpdate() {
    return lastUpdate;
}
```

20.1.7 管理order应用里的实体

`RequestBean`这个有状态会话bean不仅包含了业务逻辑，还要管理order里的实体。`RequestBean`使用`@PersistenceContext`注解来获取实体管理器实例。实体管理器用来管理order中`RequestBean`的业务方法里的实体：

```
@PersistenceContext
private EntityManager em;
```

上述Entity Manager实例是一个容器托管的实体管理器，因此容器会接管涉及管理order里实体的所有事务。

1. 创建实体

RequestBean.createPart业务方法用来创建一个新的Part实体。EntityManager.persist方法用来把新创建的实体持久化到数据库里去。

```
Part part = new Part(partNumber,
    revision,
    description,
    revisionDate,
    specification,
    drawing);
em.persist(part);
```

ConfigBean是一个单件会话bean，用来初始化order中的数据。ConfigBean使用@Startup注解，以表明EJB容器会在部署order应用的时候创建ConfigBean。createData方法使用@PostConstruct注解，通过调用RequestBean的业务方法创建order里所需要的初始实体。

2. 查找实体

RequestBean.getOrderPrice业务方法返回的是根据orderId找出的order的订单款。EntityManager.find方法用来从数据库里检索出实体。

```
Order order = em.find(Order.class, orderId);
```

EntityManager.find方法的第一个参数是实体类，第二个参数是主键。

3. 设置实体关系

RequestBean.createVendorPart业务方法创建一个关联到某个特定的Vendor的VendorPart。EntityManager.persist方法用来将刚创建出来的那个VendorPart实体持久化到数据库里去。VendorPart.setVendor方法和Vendor.setVendorPart方法用来把VendorPart和Vendor关联起来。

```
PartKey pkey = new PartKey();
pkey.partNumber = partNumber;
pkey.revision = 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);
```

4. 使用查询

RequestBean.adjustOrderDiscount业务方法更新所有订单适用的折扣，这个方法使用声明在Order中的findAllOrders命名查询：

```
@NamedQuery(
    name="findAllOrders",
    query="SELECT o FROM Order o"
)
```

`EntityManager.createNamedQuery`方法用来运行查询。因为查询返回的是所有订单的List, 所以会用到`Query.getResultList`方法。

```
List orders = em.createNamedQuery(
    "findAllOrders")
    .getResultList();
```

`RequestBean.getTotalPricePerVendor`业务方法返回对一个特定供应商的所有部件的总价款。这个方法使用命名参数`id`。`id`定义在命名查询`findTotalVendorPartPricePerVendor`里, 而这个命名查询是在`VendorPart`里定义的。

```
@NamedQuery(
    name="findTotalVendorPartPricePerVendor",
    query="SELECT SUM(vp.price) " +
    "FROM VendorPart vp " +
    "WHERE vp.vendor.vendorId = :id"
)
```

当运行这个查询时, `Query.setParameter`方法用来将命名参数`id`设置为`vendorId`的值, 而`vendorId`是`RequestBean.getTotalPricePerVendor`的参数:

```
return (Double) em.createNamedQuery(
    "findTotalVendorPartPricePerVendor")
    .setParameter("id", vendorId)
    .getSingleResult();
```

由于查询的返回结果是一个单个的值, 所以这里用到了`Query.getSingleResult`方法。

5. 删除实体

`RequestBean.removeOrder`业务方法用来删除数据库里的一个指定订单。这个方法使用`EntityManager.remove`方法删除数据库里的实体。

```
Order order = em.find(Order.class, orderId);
em.remove(order);
```

20.1.8 构建、打包、部署以及运行order应用

这部分讲述如何构建、打包、部署以及运行order应用。为此, 需要在Java DB 数据库服务器里事先创建好数据库表, 然后才能构建、部署以及运行order应用。

▼使用NetBeans IDE创建、打包、部署以及运行order应用

- (1) 在NetBeans IDE里, 选择File→Open Project。
- (2) 在Open Project 对话框里, 选择路径`tut-install/examples/persistence/`。
- (3) 选择order目录。
- (4) 勾选Open as Main Project复选框。
- (5) 单击Open Project。
- (6) 在Projects标签中, 右键单击order这个项目, 并选择Run。

NetBeans IDE会打开一个Web浏览器, 访问<http://localhost:8080/order/>。

▼使用Ant构建、打包、部署以及运行order应用

- (1) 打开一个终端窗口，切换目录到`tut-install/examples/persistence/order/`。
- (2) 输入命令`ant`并回车。该命令将运行`default`任务，即编译源代码文件、打包应用成一个WAR文件，并放置到`tut-install/examples/persistence/order/dist/order.war`。
- (3) 要部署WAR，确认GlassFish服务器已经启动，然后在终端窗口中输入命令`ant deploy`并回车。
- (4) 打开一个Web浏览器，并访问`http://localhost:8080/order/`，试着创建以及更新订单数据。

all任务

作为惯例，`all`任务会执行所有的工作，包括构建、打包、部署以及运行应用。在终端上输入`ant all`并回车就可以了。

20.2 roster 应用

`roster`应用维护的是联赛运动队的花名册。该应用有4个组件：Java Persistence API的实体（`Player`实体、`Team`实体以及`League`实体）、有状态会话bean（`RequestBean`）、应用的客户端（`RosterClient`）以及3个辅助类（`PlayerDetails`、`TeamDetails`和`LeagueDetails`）。

从功能上来说，`roster`类似于`order`应用，不过它有着`order`应用所不具备的3个新特性：多对多关系、实体继承以及部署时可以自动创建所需的表。

20.2.1 roster应用里的关系

联赛运动队有如下几种关系：

- 一个运动员可以属于多个运动队；
- 一个运动队可以有多个运动员；
- 一个运动队只能属于一个联盟；
- 一个联盟可以有多个运动队。

对应到`roster`应用，系统里包含3个实体以及它们之间的关系。这3个实体分别是`Player`（运动员）、`Team`（运动队）和`League`（联盟）。关系如下：

- `Player`和`Team`之间是多对多的关系；
- `Team`和`League`之间是多对一的关系。

roster里的多对多关系

`Player`和`Team`之间的多对多关系是通过`@ManyToMany`注解设定的。在`Team.java`文件里，`getPlayers`方法使用了`@ManyToMany`注解。

```
@ManyToMany
@JoinTable(
    name="EJB_ROSTER_TEAM_PLAYER",
    joinColumns=
```

```

        @JoinColumn(name="TEAM_ID", referencedColumnName="ID"),
        inverseJoinColumns=
        @JoinColumn(name="PLAYER_ID", referencedColumnName="ID")
    )
    public Collection<Player> getPlayers() {
        return players;
    }
}

```

`@JoinTable`注解用来设定一个数据库表,把运动员ID和运动队ID关联起来。指定了`@JoinTable`的实体是关系的拥有者,所以Team实体是与Player实体关系的拥有者。因为roster在部署阶段自动创建表,容器会创建名为EJB_ROSTER_TEAM_PLAYER的联结表。

Player不是它与Team关系的拥有者。在一对一关系和多对一关系中,非拥有者一方在关系的注解里用`mappedBy`元素标记。因为Player和Team之间的关系是双向的,可以任意指定一方为实体关系的拥有者。

在Player.java文件里, `getTeams`方法使用了`@ManyToMany`注解:

```

@ManyToMany(mappedBy="players")
public Collection<Team> getTeams() {
    return teams;
}

```

20

20.2.2 roster应用里的实体继承关系

roster应用展示了如何使用实体的继承性,实体继承性的有关内容参见19.2节。

roster应用的League实体是一个抽象实体,它有两个具体的子类: SummerLeague 和 WinterLeague。因为League是一个抽象类,它不能被实例化:

```

...
@Entity
@Table(name = "EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}

```

相反,当创建一个联盟时,客户端使用的是SummerLeague或者WinterLeague。SummerLeague和WinterLeague继承了由League定义的持久化属性,此外只增加了一个构造方法,来验证sport参数是否符合季节性联赛所允许的比赛类型。例如,下面是SummerLeague实体的代码:

```

...
@Entity
public class SummerLeague extends League
    implements java.io.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应用使用了默认的映射策略`InheritanceType.SINGLE_TABLE`，所以没必要使用`@Inheritance`注解。如果想使用不同的映射策略，则使用`@Inheritance`注解来修饰`League`，并在`strategy`元素里指定映射策略：

```

@Entity
@Inheritance(strategy=JOINED)
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}

```

roster应用使用默认的认识字段名称，所以没必要使用`@DiscriminatorColumn`注解。因为在roster中使用自动化的方式创建表，Persistence提供者将创建一个识别字段，在`EJB_ROSTER_LEAGUE`表里名为`DTYPE`，保存派生出的实体的名称，创建`league`时要用到该名称。如果想让识别字段使用不同的名称，则在`League`上使用`@DiscriminatorColumn`注解，并设置其`name`元素：

```

@Entity
@DiscriminatorColumn(name="DISCRIMINATOR")
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}

```

20.2.3 roster里的Criteria查询

roster应用使用Criteria API查询，而order应用里使用的是JPQL查询。Criteria 查询是Java编程语言实现的类型安全的查询，定义在roster应用的业务逻辑层，由`RequestBean`（无状态会话bean）实现。

1. roster应用里的元模型类

元模型类为实体的属性建模，并用于Criteria查询中，以将查询定位到实体的属性。roster应用里的每一个实体类都有对应的元模型类。元模型类是在编译时生成的，与实体所在的包名一样，但需要在类名最后追加一个下划线。例如，`roster.entity.Person`实体对应的元模型类名称为`roster.entity.Person_`。

实体类里的每一个持久化字段或者属性，在实体的元模型类里都有对应的属性。对于`Person`实体，对应的元模型类为：

```

@StaticMetamodel(Person.class)
public class Person_ {
    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;
}

```

2. 在RequestBean里获取CriteriaBuilder实例

CriteriaBuilder接口定义了用来创建Criteria查询对象的方法, 以及创建修改查询对象的表达式的方法。RequestBean通过使用注解@PostConstruct的init方法创建一个CriteriaBuilder实例:

```

@PersistenceContext
private EntityManager em;
private CriteriaBuilder cb;

@PostConstruct
private void init() {
    cb = em.getCriteriaBuilder();
}

```

EntityManager实例是运行时注入的。随后, EntityManager对象通过调用getCriteriaBuilder用来创建CriteriaBuilder实例。CriteriaBuilder实例通过用@PostConstruct注解的方法来创建, 以确保EntityManager实例已经被企业bean容器注入。

3. 在RequestBean的业务方法里创建Criteria查询

RequestBean的很多业务方法都定义了Criteria查询。getPlayerByPosition这个业务方法返回的是运动队中的司职特定位置的运动员列表:

```

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);
    }
}

```

查询对象是通过调用CriteriaBuilder对象的createQuery方法创建的。查询的类型设置成了Player, 因为查询将返回运动员列表。

查询根 (root) 是基实体 (base entity), 通过它能找到实体的属性或者相关实体。查询根是通过调用查询对象的from方法创建的。该方法将设置查询的FROM语句。

WHERE语句是通过调用查询对象的where方法设置的, 它根据条件表达式限定了查询结果的

范围。CriteriaBuilder.equal方法用于比较两个表达式。在getPlayersByPosition方法里，可以通过调用查询根的get方法访问元模型类Player_的position属性。在查询时，position属性会和传给getPlayersByPosition方法的position参数相比较。

查询的SELECT语句是通过调用查询对象的select方法设定的。查询将返回Player实体的集合，所以查询根实体作为一个参数传递给了select方法。

这里通过调用EntityManager.createQuery方法准备查询对象，该方法返回的是TypedQuery<T>对象，其类型为查询结果的类型，在这个例子里是Player。这个带类型的查询对象用来执行查询，当getResultList方法被调用时，返回运动员的集合List<Player>。

20.2.4 roster应用里的自动建表特性

在部署的时候，GlassFish服务器会自动删除并创建roster应用所用的数据库表。这通过将persistence.xml文件里的eclipselink.ddl-generation属性设置为drop-and-create-tables来实现：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="em" transaction-type="JTA">
    <jta-data-source>jdbc/__default</jta-data-source>
    <properties>
      <property name="eclipselink.ddl-generation"
        value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

不同的Java Persistence API提供者对这个特性的支持是不一样的，比如GlassFish会遵循它所选用的Java Persistence API提供者所设定的规则。就这一点来说，该特性在不同的Java EE服务器之间不具有可移植性。然而，自动建表在开发阶段很有用，当应用准备上线，或者要把它部署到其他Java EE服务器上去的时候，可能需要将eclipselink.ddl-generation从persistence.xml文件里删掉。

20.2.5 构建、打包、部署以及运行roster应用

这部分讲述如何构建、打包、部署以及运行roster应用。为此，可以使用NetBeans IDE，也可以使用Ant。

▼使用NetBeans IDE构建、打包、部署以及运行roster应用

- (1) 在NetBeans IDE里，选择File→Open Project。
- (2) 在Open Project对话框里，选择路径tut-install/examples/persistence/。
- (3) 选择roster目录。
- (4) 勾选Open as Main Project复选框以及Open Required Projects复选框。

(5) 单击Open Project。

(6) 在Projects标签中，右键单击roster项目，选择Run。

在应用客户端的Output 标签里，可以看到如下的输出信息（仅列出了一部分）：

```
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
...
```

▼使用Ant构建、打包、部署以及运行roster应用

(1) 打开一个终端窗口，目录切换到：

tut-install/examples/persistence/roster/

(2) 输入并运行如下命令：

```
ant
```

这个命令将执行default任务，该任务将编译源文件，把生成的应用打包成一个EAR文件roster.ear，并放置到*tut-install/examples/persistence/roster/dist/*。

(3) 要部署EAR，需要确保GlassFish服务器已经启动，然后输入如下命令：

```
ant deploy
```

Ant会检查Java DB数据库服务器是否正在运行，如果没有则启动它，然后部署roster.ear。在部署阶段，GlassFish服务器会删除并创建由persistence.xml指定的数据库表。

部署roster.ear以后，一个名为rosterClient.jar的客户端JAR文件生成，它包含了应用的客户端。

(4) 要运行应用客户端，输入并执行如下命令：

```
ant run
```

可以看到如下所示的输出信息：

```
[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
...
```

all任务

作为惯例，all任务将执行所有的事项，包括构建、打包、部署以及运行应用。输入并执行如下命令即可：

```
ant all
```

20.3 address-book 应用

address-book示例应用是一个简单的Web应用，用来保存联系人数据。它使用实体类Contact，该实体类使用Java的Bean验证API来验证保存在实体的持久化属性中的数据是否有效，详见19.1.2节的“验证持久化字段和属性”。

20.3.1 address-book应用里的Bean验证约束

Contact实体类在持久化属性上使用@NotNull、@Pattern以及@Past约束。

@NotNull约束标记属性的值不能为空。在实体可以持久化之前或者修改之前，属性必须设置为一个非空的值。如果属性为空，当要持久化或者修改实体的时候，Bean验证将抛出验证错误。

@Pattern约束定义了一个正则表达式，在实体可以持久化或者修改之前，属性的值必须能匹配上这个正则表达式。在address-book应用里，这个约束有两种用法。

- 声明在@Pattern注解里作用在email这个字段上的正则表达式，匹配的是电子邮件地址，格式为name@domain name.top level domain，只允许电子邮件地址使用合法字符。例如，username@example.com能通过验证，firstname.lastname@mail.example.com也可以通过验证。然而firstname.lastname@example.com不能通过验证，因为它包含了一个非法字符——逗号。

- mobilePhone和homePhone字段使用了@Pattern注解约束，它定义了一个匹配电话号码的正则表达式，格式为(xxx)xxx-xxxx。

birthday字段使用了@Past约束，说明这个字段只能接收java.util.Date类型的值，且只能是一个过去的日子。

下面是Contact实体类的部分代码：

```

@Entity
public class Contact implements Serializable {
    private static final long serialVersionUID = 1L;
    @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{3})[- ]?(\\d{4})$",
        message="{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp="^(\\d{3})\\d{3}[- ]?(\\d{3})[- ]?(\\d{4})$",
        message="{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(javax.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}

```

20.3.2 为 address-book 应用里的约束指定出错信息

Contact实体的某些约束指定了可选的message元素：

```

@Pattern(regexp="^(\\d{3})\\d{3}[- ]?(\\d{3})[- ]?(\\d{4})$",
        message="{invalid.phonenumber}")
protected String homePhone;

```

@Pattern约束里的message元素是可选的，它会覆写默认的验证信息。验证信息可以直接通过如下方式指定：

```

@Pattern(regexp="^(\\d{3})\\d{3}[- ]?(\\d{3})[- ]?(\\d{4})$",
        message="Invalid phone number!")
protected String homePhone;

```

Contact里的约束是字符串类型，它定义在资源文件里，路径为*tut-install/examples/persistence/address-book/src/java/ValidationMessage.properties*。这个机制允许在一个配置文件里设置验证信息，从而可以很容易地实现多语言支持。改写后的Bean验证消息必须放置在资源配置文件里，即默认的包中的*ValidationMessage.properties*文件。对于本地化的资源包配置文件，其文件名的格式为*ValidationMessage_local-prefix.properties*。例如，*ValidationMessage_es.properties*则表明资源包用在西班牙语为母语的地区。

20.3.3 验证 JSF 应用中输入的 Contact 数据

address-book应用使用JSF页面作为前端，使用户可以在这个页面里输入联系人信息。JSF有一种表单输入验证机制，该机制需要在Facelets的XHTML文件里使用标签，不过address-book不

使用这种机制。address-book使用的是在JSF后台bean中的bean验证约束。在本例中，后台bean是指Contact这个实体。当提交表单时，会自动触发后台bean的验证机制。

下面的代码来自Facelets文件Create.xhtml，展示了输入表单的几个输入项，这些输入项用来创建新的Contact实例。

```
<h:form>
  <h:panelGrid columns="3">
    <h:outputLabel value="#{bundle.CreateContactLabel_firstName}"
      for="firstName" />
    <h:inputText id="firstName"
      value="#{contactController.selected.firstName}"
      title="#{bundle.CreateContactTitle_firstName}" />
    <h:message for="firstName"
      errorStyle="color: red"
      infoStyle="color: green" />
    <h:outputLabel value="#{bundle.CreateContactLabel_lastName}"
      for="lastName" />
    <h:inputText id="lastName"
      value="#{contactController.selected.lastName}"
      title="#{bundle.CreateContactTitle_lastName}" />
    <h:message for="lastName"
      errorStyle="color: red"
      infoStyle="color: green" />
    ...
  </h:panelGrid>
</h:form>
```

firstName和lastName使用的<h:inputText>标签，分别绑定到Contact实体实例的属性上，对应的实体是contactController.selected，ContactController是一个无状态会话bean。每一个<h:inputText>标签都有一个对应的<h:message>标签，以显示验证出错时的提示信息。不过，表单不需要任何JSF的验证标签。

20.3.4 构建、打包、部署以及运行 address-book 应用

这部分讲述如何构建、打包、部署以及运行address-book应用。为此，可以使用NetBeans IDE，也可以使用Ant。

▼使用NetBeans IDE构建、打包、部署以及运行address-book应用

- (1) 在NetBeans IDE里，选择File→Open Project。
- (2) 在Open Project对话框里，选择路径*tut-install/examples/persistence/*。
- (3) 选择address-book目录。
- (4) 勾选Open as Main Project复选框以及Open Required Projects复选框。
- (5) 单击Open Project。
- (6) 在Projects标签里，右键单击address-book项目，选择Run。

当应用部署完成后，Web浏览器会打开并访问如下URL：

<http://localhost:8080/address-book/>

- (7) 在页面里单击Show All Contact Items，然后单击Create New Contact。在表单的文本框里输

入值，然后单击Save。

如果任何一个输入值违反了Contact的约束，出错的表单字段旁都会出现一个红色的出错信息。

▼使用Ant构建、打包、部署以及运行address-book应用

(1) 打开一个终端窗口，目录切换至：

tut-install/examples/persistence/address-book/

(2) 输入并运行如下命令：

```
ant
```

这个命令将编译并装配address-book应用。

(3) 输入如下命令：

```
ant deploy
```

这个命令将把应用部署到GlassFish服务器上。

(4) 打开一个浏览器窗口，然后访问如下URL：

<http://localhost:8080/address-book/>

20

小贴士 作为惯例，all任务将执行所有的事项，包括构建、打包、部署以及运行应用。输入并执行如下命令即可：

```
ant all
```

(5) 在页面里单击Show All Contact Items，然后单击Create New Contact。在表单的文本框里输入值，然后单击Save。

如果任何一个输入值违反了Contact的约束，出错的表单字段旁都会出现一个红色的出错信息。

Java Persistence 查询语言定义了对实体及其持久化状态的查询方法。查询语言使得我们可以编写跨平台的查询，不用关注底层使用的是什么数据存储。

查询语言使用实体的抽象持久化模式（包括它们的关系）作为查询语言的数据模型，并且定义基于这个数据模型的操作和表达式。查询的范围涉及位于同一个持久化单元里相关实体的所有抽象模式。查询语言使用的是类似SQL的语法，基于实体的抽象模式类型和实体间的关系选择对象或者值。

本章内容依赖于在前面几章讲述过的内容。有关概念性的信息，可以参见第19章。有关代码示例的内容，可以参见第20章。

本章内容

- 查询语言术语
- 使用Java Persistence查询语言创建查询
- 查询语言的简要语法
- 查询示例
- 查询语言语法全本

21.1 查询语言术语

下面列举了本章内容会用到的一些术语。

- **抽象模式** 持久化数据模型的抽象（包含的要素有持久化的实体、它们的状态以及实体间的关系），而查询建立在这个抽象之上。查询语言把基于这种抽象的查询，转换成基于实体所映射数据库数据模型之上的查询。
- **抽象模式类型** 抽象模式中实体的持久化属性的类型。也就是说，实体里的每一个持久化字段或者属性，在抽象模式里都有一个对应的状态字段，且这个字段的类型和抽象模式里的一样。实体的抽象模式类型源于实体类以及由Java注解提供的元数据信息。
- **BNF（Backus–Naur Form，巴科斯–诺尔范式）** 一种描述高级语言的语法范式。本章中的语法说明使用的就是BNF 标记法。

- 定位 在查询语言的表达式里对实体关系的遍历。定位操作符用点号来表示。
- 路径表达式 定位到实体的状态，或者实体关系的字段的表达式。
- 状态字段 实体的持久化字段。
- 关系字段 实体的持久化关系字段，它的类型与相关实体的抽象模式类型一样。

21.2 使用 Java Persistence 查询语言创建查询

通过使用Java Persistence查询语言，EntityManager.createQuery和EntityManager.createNamedQuery方法可以用来查询数据存储。

createQuery方法用来创建动态查询，它是直接定义在应用的业务逻辑内的查询：

```
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方法通过使用javax.persistence.NamedQuery注解创建静态查询，或者创建在元数据中定义的查询。@NamedQuery注解的name元素指定了查询的名称，它会用在createNamedQuery方法里。@NamedQuery注解的query元素就是查询语句：

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

下面是一个createNamedQuery示例，使用的是@NamedQuery注解：

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

21.2.1 查询里的命名参数

命名参数是以冒号作为前缀的查询参数。使用如下的方法，查询里的命名参数的值可以绑定到一个对象：

```
javax.persistence.Query.setParameter(String name, Object value)
```

在下面的例子里，通过调用Query.setParameter方法，findWithName方法的name参数绑定到查询里的:custName这个命名参数：

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", name)
        .getResultList();
}
```

命名参数是大小写敏感的，既可以用在动态查询里，也可以用在静态查询里。

21.2.2 查询里的位置参数

可以在查询里使用位置参数，而不必使用命名参数。位置参数是以问号(?)开头，紧接一个数字表明参数在查询里的位置。Query.setParameter(int position, Object value)方法可以用来设置参数的值。

下面这个例子重写了findWithName业务方法，这次使用位置参数：

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")
        .setParameter(1, name)
        .getResultList();
}
```

位置参数的编号从1开始。位置参数的值是大小写敏感的，且可以用在动态查询里，也可以用在静态查询里。

21.3 查询语言的简要语法

本节简要介绍这一查询语言的语法，从而使你可以快速开始21.4节的学习。当需要学习语法的更多知识时，可以参考21.5节。

21.3.1 选择语句

选择查询有6个语句：SELECT、FROM、WHERE、GROUP BY、HAVING以及ORDER BY。SELECT和FROM语句是必需的，而WHERE、GROUP BY、HAVING以及ORDER BY语句则是可选的。下面是查询语言的选择语句的高级BNF语法：

```
QL_statement ::= select_clause from_clause
               [where_clause][groupby_clause][having_clause][orderby_clause]
```

- SELECT语句定义了查询所返回对象的类型或者值的类型。
- FROM语句通过声明标识变量定义了查询的范围，这些变量可以在SELECT语句和WHERE语句里引用。标识变量代表了如下所列的某一种元素：
 - 实体的抽象模式名称；
 - 集合关系里的某一个元素；
 - 单值关系里的一个元素；
 - 一对多的关系里“多”的一侧对应的那个集合成员。
- WHERE语句是一个条件表达式，该表达式限定了查询返回的对象或者值。虽然WHERE语句是可选的，但是大多数查询都有WHERE语句。
- GROUP BY语句通过一组属性把查询结果分组。
- HAVING与GROUP BY语句一起使用，通过一个条件表达式进一步限定查询的结果。

□ ORDER BY语句为查询返回的结果（对象或者值）排序。

21.3.2 更新和删除语句

更新和删除语句提供了针对多个实体的批量操作。这些语句的语法如下：

```
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
```

更新和删除语句决定要更新或者删除的实体的类型。WHERE语句可以用来限定更新和删除操作的作用范围。

21.4 查询示例

下面的查询是roster应用中与Player实体有关的查询。关于roster应用，参见20.2节。

21.4.1 简单查询

如果读者还不熟悉查询语言的话，这些简单查询的例子是很好的学习起点。

1. 基本的选择查询

```
SELECT p
FROM Player p
```

□ 要查询的数据 所有的运动员。

□ 描述信息 FROM语句声明了标识变量p，省去了可选的关键字AS。如果要使用AS的话，则可以写成：

```
FROM Player AS
p
```

Player元素是Player实体的抽象模式名称，

□ 参考内容 21.5.3节的“标识变量”。

2. 限制重复记录

```
SELECT DISTINCT
p
FROM Player p
WHERE p.position = ?1
```

□ 要查询的数据 所有position等于查询参数的运动员。

□ 描述信息 DISTINCT关键字用于避免重复记录。

WHERE语句限定了查询运动员的范围，限定条件是Player实体里那些position（Player实体的持久化字段）等于某个特定值的运动员。?1表明是一个位置参数。

□ 参考内容 21.5.5节的“输入参数”以及21.5.6节的“DISTINCT关键字”。

3. 使用命名参数

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = :position AND p.name = :name
```

- **要查询的数据** 所有有着指定的position和指定姓名的运动员。
- **描述信息** position和name元素都是Player实体的持久化字段。通过使用Query.setNamedParameter方法, WHERE语句把实体的这两个值和查询的命名参数相比较。在查询语言里用冒号指定一个命名输入参数。第一个输入参数是:position, 第二个输入参数是:name。

21.4.2 需定位到相关实体的查询

在查询语言里, 表达式可以遍历或者定位到相关的实体。这些表达式就是Java Persistence查询语言与SQL的重要区别。Persistence查询按照实体间的关系定位实体, 而SQL则通过表间的联结操作实现查询。

1. 带关系的简单查询

```
SELECT DISTINCT p
FROM Player p, IN(p.teams) t
```

- **要查询的数据** 所有属于某个运动队的运动员。
- **描述信息** FROM语句声明了两个标识变量: p和t。变量p表示Player实体, 变量t表示相关的Team实体。t的声明引用了刚刚声明的变量p。IN关键字意味着team是相关实体集合。p.teams表达式可以把一个Player定位到相关的Team上去。p.teams表达式里的点号就是定位操作符。

我们还可以使用JOIN语句编写同样功能的查询:

```
SELECT DISTINCT p
FROM Player p JOIN p.teams t
```

查询还可以写成:

```
SELECT DISTINCT p
FROM Player p
WHERE p.team IS NOT EMPTY
```

2. 定位到单值关系字段

使用JOIN语句可以定位到单值的关系字段上:

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = 'soccer' OR l.sport = 'football'
```

在这个例子中, 查询将返回所有美式足球联盟, 或是橄榄球联盟中的运动队。

3. 使用输入参数遍历关系

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) AS t
WHERE t.city = :city
```

- **要查询的数据** 属于特定城市的运动队里的所有运动员。
- **描述信息** 这个查询和刚才的那个例子相似, 只不过增加了一个输入参数。FROM语句里的AS关键字是可选的。在WHERE语句里, 持久化变量city前面的那个点号是一个分隔符,

不是定位操作符。严格来说，表达式可以定位到关系字段（即相关实体），但不是持久化字段。为了访问一个持久化字段，表达式将点号作为分隔符。

表达式不能定位到集合类型的关系字段上。在表达式语法里，集合类型的字段意味着结束，不能再往下定位了。因为teams字段是集合类型，所以WHERE语句不能使用p.teams.city（这个表达式是非法的）。

□ 参考内容 21.5.4节。

4. 遍历多个关系

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league = :league
```

□ 要查询的数据 属于特定联盟的所有运动员。

□ 描述信息 这个查询里的表达式会遍历到两个关系。p.teams表达式表示的是Player-Team关系，而t.league表达式表示的是Team-League关系。

在以往的例子里，输入参数都是String类型的对象，而在这个例子里，参数是一个类型为League的对象。该对象通过WHERE语句里的比较表达式匹配了名为league的关系字段。

5. 依据相关的字段进行定位

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

□ 要查询的数据 参加了特定运动项目的所有运动员。

□ 描述信息 持久化字段sport属于League实体。为了定位到sport这个字段，查询必须首先从Player实体定位到Team（p.teams），然后再从Team定位到League实体（t.league）。因为league的关系字段不是集合类型的，所以可以在它上面继续定位持久化字段sport。

21

21.4.3 使用其他条件表达式的查询

每一个WHERE语句都必须指定条件表达式，而条件表达式有多种类型。在刚才的例子里，条件表达式是比较型表达式，判定是否相等。下面的例子展示的是其他类型的条件表达式。所有条件表达式的信息，参见21.5.5节。

1. LIKE表达式

```
SELECT p
FROM Player p
WHERE p.name LIKE 'Mich%'
```

□ 要查询的数据 名字以Mich开头的运动员。

□ 描述信息 LIKE表达式使用通配符查找字符串，看它是否匹配通配符指定的模式。本例中，查询使用LIKE表达式和%通配符来查找名字以字符串Mich开头的运动员。比如说，Michael和Michelle都匹配这个模式。

□ 参考内容 21.5.5节的“LIKE表达式”。

2. IS NULL表达式

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

- **要查询的数据** 所有不属于任何联盟的运动队。
- **描述信息** IS NULL表达式可以用来检查两个实体间是否建立了关系。本例中，查询检查的是运动队和联盟间的关系，返回的是不属于任何联盟的运动队。
- **参考内容** 21.5.5节的“NULL比较表达式”和“NULL值”。

3. IS EMPTY表达式

```
SELECT p
FROM Player p
WHERE p.teams IS EMPTY
```

- **要查询的数据** 所有不属于任何运动队的运动员。
- **描述信息** Player实体的关系字段teams是一个集合。如果一个运动员不属于任何一个运动队，则运动队（teams）集合为空，条件表达式的结果为TRUE。
- **参考内容** 21.5.5节的“空集合比较表达式”。

4. BETWEEN表达式

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

- **要查询的数据** 所有薪水位于指定范围内的运动员。
- **描述信息** BETWEEN表达式有3个算术表达式：持久化字段p.salary以及两个输入参数 :lowerSalary和:higherSalary。下面的表达式等价于BETWEEN表达式：
p.salary >= :lowerSalary AND p.salary <= :higherSalary
- **参考内容** 21.5.5节的“BETWEEN表达式”。

5. 比较运算符

```
SELECT DISTINCT p1
FROM Player p1, Player p2
WHERE p1.salary > p2.salary AND p2.name = :name
```

- **要查询的数据** 薪水高于某指定运动员的所有运动员。
- **描述信息** FROM语句声明两个标识变量p1和p2，它们同属于Player类型。这里需要两个标识变量是因为WHERE语句需要比较某运动员（p2）的薪水和其他运动员（p1）的薪水。
- **参考内容** 21.5.3节的“标识变量”。

21.4.4 批量更新和删除

下面的例子展示在查询中如何使用UPDATE和DELETE语句。UPDATE和DELETE语句通过条件或者条件组合，操作符合WHERE语句条件的多个实体。UPDATE和DELETE查询语句里的WHERE语句遵循和SELECT查询同样的规则。

1. 更新查询

```
UPDATE Player p
SET p.status = 'inactive'
WHERE p.lastPlayed < :inactiveThresholdDate
```

❑ 描述信息 该查询将一组运动员的状态设置为inactive，前提是该运动员的上一次比赛早于inactiveThresholdDate指定的日期。

2. 删除查询

```
DELETE
FROM Player p
WHERE p.status = 'inactive'
AND p.teams IS EMPTY
```

❑ 描述信息 该查询删除所有状态为inactive且不属于任何运动队的运动员。

21.5 查询语言语法全本

本节内容讨论查询语言的语法，其定义参见Java Persistence API 2.0规范，查阅地址为<http://jcp.org/en/jsr/detail?id=317>。下面的大部分内容均引自该规范。

21.5.1 BNF符号

表21-1描述了本章用到的BNF符号。

表21-1 BNF符号总结

符 号	描 述
::=	符号左边的元素由右边的元素来构造
*	前边的字符可以出现0次或者多次
{...}	大括号里的内容归为一组
[...]	中括号里的内容是可选的
	异或
黑体字	黑体字表示关键字，虽然在BNF标记法里用大写字母表示，但关键字不区分大小写
空白	空白字符，可以是空格、水平制表符，也可以是换行符

21.5.2 Java Persistence查询语言的BNF语法

下面是查询语言的BNF 描述的完整内容：

```
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_spec FETCH 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_expression AND 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

```

21.5.3 FROM语句

FROM语句通过声明一个标识变量定义查询的范围。

1. 标识符

标识符是一个或多个字符的序列。Java编程语言里的标识符，首字符必须是一个有效的字符（字母、\$以及下划线）。Java编程语言在本章以下的内容里简称Java。标识符序列中的每一个后续字符必须是有效的Java字符（字母、数字、\$以及下划线）。详细信息可以参阅Java SE API文档里对Character类的isJavaIdentifierStart 以及isJavaIdentifierPart方法的介绍。问号(?)是查询语言里的保留字符，不能用在标识符里。

查询语言标识符是大小写敏感的，不过有两个例外：

- 关键字；
- 标识变量。

标识符不能使用查询语言的关键字。下面列举的是查询语言的关键字：

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		

不建议把SQL关键字作为标识符使用，因为关键字列表未来扩展后，可能包括其他的SQL保留字。

2. 标识变量

标识变量是在FROM语句里声明的标识符。虽然SELECT语句和WHERE语句可以引用标识变量，却不能声明标识变量。所有的标识变量都必须声明在FROM语句里。

因为本质是标识符，所以标识变量和标识符有着一样的命名惯例和限制条件，其差别在于标识变量是大小写敏感的。例如，标识变量不能使用查询语言的关键字。此外，在一个持久化单元内部，标识变量的名字不能和任何实体或者抽象模式的名字相同。

FROM语句可以包含多个声明，以逗号隔开。声明可以引用其他已经声明过的标识变量（位于其左侧）。在下面的FROM语句中，变量t引用了一个已声明过的变量p：

```
FROM Player p, IN (p.teams) AS t
```

即使没有用在WHERE语句里，标识变量的声明也可以影响查询的结果。例如，比较下面两个查询。下面的查询返回所有的运动员，不管他们是否属于某个运动队：

```
SELECT p
FROM Player p
```

相比较而言，接下来这个查询因为声明了标识变量t，所以返回的是属于某个运动队的所有运动员：

```
SELECT p
FROM Player p, IN (p.teams) AS t
```

下面的查询返回的结果和刚才的例子是一样的，但是WHERE语句的使用使得它的可读性更好了：

```
SELECT p
FROM Player p
WHERE p.teams IS NOT EMPTY
```

标识变量永远是对某一个值的引用，它的类型和在声明里的表达式的类型一样。有两种类型的声明：范围变量和集成员。

3. 范围变量的声明

为了将一个标识变量声明为一个抽象模式类型，需要声明一个范围变量。换句话说，标识变量可以限定实体的抽象模式范围。在接下来的例子中，标识变量p代表Player这个实体的抽象模式：

```
FROM Player p
```

一个范围变量的声明可以包含可选的AS关键字：

```
FROM Player AS p
```

为了获得对象，查询通常使用路径表达式，以在关系中实现定位。但是对于那些不能通过定位获取的对象，可以使用范围变量声明指定一个查询起始点，或者说根。

如果查询要比较的是同一个抽象模式类型的多个值，FROM语句必须为这个抽象模式声明多个标识变量：

```
FROM Player p1, Player p2
```

这种查询的例子，参见21.4.3节的“比较运算符”。

4. 集成员的声明

在一对多的关系中，“多”的一方包含了实体的集合，可以用一个标识变量表示这个集合的一个成员。标识变量声明中的路径表达式将沿着在抽象模式里定义的关系来访问集合的成员。（有关路径表达式的更多信息，可以参阅21.5.4节。）因为路径表达式可以基于另外一个路径表达式，定位可以遍历多个关系，详见21.4.2节的“遍历多个关系”。

一个集成员的声明必须包含在IN操作符内，但是可以省略掉可选的AS关键字。

在下面这个例子中，实体由一个名为Player的抽象模式表示，且有一个名为teams的关系字段。标识变量叫做t，代表的是teams集合的一个成员。

```
FROM Player p, IN (p.teams) t
```

5. 联结

JOIN操作符用于在实体关系间遍历，从功能上来说和IN类似。

在下面的例子中，联结查询跨越了customer和orders之间的实体关系：

```
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
```

上述例子和下面使用IN操作符的查询是等效的:

```
SELECT c
FROM Customer c, IN(c.orders) o
WHERE c.status = 1 AND o.totalPrice > 10000
```

还可以联结一个单值的关系:

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = :sport
```

对于LEFT JOIN (或者LEFT OUTER JOIN) 来说, 即使不指定联结条件, 查询也可以返回一组实体。OUTER关键字是可选的。

```
SELECT c.name, o.totalPrice
FROM Order o LEFT JOIN o.customer c
```

FETCH JOIN是一种联结运算, 作为查询结果的副产品, 还返回相关的实体。在下面的例子里, 查询返回的是一组部门, 作为副产品, 也会返回相关部门的员工, 即使在SELECT语句里并没有显式地获取员工信息。

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

21.5.4 路径表达式

路径表达式是查询语言语法里一项重要的内容。之所以这么说是基于下面这么几个因素。首先, 路径表达式定义了通过抽象模式里的关系来定位的路径。这些路径定义影响查询范围以及结果。其次, 路径表达式可以出现在查询的所有主要语句 (SELECT、DELETE、HAVING、UPDATE、WHERE、FROM、GROUP BY、ORDER BY) 里。最后, 虽然很多查询语句在SQL中也有, 但是路径表达式也扩展了一些SQL里没有的内容。

1. 路径表达式的例子

在下面这个例子中, WHERE语句包含了一个单值的路径表达式 (single_valued_path_expression), p是一个标识变量, salary是Player实体的持久化字段:

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

在下面这个例子中, WHERE语句也包含一个单值的路径表达式, t是一个标识变量, league是一个单值关系字段, sport则是league的一个持久化字段。

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

在下面这个例子中，WHERE语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`是一个集合类型的关系字段。

```
SELECT DISTINCT p
FROM Player p
WHERE p.teams IS EMPTY
```

2. 表达式的类型

路径表达式的类型指的是表达式中最后的元素对应的类型，该元素代表了如下几种对象的类型：

- 持久化字段；
- 单值关系的字段；
- 集合关系的字段。

例如，表达式`p.salary`的类型是`double`，这是因为最后的元素`salary`是`double`类型的。在表达式`p.teams`里，终止元素是集合关系的字段（即`teams`）。表达式的类型是抽象模式类型（`Team`）的集合。因为`Team`是`Team`实体对应的抽象模式的名字，这个类型会映射到实体上。有关抽象模式类型映射的更多内容，参见21.5.6节的“返回类型”。

3. 定位

路径表达式使得查询可以沿着实体间的关系进行。表达式最后的元素决定定位是否合法。如果一个表达式包含了单值关系字段，则定位可以精确到该字段相关的对象。不过，表达式的定位不能超越持久化字段或者值是集合类型的关系字段。例如，`p.teams.league.sport`是非法的表达式，因为`teams`是集合类型的关系字段。为了定位到`sport`字段，FROM语句可以定义一个标识变量`t`来代表`teams`字段：

```
FROM Player AS p, IN (p.teams) t
WHERE t.league.sport = 'soccer'
```

21.5.5 WHERE语句

WHERE语句指定条件表达式以限定查询的返回值。查询将返回数据存储里所有满足条件（条件表达式为TRUE）的相关值。虽然大多数查询语句都有WHERE语句，但WHERE语句是可选的。如果没使用WHERE语句，查询就会返回所有的值。WHERE语句的简明语法如下：

```
where_clause ::= WHERE conditional_expression
```

1. 常量

常量有4种：字符串型常量、数值型常量、布尔型常量以及枚举型常量。

- 字符串型常量 字符串型常量是单引号里的内容，如'Duke'。

如果字符串类型的常量本身也包含有单引号，可以用两个单引号来表示：'Duke''s'。和Java语言里的String相似，查询语言里的字符串型常量使用的是Unicode编码格式。

- 数值型常量 有两种类型的数值型常量：精确数和约数。

精确型的数值是不带小数点的数值，比如65、-233以及+12。使用Java整数的语法，精确型的数值范围和Java语法里的long数据类型表示的数值范围一样。

约数常量是使用科学计数法表述的数值，如57、-85.7和+2.1。使用Java 浮点数的语法，约数能表示的范围和Java语言里double数据类型表示的数值范围一样。

- ❑ **布尔型常量** 布尔型的常量是TRUE或者FALSE。这两个值不是大小写敏感的。
- ❑ **枚举型常量** Java Persistence查询语言支持枚举型的常量，使用Java枚举型常量语法。枚举类名必须指定为完全限定类名：

```
SELECT e
FROM Employee e
WHERE e.status = com.xyz.EmployeeStatus.FULL_TIME
```

2. 输入参数

输入参数可以是命名参数，也可以是位置参数。

- ❑ **命名参数** 命名参数是用冒号开头的字符串，比如:name。
- ❑ **位置参数** 位置参数使用问号(?)开头，紧跟着一个数字来表明位置。例如，第一个输入参数表示为?1，第二个输入参数表示为?2，以此类推。

输入参数还要遵照下面的规则：

- ❑ 只能用在WHERE或者HAVING语句中；
- ❑ 位置参数必须用数字来表明位置，1表示第一个参数；
- ❑ 在一个查询里，命名参数和位置参数不能混用；
- ❑ 命名参数是大小写敏感的。

3. 条件表达式

WHERE语句由一个条件表达式构成，该表达式在同一优先级里是按照从左到右的顺序求值的。

我们可以用小括号修改求值的顺序。

4. 运算符及其优先级

表21-2列举了查询语言的运算符，以优先级降序进行排列。

表21-2 查询语言的运算及优先级

类 型	优先级顺序
定位	. (点)
数学运算	+ (一元运算符)
	*/ (乘除运算)
	+ (加减运算)
比较运算	=
	>
	>=
	<
	<=
	<> (不等于)

(续)

类 型	优先级顺序
逻辑运算	[NOT] BETWEEN
	[NOT] LIKE
	[NOT] IN
	IS [NOT] NULL
	IS [NOT] EMPTY
	[NOT] MEMBER OF
	NOT
	AND
	OR

5. BETWEEN表达式

BETWEEN表达式判定算术表达式的值是否落在指定的范围内。

下面这两个表达式是等效的：

```
p.age BETWEEN 15 AND 19
p.age >= 15 AND p.age <= 19
```

下面的这两个表达式也是等效的：

```
p.age NOT BETWEEN 15 AND 19
p.age < 15 OR p.age > 19
```

如果算术表达式有NULL值，则BETWEEN表达式的求值结果不确定。

6. IN表达式

IN表达式判定一个字符串是否属于一个字符串型常量的集合，或者某个数字是否出现在一组数字中。

路径表达式必须有一个字符串或者数字类型的值。如果路径表达式有一个NULL值，则IN表达式的求值结果不确定。

在下面的例子中，如果country是UK，则表达式求值结果为TRUE。如果country为Peru，则表达式求值结果为FALSE。

```
o.country IN ('UK', 'US', 'France')
```

也可以使用输入参数：

```
o.country IN ('UK', 'US', 'France', :country)
```

7. LIKE表达式

LIKE表达式判断某字符串是否匹配使用了通配符的模式。

路径表达式必须有一个字符串或者数值类型的值。如果该值为NULL，LIKE表达式的求值结果不确定。模式是一个可包含通配符的字符串常量。下划线()通配符代表任何单个的字符，%通配符代表了0个或者多个字符。ESCAPE语句为模式里的通配字符指定了一个转义字符。表21-3展示了一些使用LIKE语句的例子。

表21-3 LIKE表达式示例

表 达 式	TRUE	FALSE
address.phone LIKE '12%3'	'123'	'1234'
	'12993'	
asentence.word LIKE 'l_se'	'lose'	'loose'
aword.underscored LIKE '_%' ESCAPE '\'	'_foo'	'bar'
address.phone NOT LIKE '12%3'	'1234'	'123'
		'12993'

8. NULL比较表达式

NULL比较表达式用于测试单值路径表达式或者一个输入参数的值是否为NULL。通常，NULL比较表达式用来测试某个单值的关系是否已设置：

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

该查询找出没有设置league关系的所有运动队。注意它与下面的查询不是等效的：

```
SELECT t
FROM Team t
WHERE t.league = NULL
```

使用等号操作符(=)与NULL比较，其结果永远不确定，即使在关系没有设置的情况下也是这样。第二个查询的返回结果永远是空。

9. 空集合比较表达式

IS [NOT] EMPTY比较表达式测试的是集合值型的路径表达式是否包含元素。换句话说，它测试的是是否设置了一个集合型值的关系。

如果集合值型的路径表达式为NULL，空集合比较表达式返回NULL值。

下面这个例子查找不包含任何订单项的所有订单：

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

10. 集合成员表达式

[NOT] MEMBER [OF]集合成员表达式判断某值是不是集合的成员。值和集合成员的类型必须一致。

如果集合型值或者单值路径表达式不确定，则集合成员表达式的求值结果不确定。如果集合型值路径表达式代表一个空集合，则集合成员表达式的结果为FALSE。

OF关键字是可选的。

下面的例子测试了一个订单项是不是属于某订单：

```
SELECT o
FROM Order o
WHERE :lineItem MEMBER OF o.lineItems
```

11. 子查询

子查询可以用在查询的WHERE语句或者HAVING语句里。子查询必须用小括号括起来。

下面的例子是要找出所有下订单数多于10个的客户：

```
SELECT c
FROM Customer c
WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

子查询可以包含EXISTS、ALL以及ANY表达式。

□ **EXISTS表达式** [NOT] EXISTS表达式用于子查询，仅当子查询的结果包含一个或者多个值的时候值为TRUE，否则为FALSE。

下面的例子找出所有配偶也是本公司员工的人：

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
    SELECT spouseEmp
    FROM Employee spouseEmp
    WHERE spouseEmp = emp.spouse)
```

□ **ALL和ANY表达式** ALL表达式用于子查询，如果所有子查询返回的值都为TRUE，或者子查询结果为空，则ALL表达式为TRUE。

ANY表达式用于子查询，如果子查询返回的结果有一个为TRUE，则ANY表达式为TRUE。如果子查询的结果为空，或者所有返回结果都为FALSE，则ANY表达式为FALSE。SOME关键字是ANY关键字的同义词。

ALL和ANY表达式要和=、<、<=、>、>=以及<>等比较运算符结合起来使用。

下面的例子找出薪水比自己的部门经理薪水还高的所有员工：

```
SELECT emp
FROM Employee emp
WHERE emp.salary > ALL (
    SELECT m.salary
    FROM Manager m
    WHERE m.department = emp.department)
```

12. 函数型表达式

查询语言包括一些字符串、数学以及日期时间相关的函数，可以用在查询的SELECT、WHERE或者HAVING语句里。这些函数已列举在表21-4、表21-5和表21-6里。

表21-4 字符串相关的函数

函数语法	返回值类型
CONCAT(String, String)	String
LENGTH(String)	int

(续)

函数语法	返回值类型
LOCATE(String, String [, start])	int
SUBSTRING(String, start, length)	String
TRIM([[LEADING TRAILING BOTH] char) FROM] (String)	String
LOWER(String)	String
UPPER(String)	String

在表21-4里，start和length参数的类型是int，表示的是在String参数中的位置。字符串的第一个字符的位置是1。

CONCAT函数把两个字符串拼接成一个字符串。

LENGTH函数返回的是字符串里字符的个数，返回值为整数。

LOCATE函数返回的是某字符串在给定字符串里出现的位置。这个函数返回的是它在字符串里首次出现的位置，返回值为整数。第一个参数是被定位的字符串，第二个参数是要查找的字符串。第三个参数是可选的，它是一个整数，代表的是查找开始的位置。在默认情况下，LOCATE是从字符串的开始进行查找的。字符串的第一个位置是1。如果字符串没找到，则LOCATE返回0。

SUBSTRING函数返回的是某个字符串的子串。函数返回的子串，由第二个参数表示的开始位置以及第三个参数表示的长度决定。

TRIM函数可以截去字符串开头和（或）结尾处的指定字符。如果没指定字符，则TRIM函数会截去字符串开头或者结尾处的空格。如果指定了可选参数LEADING，TRIM函数只截去字符串开头的指定字符；如果指定了可选参数TRAILING，TRIM函数只截去字符串结尾的指定字符；默认值是BOTH，也就是说，TRIM函数会截去字符串开头和结尾的指定字符。

LOWER函数是把字符串里的字符全部变成小写字符，而UPPER函数是把字符串里的字符全部变成大写字符。

表21-5里的number 参数可以是int、float或者double类型。

表21-5 数学相关的函数

函数语法	返回值
ABS(number)	Int、float或者double
MOD(int, int)	Int
SQRT(double)	double
SIZE(Collection)	Int

ABS函数返回的是输入参数的绝对值，输入参数是数值表达式，返回值的类型和输入参数的类型一样（可能为int、float或者double）。

MOD函数返回的是第一个参数被第二个参数相除以后的余数。

SQRT函数返回的是一个数字的平方根。

SIZE函数返回的是一个整数，表示一个给定的集合里面元素的个数。

在表21-6里，日期/时间函数返回的是数据库服务器上的日期、时间或者时间戳。

表21-6 日期/时间相关的表达式

函数语法	返回值类型
CURRENT_DATE	java.sql.Date
CURRENT_TIME	java.sql.Time
CURRENT_TIMESTAMP	java.sql.Timestamp

13. case表达式

case表达式和Java编程语言里的case关键字相似，也是依据一个条件改变程序的执行流程。CASE关键字指明了一个case表达式的开始，表达式的结束用END关键字表示。WHEN和THEN关键字定义了一个分支条件，ELSE关键字定义了默认的条件，即其他条件都不满足的情况。

下面的查询选择一个人的名字，条件字符串依赖于Person实体的子类型。如果子类型为Student，就返回字符串kid。如果子类型为Guardian或者Staff，就返回字符串adult。如果实体是Person的其他子类型，返回的字符串为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
```

下面的查询设置了针对不同级别客户的折扣。金卡用户可以得到20%的折扣，银卡用户可以得到15%的折扣，铜卡用户可以得到10%的折扣，其他类型的用户则可以得到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
```

14. NULL值

如果引用的目标不在持久化数据存储器里，那么目标对象为NULL。对于包含了NULL的条件表达式，查询语言使用SQL92定义的语义。简要地列举这些语义如下：

- 如果比较运算或者算术运算有不确定的值，求值结果为NULL值。
- 两个NULL值其实并不相等，比较两个NULL值，求值结果不确定。
- IS NULL会将一个为NULL的持久化字段（或者单值关系字段）转换为TRUE。IS NOT NULL则

把它们转换为FALSE。

- 布尔类型的运算符和条件测试使用由表21-7和表21-8定义的值逻辑。在这两个表里，T代表了TRUE，F代表了FALSE，U代表不确定（unknown）。

表21-7 AND运算逻辑

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

表21-8 OR运算逻辑

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

15. 相等语义

在查询语言里，只有类型相同的值才可以比较。不过，这个规则有个例外：精确值和约数值是可以作比较的。在这种比较中，需要做的类型转换自然要遵照Java编程语言里的转换规则——数字提升。

查询语言对待要比较的值，就好像它们是Java类型，而不是底层数据存储里的类型。例如，一个持久化的字段，有可能是一个整数或者是为NULL，必须用一个Integer对象来表示，不能定义为一个基本类型int。必须使用这种处理方式，因为Java对象可以为NULL，而基本类型int不可能为NULL。

两个字符串只有在包含的所有字符都一样且顺序相同的情况下才相等。在做字符串相等比较时，把字符串的头尾空格去掉非常重要。例如，字符串'abc'和字符串'abc'并不相等。

来自于同一个抽象模式类型的两个实体，只有在它们的主键有相同值的情况下才相等。表21-9展示了NOT的运算逻辑，表21-10展示的是条件测试的真值表。

表21-9 NOT运算逻辑

NOT值	值
T	F
F	T
U	U

表21-10 条件测试真值表

条件测试	T	F	U
表达式IS TRUE	T	F	F
表达式IS FALSE	F	T	F
表达式为未知	F	F	T

21.5.6 SELECT语句

SELECT语句定义了查询所返回对象或者值的类型。

1. 返回类型

SELECT语句的返回类型是由SELECT语句里表达式的类型确定的。如果用了多个表达式，查询的结果则为对象数组Object[]，数组里元素的顺序就是SELECT语句里表达式的顺序，类型也和每一个表达式的返回值类型一致。

SELECT语句不能指定集合值类型的表达式。例如，SELECT语句p.teams是无效的，因为teams是集合类型的数据。不过，下面这个查询是有效的，因为t是teams集合里的一个单值成员。

```
SELECT t
FROM Player p, IN (p.teams) t
```

下面是一个查询的例子，SELECT语句里有多个表达式：

```
SELECT c.name, c.country.name
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

这个查询返回的是Object类型的数组（Object[]）：第一个数组元素是字符串类型，表示的是客户的姓名；第二个数组元素是字符串类型，表示的是客户所在国家的名称。

查询的结果可能是一个聚合函数的结果，列举在表21-11里。

表21-11 选择语句里的聚合函数

名 称	返回类型	描述信息
AVG	Double	返回的是字段的平均值
COUNT	Long	返回的是结果的记录数
MAX	和字段的类型一样	返回的是结果里的最大值
MIN	和字段的类型一样	返回的是结果里的最小值
SUM	Long（如果字段为整型） Double（如果字段为浮点型） BigInteger（如果字段为BigInteger型） BigDecimal（如果字段为BigDecimal型）	返回的是所有结果的求和结果

如果在SELECT语句里使用了聚合函数（AVG、COUNT、MAX、MIN还有SUM），则需要遵照如下的规则：

- ❑ 如果传入参数为空，则AVG、MAX、MIN和SUM函数返回空。
- ❑ 如果查询没有符合条件的结果，则COUNT函数返回0。

下面的例子返回的是订单数量的平均值：

```
SELECT AVG(o.quantity)
FROM Order o
```

下面的例子返回的是Roxane Coss所下订单中所有订单项的总价：

```
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

下面的例子返回的是订单的总数量：

```
SELECT COUNT(o)
FROM Order o
```

下面的例子返回的是Hal Incandenza所下订单中所有的订单项总数：

```
SELECT COUNT(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Incandenza' AND c.firstname = 'Hal'
```

2. DISTINCT关键字

DISTINCT关键字剔除返回结果中的重复记录。如果某查询返回java.util.Collection类型的数据，由于Collection是允许数据重复的，必须指定 DISTINCT关键字以剔除重复记录。

3. 构造表达式

构造表达式（constructor expression）使得我们可以以Java实例的方式保存查询的结果，而不是以对象数组Object[]的方式保存。

下面的查询为每一个匹配了WHERE语句的customer创建一个CustomerDetail实例。CustomerDetail保存了客户的姓名和所在国家的名称。所以查询返回的是一个List，List里的每一个元素都是CustomerDetail实例：

```
SELECT NEW com.xyz.CustomerDetail(c.name, c.country.name)
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

21.5.7 ORDER BY语句

从名字就可以看出来，ORDER BY语句将对查询返回的值或对象进行排序。

如果ORDER BY语句包含了多个元素，按照从左到右的顺序，决定排序的优先顺序。

ASC关键字指定按照升序排列，这是默认的排列方式，而DESC关键字则是将查询结果按照降序排列。

当使用ORDER BY语句的时候，SELECT语句必须返回可以排序的对象或值的集合。只能为SELECT语句返回的对象或值排序。例如，下面的查询是有效的，因为ORDER BY语句使用的对象就是SELECT语句返回的：

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity, o.totalcost
```

下面的例子是无效的，因为ORDER BY语句使用的值不是由SELECT语句返回的：

```
SELECT p.product_name
FROM Order o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname = 'Faehmel' AND c.firstname = 'Robert'
ORDER BY o.quantity
```


21.5.8 GROUP BY和HAVING语句

GROUP BY语句使得开发人员可以通过一组属性为查询结果分组。

下面的查询，依据客户所在的国家以及每一个国家里所有的客户数量对客户进行分组：

```
SELECT c.country, COUNT(c)
FROM Customer c GROUP BY c.country
```

HAVING语句和GROUP BY语句配合使用，用来进一步限定查询的返回值。

下面的查询依据客户的状态为订单进行分组，并且返回客户的状态以及具有同样状态的客户所有订单总价款的平均值。此外，返回结果还限定仅包含状态值为1、2或者3的客户，而其他状态的客户的订单不予考虑：

```
SELECT c.status, AVG(o.totalPrice)
FROM Order o JOIN o.customer c
GROUP BY c.status HAVING c.status IN (1, 2, 3)
```

Criteria API用于创建定义查询的对象，以此实现对实体及其持久化状态的查询。Criteria查询基于线程安全且跨平台的Java语言API编写。这种查询不依赖于具体的数据存储方式。

本章内容

- Criteria和Metamodel API概述
- 使用Metamodel API为实体类建模
- 使用Criteria API和Metamodel API创建类型安全的基本查询

22.1 Criteria 和 Metamodel API 概述

与JPQL相似，Criteria API基于持久化实体的抽象模式、实体间的关系以及嵌入的对象完成操作。通过调用Java Persistence API中的实体操作方法，开发人员可以利用Criteria API查找、修改以及删除持久化实体，以实现对象抽象模式的操作。Metamodel API与Criteria API结合在一起，为Criteria查询创建持久化实体类的模型。

Criteria API与JPQL密切相关，对实体的查询方式也几近相同。熟悉JPQL语法的开发人员可以在Criteria API中找到对象级别的对应操作方法。

下面的代码实现一个简单的Criteria查询，在数据源中查找并返回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();
```

下面是与之等效的JPQL查询：

```
SELECT p
FROM Pet p
```

这段代码展示了创建Criteria查询的基本步骤。

- (1) 使用EntityManager的实例创建一个CriteriaBuilder对象。
- (2) 通过创建CriteriaQuery接口的实例创建一个查询对象，这个查询对象的属性由传入的查

询条件来更改。

(3) 通过调用CriteriaQuery对象的from方法设置查询的根。

(4) 通过调用CriteriaQuery对象的select方法指定查询结果的类型。

(5) 通过创建一个TypedQuery<T>实例为查询的执行做准备，并指定查询结果的类型。

(6) 通过调用TypedQuery<T>对象中的getResultList方法执行查询。由于该查询返回实体的集合，因而其结果保存在List中。

本章将详细介绍上述过程中每一个步骤的具体任务。

为了创建CriteriaBuilder实例，需要调用EntityManager实例中的getCriteriaBuilder方法：

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

查询对象通过CriteriaBuilder的实例创建：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

该查询将返回Pet实体的全部实例，因此在创建CriteriaQuery对象时需指定查询的类型，以实现类型安全的查询。

通过调用查询对象的from方法，实现对查询中FROM语句以及查询根的设定。

```
Root<Pet> pet = cq.from(Pet.class);
```

通过调用查询对象的select方法并传入查询的根，实现对查询中SELECT语句的设定。

```
cq.select(pet);
```

此时，查询对象可以用于创建一个可在数据源上执行查询的TypedQuery<T>对象。对查询对象的任何修改都将被捕获并记录，以形成一个最终执行的查询：

```
TypedQuery<Pet> q = em.createQuery(cq);
```

通过调用查询对象的getResultList方法，这种面向类型的查询得以执行。因为查询可能返回实体的多个实例，结果保存在一个List<Pet>的集合对象里。

```
List<Pet> allPets = q.getResultList();
```

22.2 使用 Metamodel API 为实体类建模

Metamodel API用于在特定的持久化单元中创建托管实体的元模型（metamodel）。对于包中的每一个实体类，元模型类的名称以下划线结尾，其中的成员变量与实体类的持久化字段或属性相对应。

下面这个实体类（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;
    ...
}

```

对应的元模型类定义如下：

```

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;
}

```

元模型类及其属性用在Criteria查询中，代表托管的实体类、类的持久化状态，以及类之间的关联关系。

使用元模型类

与实体类对应的元模型类的类型如下：

```

javax.persistence.metamodel.EntityType<T>

```

元模型类通常在开发或运行阶段由注解处理器生成。使用Criteria查询的应用开发人员，可以使用持久化提供者提供的注解处理器产生静态的元模型类，或者通过调用查询根对象上的getModel方法获得元模型类，或者首先获得一个Metamodel接口的实例，再将实体的类型传递给该实例的entity方法，以获得元模型类。

下面的代码片段展示了如何通过调用Root<T>.getModel方法，获得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();

```

下面的代码片段展示了如何使用EntityManager.getMetamodel方法获得元模型的实例，进而调用元模型实例的entity方法，最终获得Pet实体的元模型类。

```

EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);

```

22.3 使用 Criteria API 和 Metamodel API 创建类型安全的基本查询

Criteria查询的基本语法与JPQL查询相似，由一个SELECT语句、一个FROM语句，以及一个可选的WHERE语句构成。Criteria查询使用Java编程语言对象设定这些语句，因而构造的查询是类型安全的。

22.3.1 创建Criteria查询

javax.persistence.criteria.CriteriaBuilder接口用于构造：

- Criteria查询；
- 选择；
- 表达式；
- 断言 (predicate)；
- 排序。

为了获得CriteriaBuilder接口的实例，可以调用EntityManager或EntityManagerFactory实例的getCriteriaBuilder方法。

下面的代码展示了如何使用EntityManager.getCriteriaBuilder方法，获得CriteriaBuilder的实例：

```
EntityManager em = ...;  
CriteriaBuilder cb = em.getCriteriaBuilder();
```

通过获得下面接口的实例创建Criteria查询：

```
javax.persistence.criteria.CriteriaQuery
```

CriteriaQuery对象定义了一个遍历一个或多个实体的特定查询。通过调用某个CriteriaBuilder.createQuery方法，可以获得CriteriaQuery的实例。为了创建类型安全的查询，可以以如下方式调用CriteriaBuilder.createQuery方法：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

CriteriaQuery对象的类型，应该设定为与查询所期望得到的结果相匹配的类型。在上面的代码中，对象的类型设定为CriteriaQuery<Pet>，表明此查询将查找类型为Pet的实体的实例。

在下面的代码片段中，CriteriaQuery对象用于创建一个返回类型为String的查询：

```
CriteriaQuery<String> cq = cb.createQuery(String.class);
```

22.3.2 查询根

对于特定的CriteriaQuery对象来说，所有的查询源自根实体，即查询根 (query root)。它类似于JPQL中的FROM语句。

通过调用CriteriaQuery实例的from方法可以创建查询根。from方法的参数可以是实体类或者实体的EntityType<T>实例。

下面的代码设置Pet实体的查询根：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);  
Root<Pet> pet = cq.from(Pet.class);
```

下面的代码使用EntityType<T>的实例，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，这种情况通常用于在多个实体中进行查询。

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);
```

22.3.3 使用join查询关联关系

对于关联的实体类，其查询必须定义一个相关实体间的关联（join），并通过调用查询根对象或另一个联结对象的某个From.join方法来实现。join方法类似于JPQL中的JOIN关键字。

联结的目标使用EntityType<T>类型的元模型类，来指定被联结实体的持久化字段或属性。

join方法返回Join<X, Y>类型的对象。其中，X代表源实体，Y代表联结目标。在下面的代码片段中，Pet是源实体，而Owner是目标：

```
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);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
```

联结可以以“链”的方式查找所有与目标实体关联的实体，而无需为每一个联结都创建一个Join<X, Y>实例：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
EntityType<Owner> Owner_ = m.entity(Owner.class);

Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = cq.join(Pet_.owners).join(Owner_.addresses);
```

22.3.4 Criteria查询中的路径定位

Path对象用在Criteria查询中的SELECT和WHERE语句中，可以是查询的根实体、联结实体或其他Path对象。Path.get方法用于定位到查询中的实体属性。

get方法的参数是对应实体元模型类的属性，可以在元模型类中通过@SingularAttribute注解的单值类型属性（single-valued attribute），也可以是通过@CollectionAttribute、@SetAttribute、@ListAttribute或@MapAttribute注解的集合属性。

下面的查询返回数据存储中所有宠物的名称。在查询根pet上调用get方法，并将Pet实体的元模型类Pet_中的属性name作为参数传给get方法：

```
CriteriaQuery<String> cq = cb.createQuery(String.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
cq.select(pet.get(Pet_.name));
```

22.3.5 过滤Criteria查询结果

调用CriteriaQuery.where方法，可以设置CriteriaQuery对象的查询条件，以此对查询结果进行过滤。调用where方法类似于设定JPQL查询中的WHERE语句。

where方法依据Expression接口实例中的表达式，对返回结果进行过滤。Expression和CriteriaBuilder接口中提供的方法可以创建Expression实例。

1. Expression接口中的方法

Expression对象用于实现查询中的SELECT、WHERE或HAVING语句。表22-1展示了可以结合Expression对象使用的用于设定查询条件的方法。

表22-1 Expression接口中用于设定查询条件的方法

方 法	描 述
isNull	验证表达式是否为null
isNotNull	验证表达式是否不为null
in	验证表达式是否在一组取值范围内

下面的查询使用Expression.isNull方法，查找所有color属性为null的宠物：

```
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(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);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).in("brown", "black"));
```

in方法还可以用于验证属性是否是集合中的成员。

2. CriteriaBuilder接口中的表达式方法

CriteriaBuilder接口定义了一组创建表达式的方法。这些方法对应于JPQL中的算术、字符串、日期、时间以及分支运算符和方法。表22-2展示了可结合CriteriaBuilder对象使用的判定方法。

表22-2 CriteriaBuilder接口中的判定方法

判定方法	描 述
equal	验证两个表达式是否相等
notEqual	验证两个表达式是否不相等
gt	验证第一个数值表达式是否大于第二个数值表达式
ge	验证第一个数值表达式是否大于或等于第二个数值表达式
lt	验证第一个数值表达式是否小于第二个数值表达式

(续)

判定方法	描 述
le	验证第一个数值表达式是否小于或等于第二个数值表达式
between	验证第一个表达式的值是否介于第二个和第三个表达式的值之间
like	验证表达式是否匹配给定模式

下面的代码使用CriteriaBuilder.equal方法:

```
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");
...
```

下面的代码使用CriteriaBuilder.gt方法:

```
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);
Date someDate = new Date(...);
cq.where(cb.gt(pet.get(Pet_.birthday), date);
```

下面的代码使用CriteriaBuilder.between方法:

```
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);
Date firstDate = new Date(...);
Date secondDate = new Date(...);
cq.where(cb.between(pet.get(Pet_.birthday), firstDate, secondDate);
```

下面的代码使用CriteriaBuilder.like方法:

```
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.like(pet.get(Pet_.name), "*do");
```

可以使用CriteriaBuilder接口中的多条件判定方法,实现对组合条件的判定,如表22-3所示。

表22-3 CriteriaBuilder接口中的多条件判定方法

方 法	描 述
and	两个布尔表达式的逻辑与
or	两个布尔表达式的逻辑或
not	给定布尔表达式的逻辑非

下面的代码在查询中使用了组合条件判定:


```
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")
        .and(cb.equal(pet.get(Pet_.color), "brown"));
```

22.3.6 处理Criteria查询结果

对于查询返回多个结果的情况,对结果进行组织非常有用。CriteriaBuilder接口定义了orderBy方法,对查询结果按照实体的属性进行排序。CriteriaBuilder接口还定义了groupBy方法(对查询结果按照实体的属性进行分组)以及having方法(对分组的结果按照一定条件进行过滤)。

1. 结果排序

对查询结果进行排序,可以调用CriteriaBuilder.orderBy方法,并作为参数传入Order对象。通过调用CriteriaBuilder.asc方法或CriteriaBuilder.desc方法,可以创建Order对象。asc方法按照传入的表达式参数值升序排列结果。desc方法按照传入的表达式参数值降序排列结果。下面的代码展示了desc方法的用法:

```
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,查询的结果将按照宠物的出生日期从高到低进行排序。也就是说,12月出生的宠物将出现在5月出生的宠物之前。

下面的代码展示了asc方法的用法:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = cq.join(Pet_.owners).join(Owner_.address);
cq.select(pet);
cq.orderBy(cb.asc(address.get(Address_.postalCode)));
```

在这个查询中,查询结果将按照宠物主人的邮政编码从低到高进行排列。也就是说,居住地邮政编码为10001的主人的宠物将出现在居住地邮政编码为90001的主人的宠物之前。

如果多个Order对象传递给orderBy方法,则其优先级将由Order对象在orderBy方法参数列表中的所在位置决定。第一个Order对象有最高优先级。

下面的代码根据多个条件对结果进行排序:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = cq.join(Pet_.owners);
cq.select(pet);
cq.orderBy(cb.asc(owner.get(Owner_.lastName), owner.get(Owner_.firstName)));
```

这个查询的结果将首先按照宠物主人的姓氏字母进行排序,再按照主人的名字进行排序。

2. 结果分组

CriteriaQuery.groupBy方法将对查询结果进行分组。组的设定可以通过给groupBy方法传递表达式来实现:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
```

这个查询返回所有的Pet实体，并按照宠物的颜色进行分组。

CriteriaQuery.having方法与groupBy方法一起使用，以实现对分组结果的过滤。条件表达式作为参数传递给having方法。通过调用having方法，查询结果将按照条件表达式进行过滤：

```
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"));
```

如同前例，在这段代码中，查询结果按照Pet实体的颜色进行分组。然而，查询最终仅返回Pet实体的color属性为brown或blonde的组。也就是说，不会查出颜色为灰色的宠物。

22.3.7 查询执行

为了执行查询，需要创建一个TypedQuery<T>对象，其类型为查询返回结果的类型，方法是将CriteriaQuery对象传递给EntityManager.createQuery方法。

可以通过调用TypedQuery<T>对象的getSingleResult或getResultList方法执行查询。

1. 单值查询结果

TypedQuery<T>.getSingleResult方法用于执行仅有一个返回值的查询：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
Pet result = q.getSingleResult();
```

2. 多值查询结果

TypedQuery<T>.getResultList方法用于执行返回一组对象的查询：

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```

Part 7

第七部分

安 全

本部分介绍基本的安全概念和相关示例。

本 部 分 内 容

- 第 23 章 Java EE 平台安全入门
- 第 24 章 Web 应用安全化入门
- 第 25 章 企业应用安全化入门

第七部分将讨论Web层与企业层应用的安全性需求。每个企业都有可能被很多用户访问的敏感信息，或者有需要保护，但允许通过无保护的开放网络（如因特网）进行访问的信息。

本章将介绍基本的安全概念及安全机制。关于安全概念或安全机制的更多信息，可以参考Java EE 6平台规范中有关安全的章节。该文档可以通过<http://www.jcp.org/en/jsr/detail?id=316>下载。

在本书中，如下各章也会讨论安全性需求。

- 第24章，介绍如何为Web组件（如servlet）增加安全性。

- 第25章，介绍如何为Java EE组件（如企业bean和应用客户端）增加安全性。

本章中讲述的内容和使用的示例需要读者对基本的安全概念有所了解。在阅读本章之前，读者可访问有关Java SE安全的网站<http://docs.oracle.com/javase/6/docs/technotes/guides/security>，了解更多关于安全性的基本概念。

本章内容

- Java EE安全性概述
- 安全机制
- 为容器增加安全性
- 为GlassFish服务器增加安全性
- 使用域、用户、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

23.1 Java EE 安全性概述

企业层与Web层应用由部署于不同容器的组件构成。这些组件结合在一起，构成了多层次的企业应用。组件的安全性由其所处的容器来提供。容器通过两种方式提供安全机制：声明方式和编程方式。

- **声明方式的安全机制** 通过部署描述文件或注解阐述应用组件对安全性的要求。部署描述文件不属于应用，它是一个外部的XML文件，阐述了应用的安全架构，包括角色、访

问控制以及认证方式。关于部署描述文件的更多信息，参见23.3.2节。

注解，也称为元数据，用于在类文件中指定安全信息。部署应用之后，这些安全信息可以被应用部署描述文件使用或覆写。注解使得开发人员不必在XML部署描述文件中编写安全性声明，取而代之的是在代码中增加注解，安全信息将自动生成。在本书中，所有需要增加安全性机制的地方均使用了注解。关于注解的更多信息，参见23.3.1节。

- **编程方式的安全机制** 通常嵌入在应用中，可以动态地决定安全机制。在声明方式的安全机制无法满足应用的安全模型时，编程方式的安全机制非常有用。有关编程方式的安全机制，详见23.3.3节。

23.1.1 简单的安全应用示例

通过研究一个包含Web客户端、用户界面和企业bean业务逻辑的简单应用，可以更好地理解Java EE平台环境中的安全机制。

下面的示例取自Java EE 6规范。Web服务器通过收集来自客户端的用户认证信息，与Web客户端建立一个认证的会话。此时，Web服务器扮演的是一个认证代理的角色。

1. 第一步：初始请求

在本示例的第一步中，Web客户端请求主应用的URL，其过程参见图23-1。

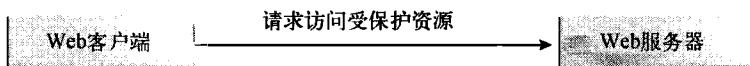


图23-1 初始请求

由于客户端还没有在服务端通过认证，负责处理应用中Web请求的服务器（此处是指Web服务器）检测请求，并调用适合该资源的认证机制。关于认证机制的更多信息，参见23.2节。

2. 第二步：初始认证

Web服务器返回一个表单，Web客户端用此表单收集用于认证用户的数据，如用户名和密码。Web客户端将认证数据提交给Web服务器，由其对数据进行验证，如图23-2所示。验证可以在服务器内部完成，或通过调用底层的安全服务完成。基于验证的结果，Web服务器为用户设置一个安全凭证。

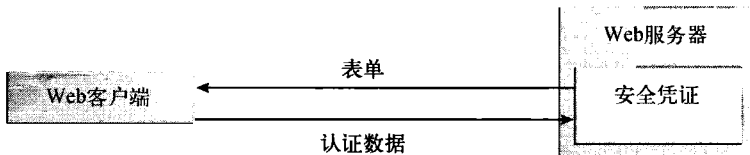


图23-2 初始认证

3. 第三步：URL授权

安全凭证用于判断用户对受限资源的后续访问是否合法。Web服务器向与Web资源相关的安

全策略发起询问，以获知有权访问该资源的角色。安全策略来自注解或部署描述文件。此时，Web容器对照角色验证用户的安全凭证，以判定是否可将用户映射到某个角色。图23-3展示了这个过程。

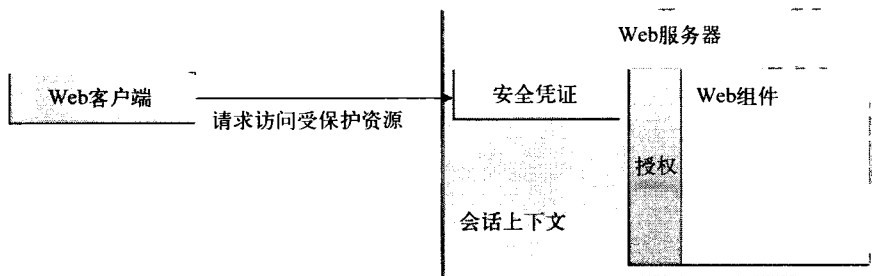


图23-3 URL授权

如果Web服务器能够将用户映射至一个角色，Web服务器的验证过程结束，并得出一个“授权”结果。如果Web服务器不能将用户映射到任何一个角色，Web服务器得出“未授权”。

4. 第四步：执行原始请求

如果用户获得授权，Web服务器返回原始URL请求的结果，如图23-4所示。

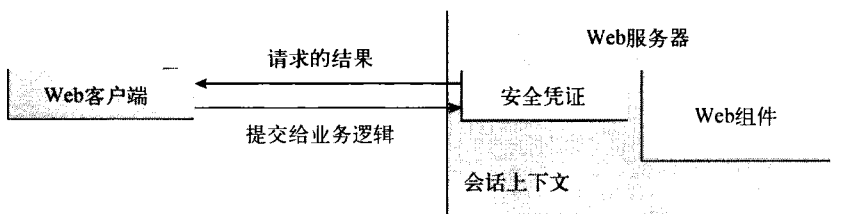


图23-4 执行原始请求

在本例中，返回给客户端的是一个Web页面的URL，用户进而可以将需要处理的表单数据提交给应用组件的业务逻辑进行处理。有关加固Web应用安全性的更多信息，参见第24章“Web应用安全化入门”。

5. 第五步：调用企业bean的业务方法

Web页面使用用户的安全凭证建立起与企业bean之间的安全通道，从而远程调用企业bean中的方法，如图23-5所示。这种安全通道通过两个相关联的、与安全相关的上下文来实现，这两个上下文一个位于Web服务器内，一个位于EJB容器内。

EJB容器负责强制执行企业bean方法上的访问控制。容器向与企业bean关联的安全策略发起询问，以获知可以访问该方法的角色。安全策略来自注解或部署描述文件。对于每一种角色，EJB容器使用与方法调用相关的安全上下文判定是否能将调用者映射至角色。

如果容器能够将调用者的安全凭证映射至一个角色，则容器对调用者的安全评估结束，并得出授权这一结果。如果容器不能将调用者映射至任何一个被允许的角色，则得出一个“未授权”

结果。“未授权”这一结果将导致容器抛出一个异常，并将其返回给发起调用的Web页面。

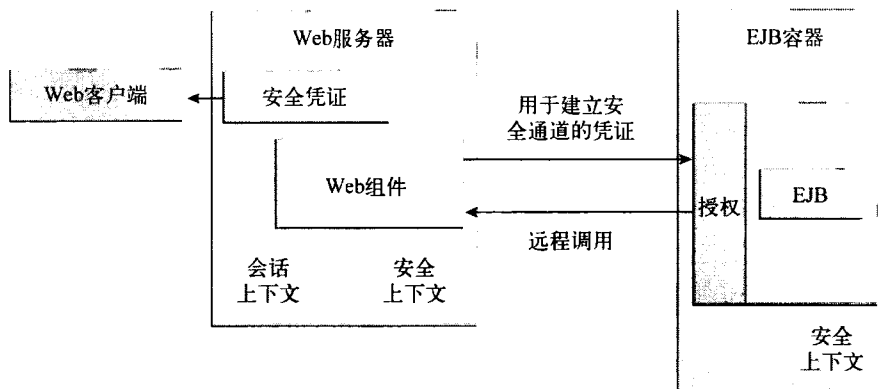


图23-5 调用企业bean的业务方法

如果调用获得授权，则容器将控制权交给企业bean的方法。该bean的执行结果将返回给Web页面，并最终通过Web服务器和Web客户端返回给用户。

23.1.2 安全机制的特性

一个正确实施的安全机制将提供如下功能：

- ❑ 阻止对应用功能、业务逻辑或个人数据的未授权访问（授权功能）；
- ❑ 记录系统用户的所有操作（不可抵赖）；
- ❑ 保护系统，使之免于服务中断以及其他导致服务质量下降的破坏。

理想情况下，一个正确实施的安全机制还将提供如下特性：

- ❑ 易于管理；
- ❑ 对系统用户透明；
- ❑ 应用系统间、不同企业应用系统间的互操作性。

23.1.3 应用安全的特征

Java EE应用由组件构成，其中既有受保护的资源，也有未受保护的资源。通常来说，需要对资源进行保护，仅使授权的用户可以访问。授权机制提供了对受保护资源的访问控制。授权以身份识别与认证为基础。身份识别是一个过程，实现系统对待认证实体的识别。认证也是一个过程，实现对用户身份、设备及计算机系统中其他实体的验证，这个过程通常是访问系统资源的前提。

访问未保护资源时，不一定需要授权与认证。访问资源时不进行认证，通常被认为是未认证或匿名访问。

正确地实现应用的安全机制有助于降低企业所面临的安全性风险，如下所示。

- **认证** 通过认证, 通信实体(如客户端和服务端)之间可以相互证明彼此都是以获得授权的身份进行访问的。这种机制确保了用户对其身份声明的认可。
- **授权或访问控制** 通过授权, 与资源的交互仅限于特定用户或程序的集合, 以此实现完整性、保密性或可用性限制。这种机制确保仅是有权限用户执行操作或访问数据。
- **数据完整性** 通过这种方式证明信息没有被第三方或信息源之外的其他实体所修改。例如, 开放网络数据的接收者必须能够检测并抛弃在发送后被修改的消息。这种机制确保仅有授权用户可以修改数据。
- **数据保密性** 通过这种方式, 数据仅对被授权访问的用户可见。这种机制确保仅有授权用户可以查看敏感数据。
- **不可抵赖性** 这种方式用于证明对数据进行过操作的用户不能对其所作所为抵赖。这种机制确保事务不可抵赖。
- **服务质量** 这种方式用于利用各种流量控制技术, 为选定的应用提供更好的传输服务。
- **审计** 这种方式用于捕捉安全事件的防篡改记录, 以便可以评估安全策略或机制的有效性。为了实现审计, 系统需要保存事务记录和安全信息。

23.2 安全机制

在制定应用的安全等级及类型时, 需要考虑应用自身的特征。本节将探讨为Java EE应用增加安全性的常用机制及其特征。每一种机制都可以独立使用, 也可以与其他机制配合使用, 从而为特定需求实现定制化的保护方式。

23.2.1 Java SE 安全机制

Java SE提供对多种安全特征和机制的支持。

- **Java认证与授权服务** JAAS (Java Authentication and Authorization Service, Java认证与授权服务) 是一组API, 能够为服务提供基于用户的认证, 还能够强制要求实施访问控制。JAAS提供一种插件式、可扩展的框架, 使开发人员能够以编程的方式实现认证与授权。JAAS属于Java SE的核心API, 是Java EE安全机制的底层技术。
- **Java通用安全服务** Java GSS-API (Java Generic Security Service, Java通用安全服务) 是一组基于令牌(token)的API, 为相互通信的应用提供安全的信息交换。GSS-API为应用开发人员提供一种对于不同底层安全机制之上的安全服务(如Kerberos)的统一访问方式。
- **Java加密扩展** JCE (Java Cryptography Extension, Java加密扩展) 为加密、密钥产生和密钥协议, 以及MAC (Message Authentication Code, 消息认证码) 算法, 提供了一个框架和多种实现方法。其支持的加密方式包括对称加密、非对称加密、块加密(block cipher)和流加密(stream cipher)。块加密是在多个字节上进行, 流加密则一次操作一个字节。JCE软件支持安全信息流和对象保护(sealed object)。

- **Java安全套接字扩展 JSSE** (Java Secure Socket Extension, Java安全套接字扩展) 为Java语言版本的SSL和TLS协议提供了一个框架及一种实现方案, 并且提供数据加密功能、服务器认证、消息完整性检查以及可选的客户端认证, 从而实现安全的因特网通信。
- **简单认证和安全层 SASL** (Simple Authentication and Security Layer, 简单认证和安全层) 是一个因特网标准 (RFC 2222), 该标准为认证以及在客户端和服务器间建立安全连接 (非强制性要求) 定义了一个协议。SASL定义了认证数据的交换方式, 但本身不定义认证数据的内容。SASL是一个框架, 定义了能够运行在这个框架下的认证信息和认证语法的认证协议。

Java SE同时提供一组工具, 以管理密钥仓库 (keystore)、证书以及策略文件, 产生及验证JAR文件签名, 以及获取、排列和管理Kerberos票据。

有关Java SE安全性的更多信息, 可以参考 <http://docs.oracle.com/javase/6/docs/technotes/guides/security/>。

23.2.2 Java EE安全机制

Java EE安全服务由组件容器提供, 可以通过声明方式或编程方式实现 (参见23.3节)。Java EE安全服务提供一种健壮的、易于配置的安全机制, 在不同层面上实现对用户的认证和授权访问应用提供的功能以及相关数据。Java EE安全服务不属于操作系统提供的安全机制。

1. 应用层安全性

在Java EE中, 组件容器负责提供应用层的安全性, 以及基于应用需求的为特定类型应用提供的安全服务。在应用层中, 应用防火墙可以保护通信流以及所有相关联的应用资源, 使其免受攻击, 从而增强应用的安全性。

Java EE的安全性易于实现和配置, 可以提供对应用功能和数据访问的精细化控制。然而, 由于这种安全策略是应用层所特有的, 其安全属性无法传导至运行于其他环境的应用, 因而只能保护当前应用环境中的数据。在传统的企业应用环境中, 这不是一个问题。但在Web服务应用中, 数据通常需要跨应用传递, 此时Java EE安全机制需与传输层和消息层的安全机制一起使用, 以提供一个完整的解决方案。

使用应用层安全机制的优势在于:

- 安全性与应用的需求紧密贴合;
- 安全性是细粒度的, 是为应用定制的。

使用应用层安全机制的劣势在于:

- 应用依赖于一些不能在不同类型应用之间传导的安全属性;
- 支持多种协议, 使其易受到攻击;
- 敏感数据靠近或位于攻击点。

有关应用层安全性的更多信息, 参见23.3节。

2. 传输层安全性

客户端和服务提供者之间通过线路来传输数据, 而传输层的安全性由线路所使用的传输机制

来提供。因此，传输层的安全性依赖于使用SSL（Secure Sockets Layer，安全套接层）的安全的HTTP传输协议，即HTTPS。传输层安全性是一种点对点的安全机制，可用于认证、消息完整性检查以及保密性。当服务器与客户端运行于一个受SSL保护的会话中，它们可以在应用协议发送或接收第一个字节前相互认证并协商加密算法及加密密钥。安全性作用于数据离开客户端，直至抵达目的地的整个周期内。反之亦然，即使是跨系统也可以。问题是，数据一旦抵达目的地便不再受保护。在数据发送前进行加密是一个解决办法。

传输层安全性实现于多个阶段，包括：

- ❑ 客户端和服务端就加密算法达成一致；
- ❑ 使用公钥加密和基于证书的认证交换密钥；
- ❑ 在交换信息时使用对称加密。

当使用SSL运行HTTPS时，数字证书不可或缺。必须安装数字证书，否则大部分Web服务器的HTTPS服务无法运行。GlassFish服务器已有内置的数字证书。

传输层安全性的优势在于：

- ❑ 相对简单，易于理解，是标准技术；
- ❑ 可作用于消息体及其附件。

传输层安全性的劣势如下。

- ❑ 与传输层的协议紧耦合。
- ❑ 安全性走极端——全部安全或全部不安全。也就是说，这种安全性机制不区分消息的内容，因此无法选择性地对消息的部分内容应用安全策略（消息层的安全性可以实现这种选择性）。
- ❑ 保护是暂时性的。消息仅在传输的过程中获得保护。当最终接收方接收到消息后，保护自动结束。
- ❑ 它仅是一个点对点，而非端到端的解决方案。

有关传输层安全性的更多信息，参见23.6节。

3. 消息层安全性

在消息层安全机制中，安全信息保存于SOAP消息和（或）SOAP消息附件中。这种机制允许安全信息随着消息或附件一起传输。例如，消息中的一部分可以被发送者签名，并为特定的接收者加密。消息从初始发送者发出后，在抵达最终接受者之前，可能经过多个中间节点。在这种场景中，任何中间节点都无法读取消息中加密部分的内容，这些内容仅可以被最终接收者解密。因此，消息层的安全性通常被认为是一种端到端的安全性。

消息层安全性的优势如下。

- ❑ 安全性在消息途经多个节点的过程中始终存在，即使是在其抵达目的地之后，也依旧有效。
- ❑ 安全性可以有选择地应用于消息的不同部分。如果使用XML Web Services安全机制，甚至可以应用于附件。
- ❑ 消息的安全性可以应用到其所途经的多个节点。

□ 消息的安全性独立于应用的环境及传输协议。

消息层安全性的劣势在于其相对复杂，且会增加消息处理的负担。

GlassFish服务器使用Metro支持消息安全性。Metro是一个Web服务栈，使用WSS (Web Service Security) 为消息增加安全性。由于该安全机制与Metro紧密相关，且不属于Java EE平台的一部分，因而本书不讨论如何通过WSS为消息增加安全性。更多信息，请参考<http://metro.java.net/guide/>处的“Metro User Guide”。

23.3 为容器增加安全性

在Java EE中，组件容器负责应用的安全性。容器通过两种方式实现安全性：声明式和编程式。

23.3.1 用注解为应用增加安全性

注解是一种在编程时使用的声明，可以实现声明式和编程式的安全属性设定。用户可以在类文件中使用注解，以指定应用的安全属性。当应用在GlassFish服务器上部署后，服务器可以读取并使用这些设定。并不是所有的安全属性都可以通过注解的方式实现，有些属性只能在部署描述文件中设定。

特定的注解可以在企业bean的类文件中定义以指定安全信息，参见25.1.1节。第24章介绍了使用注解为Web应用增加安全性的一些场景。本书仅在需要的时候，介绍通过部署描述文件增加安全性的方法。

有关注解的更多信息，参见23.7节。

23.3.2 用部署描述文件为应用增加安全性

声明式安全机制可以通过部署描述文件定义组件的安全需求。由于部署描述文件中的信息都是以声明的方式来定义，因此可以在不修改源代码的情况下改变设置。在系统运行时，Java EE服务器读取部署描述文件中的信息，并将其作用于相应的应用、模块或组件。如果安全属性没有通过注解的方式声明，或不使用默认设置，则必须在部署描述文件中按照约定的格式为各个组件进行设定。

本章不介绍如何创建部署描述文件，仅介绍部署描述文件中与安全性相关的元素。NetBeans IDE提供了创建及修改部署描述文件的工具。

不同类型的组件在部署描述文件中有不同的格式或结构。本书讲解了两种类型的部署描述文件。

□ Web组件可以使用名为web.xml的Web应用部署描述文件。

Java Servlet 3.0规范 (JSR 315) 中的第14章介绍了Web组件的部署描述文件结构，该规范可以从<http://jcp.org/en/jsr/detail?id=315>下载。

- ❑ 企业JavaBeans组件可以使用名为META-INF/ejb-jar.xml的EJB部署描述文件。该文件位于EJB的JAR文件中。
- EJB 3.1规范（JSR 318）的第19章介绍了企业bean的部署描述文件结构。该规范可以从<http://jcp.org/en/jsr/detail?id=318>下载。

23.3.3 使用编程式的安全机制

编程式安全性通常嵌入在应用中，用于定义安全策略。当单独使用声明式安全策略不能满足应用的安全模型的需求时，编程式安全策略可以作为补充。编程式安全机制的API由EJBContext接口和HttpServletRequest接口组成。接口中定义的方法支持组件在应用运行时根据方法调用者或远程用户的角色动态选择安全策略。

编程式安全性将在如下部分详细介绍：

- ❑ 在Web应用中使用编程式安全性24.3节；
- ❑ 通过编程方式为企业Bean增加安全性25.1.2节。

23.4 为 GlassFish 服务器增加安全性

本书介绍了如何在GlassFish服务器上部署应用。GlassFish服务器为应用提供基于Java EE安全模型的安全性、交互性和分布式组件计算能力。GlassFish服务器支持Java EE 6的安全模型。开发人员通过配置GlassFish服务器，可以实现如下目标。

- ❑ 增加、删除及修改认证的用户，详见23.5节。
- ❑ 配置安全的HTTP连接和IIOP（Internet Inter-Orb Protocol，因特网内部对象请求代理协议）监听器。
- ❑ 配置安全的JMX（Java Management Extension，Java管理扩展）连接器。
- ❑ 增加、删除、修改及定制域（realm）。
- ❑ 使用JACC（Java Authentication Contract for Container）为插件式的授权策略提供者定义一个接口。JACC定义了GlassFish服务器和授权策略模块间的安全协议。这些协议规定了在访问策略中，如何安装、配置以及使用授权策略提供者。
- ❑ 使用插件式的审计模块。
- ❑ 可定制的认证机制。所有兼容Java EE 6 Servlet容器规范的实现都需要支持JSR 196的Servlet Profile，该规范为一个或多个应用提供了定制Web容器所用认证机制的方式。
- ❑ 设置及修改应用的权限策略。

23.5 使用域、用户、用户组和角色

资源通常需要保护，以确保仅有获得授权的用户可以访问。关于认证、识别与授权概念的介绍，请参见23.1.3节。

本节将介绍如何进行用户设置,以使用户能被正确识别,并给予其对受保护资源的访问权限,或拒绝未被授权用户访问受保护资源。执行如下基本步骤可实现用户认证。

(1) 应用开发人员编写代码,提示用户输入用户名和密码。认证的多种方法见24.2.2节中的“在部署描述文件中设定认证机制”。

(2) 应用开发人员需要明确如何通过元数据注解或部署描述文件设定应用的安全性,详见23.5.3节。

(3) 服务器管理员在GlassFish服务器上设定授权用户和用户组,详见23.5.2节。

(4) 应用部署人员将应用的安全角色与GlassFish服务器上定义的用户、用户组和主体进行映射,详见23.5.4节。

23.5.1 什么是域、用户、用户组和角色

域 (realm) 是Web或应用服务器中安全策略的作用范围。域包括一个用户集合,而这些用户可以被分配至一个用户组,但非必须。关于如何在GlassFish服务器上管理用户,详见23.5.2节。

应用通常在允许用户访问受保护资源前,提示用户输入用户名和密码。输入用户名和密码后,信息将被传递至服务器。在这里,服务器认证用户并发送受保护资源,或者在用户认证失败时阻止其对受保护资源的访问。这种类型的用户认证机制,将在23.5.3节中讨论。

在某些应用中,获得认证的用户被分配一个角色。在这种情况下,应用分配给用户的角色必须与服务器上定义的主体 (principal) 或用户组相匹配。图23-6展示了这个过程。有关将角色映射至用户和用户组的更多内容,参见23.5.3节。

创建用户和/或用户组 → 在应用中定义角色 → 将角色映射至用户和/或用户组

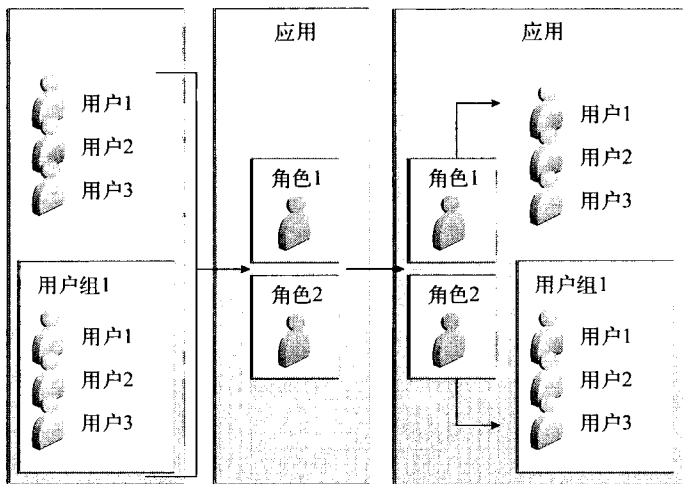


图23-6 将角色映射至用户和用户组

接下来将讲述关于域、用户、用户组和角色的更多内容。

1. 什么是域

域 (realm) 是 Web 或应用服务器中安全策略的作用范围。服务器上的受保护资源可以以分区的方式分配至一组不同的受保护空间内, 而每一个空间都有自己的认证机制, 以及/或包含一组用户和用户组的授权数据库。对于 Web 应用来说, 域是一个包含有效用户和用户组的完整数据库, 这个用户和用户组的集合对某个 Web 应用或一组 Web 应用有效, 并受相同的认证策略控制。

Java EE 服务器认证服务可以在多个域中管理用户。GlassFish 服务器预先配置了几个域, 如 file、admin-realm 和 certificate。

在 file 域里, 服务器在本地文件 keyfile 中存储用户证书。可以使用管理控制台管理 file 域中的用户。当使用 file 域时, 服务器的认证服务通过检查 file 域验证用户身份。这个域用于验证除使用 HTTPS 和证书的 Web 浏览器之外所有的客户端。

在 certificate 域里, 服务器将用户证书存储在证书数据库中。当使用 certificate 域时, 服务器使用 HTTPS 的证书认证 Web 客户端。为了在 certificate 域中验证用户身份, 认证服务需要验证 X.509 证书的有效性。关于创建此类证书的详细步骤, 参见 23.6.2 节。X.509 证书的 name 字段通常用作主体 (principal) 的名称。

admin-realm 也是一种 file 域, 在一个名为 admin-keyfile 的本地文件中存储管理员用户的证书。可以使用管理控制台管理这个域中的用户, 其方式与在 file 域中管理用户的方式相同。更多信息, 参见 23.5.2 节。

2. 什么是用户

用户是 GlassFish 服务器上定义的人或应用程序标识。在 Web 应用中, 可以为用户关联一组角色, 使其可以访问为该角色定义的一组受保护资源。用户可以关联一个用户组。

Java EE 中的用户类似于操作系统中的用户。通常, 两者的用户都代表具体的人。然而, 两者之间也存在一定的区别。Java EE 的服务器认证服务不关注用户在登录操作系统时提供的用户名和密码。Java EE 的服务器认证服务不使用操作系统的安全机制。这两种安全服务管理不同域中的用户。

3. 什么是用户组

用户组是 GlassFish 服务器中按照用户共同属性划分出的一组认证用户。file 域中的一个 Java EE 用户可以属于 GlassFish 服务器上的一个用户组。(certificate 域中的用户不属于 GlassFish 服务器上的某个用户组。) GlassFish 服务器上的用户组是按照用户共同属性进行分类的一个集合, 如按职位或客户档案分类。例如, 大多数电子商务应用的客户属于 CUSTOMER 组, 而消费多的用户属于 PREFERRED 组。当用户量较大时, 将用户分配至不同的组, 能够简化访问控制。

在 GlassFish 服务器中, 用户组的范围与角色不同。用户组针对整个 GlassFish 服务器, 而角色仅关联 GlassFish 服务器上的某个具体应用。

4. 什么是角色

角色是应用中访问一组特定资源所需具备的权限的抽象名称。通过钥匙与锁的形象比喻, 可以进一步理解角色这个概念。不同的人都可以去配同一把钥匙。锁并不关心使用钥匙的人是谁,

而只认能够打开锁的钥匙。

5. 其他术语

下列这些术语也常见于Java EE平台的安全策略中。

- **主体 (principal)** 是可以被部署在企业中的安全服务通过某种认证协议认证的实体。主体通过名称识别, 并通过认证数据获得认证。
- **安全策略域 (security policy domain)** 也称为安全域 (security domain) 或域 (realm), 是安全管理员定义并执行的安全策略所作用的范围。
- **安全属性 (security attribute)** 与每一个主体都关联的一组属性。安全属性有多种用途。例如, 用于对受保护资源的访问和用户的审计。安全属性可以通过认证协议与一个主体相关联。
- **安全凭证 (credential)** Java EE服务认证一个主体时使用的对象, 它包含或引用安全属性。主体在认证后获得凭证, 或从另一个主体获得凭证。

23.5.2 在GlassFish服务器中管理用户和用户组

在运行示例程序之前, 请执行如下步骤, 以管理用户。

▼ 为GlassFish服务器添加用户

(1) 启动GlassFish服务器。

关于启动GlassFish服务器方法的介绍, 参见2.2节。

(2) 启动管理控制台。

请打开Web浏览器, 并在地址栏中输入<http://localhost:4848>, 以启动管理控制台。如果在安装GlassFish服务器时修改了默认的Admin端口, 请输入正确的端口号以代替4848。

(3) 在导航树中, 展开Configuration结点。

(4) 展开Security结点。

(5) 展开Realms结点。

(6) 选择要增加用户的域。

- 选择file, 为这个域增加用户, 使其可以访问运行于其中的应用。

对于示例程序来说, 选择file域。

Edit Realm页面打开。

- 选择admin-realm, 为GlassFish服务器增加可以作为系统管理员的用户。

Edit Realm页面打开。

不能使用管理控制台在certificate域中增加用户。在certificate中, 仅能增加证书。关于为certificate增加 (导入) 证书的方法, 参见接下来的“将用户加入Certificate域”。

(7) 在Edit Realm页面中, 单击Manage Users按钮。

File Users或Admin Users页面打开。

(8) 在File Users或Admin Users页面中, 单击New, 为域添加一个新用户。

New File Realm User页面打开。

(9) 在User ID、Group List、New Password、Confirm New Password中输入相应值。

对于Admin Realm, Group List字段只读, 分组名称是asadmin。在Admin Realm中添加用户之后, 重启GlassFish服务器和管理控制台。

关于这些属性的更多信息, 参见23.5节。

对于示例程序, 请根据喜好为用户指定一个名称和密码, 并确保将该用户分配至TutorialUser用户组。用户名和密码对大小写敏感。记录用户名和密码, 以便在本书的后续示例中使用。

(10) 单击OK将用户添加至域, 或单击Cancel放弃操作。

将用户加入Certificate域

在certificate域中, 用户标识在GlassFish服务器的安全上下文环境中建立, 而用户数据则从加密的数字证书中获得。创建这种证书的具体步骤参见23.6.2节。

23.5.3 设置安全角色

在设计企业bean或Web组件时, 一定要考虑用户对组件的访问权限问题。例如, 在人力资源部的Web应用系统中, 角色为DEPT_ADMIN的用户和角色为DIRECTOR的用户在访问系统时可能使用不同的URL。DEPT_ADMIN角色的用户可以查看员工数据, 而DIRECTOR角色的用户可以修改员工数据, 如薪酬。每一个用户角色, 都是应用开发人员定义的, 是对用户抽象的逻辑分组。在部署应用后, 部署人员在该运行环境下将角色映射至安全身份, 如图23-6所示。

对于Java EE组件, 使用元数据注解@DeclareRoles和@RolesAllowed来定义安全角色。

在下面的示例应用中, DEPT-ADMIN角色的用户有权调查看员工工资的方法, 而DIRECTOR角色的用户有权调用修改用户工资的方法。

下面的代码展示了如何在企业bean中利用注解进行角色声明:

```
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，指定可以访问该servlet的角色。例如，servlet可以以如下方式注解：

```

@WebServlet(name = "PayrollServlet", urlPatterns = {"/payroll"})
@ServletSecurity(
    @HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
        rolesAllowed = {"DEPT-ADMIN", "DIRECTOR"}))
public class GreetingServlet extends HttpServlet {

```

关于上述注解的详细介绍，参见24.4.1节中的“使用注解实现基本认证”，以及25.1.1节。

在用户提供了登录信息，且应用也声明了可以访问其中受保护部分的用户角色之后，下一步就是要将角色与用户名或主体进行映射。

23.5.4 将角色映射至用户和用户组

在开发Java EE应用时，无需知晓定义了哪些种类的用户在运行应用的域中。在Java EE平台中，安全架构提供了一种动态映射机制，将应用中定义的角色映射至应用实际运行的域中的用户或用户组。

在GlassFish服务器中，应用中定义的角色名称通常与用户组的名称相同。在这种情况下，可以通过GlassFish服务器的管理控制台，配置默认的主体到角色的映射机制。24.4节中的“为运行示例程序设置系统”部分将介绍具体的做法。本书中关于安全性的示例均使用默认的主体到角色的映射机制。

如果应用中定义的角色名称与服务器中的用户组名称不同，可以使用运行时的部署描述文件建立二者的对应关系。对于一个Web应用，下面的示例展示了如何在sun-web.xml文件中建立映射关系：

```

<sun-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>
    ...
</sun-web-app>

```

角色可以映射至特定的主体、用户组或两者的组合。主体或用户组的名称必须是当前默认域，或在login-config元素所指定的域中有效的主体或用户组。在本例中，应用中的角色Mascot映射

至名为Duke的主体（存在于应用服务器中）。当具备某个角色的用户可能改变时，将角色映射至特定的主体非常有用。对于这种应用，仅需修改运行时的部署描述文件，而无需在这个应用中查找并替换对这个主体的所有引用。

同样是在这个示例中，Admin角色映射至一组隶属于Director用户组的用户。这很有用，因为被授权访问“Director”级别管理数据的人所在的组只能在GlassFish服务器上进行维护。应用开发人员不需要知道这些人到底是谁，而仅需要定义一个可以访问这些数据的用户组。

role-name必须与部署描述文件中security-role元素中的role-name，或@DeclareRoles注解中定义的角色名称相匹配。

23.6 使用SSL建立安全连接

SSL（Secure Socket Layer，安全套接层）是一种在传输层实现的安全技术。（关于传输层安全性的更多信息，参见23.2.2节中的“传输层安全性”。）SSL允许Web浏览器与Web服务器通过安全的连接进行通信。在这个安全的连接中，数据在发送前进行加密，在抵达接收方后并进行进一步处理前解密。浏览器和服务器在发送任何数据前，将加密所有数据流。

SSL技术满足如下安全性要求。

- **认证** 在通过一个安全通道与Web服务器建立初始连接时，服务器将以服务器证书的形式为浏览器提供一组安全凭证。证书用于验证该站点是谁，以及它声明自己是谁。在某些情况下，服务器可能请求客户端提供证书，以证明它（客户端）是谁及它（客户端）声明是谁。这种机制通常称为客户端验证。
- **机密性** 当通过网络在客户端与服务器间传递数据时，第三方可以查看并截取数据。SSL应答数据会被加密，使得第三方无法进行解密，从而保证数据的机密性。
- **完整性** 当通过网络在客户端与服务器间传递数据时，第三方可以查看并截取数据。SSL确保数据在传输的过程中不会被第三方修改。

SSL协议设计的目标是尽可能安全和高效。然而，从性能的角度来看，加密与解密的运算量很大。无需严格地要求整个Web应用运行于SSL之上，通常由开发人员决定哪些页面需要安全连接，哪些不需要。可能需要安全连接的页面包括与登录、个人信息、购物车结算单或信用卡信息传送有关的页面。可以以安全套接的方式请求应用中的任何页面，将请求地址中的http换成https即可。对于那些必须使用安全连接的页面，必须检查当前请求所使用的协议，如果不是使用https的方式则需要采取必要的措施。

在安全的连接上使用基于名称的虚拟主机可能会有问题。这是SSL协议自身设计上的限制。借以SSL握手（客户端浏览器接受服务器证书的过程），必须在处理HTTP请求之前完成。因此，包含虚拟主机名称的请求信息无法在认证之前进行判定，进而也就不能为一个IP地址分配多个证书。如果单一IP地址上的所有虚拟主机均使用同一证书进行认证，多个主机的叠加不应该妨碍服务器上SSL的正常操作。然而，如果证书中有域名，大多数客户端浏览器都会将服务器的域名与证书中的域名进行比较（这种情况通常适用于那些通过CA认证机构签名的证书）。如果域名不匹

配，浏览器会显示提示信息。通常来说，仅有基于IP地址的虚拟主机在生产环境中使用SSL。

23.6.1 验证及配置SSL

作为基本规则，下列要求必须满足，才能使服务器上的SSL生效：

- 在服务器上的部署描述文件必须有一个声明SSL连接器的Connector元素；
- 必须有有效的keystore和证书文件；
- keystore文件的位置及其密码必须在服务器的部署描述文件中指定。

GlassFish服务器已经配置一个SSL的HTTPS连接器。

如果为了测试并验证SSL已正确安装，可以打开使用如下URL的默认欢迎页面。这个URL包含了在部署描述文件中定义的连接端口：

`https://localhost:8181/`

URL中的https表明浏览器需使用SSL协议。本例中使用localhost，假定示例运行在本地的开发环境中。本例中的8181是安全连接端口，是在创建SSL连接器时指定的。如果使用其他服务器或端口，请修改对应的值。

当首次使用该应用时，会看见New Site Certificate或Security Alert对话框。选择Next，逐一响应弹出的所有对话框，直至在最后一个对话框中选择Finish。证书仅在第一次使用时显示。接受了证书，即表明在随后对该站点的访问过程中始终信任其中的内容。

23.6.2 使用数字证书

GlassFish服务器的数字证书已经生成，并保存在`as-install/domain-dir/config/`目录下。这些数字证书采用自签署的方式，主要应用于应用开发阶段，并不用于生产环境中。在生产环境中，需要生成自己的证书并获得CA认证机构的签署。

为了使用SSL，应用或Web服务器必须为每一个接受安全连接的外部接口或IP地址提供一个关联的证书。这种设计的理由是，服务器必须提供某种合理的保证，以表明它的所有者正是其人，特别是在接收任何敏感信息之前。可以将数字证书理解为某因特网地址的“数字驾驶执照”。证书阐述了站点所关联的组织，以及一些诸如站点所有人或管理员的基本联系信息。

数字证书使用加密的方式由其所有人签署，因而很难被其他人伪造。对于电子商务网站，或那些身份认证在其中起至关重要作用的交易，可以向知名的CA认证机构（如VeriSign或Thawte）购买证书。如果服务器的证书是自签署的，则必须将其安装至GlassFish服务器的keystore文件（keystore.jks）。如果客户端的证书是自签署的，需要将其安装至GlassFish服务器的truststore文件（cacerts.jks）。

有时，认证并非那么重要。例如，管理员仅是希望确保服务器传输和接收的数据是私有的且不能被连接线路上的窃听者窃取。在这种情况下，无需费时费钱地申请CA证书，使用自签署的即可。

SSL使用基于密钥对的公钥加密机制。密钥对包含一个公钥和一个私钥。用其中一个密钥加

密的数据仅能用密钥对中的另一个密钥解密。这种机制是在交易中建立信任和保密性的基础。例如，服务器使用SSL计算一个值，并使用它的私钥进行加密。被加密的值称为数字签名。客户端使用服务器的公钥解密这个加密的值，并将其与自己计算出的值进行比较。如果两个值一致，客户端可以信任这个签名的真实性，因为仅有那个私钥可以产生这个签名。

数字证书还与HTTPS一起，用于Web客户端的认证。大多数Web服务器的HTTPS服务仅在安装数字证书之后才能使用。参照下面“生成服务器证书”的内容设置数字证书，使应用或Web服务器可以支持SSL。

keytool是一个设置数字证书的工具。它作为密钥和证书的管理工具集成在JDK中。这个工具可以使用户管理自己的密钥对及用于自我认证的证书。自我认证是使用数字证书，向其他用户、服务或数据完整性和认证服务认证自己。这个工具支持用户以证书的形式缓存通信对象的公钥。为了更好地理解keytool和公钥加密机制，可以参考keytool的在线文档，地址为<http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html>。

生成服务器证书

GlassFish服务器的证书已经生成，并保存在`domain-dir/config/`目录下。服务器的证书存放于`keystore.jks`文件中。`cacerts.jks`保存了所有可信任的证书，包括客户端证书。

如果需要，可以使用keytool生成新的证书。keytool工具将密钥和证书存放于文件`keystore`中，该文件是一个用于识别服务器或客户端的证书存储库。通常情况下，一个keystore对应一个文件，保存一个客户端或一个服务器的标识。keystore通过密码保护私钥。

如果在指定keystore文件名时没指定目录，则keystore会在运行keytool命令的目录下生成。可以是某个应用所在的目录，也可以是多个应用共享的目录。

生成一个服务器证书的通常步骤如下。

(1) 创建keystore。

(2) 从keystore中导出证书。

(3) 签署证书。

(4) 将证书导入truststore。`truststore`是一个用于验证证书的证书存储库。`truststore`通常保存多个证书。

下面详细介绍使用keytool实用工具执行上述步骤的具体方法。

▼使用keytool生成服务器证书

使用keytool生成新的密钥对，保存在默认的keystore文件`keystore.jks`中。这个示例使用别名`server-alias`生成一个新的公钥/私钥密钥对，并在`keystore.jks`中将公钥封装成一个自签署的证书。这里的密钥对是使用RSA算法产生的，其默认密码为`changeit`。关于keytool的更多信息，以及创建和管理keystore文件的信息及其他示例，请参考keytool的在线帮助文档<http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html>。

注意 RSA是RSA Data Security 公司开发的基于公钥的加密技术。

在创建密钥对的目标下,按照如下步骤运行keytool。

(1) 生成服务器证书。

在一行中输入keytool命令的全部内容:

```
java-home/bin/keytool -genkey -alias server-alias -keyalg RSA -keypass changeit
-storepass changeit -keystore keystore.jks
```

在输入上述命令并执行后, keytool会给出提示要求输入服务器名、组织部门、组织、地区、国家和国家代码。

对于keytool的第一个提示(提示输入名和姓),必须输入服务器的名称。如果以测试为目的,可以输入localhost。

当运行示例应用时,在keystore中指定的主机(服务器名)必须与*tut-install/examples/bp-project/build.properties*文件中指定的javaee.server.name属性的值保持一致。

(2) 将keystore.jks文件中的服务器证书导出至文件server.cer。

在一行中输入keytool命令的全部内容:

```
java-home/bin/keytool -export -alias server-alias -storepass changeit
-file server.cer -keystore keystore.jks
```

(3) 如果希望证书被CA认证机构签署,请参阅<http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html>处的示例。

(4) 要将服务器证书添加至truststore文件cacerts.jks中,在创建keystore和服务器证书的目录下运行keytool。

使用如下参数:

```
java-home/bin/keytool -import -v -trustcacerts -alias server-alias
-file server.cer -keystore cacerts.jks -keypass changeit -storepass changeit
```

运行后,终端上将出现如下证书信息:

```
Owner: CN=localhost, OU=Sun Micro, O=Docs, L=Santa Clara, ST=CA,
C=USIssuer: CN=localhost, OU=Sun Micro, O=Docs, L=Santa Clara, ST=CA,
C=USSerial number: 3e932169Valid from: Tue Apr 08Certificate
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:
Trust this certificate? [no]:
```

(5) 输入yes,并按下回车键。

终端上将出现如下信息:

```
Certificate was added to keystore[Saving cacerts.jks]
```

23.7 有关安全性的更多信息

有关Java EE应用安全性的更多信息,可参见如下内容。

❑ Java EE 6规范:

<http://jcp.org/en/jsr/detail?id=316>

❑ 文档“Oracle GlassFish Server 3.0.1 Application Development Guide”为开发人员介绍安全信息,如GlassFish服务器上有关部署描述文件的特定设置。

❑ 文档“Oracle GlassFish Server 3.0.1 Application Administration Guide”,包含GlassFish服务器上的安全设置方法。

❑ 企业Java Beans 3.1规范:

<http://jcp.org/en/jsr/detail?id=318>

❑ 企业Web Service 3.1实现规范:

<http://jcp.org/en/jsr/detail?id=109>

❑ Java SE安全信息:

<http://docs.oracle.com/javase/6/docs/technotes/guides/security/>

❑ Java Servlet 3.0规范:

<http://jcp.org/en/jsr/detail?id=315>

❑ Java Authorization Contract for Container 1.3规范:

<http://jcp.org/en/jsr/detail?id=115>

❑ 文档“Java Authentication and Authorization Service (JAAS) Reference Guide”:

<http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html>

❑ 文档“Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide”:

<http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASLMDevGuide.html>

Web应用是指使用Web浏览器，通过网络（如因特网或公司局域网）访问的应用。正如1.3节所述，Java EE平台使用分布式多层应用模型，而Web应用运行于Web层。

Web应用包含可以被许多用户访问的资源。这些资源通常在无保护的开放网络（如因特网）上传输。在这样的环境中，一些重要的Web应用需要某种类型的安全保护。为Java EE应用增加安全保护的通常做法，已在23.3节介绍。本章将介绍一些与Web组件相关的安全服务示例程序，并对前述内容进行深入探讨。

关于在业务层和EIS（企业信息系统）层为应用及其客户端进行安全加固的内容，请参见第25章。

本章内容

- Web应用安全性概述
- 为Web应用增加安全性
- 在Web应用中使用编程式安全机制
- 为Web应用增加安全性的示例

24.1 Web 应用安全性概述

在Java EE平台中，Web组件为Web服务器提供了动态扩展能力。Web组件可以是Java servlet或Java Server Faces页面。Web客户端与Web应用间的交互如图24-1所示。

Web应用安全性的某些属性可以在安装或部署应用时配置到Web容器中。注解和部署描述文件可用于将应用的安全性及其他信息传达给应用部署人员。在注解或部署描述文件中设置这些信息，将帮助应用部署人员正确设置Web应用的安全策略。任何在部署描述文件中显式指定的属性设置，都将覆写注解中的设定。

Java EE Web应用的安全性可以通过如下方式实现。

- 声明式安全机制 可以通过元数据注解或应用部署描述文件实现，详见23.1节。

关于如何在Web应用中使用声明式安全机制，请参见24.2节。

- 编程式安全机制 通常嵌入在应用中，在声明式安全机制无法满足应用的安全需要时，

该方式可以用于确定安全策略。当特定工作流程（并非所有场景）中的某个环节需要有条件登录时，单独使用声明式安全机制可能无法实现足够的安全控制。更多信息，请参见23.1节。

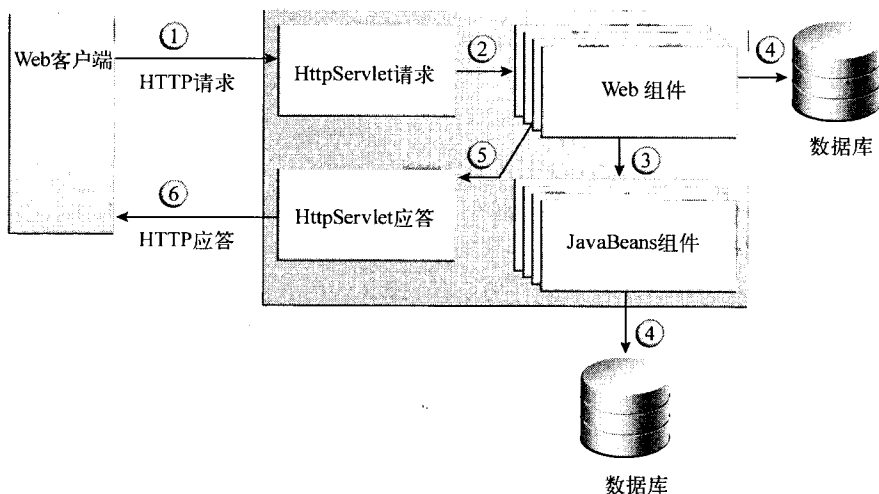


图24-1 Java Web应用请求处理

Servlet 3.0规范为HttpServletRequest接口提供了authenticate、login和logout方法。在添加authenticate、login和logout方法至Servlet规范后，对于Web应用来说部署描述文件的作用在淡化，它仅用于在默认的基础设置之上进一步完善安全策略。

编程式安全机制将在24.3节中讨论。

- 消息安全机制 利用Web 服务，将一些安全特性（如数字签名与加密）整合至SOAP消息的头部。该机制运行于应用层，确保端到端的安全性。消息安全机制不是Java EE 6的组件，此处仅是简单提及。

本章中的一些讨论依赖于前面章节的内容，且本章将假定读者已经熟悉下述各章的内容：

- 第3章，Web应用初步；
- 第4章，JSF技术；
- 第10章，Java Servlet技术；
- 第23章，Java EE平台安全入门。

24.2 为 Web 应用增加安全性

Web应用由应用开发人员创建，并由他们以某种方式提交给应用部署人员，以将应用安装至运行时环境中。应用开发人员通过注解或部署描述文件，将如何为已部署的应用设置安全性的信息传达出去。这些信息将传达给应用部署人员，他们使用这些信息为安全角色定义方法的访问权

限并设置用户认证方式以及适当的数据传输机制。如果应用开发人员没有提出对安全性的需求，应用部署人员需要自己决定安全性设定。

有些对Web应用来说必要的安全属性，不能通过注解的方式为所有类型的Web应用设定。本章将介绍如何使用注解为Web应用进行安全加固。本章还将介绍在不能使用注解时，如何通过部署描述文件实现安全性。

24.2.1 设定安全限制

安全限制通过使用URL映射为一组资源定义访问权限。

如果Web应用使用servlet，可以使用注解实现安全限制。此时，可以在@ServletSecurity中使用@HttpConstraint和@HttpMethodConstraint注解（可选），以设定安全限制。

如果Web应用不使用servlet，必须在部署描述文件中设定security-constraint元素。在这种情况下，认证机制不能通过注解的方式声明，因此当应用要求使用除BASIC（默认）方式之外的其他认证方式时，必须使用部署描述文件。

下面这些元素可以作为security-constraint元素的子元素。

- ❑ **Web资源集合（web-resource-collection）** 一组URL模式列表（在主机名和端口之后希望限定的URL部分）和HTTP操作（文件中的一组方法，能够匹配需要限定的URL模式），描述了一组需要保护的资源。
- ❑ **授权限制（auth-constraint）** 指定是否需要认证，以及授权的角色名称，以执行受限的请求。
- ❑ **用户数据限制（user-data-constraint）** 指定在客户端与服务器间传递数据时如何保护数据。

1. 设定Web资源集合

Web资源集合由下面的子元素组成。

- ❑ **web-resource-name**是资源对应的名称，可选。
- ❑ **url-pattern**列举需要保护的请求URI。对于大多数应用来说，不是所有的资源都需要保护。对于那些无需访问限制的资源，不必配置对请求URI的安全限制。

请求URI是URL在主机名和端口之后的那部分。例如，电子商务网站的产品目录允许任何人访问并浏览，而对于购物车，只有注册并登录的用户可以访问。可以为这个Web应用设置一个路径模式，如/cart/*，以便仅保护匹配这种模式的资源，而其他资源不在保护之列。假定应用安装在/myapp路径下，则：

- http://localhost:8080/myapp/index.xhtml无保护；
- http://localhost:8080/myapp/cart/index.xhtml有保护。

用户在首次访问cart/子目录下的资源时，会被提示需要登录。

- ❑ **http-method**或**http-method-omission**用于指定哪些HTTP方法需要保护，或哪些方法无需保护。在以下的情况下，HTTP方法将通过web-resource-collection获得保护：

- 如果集合中没有HTTP方法的名称（意味着对所有方法进行保护）；
- 如果集合在http-method子元素中明确指出了HTTP方法的名称；
- 如果集合中包括一个或多个http-method-omission元素，但都没有指定HTTP方法名称。

2. 设定授权限制

授权限制（auth-constraint）包括role-name元素。注意，可以按照需要使用多个role-name。

授权限制制定了一些认证要求，对于限制中声明的可以访问的URL模式和HTTP方法分别定义了授权的角色。如果没有授权限制，容器必须在不要求用户认证的情况下接受请求。如果设定了授权限制，但没有在其中指定角色，则容器将拒绝所有对受限资源的访问。此处设定的角色名，必须与在Web应用中通过security-role元素定义的角色名称相对应，或者使用保留的关键字“*”（代表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至服务器上定义的用户和用户组，除非使用默认的主体到角色的映射。

关于安全角色的更多内容，参见24.2.3节。关于映射安全角色的更多内容，参见23.5.4节。

对于servlet，@HttpConstraint和@HttpMethodConstraint注解支持rolesAllowed元素，以指定授权的角色。

3. 设定安全连接

用户数据限制（对应部署描述文件中的user-data-constraint）包含transport-guarantee子元素。用户数据限制可以要求，对于所有在安全限制中指定的受限URL模式和HTTP方法使用受保护的传输层连接，如HTTPS。传输模式可以选择CONFIDENTIAL、INTEGRAL或NONE中的一种。如果选择CONFIDENTIAL或INTEGRAL，则意味着要访问所有在Web资源集合中匹配URL模式的资源，均需要使用SSL连接，而不仅限于登录对话框。

对受限资源的保护强度由选择的传输模式来决定。

- **CONFIDENTIAL** 当应用需要防止其他人截取并查看传输的数据时，使用该参数。
- **INTEGRAL** 当应用要求数据在客户端和服务器间传递时不能被修改时，使用该参数。
- **NONE** 声明容器必须接受来自任何连接方式的对受保护资源的请求，包括非保护资源。

注意 在实际使用中，对于CONFIDENTIAL和INTEGRAL这两个选项，Java EE服务器在实现方式上是一样的。

用户数据限制很容易与基本认证和基于表单的用户认证结合起来使用。当登录认证方法设定为BASIC或FORM时，密码无法得到保护，也就是说密码在客户端和服务器间的传输是在一个未受保护的会话中进行的，因此密码可能会被第三方截取并查看。与用户认证机制相结合的用户数据限制可以在一定程度上规避这种隐患。配置用户认证机制将在24.2.2节中的“在部署描述文件中指定认证机制”部分介绍。

为了保证数据通过安全的连接方式进行传输，请确保服务器正确配置了SSL。SSL已经配置在GlassFish服务器上。

注意 在将会话切换至SSL后，请不要让会话再接收来自非SSL的连接请求。例如，一个购物网站只在结算页面才使用SSL，在此处切换为使用SSL，以便处理信用卡信息。在切换为使用SSL后，应用应该终止监听对这一会话的非SSL请求。这样做的原因是会话ID本身未在之前的通信中进行加密。如果是单纯的购物，似乎没什么问题。但如果信用卡的信息保存在会话中，恐怕没有人愿意他人盗用自己的信用卡并用其购物。在实际情况中，增加一个过滤器即可。

4. 为不同的资源设定不同的安全限制

开发人员可以在应用中为不同的资源指定不同的安全限制。例如，可以允许PARTNER角色的用户访问URL模式为/acme/wholesale/*的所有资源的GET和POST方法，以及允许角色为CLIENT的用户访问URL模式为/acme/retail/*的所有资源的GET和POST方法。下面这个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>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </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>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </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>
```

当相同的url-pattern和http-method出现在多个安全限制中，对于模式和方法的限制将是每一个限制组合后共同作用的结果，这可能导致无意中拒绝正常的访问。

24.2.2 设定认证机制

用户认证机制指定如下内容：

- 用户获得Web内容访问权的方式;
- 当使用基本认证时, 认证用户的域;
- 当使用表单认证时的一些其他属性。

当指定了某个认证机制时, 用户在访问任何设定了安全限制的資源前, 必须先进行认证。多个安全限制可以应用于多个资源上, 但应用中所有受限资源上的认证机制均相同。

在认证用户之前, 必须有记录用户名、密码和在Web或应用服务器上所配置角色的数据库。关于设置用户数据库的更多内容, 参见23.5.2节。

HTTP基本认证和基于表单的认证并非是非常安全的认证机制。基本认证是通过因特网发送经过Base64加密的用户名和密码, 而表单认证以纯文本的形式发送这些数据。在这两种情况下, 目标服务器并未经过认证。因此, 这两种形式将暴露用户数据, 使其易受攻击。如果第三方截获传输的数据, 则用户名和密码很容易被破解。然而, 当使用安全传输机制(如SSL), 或网络层的安全协议(如IPSec, 即Internet Protocol Security), 或虚拟个人专网VPN策略, 并将这些机制与基本认证或基于表单的认证相结合, 上述的风险将得到缓解。要指定安全传输机制, 请使用24.2.1节的“设定安全连接”部分介绍的参数。

1. HTTP基本认证

如果使用HTTP基本认证, 那么服务器要求Web客户端提供用户名和密码, 并将收到的用户名和密码与在指定或默认域中定义的授权用户数据库中的对应信息进行比较。

不指定认证机制时, 默认使用基本认证机制。

当使用基本认证时, 认证的过程如下。

- (1) 客户端请求访问受保护资源。
- (2) Web服务器返回一个对话框, 要求用户输入用户名和密码。
- (3) 客户端将用户名和密码提交给服务器。
- (4) 服务器在指定的域中认证用户。如果成功, 返回请求的资源。

图24-2展示了基本认证的处理过程。

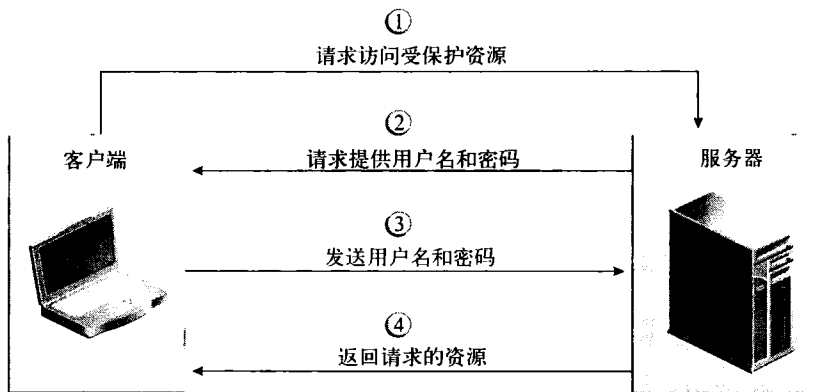


图24-2 HTTP基本认证

2. 表单认证

表单认证允许开发人员定制HTTP浏览器呈现给终端用户的登录页面和错误提示页面，以此设定个性化的登录认证界面的外观。当声明使用表单认证方式时，认证的过程如下。

- (1) 客户端请求访问受保护资源。
- (2) 如果客户端尚未认证，则服务器将客户端重定向至登录页面。
- (3) 客户端将登录表单提交给服务器。
- (4) 服务器尝试认证用户。
 - (a) 如果认证成功，检查获得认证的用户的主体，以确保其具有访问该资源的角色。如果用户获得授权，服务器使用保存的URL地址将客户端重定向至原先请求的资源上。
 - (b) 如果认证失败，客户端将被转发或重定向至错误提示页面。

图24-3展示了表单认证的处理过程。

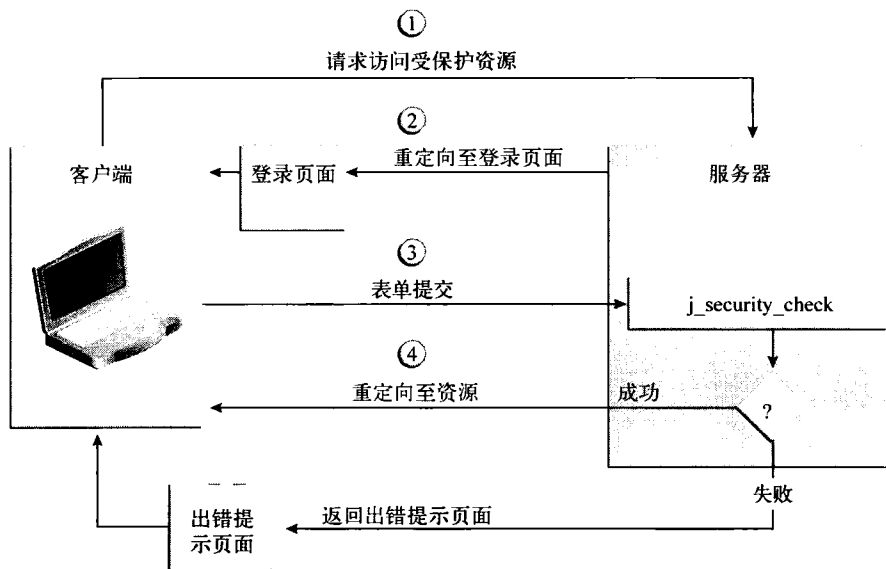


图24-3 表单认证

24.4.2节展示了一个使用表单认证的示例应用。

当创建基于表单的登录方式时，请确保使用cookie或SSL管理会话。

为了确保认证能正确执行，登录表单的action必须是`j_security_check`。之所以有这个限制，是为确保无论访问什么样的资源，登录表单都可以正常工作，且无需特别考虑应答表单的action字段。下面的代码片段展示了如何在HTML页面中编写表单代码：

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

3. 摘要认证

与基本认证类似，摘要认证通过用户名和密码对用户进行认证。然而，不同于基本认证，摘要认证不通过网络发送用户密码。取而代之的是，客户端发送密码和附加数据的单向加密散列。虽然密码没有以明文的方式传输，摘要认证要求认证容器保存相同明文的密码，以便通过计算摘要验证接收到的认证信息是否正确。

4. 客户端认证

Web服务器使用客户端的公钥证书，实现对客户端的认证。相比于基本认证和表单认证，客户端认证是一种更安全的方式。该认证使用基于SSL的HTTP协议（HTTPS），在协议的基础上，服务器使用客户端的公钥证书实现对客户端的认证。SSL技术提供数据加密、服务器认证、消息完整性检查，并可为TCP/IP连接验证客户端（可选）。可以将公钥证书想象成数字化证件，如护照。证书由可信赖的机构（如CA认证中心）颁发，为持有者提供身份证明。

在使用客户端认证前，需要确保客户端具有有效的公钥证书。关于创建和使用公钥证书的更多内容，参见23.6.2节。

5. 双向认证

使用双向认证机制，服务器和客户端可以相互认证对方。双向认证有两种类型：

- 基于证书的双向论证（参见图24-4）；
- 基于用户名和密码的双向认证（参见图24-5）。

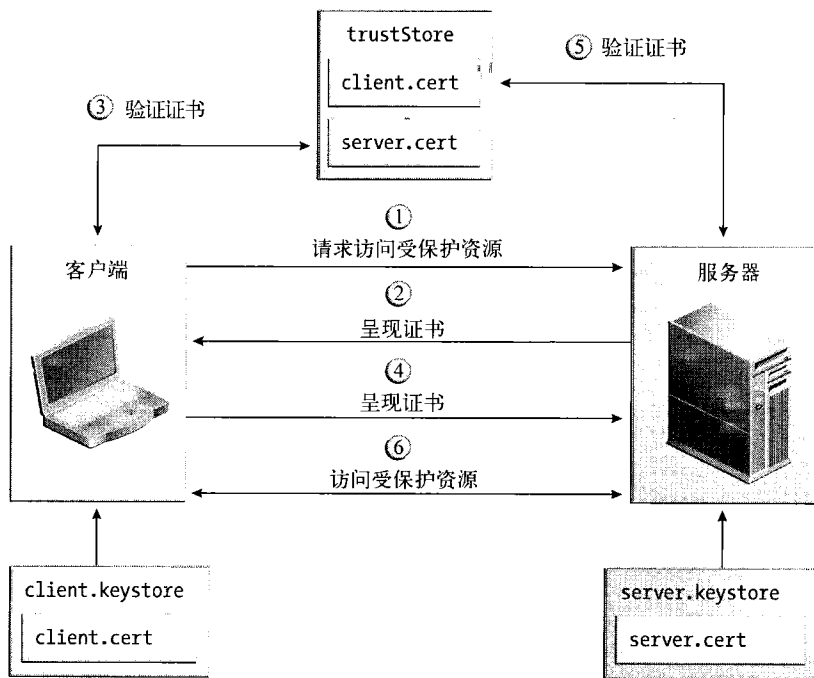


图24-4 基于证书的双向认证

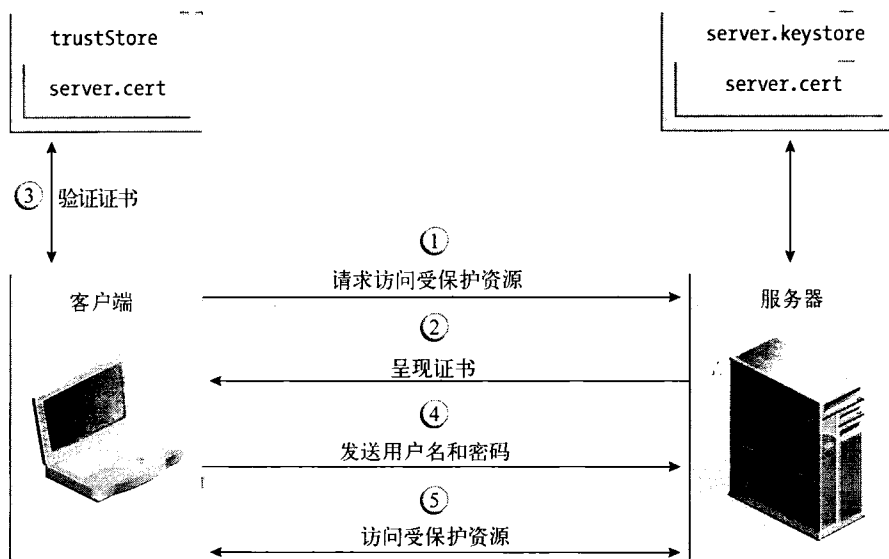


图24-5 基于用户名和密码的双向认证

当使用基于证书的双向认证机制时，认证的过程如下。

- (1) 客户端请求访问受保护资源。
- (2) Web服务器将其证书呈现给客户端。
- (3) 客户端验证服务器的证书。
- (4) 如果验证成功，客户端将自己的证书提交给服务器。
- (5) 服务器验证客户端的凭据。
- (6) 如果验证成功，服务器将授权客户端对受保护资源的访问。

图24-4展示了基于证书的双向认证过程。

对于基于用户名和密码的双向认证机制，认证过程如下。

- (1) 客户端请求访问受保护资源。
- (2) Web服务器将其证书呈现给客户端。
- (3) 客户端验证服务器的证书。
- (4) 如果验证成功，客户端将用户名和密码发送给服务器，由服务器验证客户端凭证。
- (5) 如果验证成功，服务器将授权客户端访问受保护资源。

图24-5展示了基于用户名和密码的双向认证过程。

6. 在部署描述文件中指定认证机制

使用login-config元素指定认证机制。该元素包括如下子元素。

- auth-method 为Web应用配置认证机制。可以设定的值为NONE、BASIC、DIGEST、FORM或CLIENT-CERT。

- **realm-name** 当Web应用使用基本认证机制时，该元素声明了域的名称。
- **form-login-config** 当使用基于表单的登录机制时，该元素声明登录及出错页面。

注意 另外一种指定表单认证机制的方法是使用HttpServletRequest的authenticate、login和logout方法。这部分内容将在24.3.1节中讨论。

当访问通过security-constraint元素加以限制的Web资源时，Web容器将触发为该资源配置的认证机制。认证机制将指定用户该以何种方式登录。如果配置了login-config，且auth-method的值不为NONE，则用户只有在获得认证之后才能访问资源。如果没指定认证机制，则无需对用户进行认证。

下面的示例展示如何在部署描述文件中声明表单认证机制：

```
<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>
```

登录和出错提示页面的位置是一个相对路径（相对于部署描述文件）。24.4.2节中的“创建登录表单及出错提示页面”部分将展示如何对其进行配置。

下面的示例展示如何在部署描述文件中声明摘要认证机制：

```
<login-config>
  <auth-method>DIGEST</auth-method>
</login-config>
```

下面的示例展示如何在部署描述文件中声明客户端认证机制：

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

24.2.3 声明安全角色

通过使用部署描述文件中的security-role元素，可以声明Web应用中的安全角色名。使用该元素可罗列应用使用到的所有安全角色名。

下面的部署描述文件代码片段使用security-role元素声明了应用中将要使用的角色，并通过auth-constraint元素指定可以访问受保护资源的角色。

```
<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>

```

在本例中，`security-role`元素列举了应用中用到的全部角色，即`manager`和`employee`。应用部署人员可以就此将应用中定义的所有角色映射至GlassFish服务器上定义的用户和用户组。

`auth-constraint`元素指定`manager`角色可以访问通过`url-pattern`元素指定的目录（`/jsp/security/protected/*`）下的HTTP方法，包括PUT、DELETE、GET和POST。

由于`@ServletSecurity`注解中的限制条件将作用于所有通过`@WebServlet`注解设定的URL模式，因而不能应用于上述场景中。

24.3 在 Web 应用中使用编程式安全机制

当应用对安全性有特殊需求，而声明式的安全机制不能满足要求时，需要使用编程式安全机制。

24.3.1 以编程方式实现用户认证

Servlet 3.0规范在`HttpServletRequest`接口中提供了如下方法，用以通过编程为Web应用认证用户。

- ❑ **authenticate** 允许应用在接收到未经认证的请求时，触发容器对请求发起者进行认证。显示登录对话框，并接收用户输入的用户名和密码，以满足认证的要求。
- ❑ **login** 允许应用接收用户名和密码数据，作为表单认证机制（在部署描述文件中指定的）的替代方式。
- ❑ **logout** 允许应用重置请求发起者的身份。

下面的代码展示了如何使用`login`和`logout`方法：

```

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();
        }
    }
}

```

下面的代码展示了如何使用authenticate方法:

```

package com.sam.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();
    }
}

```

24.3.2 以编程方式检查发起者身份

通常来说，安全管理应该对Web组件透明，由容器强制执行。本节介绍的与安全相关的API仅应用于不常见的场合，其中Web组件中的方法需要访问有关安全的上下文信息。

Servlet 3.0规范提供了如下方法，以访问与组件调用的发起者相关的安全信息。

- ❑ **getRemoteUser** 获得客户端在验证时使用的用户名。**getRemoteUser**方法返回请求在访问容器时使用的远程用户（发起者）的名称。如果用户未通过认证，则该方法返回null。
- ❑ **isUserInRole** 判定远程用户是否具有特定的安全角色。如果用户未通过认证，则该方法返回false。该方法需要一个String类型的role-name（用户角色名）作为参数。在部署描述文件的security-role-ref元素中，需要声明一个role-name子元素，以此定义一个角色名，以用于在调用该方法时将其作为参数传递。关于安全角色引用的内容，将在24.3.4节中讨论。
- ❑ **getUserPrincipal** 判定当前用户的主体（principal）名称，并返回java.security.Principal对象。如果用户未获得认证，则该方法返回null。调用getUserPrincipal方法返回的Principal对象中的getName方法，将返回远程用户的名称。应用可以根据调用上述API返回的结果，实现用户认证逻辑。

24.3.3 编程安全性的代码示例

下面的代码通过定制的登录逻辑，展示了编程安全性的使用方法。该servlet具备如下功能：

- (1) 显示当前用户的信息；
- (2) 提示用户登录；
- (3) 再次输出当前用户的信息，以展示执行login方法的结果；
- (4) 实现用户退出；
- (5) 再次输出当前用户的信息，以展示执行logout方法的结果。

```

package enterprise.programmatic_login;

import java.io.*;
import java.net.*;
import javax.annotation.security.DeclareRoles;
import javax.servlet.*;
import javax.servlet.http.*;

@DeclareRoles("javaee6user")
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("javaee6user")+"<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("javaee6user")+"<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("javaee6user")+"<br>");
            out.println("getRemoteUser?.." + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.." + request.getAuthType()+"<br>");
        } finally {
            out.close();
        }
    }
    ...
}

```

24.3.4 声明并关联角色引用

Web组件在调用`isUserInRole(String role)`时使用了一个角色名称，应用自身定义了一组角色名称，安全角色引用定义了两者的映射关系。如果没有在部署描述文件中声明`security-role-ref`元素，而`isUserInRole`方法被调用，则在默认情况下，容器将按照Web应用中定义的所有安全角色的列表，逐一比较提供的角色名称。在这种情况下，当在应用中修改角色名

称后，需要重新编译调用该方法的servlet。这种做法限制了应用的灵活性。

当应用使用HttpServletRequest.isUserInRole(String role)方法时，通常配合使用security-role-ref元素。传递给isUserInRole方法的参数是String类型，代表用户的角色名。role-name元素的值必须是给HttpServletRequest.isUserInRole(String role)方法传递参数时使用的String值。role-link至少要包含一个在security-role中定义的安全角色名。容器依据security-role-ref到security-role的映射决定方法的返回值。

例如，将安全角色引用cust映射至安全角色bankCustomer，其语法如下：

```
<servlet>
...
    <security-role-ref>
        <role-name>cust</role-name>
        <role-link>bankCustomer</role-link>
    </security-role-ref>
...
</servlet>
```

如果servlet的方法由一个具有bankCustomer安全角色的用户调用，则isUserInRole("cust")方法返回true。

security-role-ref元素中的role-link元素必须与在同一个web.xml部署描述文件中的security-role元素中定义的role-name相匹配，如下面代码所示：

```
<security-role>
    <role-name>bankCustomer</role-name>
</security-role>
```

对于安全角色引用，包括引用所定义的名称，其作用域仅限于部署描述文件中定义了security-role-ref元素的组件。

24.4 为 Web 应用增加安全性的示例

在任何示例应用正常运行之前，需要进行基本的设置。本示例使用了注解、编程式安全机制以及（或）声明式安全机制，以展示为已有Web应用增加安全性的方法。

在下述的章节中，可以看到为不同应用增加安全性的更多示例：

- 25.2.1节；
- 25.2.2节；
- GlassFish示例见<http://glassfish-samples.java.net/>。

▼ 为运行示例程序设置系统

要运行安全性示例，需要对系统进行设置，即配置一个用户数据库，以使应用可以通过该数据库进行用户认证。在开始之前，请执行如下操作。

(1) 将授权用户添加至GlassFish服务器。对于本章和第25章中的示例，需要将一个用户添加至GlassFish服务器的file域（realm）中，并将该用户分配至用户组TutorialUser。

(a) 在管理控制台中，展开Configuration结点。

- (b) 展开Security结点。
- (c) 展开Realms结点。
- (d) 选择File结点。
- (e) 在Edit Realm页面中, 单击Manage Users。
- (f) 在File Users页面中, 单击New。
- (g) 在User ID字段中, 输入一个用户ID。
- (h) 在Group List字段中, 输入TutorialUser。
- (i) 在New Password和Confirm New Password字段中, 输入密码。
- (j) 单击OK。

请牢记新创建的用户名和密码, 测试示例时应时会用到。认证区分用户名和密码的大小写, 因此请准确记录这些信息。这个主题已在23.5.2节中详尽地介绍过。

(2) 在GlassFish服务器上设置默认的主体到角色的映射。

- (a) 在管理控制台中, 展开Configuration结点。
- (b) 选择Security结点。
- (c) 选择Default Principal to Role Mapping Enabled复选框。
- (d) 单击Save。

24.4.1 在servlet中使用基本认证的示例

本示例展示如何在servlet中使用基本认证。对于servlet中的基本认证, Web浏览器将显示一个标准的登录对话框, 该对话框不可定制。在用户提交用户名和密码后, 服务器将判定该用户是否为授权的用户。如果授权用户访问, 则服务器将返回用户请求的Web资源。

通常来说, 为一个未进行安全加固的servlet增加基本认证, 需要执行一系列必要的步骤, 如第3章所述。在本书介绍的示例应用中, 上述的多数步骤均已完成, 而此处仅展示当要创建一个类似应用时所要做的的工作。这个示例应用的完整代码可以在*tut-install/examples/security/hello2_basicauth/*中找到。

(1) 执行24.4节中“为运行示例程序设置系统”部分介绍的步骤。

(2) 如第3章所述为名为hello2的servlet创建一个Web模块。

(3) 在servlet中添加合适的安全性注解。关于安全性注解的介绍, 参见接下来的“使用注解实现基本认证”。

(4) 参照接下来的“使用NetBeans IDE构建、打包及部署servlet基本认证示例”或“使用Ant构建、打包及部署servlet基本认证示例”执行相应操作。

(5) 参照接下来的“运行servlet基本认证示例”运行Web应用。

使用注解实现基本认证

基本认证是GlassFish服务器的默认安全认证机制。利用基本认证, GlassFish服务器通过标准的登录对话框获取访问受保护资源时用户提交的用户名和密码。一旦用户通过认证, 便可访问受

保护资源。

为servlet设定安全性，可以使用@ServletSecurity注解。这个注解既可以为特定的HTTP方法设定访问限制规则，也可以为那些没有特定访问限制的HTTP方法设定一般性访问限制规则。在@ServletSecurity注解中，可以进一步使用如下注解。

- ❑ @HttpMethodConstraint注解，应用于特定的HTTP方法。
- ❑ 更通用的@HttpConstraint注解，应用于没有通过@HttpMethodConstraint注解的其他HTTP方法。

@ServletSecurity注解中的@HttpMethodConstraint和@HttpConstraint注解均可使用如下元素。

- ❑ transportGuarantee元素，指定数据在传输过程中的保护方式（是否需要SSL/TLS）。该元素可接受的值为NONE或CONFIDENTIAL。
- ❑ rolesAllowed元素，指定授权访问的角色。

对于hello2_basicauth应用，GreetingServlet使用如下注解：

```
@WebServlet(name = "GreetingServlet", urlPatterns = {"/greeting"})
@ServletSecurity(
    @HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
        rolesAllowed = {"TutorialUser"}))
```

该注解声明，对于请求URI/greeting，仅有通过验证且具有TutorialUser角色的用户可以访问此URL。数据将在受保护的情况下传送，以确保用户名和密码在传输的过程中不被窃取。

▼使用NetBeans IDE构建、打包及部署servlet基本认证示例

- (1) 执行24.4节中的“为运行示例程序设置系统”介绍的步骤。
 - (2) 在NetBeans IDE中，选择File → Open Project。
 - (3) 在Open Project对话框中，选择路径*tut-install/examples/security*。
 - (4) 选择hello2_basicauth目录。
 - (5) 选择Open as Main Project复选框。
 - (6) 单击Open Project。
 - (7) 在Projects窗格中右键单击hello2_basicauth，选择Deploy。
- 该选项将构建并部署应用至GlassFish服务器实例。

▼使用Ant构建、打包及部署servlet基本认证示例

- (1) 执行24.4节中的“为运行示例程序设置系统”介绍的步骤。
- (2) 在终端窗口中，切换目录到*tut-install/examples/security/hello2_basicauth/*。
- (3) 键入如下命令：

```
ant
```

该命令将执行default任务，构建并将应用打包至WAR文件hello2_basicauth.war中，它位于dist目录下。

(4) 确保GlassFish服务器已经启动。

(5) 键入下面的命令，以部署应用：

```
ant deploy
```

▼ 运行servlet基本认证示例

(1) 在Web浏览器中，输入如下地址：

`https://localhost:8181/hello2_basicauth/greeting`

用户可能会被提示接受服务器端的安全证书。如有提示，接受安全证书即可。如果浏览器发现该证书是自签署的并给出提示，请将其加入安全列表中。

页面中将会出现一个Authentication Required对话框。该对话框的外观可能因浏览器的不同而有差异。图24-6是其中一例。

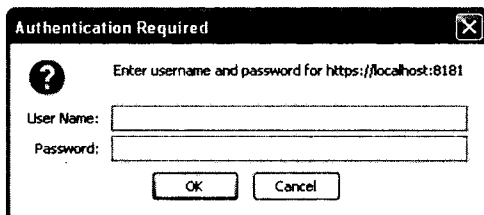


图24-6 基本认证对话框示例

(2) 输入用户名和密码，确保它是可以在GlassFish服务器的file域中找到的，且已被分配至TutorialUser用户组。单击OK。

基本认证区分用户名和密码的大小写，因此在输入时，请仔细核对，以确保输入与在GlassFish服务器上定义的完全一致。

如果下列条件全部满足，服务器返回请求的资源。

- ☐ 用户输入的用户名在GlassFish服务器上有定义。
- ☐ 对应输入的用户名输入正确的密码。
- ☐ 用户属于GlassFish服务器上的TutorialUser用户组。
- ☐ 应用中定义的TutorialUser角色可以映射至GlassFish服务器中定义的TutorialUser用户组。

当上述条件满足，且用户通过了服务器认证，应用将展示图3-2中所示的外观，只是URL略有不同。

(3) 在文本框中输入名称，并单击Submit按钮。

由于用户已被认证，此处输入的用户名无限制。此时，用户可以毫无限制地访问应用。

应用返回给用户一个“Hello”，如图3-3所示，只是URL略有不同。

后续操作 如果需要重复测试该应用，可能需要关闭并重新打开浏览器。也可以运行antundeploy或ant clean任务，或在NetBeans IDE中选择Clean和Build选项，以重新启动应用。

24.4.2 在JSF中使用表单认证机制的示例

本示例将展示如何在JSF应用中使用表单认证机制。利用表单认证机制，开发人员可以定制显示在Web客户端上的登录和出错提示页面，以依据用户名和密码认证用户。在用户提交用户名和密码之后，服务器将判定提交的用户名和密码是否对应某授权用户，如果认证通过则发送请求的Web资源。

本示例，即hello1_formauth，为3.3节中介绍的基本JSF示例应用增加了安全设置。

通常来说，为尚未进行安全设置的JSF应用增加表单认证机制的过程与在24.4.1节介绍的过程基本类似。主要的差别在于，在部署描述文件中需要指定使用表单认证机制，参见接下来的“为表单认证示例设定安全性”。除此之外，必须创建登录表单页面和出错提示页面，参见接下来的“创建登录表单及出错提示页面”。

1. 创建登录表单及出错提示页面

当使用表单登录机制时，必须指定一个页面（包含表单），用于接收用户提交的用户名和密码，以及当登录认证失败时展示错误信息的页面。接下来的“为表单认证示例设定安全性”部将展示如何在部署描述文件中指定这些页面。

登录页面可以是HTML、JSF、JSP页面或servlet。该页面必须返回遵循命名规范（关于命名规范，参见Java Servlet 3.0规范），且包含表单的HTML页面。此要求可通过在登录页面中添加<form></form>标签间的元素（接收用户名和密码）来实现。用于登录的HTML、JSF、JSP页面或servlet的代码形式如下：

```
<form method=post action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
</form>
```

本示例中用到的登录页面的完整代码，参见tut-install/examples/security/hello1_formauth/web/login.xhtml。运行登录表单页面的示例将在随后介绍，页面参见图24-7。下面是页面的代码：

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Login Form</title>
  </h:head>
  <h:body>
    <h2>Hello, please log in:</h2>
    <form name="loginForm" method="POST" action="j_security_check">
      <p><strong>Please type your user name: </strong>
        <input type="text" name="j_username" size="25"></p>
      <p><strong>Please type your password: </strong>
        <input type="password" size="15" name="j_password"></p>
    </form>
```

```

        <input type="submit" value="Submit"/>
        <input type="reset" value="Reset"/></p>
    </form>
</h:body>
</html>

```

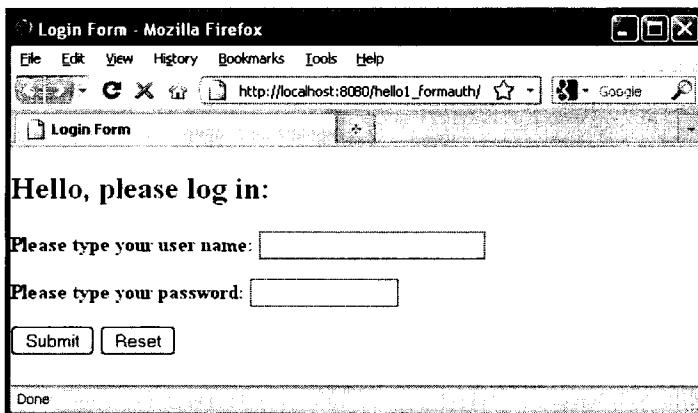


图24-7 基于表单的登录页面

当用户提供基于用户名和密码组合的请求，但未获得对访问受保护URI的授权，服务器将返回登录出错提示页面。对于本例，登录出错提示页面位于 *tut-install/examples/security/hello1_formauth/web/error.xhtml*。这个页面将显示登录失败的原因，并提供一个链接，以允许用户再次尝试登录。下面是本页面的代码：

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Login Error</title>
  </h:head>
  <h: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>
    <h:link outcome="login">Return to login page</h:link>

  </h:body>
</html>

```

2. 为表单认证示例设定安全性

本示例将实现一个非常简单的、基于servlet的Web应用，并为其增加基于表单的安全认证机制。如果为JSF应用增加基于表单的安全认证机制，而不是使用基本认证机制，则必须使用部署描述文件。

下面的代码展示了本例用于设定安全机制而需要添加至部署描述文件中的元素。完整的代码

可以参见 *tut-install/examples/security/hello1_formauth/web/WEB-INF/web.xml*。

```
<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>
```

▼ 使用NetBeans IDE构建、打包及部署表单认证示例

- (1) 执行24.4节中的“为运行示例程序设置系统”介绍的步骤。
- (2) 在NetBeans IDE中，选择File → Open Project。
- (3) 在Open Project对话框中，选择路径*tut-install/examples/security*。
- (4) 选择*hello1_formauth*目录。
- (5) 选择Open as Main Project复选框。
- (6) 单击Open Project。
- (7) 在Projects窗格中右键单击*hello1_formauth*，选择Deploy。

▼ 使用Ant构建、打包及部署表单认证示例

- (1) 执行24.4节中的“为运行示例程序设置系统”介绍的步骤。
- (2) 在终端窗口中，切换目录到*tut-install/examples/security/hello2_formauth/*。
- (3) 键入如下命令：

ant

该任务将编译文件，复制文件到*tut-install/examples/security/hello2_formauth/build/*目录中，创建一个WAR文件，并将该文件复制到*tut-install/examples/security/hello2_formauth/dist/*目录中。

- (4) 将*hello2_formauth.war*部署至GlassFish服务器，键入下面的命令：

ant deploy

▼ 运行表单认证示例

执行下面的步骤，运行hello1_formauth的Web客户端：

(1) 在Web浏览器中打开如下地址：

https://localhost:8181/hello1_formauth/

登录表单页面会显示在浏览器中，如图24-7所示。

(2) 输入用户名和密码。用户名和密码的组合必须存在于GlassFish服务器的file域中，且相应用户已被分配至TutorialUser用户组。

表单认证区分用户名和密码的大小写，因此在输入时请仔细核对，以确保输入与在GlassFish服务器上所定义的完全一致。

(3) 单击Submit按钮。

如果满足下列条件，当在名称和密码输入框中分别输入My_Name和My_Pwd后，服务器返回请求的资源。

- ☐ 用户输入的名为My_Name的用户在GlassFish服务器上已存在。
- ☐ 在GlassFish服务器上名为My_Name的用户的密码为My_Pwd。
- ☐ 用户名为My_Name，密码为My_Pwd的用户属于GlassFish服务器上的TutorialUser用户组。
- ☐ 应用中定义的TutorialUser角色可以映射至GlassFish服务器中定义的TutorialUser用户组。

当上述条件满足，且用户通过了服务器的认证，应用将出现在客户端的浏览器中。

(4) 在文本框中输入名字，并单击Submit按钮。

由于用户已被认证，此处输入的用户名无限制。此时，用户可以毫无限制地访问应用。

应用返回给用户一个“Hello”作为应答。

后续操作 如果需要测试登录出错提示页面，关闭并重新打开浏览器，输入应用的URL，并输入一个无法通过认证的用户名和密码组合。

注意 如果需要重复测试该应用，需要关闭并重新打开浏览器。如果使用Ant，还应运行ant clean或ant undeploy任务，以重新构建和部署应用；或在NetBeans IDE中选择Clean、Build及Deploy选项，重新构建和部署应用。

下述这些角色负责管理企业应用的安全性。

- **系统管理员** 负责创建用户数据库，并将用户分配至适当的用户组。系统管理员还负责 GlassFish 服务器属性的设置，以保障应用的正常运行。在本书中，部分与安全相关的示例应用设置了默认的主体到角色的映射关系、匿名用户、默认用户以及用于身份传播的标识。本书在介绍示例应用的过程中，穿插了对相关设置方法的介绍。
- **应用开发人员/bean提供者** 负责为企业应用的类和方法设置注解，以告知应用部署人员哪些特定的方法需要设置访问控制策略。本书将介绍完成设置所必须执行的操作。
- **应用部署人员** 负责考虑应用开发人员提出的安全需求，将其在部署的过程中加以实现。本书将介绍在部署阶段完成安全设置所必需的操作。

本章内容

- 为企业 bean 增加安全性
- 为企业 bean 增加安全性的一组示例
- 为应用客户端增加安全性
- 为企业信息系统应用增加安全性

25.1 为企业 bean 增加安全性

企业 bean 此处是指通过 EJB 技术实现的 Java EE 组件。企业 bean 运行在 EJB 容器中，对于 GlassFish 服务器来说，该容器是应用的运行时环境。虽然 EJB 容器对应用开发人员透明，但它为企业 bean 提供了一组系统级服务，如企业 bean 的事务管理、安全管理等。这些系统级服务构成了事务型 Java EE 应用的核心。

可以通过如下方法为企业 bean 方法增加安全性。

- **声明式安全性（推荐方案）** 通过部署描述文件或注解的方式提出对应用组件的安全性要求。想要在特定场景下进行方法保护与认证，只需要在企业 bean 类的业务方法中指定注解。本节将讨论这种为企业 bean 增加安全性的简单且高效的方案。

在某些特定的场景下，通过注解方式加固企业 bean 的安全性的做法有一定局限性，此时

还需要使用部署描述文件进行安全设定。认证机制必须在服务器上进行配置才能生效。基本认证机制是GlassFish服务器的默认认证方法。

本书将介绍如何通过为企业应用的业务方法增加访问控制注解，来实现基于用户名和密码的认证机制。

为了简化部署人员的工作，应用开发人员可以定义安全角色。安全角色是一组权限的组合，特定类型的用户必须具备这些权限才可以访问应用。例如，在一个工资单管理系统中，某些用户可查看自己的工资信息（即员工），某些用户可以查看其他用户的工资信息（如经理），某些用户可以修改其他用户的工资信息（如人事部门）。应用开发人员将负责确定应用的潜在用户，以及哪些用户可以访问哪些方法，即用户与方法访问的对应关系。于是，应用开发人员为企业bean的类或方法增加注解，声明授权访问方法的用户类型。关于如何通过注解的方式声明授权用户，将在25.1.1节中的“通过角色声明指定授权用户”进行介绍。

如果使用某种注解定义方法的访问权限，则部署系统在运行并收到请求时，将自动触发用户名和密码的验证。在这种认证机制中，用户被提示输入用户名和密码。这些数据将被传送至服务器端，与用户数据库中的已知用户信息进行比较。如果找到用户且密码匹配，则服务器将比较用户的角色与被授权访问该方法的角色。如果用户认证通过，且具备可访问该方法的角色，则数据将返回给用户。

25.1.1节将详细讨论如何实现声明式安全性。

□ **编程式安全性** 对于企业bean，嵌入至业务方法中的代码片段将以编程的方式访问调用者的用户信息，并使用该信息进行进一步的判定。当声明式的安全机制不能完全满足应用的特定安全需求时，编程式安全机制是一种重要的补充手段。

通常来说，应用的安全策略应该对企业bean中的业务方法透明，且由容器强制执行。本章介绍的编程式安全性API并不常用。当企业bean的业务方法需要访问有关安全的上下文信息时，如按照时段设定访问权限，或为某个角色增加其他重要检查时，才会用到这些API。

25.1.2节将详细讨论编程式安全性。

本章中讲述的某些内容，需要读者首先阅读并掌握第14章、第15章及第23章的内容。

如前所述，企业bean运行在EJB容器中，对于GlassFish服务器来说，EJB容器是一个应用运行时环境。如图25-1所示。

本节将讨论如何为Java EE应用增加安全性。Java EE应用通常由一个或多个模块组成，如EJB JAR文件，并打包至EAR文件中（保存应用的归档文件）。安全注解用在Java的类文件中，设定授权用户，以及声明使用基本认证机制（基于用户名和密码）。

企业bean通常为Web应用提供业务逻辑。在这种场景中，将企业bean打包至Web应用的WAR模块中，会简化应用的部署过程和组织方式。企业bean可以以Java类文件的形式打包至WAR模块中，或先打包成JAR文件，再集成至WAR模块中。当servlet或JSF页面处理Web前端请求，而应用以Java类文件的形式打包至WAR模块中时，应用的web.xml文件将负责设定并处理应用的安全性

问题。如果需要的话，WAR中的EJB可以有单独的部署描述文件ejb-jar.xml。使用web.xml为Web应用增加安全性的内容已在第24章介绍。

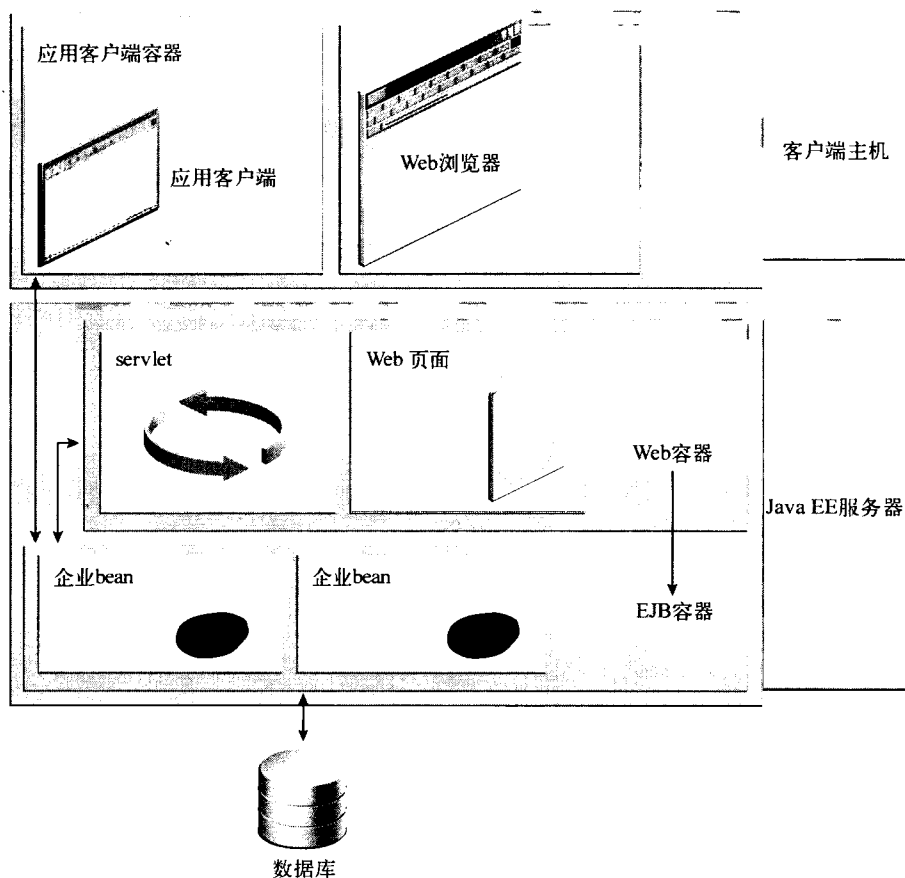


图25-1 Java EE服务器和容器

本章随后的内容将介绍声明方式和编程方式的安全机制，这两种机制将用于保护企业bean资源。受保护的资源包括应用客户端调用的企业bean方法、Web组件以及其他企业bean。

关于这个主题的更多信息，可阅读EJB 3.1规范，该文档可以从<http://jcp.org/en/jsr/detail?id=318>下载。该规范的第17章“Security Management”将介绍企业bean的安全管理。

25.1.1 使用声明方式为企业bean增加安全性

声明式安全性支持应用开发人员指定授权哪些用户访问企业bean中的哪些方法，同时支持其

使用基本认证方式（基于用户名和密码）对用户进行认证。通常情况下，企业应用的开发与部署工作是由不同的人承担的。应用开发人员使用声明方式定义方法的访问权限和认证机制，并将这些信息以EJB JAR中的企业bean安全视图的方式转交给应用部署人员。应用部署人员获得安全视图后，即可以为相应的角色定义访问权限。如果应用开发人员不定义安全视图，则部署人员需要自行判断每一个业务方法的作用并确定其授权方式。

安全视图由一系列安全角色构成，它是一种对权限在语义上的分组，定义哪些类型的用户可以访问应用。安全角色是一种逻辑角色，代表一类用户。可以为每一个安全角色定义方法的访问权限。方法权限是指访问企业bean的一组业务方法的权限，这些方法可以位于业务接口、本地（home）接口、组件接口，以及Web服务端点中。在为方法定义了权限之后，基于用户名和密码的认证机制将用于对用户身份的验证。

需要牢记的是，安全角色用于定义应用的逻辑安全视图，不能与服务器（如GlassFish）上的用户组、用户、主体以及其他一些概念相混淆。需要额外的步骤将应用中定义的角色与应用服务器（如GlassFish）file域中用户数据库里定义的用户、用户组和主体进行映射。这些步骤已在23.5.4节进行了详细介绍。

随后的内容将展示应用开发人员如何通过实现声明式安全性对应用进行安全加固，以及如何创建一个安全视图并将其交付给部署人员。

1. 通过角色声明指定授权用户

本节将介绍如何使用注解为bean类中的方法设定访问权限。有关注解的更多信息，可以参考Java平台常用注解（Common Annotations for Java Platform），该文档位于<http://jcp.org/en/jsr/detail?id=250>。

方法访问权限可以设定在类级别或方法级别，也可以在这两个级别上同时设定。方法访问权限可以设定在bean类的方法上，从而覆写在整个bean类上设定的权限。下面的注解用于方法访问权限的设定。

❑ **@DeclareRoles** 指定应用中使用到的所有角色，包括在@RolesAllowed中没有显式声明的角色。应用中用到的安全角色的集合全部来自于@DeclareRoles和@RolesAllowed中定义的角色。

@DeclareRoles定义在bean的类级别，其中定义的角色所适用的范围为所注释的类的所有方法（可通过调用isCallerInRole进行验证）。当声明一个角色的名称并作为参数传递给isCallerInRole(String roleName)方法时，声明的名称必须与传递给方法的参数值完全一致。

下面的代码展示了@DeclareRoles注解的使用方法：

```
@DeclareRoles("BusinessAdmin")
public class Calculator {
    ...
}
```

下面的代码示例展示了一次声明多个角色的语法：

```
@DeclareRoles({"Administrator", "Manager", "Employee"})
```


- **@RolesAllowed** 指定允许访问应用中方法的安全角色。这个注解可以用于类级别，也可以应用于一个或多个方法上。当定义在类级别上时，这一注解将应用于类中的全部方法。如果定义在方法级别，则该注解仅适用于该方法，并覆写在类级别上对方法权限的定义。

如果声明所有角色均不可以访问应用中的某个方法，可以使用@DenyAll注解。如果指定任何用户均可以访问应用，则使用@PermitAll注解。

当这个注解与@DeclareRoles注解联合使用时，这两个注解中定义的全部角色将作为应用的角色集合。

下面的代码展示了@RolesAllowed注解的使用方法：

```
@DeclareRoles({"Administrator", "Manager", "Employee"})
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        ...
    }
}
```

- **@PermitAll** 声明所有安全角色均可以访问指定的一个或多个方法。在这种设置下，服务器不会在数据库中查询用户以评估其是否具有访问应用的权限。

这个注解可以用于类级别，也可以应用于一个或多个方法。当定义在类级别时，该注解将应用于类中的全部方法。如果定义在方法级别，则该注解仅适用于该方法。

下面的代码展示了@PermitAll注解的使用方法：

```
import javax.annotation.security.*;
@RolesAllowed("RestrictedUsers")
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        //...
    }
    @PermitAll
    public long convertCurrency(long amount) {
        //...
    }
}
```

- **@DenyAll** 声明任何角色均不可以访问指定的一个或多个方法。这就意味着，这些方法不能在Java EE容器中执行。

下面的代码展示了@DenyAll注解的使用方法：

```
import javax.annotation.security.*;
@RolesAllowed("Users")
public class Calculator {
    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        //...
    }
    @DenyAll
    public long convertCurrency(long amount) {
```

```

        //...
    }
}

```

下面的代码片段展示了@DeclareRoles注解及isCallerInRole方法的用法。在这个示例中，@DeclareRoles注解声明了一个角色，PayrollBean这个企业bean使用该角色，通过调用isCallerInRole("payroll")验证调用者是否拥有修改薪酬数据的权限：

```

@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(...);
        }
        ...
    }
    ...
}

```

下面的代码展示了@RolesAllowed注解的使用方法：

```

@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 () {...}
    ...
}

```

在本示例中，假定aMethod、bMethod和cMethod是业务接口A中的方法。aMethod和bMethod的访问权限分别设置为@RolesAllowed("HR")和@RolesAllowed("admin")，而cMethod方法的访问权限未做设定。

需要指出的是，类级别的注解不会被子类继承。不过，方法级别的注解可以应用到继承自父类的子类方法。

2. 设定认证机制和安全连接

为方法设定访问权限后，服务器（如GlassFish）将调用基于用户名和密码的基本认证。

如果使用其他认证机制，或规定必须使用SSL安全连接，则需要在应用的部署描述文件中进

行设定。

25.1.2 使用编程方式为企业bean增加安全性

程式安全是一段嵌入在业务方法中的代码，以编程的方式访问调用者的身份信息，并利用这些信息在方法内部对安全策略进行判定。

访问企业bean调用者的安全上下文

通常来说，安全管理应该由容器强制执行，并保证对企业bean的业务方法透明。本节中介绍的与安全相关的API仅用于一些特定的场合，此时企业bean的业务方法需要访问安全上下文信息，如在特定的时间段内限制方法的访问。

javax.ejb.EJBContext接口提供两种方式，支持bean的提供者访问企业bean调用者的安全上下文信息：

- **getCallerPrincipal** 允许企业bean的方法获得当前调用者的主体名称。例如，该方法可能使用主体名称作为在数据库中查询用户信息的关键字。

下面的代码展示了getCallerPrincipal方法的使用：

```
@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(...);

        ...
    }
}
```

本例中，企业bean获得了当前调用者的主体名称，并将其作为查询主键，用以定位EmployeeRecord实体。本例假定应用已经部署，且当前调用者的主体包含用于标识员工的主键，如员工号。

- **isCallerInRole** 在企业bean的代码中使用该方法，可以使bean提供者或应用开发人员编写定制的安全检查逻辑，以弥补方法权限定义的不足。例如，可以为请求增加一个强制的角色检查，或者到数据库中查询特定信息。

企业bean的代码可以使用isCallerInRole方法，验证是否已为当前的调用者分配一个安全角色。安全角色由bean的提供者或应用装配人员设定，并由应用部署人员分配至应用运

行环境中的主体或主体组。

下面的代码展示了isCallerInRole方法的使用法：

```
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;

    public void updateEmployeeInfo(EmplInfo info) {
        oldInfo = ... read from database;

        // The salary field can be changed only by callers
        // who have the security role "payroll"
        if (info.salary != oldInfo.salary &&
            !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }
        ...
    }
    ...
}
```

当需要在特定的时段以外拒绝访问时，可以使用编程的方式设定安全性，以实现对方法访问的动态控制。关于使用方法getCallerPrincipal和isCallerInRole的示例应用，请参见25.2.2节。

25.1.3 用于身份传播的安全标识（run-as）

可以为企业bean的特定方法设定是否在其执行时使用调用者的安全标识，或者指定是否使用特定的run-as标识。图25-2将解释这个概念。



图25-2 安全标识传播

在上面的图示中，应用客户端发起对EJB容器中某个企业bean方法的调用。这个企业bean的方法在收到请求后，调用另一个容器中企业bean的方法。第一次调用时的安全标识是调用者的标识。第二次调用时的标识可以为下列选项。

- 在默认情况下，中间组件的调用者标识将传递给目标企业bean。该方案用于目标容器信赖中间容器的情景。
- 特定的标识传递给目标企业bean。该方案用于目标容器仅接受来自特定标识的访问时。如接下来的“配置组件的身份传播安全标识”所述，为了将标识传递给目标企业bean，需要为企业bean配置一个run-as标识。为企业bean创建一个run-as标识不影响其调用者的标识，此时调用者的标识在访问企业bean的方法时已经通过验证。企业bean在发起新的调用时，将使用run-as标识。

run-as标识将应用于企业bean，包括企业bean业务接口、本地和远程接口、组件接口以及Web服务端点接口中的所有方法，以及消息驱动bean的消息监听方法、企业bean的超时方

法和bean内部相互调用的所有方法。

1. 配置组件的身份传播安全标识

使用@RunAs注解，可以配置企业bean的run-as（或身份传播）安全标识。该注解定义了应用在Java EE容器中运行时的角色。该注解可以定义在类级别，以支持开发人员以特定的角色运行应用。角色必须映射至容器的安全域中定义的用户和用户组。@RunAs注解以参数的形式指定安全角色的名称。

下面的代码展示了@RunAs注解的使用方法：

```
@RunAs("Admin")
public class Calculator {
    //....
}
```

当引用的角色可能关联至多个用户主体时，需要将run-as的角色名称映射至服务器（如GlassFish）中定义的特定主体。

2. 容器间的信任关系

当企业bean的实现要求使用原始调用者的标识或指定的标识调用目标bean时，目标bean仅收到身份传播标识。目标bean不会收到任何认证数据。

目标容器无法实现对身份传播标识的认证。但是，由于安全标识用于认证检查（如方法访问权限或使用isCallerInRole方法），标识的真实性尤显重要。由于无法为身份传播标识提供认证数据，目标bean或容器必须信赖调用容器所使用的身份传播标识的真实性。

默认情况下，GlassFish服务器信任来自不同容器的身份传播标识。因此，无需额外的操作来建立这种信任关系。

25.1.4 部署经过安全加固的企业bean

应用部署人员负责组装后的应用在部署至目标环境之后的安全性。如果开发人员使用注解或以部署描述文件的方式，为应用部署人员提供一个安全视图，则该视图需与目标运行环境（如GlassFish服务器）的安全认证机制及策略相匹配。如果开发人员没有提供安全视图，则应用部署人员需要自行为企业bean应用设定适当的安全策略。

部署信息通常与Web或应用服务器的特定实现紧密相关。

25

25.2 为企业 bean 增加安全性的一组示例

下面的这组示例展示了如何通过声明的方式和编程的方式为企业bean增加安全性。

25.2.1 使用声明方式为企业bean增加安全性的示例

本节将讨论如何配置企业bean以实现基于用户名和密码的认证。使用这种认证机制的bean在收到请求的时候，服务器要求客户端提供用户名和密码，并将客户端提供的用户名和密码与服务器（如GlassFish）中的授权用户数据库中的数据进行比较，以验证其有效性。

关于认证相关的基础概念, 请阅读24.2.2节中的“在部署描述文件中指定认证机制”部分。

本示例应用将以一个不安全的企业bean应用 (cart) 作为起点, 来展示应用安全化的过程。该应用位于 *tut-install/examples/ejb/cart/* 目录下, 并在16.1节中进行过详细介绍。

通常来说, 以下是改造一个包含企业bean的应用, 为其增加基于用户名和密码的认证机制所必需的步骤。在本书介绍的这个示例应用中, 这些步骤已经完成。希望通过此处的介绍, 能够帮助读者更好地理解并运用这个方法开发类似的应用。

(1) 创建一个与16.1节中cart示例相似的应用。本书以这个示例应用为起点, 展示如何为这个应用增加基本认证机制, 实现对客户端的认证。本节介绍的示例应用的代码位于 *tut-install/examples/security/cart-secure/* 目录下。

(2) 请先按照24.4节中“为运行示例程序设置系统”介绍的系统设置方法对系统进行基本配置。

(3) 修改企业bean *CartBean.java* 的源代码, 指定哪些角色可以访问其中受保护的哪些方法。这个步骤将在接下来的“为bean增加注解”部分详细介绍。

(4) 构建、打包并部署企业bean。然后按照接下来的“使用NetBeans IDE构建、打包、部署及运行安全示例应用cart”或“使用Ant构建、打包、部署及运行安全示例应用cart”部分介绍的过程构建并运行客户端应用。

为bean增加注解

下面的代码片段展示了修改后的cart应用 (改动的部分以**粗体标注**)。最终修改后的 *CartBean.java* 文件位于 *tut-install/examples/security/cart-secure/cart-secure-ejb/src/java/cart/ejb/* 路径下。

```
package cart.ejb;

import cart.util.BookException;
import cart.util.IdVerifier;
import java.util.ArrayList;
import java.util.List;
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 {
    List<String> contents;
    String customerId;
    String customerName;

    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<String>();
    }
}
```

```
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<String>();
}

@RolesAllowed("TutorialUser")
public void addBook(String title) {
    contents.add(title);
}

@RolesAllowed("TutorialUser")
public void removeBook(String title) throws BookException {
    boolean result = contents.remove(title);

    if (result == false) {
        throw new BookException "\"" + title + "\" not in cart.");
    }
}

@RolesAllowed("TutorialUser")
public List<String> getContents() {
    return contents;
}

@Remove()
@RolesAllowed("TutorialUser")
public void remove() {
    contents = null;
}
}
```

@RolesAllowed注解应用在需要进行访问控制的方法上。在本示例中，仅有具备TutorialUser角色的用户可以为购物车添加或删除书籍，以及浏览购物车中的内容。@RolesAllowed注解隐式地声明了应用中引用的一个角色，因此无需再使用@DeclareRoles注解声明了。@RolesAllowed注解同时还隐式地声明了用户在访问这些方法之前需要进行认证。如果在部署描述文件中没有特别指明认证方法，则默认使用基于用户名和密码的认证方法。

▼使用NetBeans IDE构建、打包、部署及运行安全示例应用cart

- (1) 按照24.4节中的“为运行示例程序设置系统”部分介绍的方法进行系统初始配置。
- (2) 在NetBeans IDE中，选择File → Open Project。

(3) 在Open Project对话框中, 选择路径*tut-install/examples/security/*。

(4) 选择cart-secure目录。

(5) 选择Open as Main Project和Open Required Projects复选框。

(6) 单击Open Project。

(7) 在Projects标签中, 右键单击cart-secure项目并选择Build。

(8) 在Projects标签中, 右键单击cart-secure项目并选择Deploy。

这个步骤将构建并打包应用至cart-secure.ear文件中(该文件位于*tut-install/examples/security/cart-secure/dist/*路径下), 并将这个EAR文件部署至GlassFish应用服务器实例上。

(9) 右键单击cart-secure项目并选择Run, 以运行应用客户端。

此时将出现一个用户登录对话框。

(10) 在对话框中, 键入一个在GlassFish服务器的file域中定义的, 且属于TutorialUser用户组的用户名及其密码, 单击OK。

如果输入的用户名和密码通过认证, 应用客户端的输出结果将显示在Output窗格中:

```
...
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通过认证, 对话框将再次出现, 直至输入正确的用户名和密码。

▼使用Ant构建、打包、部署及运行安全示例应用cart

(1) 按照24.4节中的“为运行示例程序设置系统”部分介绍的方法进行系统初始配置。

(2) 在终端窗口中, 切换目录至*tut-install/examples/security/cart-secure/*。

(3) 在终端窗口或命令提示符中键入如下命令, 以构建应用并将其打包至EAR文件中:

```
ant
```

(4) 键入如下命令, 将应用部署至GlassFish服务器:

```
ant deploy
```

(5) 键入如下命令, 以运行应用客户端:

```
ant run
```

该任务将读取并执行应用客户端的JAR文件。此时将出现一个用户登录对话框。

(6) 在对话框中, 键入一个在GlassFish服务器的file域中定义的, 且属于TutorialUser用户组的用户名及其密码, 单击OK。

如果输入的用户名和密码通过认证, 应用客户端的输出结果将显示在Output窗格中:

```
[echo] running application client container.
[exec] Retrieving book title from cart: Infinite Jest
[exec] Retrieving book title from cart: Bel Canto
```



```
[exec] Retrieving book title from cart: Kafka on the Shore
[exec] Removing "Gravity's Rainbow" from cart.
[exec] Caught a BookException: "Gravity's Rainbow" not in cart.
[exec] Result: 1
```

如果用户名和密码未通过认证，对话框将再次出现，直至输入正确的用户名和密码。

25.2.2 使用编程方式为企业 bean 增加安全性的示例

本示例应用将展示如何在企业bean中使用`getCallerPrincipal`和`isCallerInRole`方法。本示例应用将以一个简单的EJB应用`converter`为基础，修改`ConverterBean`中的方法，使得仅当请求发起者的角色为`TutorialUser`时汇率的转换才能执行。

本示例的完整代码位于`tut-install/examples/security/converter-secure`目录下。本示例源于一个不安全的企业bean应用`converter`——已在第15章进行过详细介绍，其代码位于`tut-install/examples/ejb/converter/`目录下。本节将对应用进行必要的修改，使用`getCallerPrincipal`和`isCallerInRole`方法为应用增加安全性。有关这两个方法的详细内容，参见25.1.2节中的“访问企业bean调用者的安全上下文”。

通常来说，为了在企业bean中使用`getCallerPrincipal`和`isCallerInRole`方法，必须执行下列步骤。在本书介绍的这个示例应用中，下述的这些步骤很多已经完成。希望通过此处的介绍，能够帮助读者更好地理解并开发类似的应用。

- (1) 创建一个简单的企业bean应用。
- (2) 在GlassFish服务器上，在file域中设置一个用户，将其分配至`TutorialUser`用户组，并设置默认的主体到角色的映射关系。这个过程可以参考24.4节中的“为运行示例程序设置系统”。
- (3) 修改bean的代码，增加`getCallerPrincipal`和`isCallerInRole`方法。
- (4) 如果应用包含一个用servlet实现的Web客户端，需要为servlet指定安全属性。具体方法参见24.4.1节中的“使用注解实现基本认证”。
- (5) 构建、打包、部署并运行应用。

1. 修改ConverterBean

本示例应用对`ConverterBean`的源代码进行了修改，增加了`if...else`语句，以验证请求发起者是否具备`TutorialUser`角色。如果用户具备正确的角色，汇率转换将执行，并显示转换后的结果。如果用户不具备正确的角色，转换操作将不执行，且应用将以0作为转换结果。代码位于如下文件中：

`tut-install/examples/ejb/converter-secure/converter-secure-ejb/src/java/converter/ejb/ConverterBean.java`

修改过的代码片段展示如下（修改处用粗体标注）：

```
package converter.ejb;

import java.math.BigDecimal;
import javax.ejb.Stateless;
import java.security.Principal;
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 BigDecimal yenRate = new BigDecimal("89.5094");
    private BigDecimal euroRate = new BigDecimal("0.0081");

    @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);
        }
    }
}

```

2. 修改ConverterServlet

下面的注解为converter的Web客户端（即ConverterServlet）设定了安全性：

```

@WebServlet(name = "ConverterServlet", urlPatterns = {"/"})
@ServletSecurity(
    @HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
        rolesAllowed = {"TutorialUser"}))

```

▼使用NetBeans IDE构建、打包及部署安全示例应用converter

- (1) 按照24.4节中“为运行示例程序设置系统”介绍的方法进行系统初始配置。
- (2) 在NetBeans IDE中，选择File → Open Project。
- (3) 在Open Project对话框中，选择路径tut\install\examples\security/。
- (4) 选择converter-secure目录。
- (5) 选择Open as Main Project复选框。
- (6) 单击Open Project。
- (7) 右键单击converter-secure项目并选择Build。
- (8) 右键单击converter-secure项目并选择Deploy。

▼ 使用Ant构建、打包及部署安全示例应用converter

- (1) 按照24.4节中“为运行示例程序设置系统”介绍的方法进行系统初始配置。
- (2) 在终端窗口中，切换目录到`tut-install/examples/security/converter-secure/`。
- (3) 键入如下命令：

```
ant all
```

该命令将实现示例应用的构建及部署。

▼ 运行安全示例converter

- (1) 打开Web浏览器，输入如下地址：
`http://localhost:8080/converter`
浏览器中将出现Authentication Required对话框。
- (2) 在对话框中，键入一个在GlassFish服务器的file域中定义的，且属于TutorialUser用户组的用户名及其密码，单击OK。
浏览器界面如图15-1所示。
- (3) 在输入框中输入100并单击Submit。
新的页面将出现，显示转换后的值。

25.3 为应用客户端增加安全性

Java EE应用客户端的认证要求以及使用的认证技术与其他Java EE组件相同。访问不受保护的Web资源时，无需认证。

当访问受保护的Web资源时，常用的认证方法均可使用，如HTTP基本认证、SSL客户端认证和HTTP登录表单认证。有关这些认证方法的介绍，参见24.2.2节中的“在部署描述文件中指定认证机制”。

当访问受保护的企业bean时，需要进行认证。关于企业bean访问的认证，参见25.1节。

应用客户端使用应用客户端容器提供的认证服务，实现对其用户的认证。容器的服务可以集成至本地平台的认证系统中，以便实现单点登录。容器可以在应用启动时，或当收到对受保护资源的访问请求时，对用户进行认证。

应用客户端可以提供一类，如登录模块，以接收认证信息。如果采用这种方式，必须实现`javax.security.auth.callback.CallbackHandler`接口，且接口实现类的名称必须在部署描述文件中指定。应用的回调处理程序必须完全支持`javax.security.auth.callback`包中指定的Callback对象。

25.3.1 使用登录模块

应用客户端可以使用JAAS (Java Authentication and Authorization Service, Java认证与授权服务)创建用于认证的登录模块。基于JAAS的应用需实现`javax.security.auth.callback.Callback-`

Handler接口,从而与用户进行交互,以获取用户提供的认证信息(如用户名和密码),或显示出错与提示信息。

应用需实现CallbackHandler接口,并将其传递给登录上下文,而上下文会直接将其转发给底层的登录模块。登录模块使用回调处理程序,一方面接收来自用户的输入(如密码、智能卡PIN码),一方面反馈信息(如状态信息)给用户。由于应用指定的是回调处理程序,因而底层的登录模块保持相对独立,可以通过不同方式实现与用户的交互。

例如,对于图形用户界面的应用,其回调处理程序可以设计成显示一个窗口以接收用户输入,而对于命令行方式的应用,其回调处理程序可以设计成提示用户直接在命令行上进行输入。

登录模块传递一个回调对象数组(如获取用户名的NameCallback和获取密码的PasswordCallback)给回调处理程序的handle方法。回调处理程序执行用户请求,为回调对象设置适当的值。例如,为了处理NameCallback对象,CallbackHandler提示用户输入用户名,并接收输入结果,进而调用NameCallback对象的setName方法记录用户名。

有关在登录模块中使用JAAS的更多信息,参考下列资源(这些资源的URL参见23.7节):

- 文档“Java Authentication and Authorization Service (JAAS) Reference Guide”;
- 文档“Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide”。

25.3.2 使用程式用户登录

程式用户登录使得客户端可以以编程的方式获取用户凭证。如果是EJB客户端,可以使用com.sun.appserv.security.ProgrammaticLogin类及其提供的login和logout方法。程式用户登录与特定的服务器相关。

25.4 为企业信息系统应用增加安全性

在企业信息系统应用中,组件请求连接企业信息资源。作为连接的一部分,企业信息系统可以要求发起请求的一方在登录后才能访问资源。应用组件提供者有两种方式登录企业信息系统。

- **容器管理登录** 应用组件让容器负责配置并管理企业信息系统的登录。容器为建立到企业信息系统实例的连接,需对用户名和密码进行验证。更多信息,参见25.4.1节。
- **组件管理登录** 应用组件通过在内部封装登录企业信息系统的逻辑,实现对企业信息系统的管理。更多信息,参见25.4.2节。

开发人员可以为资源适配器配置安全性。参见25.4.3节以获取更多信息。

25.4.1 容器管理登录

在容器管理登录的模式下,应用组件可以不向getConnection()方法传递任何与登录相关的信息。容器负责提供安全信息,如下面的代码所示:

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

25.4.2 组件管理登录

在组件管理登录的模式下，应用组件负责将用于访问资源的登录用安全信息传递给 `getConnection` 方法。例如，安全信息可以是用户名和密码，如下面的代码所示：

```
// 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);
...
```

25.4.3 配置资源适配器安全性

资源适配器是系统级的软件组件，通常实现到外部资源管理器的网络连接。资源适配器可以通过两种方式扩展Java EE平台的功能，即实现某种Java EE标准服务API（如JDBC驱动），或为一个外部应用系统的连接器定义并实现一个资源适配器。资源适配器也可以提供本地服务能力，与本地资源进行交互。资源适配器通过Java EE SPI（Service Provider Interface，服务提供接口）与Java EE平台交互。使用Java EE SPI挂接至Java EE平台上的资源适配器，可以与所有Java EE服务器的实现协作。

为资源适配器配置安全性，开发人员需要编辑资源适配器的描述文件 `ra.xml`。下面的示例是 `ra.xml` 文件中的一部分，为一个资源适配器配置了一些安全属性：

```
<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>
```

浏览`as-install/lib/dtds/connector_1_0.dtd`文件，可以了解在配置资源适配器安全性时可用到的更多选项。可以在资源适配器部署描述文件中配置如下元素。

- **认证机制** 使用`authentication-mechanism`元素指定资源适配器支持的某种认证机制。此处指定的认证针对的是资源适配器，而非底层企业信息系统实例。

有两种认证机制，如下。

- **BasicPassword**，支持下列接口：

```
javax.resource.spi.security.PasswordCredential
```

- **Kerbv5**，支持下列接口：

```
javax.resource.spi.security.GenericCredential
```

Glass Fish 服务器目前不支持这种认证机制。

- **重认证支持** 使用`reauthentication-support`元素指定资源适配器的实现是否支持已有`Managed-Connection`实例的重认证。其值为`true`或`false`。

- **安全权限** 使用`security-permission`元素指定资源适配器代码必须使用的安全权限。对安全权限的支持不是强制性的，目前版本的GlassFish服务器尚不支持此功能。然而，可以通过手工更新`server.policy`文件的方式，为资源适配器增加相关的权限。

在部署描述文件中列举的安全权限不同于在连接器规范中要求的默认权限集合。

有关安全权限规范实现的更多信息，请访问<http://docs.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html#FileSyntax>。

除了在`ra.xml`文件中指定资源适配器的安全性之外，还可以为连接器的连接池创建一个安全映射表，以将应用中的主体或用户组映射至后端的企业信息系统中的主体。当应用中不同的主体或用户组（通过一个或多个企业信息系统中的主体）执行对企业信息系统后端的多个操作时，通常使用安全映射表。

▼ 实现应用中的主体到企业信息系统中主体的映射

当使用GlassFish服务器时，在容器管理的基于事务的场景中，可以使用安全映射表，将应用的调用者标识（主体或用户组）映射至合适的企业信息系统主体。当应用的主体发起一个到企业信息系统的请求时，GlassFish服务器首先使用为连接器的连接池定义的安全映射表，通过严格匹配的方式，检查对应的后端企业信息系统的主体。如果未找到严格匹配的主体，GlassFish服务器将使用通配符进行模糊匹配，判定映射到的后端系统的主体。当应用的用户需要执行一个在企业信息系统中必须具备特定身份才能执行的操作时，会用到安全映射表。

配置安全映射表，需要使用管理控制台。在管理控制台中，参照如下步骤配置安全映射表。

- (1) 在导航树中，展开`Resource`结点。
- (2) 展开`Connector`结点。
- (3) 选择`Connector Connection Pools`结点。

(4) 在Connector Connection Pools页面中，单击将要为其创建一个安全映射表的连接池的名称。

(5) 单击Security Maps标签。

(6) 单击New，为该连接池创建一个新的安全映射表。

(7) 为安全映射表定义一个名字，并提供其他相关信息。

单击Help按钮可以获取关于每个选项的更多信息。

Part 8

第八部分

Java EE 支持技术

该部分主要讲述一些 Java EE 平台的支持技术。

本 部 分 内 容

- 第 26 章 Java EE 支持技术简介
- 第 27 章 事务
- 第 28 章 资源连接

Java EE平台包含了一些用来扩展自身功能的技术和API，这些技术使得应用可以以统一的方式访问很多服务。这些技术将在第27章和第28章详细讲述。

本章内容

- 事务
- 资源

26.1 事务

在Java EE应用里，事务是指一组有关联的行为，而这些行为或者全部执行成功，或者一旦有失败的行为发生，则所有的行为都必须撤销。事务要么以提交结束，要么以回滚结束。

JTA（Java Transaction API，Java事务API）使得应用可以以一种独立于特定实现的方式访问事务。JTA为事务管理器和分布式事务系统里参与通信的各方之间制定标准的Java接口。分布式事务系统参与各方一般是指事务型应用、Java EE服务器以及事务管理器。事务管理器能控制对事务所影响到的共享资源的访问。

JTA定义了UserTransaction接口，应用可以通过这个接口来开始、提交以及终止事务。应用组件可以通过名称`java:comp/UserTransaction`，借助于JNDI查找获取UserTransaction对象，也可以通过EJB资源注入的方式实现对UserTransaction对象的访问。应用服务器使用一组JTA定义的接口与事务管理器进行通信。事务管理器使用JTA定义的接口与资源管理器进行交互。

参见第27章以了解更多细节。JTA 1.1规范可以在如下网址访问：<http://www.oracle.com/technetwork/java/javace/tech/jta-138684.html>。

26.2 资源

资源是指一种程序代码中的对象，该对象提供对数据库服务器或者消息系统的连接。

26.2.1 Java EE连接器架构和资源适配器

借助于Java EE连接器架构，Java EE组件可以与EIS（Enterprise Information System，企业信

息系统)交互。企业信息系统软件包括ERP(Enterprise Resource Planning, 企业资源计划系统)、大型联机事务处理系统以及非关系型数据库等。连接器架构使得多种企业信息系统的整合得到简化, 每一个企业信息系统只需要实现一个连接器架构就可以了。因为连接器架构的实现遵照连接器规范, 所以能移植到任何兼容Java EE的服务器上。

连接器规范定义了应用服务器的契约, 也定义了资源适配器的契约。对特定的企业信息系统资源来说, 资源适配器是系统级的软件驱动。这些标准契约在应用服务器和企业信息系统之间提供了插件式的集成能力。Java EE连接器架构规范1.6定义了新的系统契约, 比如Generic Work Context和Security Inflow。Java EE连接器架构规范1.6参见<http://jcp.org/en/jsr/detail?id=322>。

资源适配器是指为特定企业信息系统实现连接器架构的Java EE组件。资源适配器可以用来支持如下几种类型的事务。

- ❑ **NoTransaction** 不提供事务支持。
- ❑ **LocalTransaction** 支持资源管理器的本地事务。
- ❑ **XATransaction** 资源适配器支持XA分布式事务处理模型, 支持JTA的XATransaction接口。参见第28章以了解有关资源适配器的更详尽内容。

26.2.2 Java Message Service API

软件组件或应用之间可以以传递消息的方式进行通信。消息系统是点对点的系统, 也就是说, 消息的客户端可以发送消息给其他的客户端, 也可以接收其他客户端发来的消息。每一个客户端连接到一个消息代理, 而消息代理提供了诸如创建、发送、接收以及读取消息的功能。

通过使用JMS(Java Message Service, Java消息服务)API, 应用可以创建、发送、接收以及读取消息。JMS API定义了一套通用的接口和相关的语义, 允许用Java开发的程序与其他实现了这套接口的应用互相通信。

JMS API最大程度地减少了程序员为使用基于消息的应用所必须掌握的概念, 提供了足够多的特性以支持复杂多变的消息应用。JMS API还使得JMS应用能够最大程度地在不同的JMS提供者间移植, 只要这些应用是在同一个消息域内即可。

26.2.3 Java数据库连接软件

大多数应用使用关系型数据库保存、组织以及查询数据。Java EE应用通过JDBC API访问关系型数据库。

JDBC资源, 也叫数据源, 为应用提供了到数据库的连接。一般来说, 为了让部署到同一个域中的应用访问数据库, 需要为每个数据库都创建一个JDBC资源。servlet、JSF页面以及企业bean都可以以事务的方式访问JDBC资源。JDBC驱动支持连接池和分布式事务的特性, 以实现与应用服务器的交互。更多信息, 请参考28.2节。

典型的企业应用需要访问一个或多个数据库，以进行数据存取。因为信息对企业运营来说至关重要，所以必须准确、及时且可靠。如果多个应用同时更新同一信息，数据完整性就可能遭到破坏。若系统在处理某个事务时失败，可能会导致数据仅被局部更新，这也会破坏数据的完整性。为了避免这两种情况的发生，需要引入事务，来确保数据的完整性。事务控制了多个应用对数据的并发访问。一旦系统出现了故障，事务机制可以确保在系统恢复之后，数据依然是完整的。

本章内容

- 什么是事务
- 容器托管的事务
- bean托管事务
- 事务超时
- 更新多个数据库中的数据
- Web组件里的事务
- 有关事务的更多信息

27.1 什么是事务

为了模拟一个真实的事务，程序需要执行一组操作。例如，一个财务程序需要在两个账户间转款，其步骤可用如下伪代码表示：

```
begin transaction
    debit checking account
    credit savings account
    update history log
commit transaction
```

上面所列举的3个步骤，要么全部完成，要么全部撤销。否则，数据的完整性就会遭到破坏。因为事务里的步骤通常被视作一个整体，所以事务通常定义为一项不能再分割的工作单元。

事务要么以提交结束，要么以回滚结束。在事务提交之后，对数据的修改会被保存下来。如果事务里的某个语句执行失败，则事务会回滚，以撤销事务对数据所做的修改。例如，在刚才的伪代码中，如果在执行credit那一步时，磁盘驱动器写数据失败，事务就会回滚，debit步骤里

对于数据的修改也会被撤销。虽然事务失败了，数据依旧完好，此时账户依然是结平的。

在刚才的伪代码里，`begin`和`commit`语句指定了事务的边界。当设计企业bean的时候，开发人员需要确定事务的边界。对事务边界的设定有两种方式，即指定容器托管的事务或者指定bean托管的事务。

27.2 容器托管的事务

在使用容器托管事务边界的企业bean中，EJB容器会设置事务的边界。开发人员可以将会话bean或消息驱动bean与容器管理的事务结合在一起使用。因为企业bean的代码不需要显式地指定事务的边界，所以容器托管的事务能简化开发工作。开发人员不需要编写开始事务或者结束事务对应的代码。在默认情况下，如果没有指定事务边界，企业bean使用容器托管的事务边界。

一般来说，在企业bean的方法开始之前容器立刻开始一个事务，并在企业bean的方法退出之前提交事务。每一个方法可以关联一个单独的事务。不支持在一个方法里使用嵌套的事务或者多个事务。

容器托管的事务不要求所有的方法都关联到事务。当开发bean的时候，可以设置事务的属性，以声明哪些bean方法需要关联事务。

使用了容器托管事务边界的企业bean，不能调用任何与容器事务的边界交互的事务管理方法。事务管理方法包括`java.sql.Connection`类的`commit`、`setAutoCommit`以及`rollback`方法，还包括`javax.jms.Session`类的`commit`和`rollback`方法。如果确实需要自己控制事务边界，只能使用应用托管的事务边界。

使用了容器托管事务边界的企业bean，也不能使用`javax.transaction.UserTransaction`接口。

27.2.1 事务属性

事务属性控制了事务的作用域。图27-1展示了控制事务作用域的重要性。在图27-1里，`method-A`方法开始事务，然后调用Bean-2的`method-B`方法。当`method-B`方法执行的时候，它是在`method-A`开始的事务作用域之内，还是在一个新事务里执行？答案依赖于`method-B`方法的事务属性。

事务属性的值可以是如下几种：

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

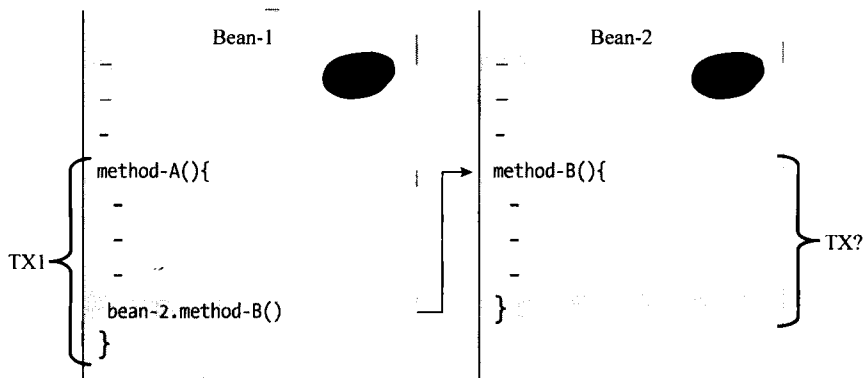


图27-1 事务的作用域

1. Required属性

如果客户端运行时在事务内部调用了企业bean的方法，则方法也是在客户端关联的事务内部执行。如果客户端没有和事务关联起来，则容器在运行这个方法之前开始一个新的事务。

对所有运行在容器托管的事务边界之内的企业bean方法来说，Required属性是隐式的事务属性。一般不需要设置Required属性，除非需要用它覆写另外一个事务属性。因为事务属性是以声明的方式定义的，后期很容易修改。

2. RequiresNew属性

如果客户端运行时在事务内部调用了企业bean的方法，则容器会执行如下操作：

- (1) 挂起客户端事务；
- (2) 开始一个新的事务；
- (3) 接管对方法的调用；
- (4) 在方法执行完毕后恢复客户端事务。

如果客户端没有关联事务，则容器在运行这个方法之前开始一个新的事务。

要想确保方法总是在新的事务里运行，必须使用RequiresNew属性。

3. Mandatory属性

如果客户端运行在事务内部，并调用了企业bean的方法，则方法运行在客户端的事务内部。

如果客户端没有关联事务，则容器会抛出TransactionRequiredException异常。

如果企业bean的方法必须使用客户端的事务，则使用Mandatory属性。

4. NotSupported属性

如果客户端运行在事务内部，并调用了企业bean的方法，则容器会在调用方法之前挂起客户端的事务。容器会在企业bean方法结束之后恢复客户端的事务。

如果客户端没有和事务关联起来，容器在运行这个方法之前不会开始一个新的事务。

对于不需要事务的方法可以使用NotSupported属性。因为省去了事务的开销，所以这个属性能提升性能。

5. Supports属性

如果客户端运行在事务内部，并调用了企业bean的方法，则方法运行在客户端的事务内部。如果客户端没和事务关联起来，则容器在运行这个方法之前，不会开始一个新的事务。

由于不同的JTA提供者对于方法的事务性支持有差异，导致其行为表现有可能不一样，因此应当谨慎使用Supports属性。

6. Never属性

如果客户端运行在事务内部，并调用了企业bean的方法，则容器会抛出RemoteException异常。如果客户端没有和事务关联起来，容器在运行这个方法之前，不会开始新事务。

7. 事务属性总结

表27-1总结了事务属性的作用。T1和T2事务都是由容器控制的。T1事务关联到客户端，客户端调用了企业bean的方法。在大多数情况下，客户端是另外一个企业bean。T2事务是在方法执行之前由容器启动的。

表27-1 事务属性及作用域

事务属性	客户端事务	业务方法的事务
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Mandatory	None	Error
	T1	T1
NotSupported	None	None
	T1	None
Supports	None	None
	T1	T1
Never	None	None
	T1	Error

在表27-1的最后一列里，None意味着业务方法不会在容器控制的事务内部执行。然而，在这样的业务方法里对数据库的调用，有可能是由数据库管理系统的事务管理器来控制。

8. 设置事务属性

通过使用javax.ejb.TransactionAttribute注解修饰企业bean的类或者方法，并把TransactionAttribute设置成javax.ejb.TransactionAttributeType常量定义的某一个值，从而实现事务属性的设定。

如果用@TransactionAttribute注解来修饰企业bean类，那么指定的TransactionAttributeType会应用到类里所有的业务方法上。如果用@TransactionAttribute注解来修饰业务方法，则指定的TransactionAttributeType仅会应用到这个业务方法上。如果类和方法都用了@TransactionAttribute注解，则方法的TransactionAttributeType则会覆写类的TransactionAttributeType。

表27-2列举了TransactionAttributeType常量，这些常量对应了本节里前面介绍的事务属性。

表27-2 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

下面的代码片段演示了如何使用@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类的事务属性设置成了NotSupported，firstMethod方法的事务属性设置成了RequiresNew，secondMethod方法的事务属性则设置成了Required。因为设置在方法上的@TransactionAttribute注解会覆写设置在类上的@TransactionAttribute，对firstMethod方法的调用会创建一个新的事务，而对secondMethod方法的调用有两种可能：要么是在当前的事务里运行，要么创建一个新的事务。对thirdMethod方法或者fourthMethod方法的调用不需要支持事务。

27.2.2 回滚容器托管的事务

有两种方式可以回滚容器托管的事务。第一种方式，如果出现一个系统异常，容器会自动地回滚事务。第二种方式，通过调用EJBContext接口的setRollbackOnly方法，bean方法通知容器回滚事务。如果bean抛出应用异常，回滚不会自动执行，但是通过调用setRollbackOnly方法可以发起回滚。

27.2.3 同步会话bean的实例变量

SessionSynchronization接口是一个可选的接口，使得有状态会话bean实例可以接收事务同步通知。例如，可以把某个企业bean的实例变量与它们在数据库里的对应值进行同步。容器可以

在事务的每一个主要阶段调用SessionSynchronization接口的相应方法：afterBegin、beforeCompletion以及afterCompletion。

afterBegin方法通知实例新的事务开始了。容器在调用业务方法的前一刻调用afterBegin方法。

容器在业务方法执行完之后，但在事务提交之前调用beforeCompletion方法。对于会话bean来说，beforeCompletion方法是回滚事务最后的机会。回滚事务可以通过调用setRollbackOnly方法实现。

afterCompletion方法指明事务已经结束。这个方法有一个boolean类型的参数，它的值为true表明事务已经提交，如果事务已经回滚则这个值为false。

27.2.4 容器托管事务里不允许使用的方法

不能调用任何有可能妨碍容器定义的事务边界的方法。禁止使用的方法列举如下：

- java.sql.Connection的commit、setAutoCommit以及rollback方法；
- javax.ejb.EJBContext的getUserTransaction方法；
- javax.transaction.UserTransaction中的任何方法。

但是，可以使用这些方法设置应用托管事务的边界。

27.3 bean 托管事务

对于bean托管事务的边界，需要在会话bean或者消息驱动bean的代码里显式地指定事务的边界。虽然容器托管事务的bean需要较少的编码工作，但是它有一个限制：当执行一个方法的时候，方法可以关联到一个事务上，也可以不关联任何事务。如果这个限制（究竟是关联还是不关联）使得开发人员在编码时感到困惑，那么应当考虑使用bean托管的事务。

下面的伪代码展示了使用应用托管事务（即bean托管事务）带来的灵活性。通过检查各种条件，伪代码决定在业务方法内部是否开始或者结束一个特定的事务：

```
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
```

当为会话bean或者消息驱动bean编写应用托管事务的代码时，开发人员必须判定使用JDBC（Java Database Connectivity）事务还是JTA事务。接下来讨论这两种事务。

27.3.1 JTA事务

JTA (Java Transaction API, Java事务API) 允许开发人员通过一种不依赖于事务管理器实现的方式设定事务的边界。GlassFish服务器通过JTS (Java Transaction Service, Java事务服务) 实现事务管理器。然而, 应用的代码不直接调用JTS的方法, 而是调用JTA的方法, JTA会调用底层的JTS例程。

JTA事务由Java EE事务管理器来控制。使用JTA事务的优势在于它可以在一个事务里面更新不同厂家的多个数据库里的数据。一般来说, 一个特定的数据库管理系统的事务管理器不能操作异构的数据库。然而, Java EE事务管理器确实有一个限制: 它不支持嵌套的事务。换句话说, 在前一个事务结束之前, 不能开始一个新的事务。

为了设定JTA事务的边界, 可以调用`javax.transaction.UserTransaction`接口的`begin`、`commit`以及`rollback`方法。

27.3.2 不提交的返回

在使用bean托管事务的无状态会话bean里, 业务方法必须在返回之前提交或者回滚事务。然而, 有状态会话bean没有这个限制。

在使用了JTA事务的有状态会话bean里, 即使事务涉及客户端的多次调用, bean实例与事务之间的关联也能得到保持。即使每一个被客户端调用的业务方法需要打开和关闭数据库连接, 上述的关联仍会保持, 直到实例完成这个事务。

在一个使用JDBC事务的有状态会话bean里, 即使事务涉及客户端的多次调用, JDBC连接也能保持bean实例和事务之间的关联。如果连接关闭, 则关联不再保持。

27.3.3 bean托管事务里不允许使用的方法

不要在bean托管的事务里调用EJBContext接口的`getRollbackOnly`方法和`setRollbackOnly`方法。这些方法仅能在容器托管事务里使用。对于bean托管的事务来说, 可以调用`UserTransaction`接口的`getStatus`方法和`rollback`方法。

27.4 事务超时

对于容器托管的事务来说, 可以使用管理控制台来配置事务超时的时间间隔。关于如何启动管理控制台参见2.3节。

对于使用bean托管JTA事务的企业bean来说, 可以调用`UserTransaction`接口的`setTransactionTimeout`方法来设置事务超时的时间间隔。

▼ 设置事务超时

- (1) 在管理控制台里, 展开Configuration结点, 选择Transaction Service。

(2) 在Transaction Service页面中, 将Transaction Timeout字段设置为自己期望的值(比如5)。在这个设定下, 如果事务在5 s之内没有完成, 则EJB容器会自动回滚事务。

该属性的默认值是0, 意味着事务永远不会超时。

(3) 单击Save。

27.5 更新多个数据库中的数据

Java EE事务管理器控制着除bean托管的JDBC事务之外的其他所有企业bean事务。Java EE事务管理器允许企业bean在一个事务里面更新多个数据库中的数据。图27-2和图27-3展示了一个事务里更新多个数据库的两种情形。

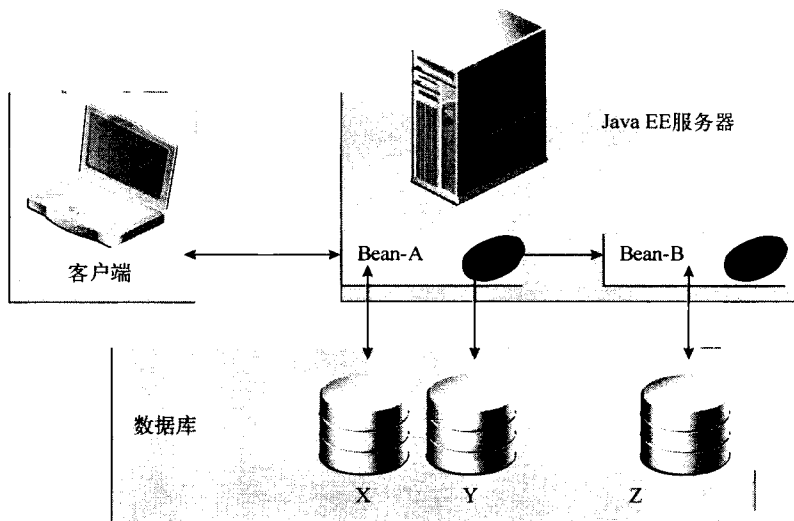


图27-2 更新多个数据库

在图27-2里, 客户端调用了Bean-A的业务方法。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的业务方法。第二个业务方法更新数据库Z, 然后返回控制给Bean-A的业务方法, 以提交事务。所有这3个数据库的更新操作都发生在同一个事务里。

在图27-3里, 客户端调用Bean-A的业务方法, 它开始一个事务, 然后更新数据库X。随后, Bean-A调用Bean-B的业务方法, 这个方法位于远程的Java EE服务器上。Bean-B的方法更新的是数据库Y。Java EE服务器的事务管理器确保两个数据库都是在同一个事务里更新的。

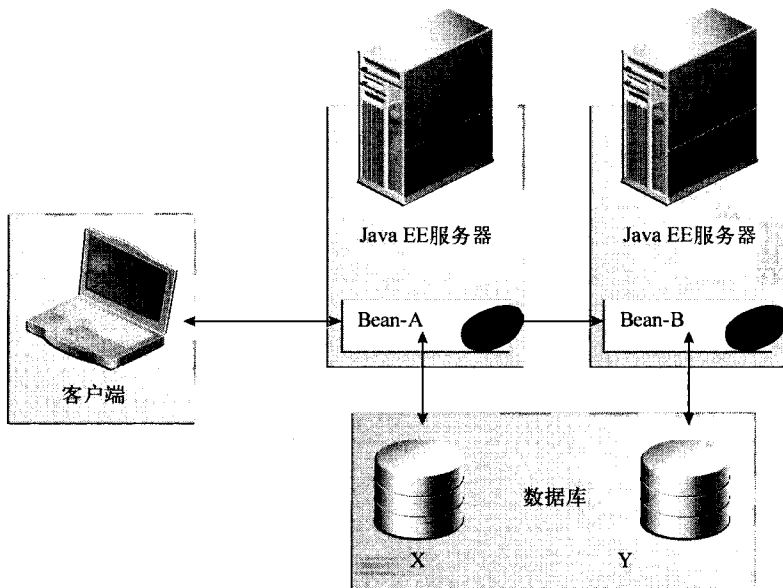


图27-3 跨Java EE服务器更新多个数据库

27.6 Web 组件里的事务

通过使用`java.sql.Connection`或者`javax.transaction.UserTransaction`接口，可以在Web组件里设定事务边界。使用bean托管事务的会话bean也可以使用这两个接口。使用`UserTransaction`接口的事务边界，在27.3.1节已经讲述过了。

27.7 有关事务的更多信息

有关事务的更多信息，可以参考如下文档。

□ Java Transaction API 1.1规范：

<http://www.oracle.com/technetwork/java/javaee/tech/jta-138684.html>

Java EE组件可以访问各种类型的资源，包括数据库、邮件会话、JMS（Java消息服务）对象以及URL。Java EE 6平台提供了以相似的方式访问所有这些资源的机制。本章主要讲述如何连接各种类型的资源。

本章内容

- 资源和JNDI命名
- DataSource对象和连接池
- 资源注入
- 资源适配器和契约
- 元数据注解
- 公共客户端接口
- 参考资源

28.1 资源和 JNDI 命名

在分布式应用里，组件需要访问其他组件和资源，比如数据库。例如，servlet可能调用远程企业bean上的方法从数据库中查询信息。在Java EE平台中，JNDI（Java Naming and Directory Interface，Java命名和目录接口）服务使得组件可以找到其他组件和资源。

资源是指程序代码中的一种对象，提供了到系统的连接，比如数据库服务器或消息系统。（JDBC资源有时候指的是数据源）。每一个资源对象都是由一个唯一的、简单易记的JNDI名称来标识。例如，对于Java DB数据库（假定使用的是GlassFish服务器）的JDBC资源来说，其JNDI名称在GlassFish服务器中定义为jdbc/_default。

JNDI命名空间里的资源是由管理员创建的。在GlassFish服务器里，可以使用管理控制台或者asadmin命令来创建资源。此时，应用可以使用注解来注入这些资源。如果应用使用资源注入，GlassFish服务器会调用JNDI的API，这种情况下，应用没必要再调用JNDI的API。然而，应用是可以通过直接调用JNDI的API找到资源的。

资源对象和它的JNDI名称会通过命名和目录服务绑定到一起。要创建一个新的资源，需

要把一个新的名称/对象绑定加到JNDI命名空间里去。在应用里,可以使用@Resource注解注入资源。

可以使用部署描述文件覆写在注解里指定的资源映射。使用部署描述文件允许开发人员通过重新打包的方式改变应用,而不是通过重新编译源文件然后再打包的方式。然而,对于大多数应用来说,部署描述文件并不是必须有的。

28.2 DataSource 对象和连接池

大多数应用使用关系型数据库保存、组织以及查询数据。Java EE 6组件可以通过JDBC API来访问关系型数据库。有关JDBC API的更多内容,可以参考<http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html>。

在JDBC API里,数据库是通过使用DataSource对象访问的。一个DataSource有一组属性,可以用来标识和描述它所代表的真实数据源。属性中的信息包括数据库服务器的位置、数据库的名称、与服务器通信所使用的网络协议等。在GlassFish服务器里,数据源称作JDBC资源。

应用通过连接访问数据源。对于DataSource实例代表的特定数据源来说,DataSource对象可以认为是一个连接的工厂。在一个基本的DataSource实现中,调用getConnection方法会返回一个连接对象,它是对数据库的物理连接。

DataSource对象可以注册为JNDI命名服务。注册之后,应用可以使用JNDI API访问该DataSource对象,进而可以用来连接它所代表的数据源。

实现了连接池的DataSource对象,也能生成一个到DataSource类所代表特定数据源的连接。getConnection方法返回的连接对象是一个到PooledConnection对象的句柄,而不是一个物理连接。应用与使用连接一样的方式来使用这个连接对象句柄。连接池对应用的代码来说无需特殊处理,除了得到的是池化的连接外,像通常的连接一样,需要显式地关闭连接。当应用关闭池化的连接时,连接会返回至一个可重用连接的池中。下一次调用getConnection的时候,如果连接池里有空闲的连接,则返回一个连接句柄供应用使用。使用连接池使得我们不必在每一次请求连接的时候都创建新的物理连接,因此应用运行起来会快很多。

JDBC连接池是一组针对某个特定数据库的可重用连接。因为创建每一个新的物理连接很耗时,所以服务器维护了一个保存所有可用连接的池子,这样可以提升应用的性能。当应用需要连接的时候,可以从池子里拿到一个可用的连接。当应用关闭连接的时候,连接将归还给池子。

使用PersistenceAPI的应用可以在persistence.xml里的jta-data-source元素里指定应用里所使用的DataSource对象:

```
<jta-data-source>jdbc/MyOrderDB</jta-data-source>
```

通常来说,对于某个持久化单元,这是唯一一处引用JDBC对象的地方。应用的代码不引用任何JDBC对象。

28.3 资源注入

`javax.annotation.Resource`注解可以用来声明对资源的引用。`@Resource`定义在类级别，也可以定义在字段级别或者方法级别。容器会把`@Resource`引用的资源注入到组件里去，这个过程可以发生在应用运行的时候，也可以发生在组件初始化的时候。资源在何时注入取决于采用的是字段或者方法级别的注入，还是类级别的注入。对于字段和方法级别的注入，容器会在应用初始化的时候注入资源。对于类级别的注入，应用会在运行时查找具体的资源。

`@Resource`注解有如下元素：

- `name`，资源的JNDI名称；
- `type`，资源对应的Java语言里的类型；
- `authenticationType`，使用资源时采用的认证方式；
- `shareable`，表明资源是否可以共享使用；
- `mappedName`，资源可以映射成的名字，这个元素不具有移植性，与特定实现相关；
- `description`，资源的描述信息。

`name`元素是资源的JNDI名称，对于字段级别或者方法级别的注入方式，该字段是可选的。对于字段级别的注入方式来说，默认的名称就是类名加上字段名。对于方法级别的注入方式，默认的名称就是该方法对应的JavaBeans的属性名称。对于类级别的注入方式，`name`元素必须指定。

资源的类型由如下因素决定：

- 对于字段级别的注入方式，`@Resource`注解修饰的那个字段的类型；
- 对于方法级别的注入方式，`@Resource`注解修饰的JavaBeans的属性的类型；
- `@Resource`注解的`type`元素。

对于类级别的注入方式，`type`元素是必需的。

`authenticationType`元素仅用在连接工厂资源上，如连接器资源（也称为资源适配器）或数据源。这个元素可以设置成`javax.annotation.Resource.AuthenticationType`中的枚举值：`CONTAINER`和`APPLICATION`；默认值是`CONTAINER`。

`shareable`元素仅用在ORB（Object Resource Broker，对象资源代管者）实例的资源或者连接工厂资源上。该元素指明了资源是否可以与别的组件共享。`shareable`元素的值为`true`或`false`，默认值是`true`。

`mappedName`元素是不可以移植的，资源具体映射成什么名称与具体实现有关。当`name`元素指定了值或者使用默认值时，对应用来说只能是本地的资源。然而有一些Java EE服务器提供了跨应用服务器引用资源的能力，就是通过设置`mappedName`元素来实现的。这种用法也是不可以跨Java EE服务器实现移植的。

`description`元素是资源的描述信息，通常使用应用被部署到的运行环境的默认语种。这个元素有助于标识资源，并帮助开发人员选择正确的资源。

28.3.1 字段级别的注入方式

为了使用字段级别的资源注入，需要声明一个字段，然后用`@Resource`注解修饰这个字段。如果注解的`name`元素和`type`元素没有指定的话，容器会推断资源的名称和类型。如果开发人员明确地指定`type`元素，那么它必须匹配字段的类型。

在下面的代码里，容器基于类名以及字段名，推断资源的名称为`com.example.SomeClass/myDB`。此处推断出资源的类型为`javax.sql.DataSource.class`：

```
package com.example;

public class SomeClass {
    @Resource
    private javax.sql.DataSource myDB;
    ...
}
```

在下面的代码里，JNDI名称是`customerDB`，推断出的类型是`javax.sql.DataSource.class`：

```
package com.example;

public class SomeClass {
    @Resource(name="customerDB")
    private javax.sql.DataSource myDB;
    ...
}
```

28.3.2 方法级别的注入方式

为了使用方法级别的注入，需要声明一个设置方法，并用`@Resource`修饰这个方法。如果注解的`name`元素和`type`元素没有指定的话，容器会推断资源名称和资源类型。设置方法必须遵循JavaBeans关于属性命名的惯例，即方法名必须以`set`开始，返回值的类型为`void`，且只有一个参数。如果指定了`type`元素，那么它必须匹配字段的类型。

在下面的代码里，容器基于类名以及字段名，推断出资源名称为`com.example.SomeClass/myDB`。此处推断出资源类型为`javax.sql.DataSource.class`：

```
package com.example;

public class SomeClass {

    private javax.sql.DataSource myDB;
    ...
    @Resource
    private void setMyDB(javax.sql.DataSource ds) {
        myDB = ds;
    }
    ...
}
```

在下面的代码里，JNDI名称是`customerDB`，推断出资源的类型为`javax.sql.DataSource.class`：

```
package com.example;

public class SomeClass {
```



```

    private javax.sql.DataSource myDB;
    ...
    @Resource(name="customerDB")
    private void setMyDB(javax.sql.DataSource ds) {
        myDB = ds;
    }
    ...
}

```

28.3.3 类级别的注入方式

为了使用类级别的注入，可以用`@Resource`修饰这个类，且必须设置`name`元素和`type`元素：

```

@Resource(name="myMessageQueue",
          type="javax.jms.ConnectionFactory")
public class SomeMessageBean {
    ...
}

```

`@Resources`注解用于为类级别的注入方式的多个`@Resource`注解分组。下面的代码展示了`@Resources`注解，它包含了两个`@Resource`声明。一个是有关Java消息服务的消息队列，另一个是有关JavaMail的会话：

```

@Resources({
    @Resource(name="myMessageQueue",
              type="javax.jms.ConnectionFactory"),
    @Resource(name="myMailSession",
              type="javax.mail.Session")
})
public class SomeMessageBean {
    ...
}

```

28.4 资源适配器和契约

资源适配器是一种Java EE组件，实现了针对某个特定EIS的Java EE连接器架构。常见的EIS包括ERP、大型联机事务处理系统以及数据库系统。如图28-1所示，资源适配器使得Java EE应用和EIS之间的通信更为方便。

资源适配器保存在RAR（Resource Adapter Archive）文件里，可以部署到任何Java EE服务器上，这一点很像Java EE应用。RAR文件可能会进一步打包到EAR文件里，当然也可以以单独文件的方式存在。

资源适配器类似于JDBC驱动，都提供标准的API。通过这些API，应用可以访问位于Java EE服务器之外的资源。对于资源适配器来说，其目标系统是EIS；对于JDBC驱动来说，其目标系统是DBMS。资源适配器和JDBC驱动一般不是由应用开发人员创建的。大多数情况下，这两种类型的软件是产品提供商（此外还销售工具、服务器或者集成软件）所提供的。

资源适配器以契约的方式，协调Java EE服务器和EIS之间的通信。应用契约定义了一组API，通过这些API，Java EE组件（如企业bean）可以访问EIS。这些API是组件操作EIS的唯一途径。

系统契约把资源适配器和Java EE服务器托管的重要服务联系起来。资源适配器自身和它的系统契约对于Java EE组件来说是透明的。

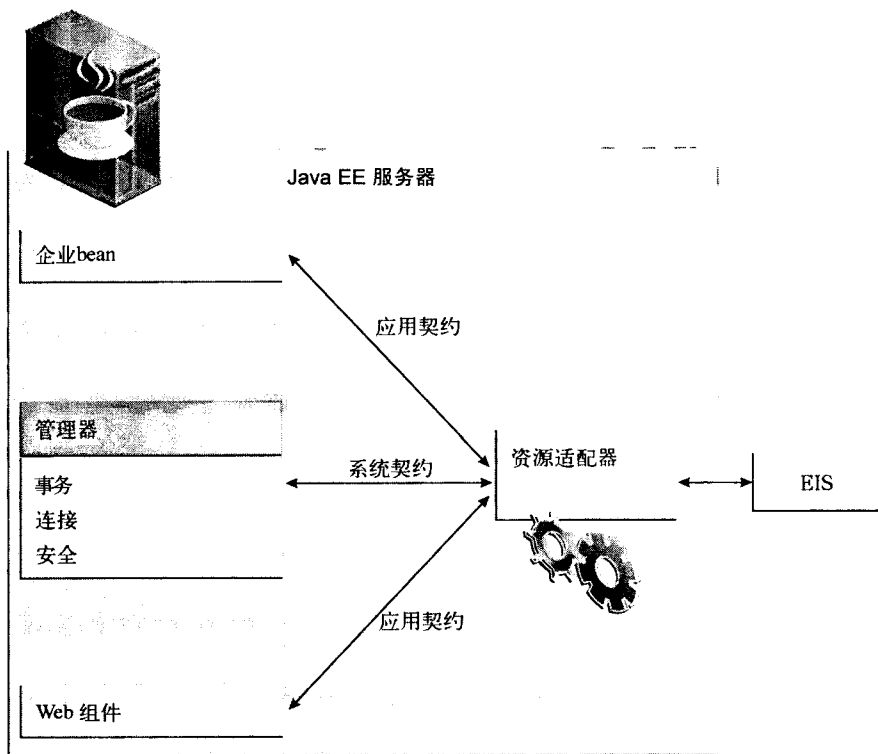


图28-1 资源适配器

28.4.1 管理契约

Java EE Connector Architecture (连接器架构) 定义了系统契约, 借助这个契约, 资源适配器有了生命周期和线程管理的能力。

1. 生命周期管理

Connector Architecture 设定了生命周期管理契约, 该契约使得应用服务器可以管理资源适配器的生命周期。这一契约提供了一种机制, 基于这种机制, 应用服务器可以在部署应用的时候或者自己启动的时候, 引导创建一个资源适配器实例。该契约还为应用服务器提供了一种为资源适配器实例发送通知的方式。当卸载应用, 或应用服务器正常关闭的时候, 资源适配器会收到消息。

2. 工作管理契约

Connector Architecture 的工作管理契约可以确保资源适配器以推荐的方式合理地使用线程。

这一契约还使得应用服务器可以管理资源适配器的线程。

资源适配器如果使用线程不当的话，会危及整个应用服务器的环境。例如，资源适配器可能创建过多的线程，或者没有正确地释放它所创建的线程。因为创建以及销毁线程是开销非常大的操作，所以线程处理不好会干扰应用服务的关闭，影响应用服务器的性能。

工作管理契约使得应用服务器可以使用线程池并重用线程，这一点和连接池很像。只要遵守这个契约，资源适配器无需自己管理线程。相反，资源适配器让应用服务器创建和提供所需的线程。当线程完成任务，资源适配器会把线程返还给应用服务器。应用服务器对线程的管理，要么是将其返回给线程池供以后复用，要么是销毁它。以这样的方式处理线程能提高应用服务器的性能，资源使用也会更高效。

除了把线程管理转交给应用服务器，Connector Architecture还为使用线程的资源适配器提供一种灵活的模型。

- 请求线程可以选择阻塞（暂停自身的执行）直到工作线程完成。
- 当请求线程等待工作线程的时候，请求线程可以阻塞住。在应用服务器提供了工作线程以后，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需要由线程完成的工作提交到一个队列里，线程稍后再执行这项工作。资源适配器可以在提交工作至队列后，继续自己的任务，无需关注线程何时执行这项工作。

对于后两种方式，提交线程和工作线程可以同时执行，也可以各自独立地执行。对于这两种方式，这一契约指定了监听器机制，以通知资源适配器线程已经完成了自己的任务。资源适配器还可以为线程指定执行的上下文，而工作管理契约控制执行线程的上下文。

28.4.2 通用工作上下文契约

借助应用服务器和资源适配器之间的工作管理契约，资源适配器可以将Work实例交给线程去执行一项任务，比如与EIS通信或者发送一个消息。

借助通用工作上下文契约，资源适配器可以控制Work实例运行的上下文。Work实例是由应用服务器的WorkManager管理的。通用工作上下文机制还使得应用服务器可以实现新消息的收发以及解析。它还资源适配器提供了丰富的上下文Work执行环境。与此同时，在一个托管的环境下，仍然维持对并发表行的控制。

通用工作上下文契约为事务上下文以及安全上下文建立了标准。

28.4.3 外向型和内向型契约

Connector Architecture定义了如下的外向型（outbound）契约，它们是应用服务器和EIS之间的系统级契约，使得应用服务器可以连接到外部的EIS。

- 连接管理契约支持连接池，这项技术可以提升应用的性能以及伸缩性。连接池对应用来说是透明的，它简化了获取EIS连接的过程。

- 事务管理契约扩展了连接管理契约，提供了对于管理本地事务和分布式事务的支持。本地事务局限在单个EIS系统范围之内，EIS资源管理器自己管理这样的事务，而分布式事务或者说全局事务可以跨越多个资源管理器。这种类型的事务需要通过外部的事务管理器来协调事务。一般来说，外部事务管理器打包在应用服务器里。事务管理器使用两阶段提交协议来管理跨越多个资源管理器或者多个EIS的事务。如果只有一个资源管理器参与了分布式事务，则使用单阶段提交优化。
- 安全管理契约为Java EE服务器和EIS之间的认证、授权以及安全通信提供了一种机制，以保护EIS的信息。

工作安全映射表将EIS的标识映射成为应用服务器所在域中的标识。

内向型（inbound）契约是Java EE服务器和EIS之间的系统级契约，允许外部EIS反向连接到应用，这是消息提供者的插件式契约和导入事务的契约。

28.5 元数据注解

Java EE Connector Architecture 1.6引入了一组注解，以最大程度地降低应用对于部署描述文件的依赖。

- 资源适配器的开发人员可以使用@Connector注解将某JavaBeans组件指定成一个具备资源适配器功能的JavaBeans组件。这个注解用来声明定义资源适配器能力的元数据。此外，开发人员可以提供一个实现ResourceAdapter接口的JavaBeans组件，如下面的例子所示：

```
@Connector(
    description = "Sample adapter using the JavaMail API",
    displayName = "InboundResourceAdapter",
    vendorName = "My Company, Inc.",
    eisType = "MAIL",
    version = "1.0"
)
public class ResourceAdapterImpl
    implements ResourceAdapter, java.io.Serializable {
    ...
    ...
}
```

- @ConnectionDefinition注解定义了一组与特定连接类型相关的连接接口和连接类，如下面的例子所示：

```
@ConnectionDefinition(
    connectionFactory =
        samples.mailra..api.JavaMailConnectionFactory.class,
    connectionFactoryImpl =
        samples.mailra.ra.outbound.JavaMailConnectionFactoryImpl.class,
    connection =
        samples.connectors.mailconnector.api.JavaMailConnection.class,
    connectionImpl =
        samples.mailra..ra.outbound.JavaMailConnectionImpl.class
)
public class ManagedConnectionFactoryImpl implements
    ManagedConnectionFactory, Serializable {
```

```

    ...
    ...
    @ConfigProperty(defaultValue = "UnknownHostName")
    public void setServerName(String serverName) {
        ...
    }
}

```

□ **@AdministratedObject**注解表明一个JavaBeans组件是一个受管理对象。

□ **@Activation**注解包含了与来自外部EIS实例的访问请求相关的配置信息，如下面的例子所示：

```

@Activation(
    messageListeners = {
        samples.mailra.api.JavaMailMessageListener.class
    }
)
public class ActivationSpecImpl
    implements javax.resource.spi.ActivationSpec,
               java.io.Serializable {
    ...
    @ConfigProperty()
    // serverName property value
    private String serverName = new String("");

    @ConfigProperty()
    // userName property value
    private String userName = new String("");

    @ConfigProperty()
    // password property value
    private String password = new String("");

    @ConfigProperty()
    // folderName property value
    private String folderName = new String("Inbox");

    // protocol property value
    // Normally imap or pop3
    @ConfigProperty(
        description = "Normally imap or pop3"
    )
    private String protocol = new String("imap");
    ...
    ...
}

```

□ **@ConfigProperty**注解可以用在JavaBeans组件上，以提供额外的配置信息，供部署人员以及资源适配器提供者使用。刚才的例子展示了一些**@ConfigProperty**注解的使用方法。

资源适配器规范允许开发人员以混合的方式开发资源适配器，即在应用中既可使用元数据注解，也可使用部署描述文件。应用装配人员或者部署人员可以使用部署描述文件覆写开发人员所指定的元数据注解。

资源适配器的部署描述文件为ra.xml。名为metadata-complete的属性定义了资源适配器模块的部署描述文件是否完整，以及对模块可用且与资源适配器打包在一起的类文件是否需要指定部署信息的注解做检查。

关于完整的注解以及由Java EE 6平台所引入的JavaBean组件的列表，可以参考Java EE Connector Architecture规范1.6。

28.6 公共客户端接口

本节讲述组件如何使用通用Connector Architecture的CCI (Common Client Interface, 公共客户端接口) API, 以及如何使用资源适配器来访问EIS里的数据。CCI API定义了一组接口和类, 其中方法允许客户端执行一些典型的数据访问操作。CCI接口和类列举如下。

- `ConnectionFactory`, 为应用组件提供了用于连接EIS的`Connection`实例。
- `Connection`, 代表对底层EIS的连接。
- `ConnectionSpec`, 当发起连接请求时, 为应用的组件提供一种方式, 以把特定于连接请求的属性传递到`ConnectionFactory`里。
- `Interaction`, 为应用的组件提供了一种调用EIS功能的方式, 比如执行数据库的存储过程。
- `InteractionSpec`, 保存了与EIS交互的应用组件的有关属性。
- `Record`, 这是一个超级接口, 可以用于不同类型的记录实例。记录实例可以是`MappedRecord`、`IndexedRecord`或者`ResultSet`实例。这几种类型都继承自`Record`接口。
- `RecordFactory`, 为应用组件提供`Record`实例。
- `IndexedRecord`, 表示一个已经排序的`Record`实例集合, 基于`java.util.List`接口。

使用了CCI与底层EIS交互的客户端或者应用组件, 在与EIS交互时, 按照规定的方式执行。组件必须使用`ConnectionFactory`建立到EIS资源管理器的连接。`Connection`对象代表到EIS的连接, 可用于后续与EIS的交互。

组件使用`Interaction`对象执行与EIS的交互, 比如访问某个表里的数据。应用组件使用`InteractionSpec`对象来定义`Interaction`对象。当应用组件从EIS (如数据库表) 读取数据, 或者写数据到这些表的时候, 需要使用某个特定类型的`Record`实例, 比如`MappedRecord`、`IndexedRecord`或者`ResultSet`实例。

需要指出, 依赖于CCI资源适配器的客户端应用, 和任何其他使用企业bean方法的Java EE客户端很相似。

28.7 参考资源

有关资源以及注解的更多信息, 可以参阅如下内容。

- Java EE 6平台规范 (JSR 316):
<http://jcp.org/en/jsr/detail?id=316>
- Java EE Connector Architecture 1.6规范:
<http://jcp.org/en/jsr/detail?id=322>
- EJB 3.1规范:
<http://jcp.org/en/jsr/detail?id=318>
- Java平台的常见注解:
<http://www.jcp.org/en/jsr/detail?id=250>