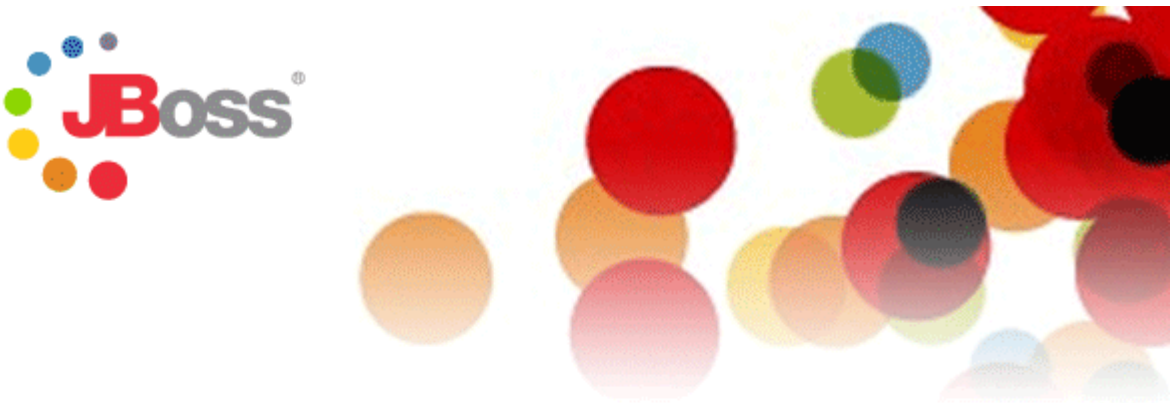


最新企业应用开发核心技术

EJB3.0 实例教程

《EJB3.0入门经典》的精简版

. Struts+EJB3.0实战 JSF+EJB3.0实战

. Struts1/2+Spring2.5+EJB3.0整合开发

. 深入了解事务管理、JMS、WebService开发

前言

《EJB3.0 实例教程》自 2006 年 7 月发布以来，已经成为相关开发人员学习 EJB3.0 的第一手中文资料，经过和广大读者的答疑和交流，《EJB3.0 实例教程》得以为不断补充和完善，应广大读者的热情呼声，本书得以整理出版，并改名为《EJB3.0 入门经典》。本书最大的特点是：通俗，易懂，非常实用，且包含了众多的新技术。书中的每一个例子都是经过作者精心构思，目得是达到最好的学习效果。在本书你基本找不到晦涩难懂的原理，因为这些原理都已转化成几句更具说明意义的代码。

《EJB3.0 入门经典》一书的标价为 59.8 元，考虑到部分家境并不富裕，强烈渴望学习 EJB3.0 的在校学生，征得出版商的同意。作者把《EJB3.0 入门经典》作了精简，仍以《EJB3.0 实例教程》电子版发布。尽管是精简版，但对于入门 EJB3.0 已经足够了，建议这部分学生看完《EJB3.0 实例教程》后，到书店看《EJB3.0 入门经典》。当然对于有经济条件的读者，如果你从本书学到了知识，那么购买《EJB3.0 入门经典》就是对本书的认可。

本教程适合人群

本书适合 Java 程序员、项目经理和系统构架师。学习本书，你不需要 EJB2.x 知识，如果具备一些 web 和 jdbc 知识，学习效率会更快些。学习本书，初级 java 程序员时间应在 1 个月左右，有 1-2 年开发经验的程序员应在 15 天之内，具有 3 年以上开发经验的程序员只需 5 天。

作者介绍

黎活明，Java EE 高级架构师与网站运营总监，毕业于中国农业大学，有着丰富的 B/S 系统开发与网站运营经验，主持或参与了像《一号通》、《固话彩铃》、《移动办公 OA》、《统一信息发送机》、《MSN 业务支撑平台》和《国内与国际机票预定系统》等项目，成功运营过中国农业网/游易网等电子商务网站。

2007 年初作者开始研发网上商城系统，主要因为在众多的商城产品中找不到适合大中型企业使用的解决方案。这些产品没有考虑到企业今后可能会围绕交易系统不断增加各种业务系统的情况，必然会遇到像游易/当当/卓越前期存在的系统构架问题，这些问题给企业带来的是：系统扩展难，重复性开发严重，有时一个业务变更往往需要对多个业务系统（如：网站预订系统、电话预订系统、内部办公系统等）进行修改，对于大一点的系统升级，更是担心因改动数据库结构而影响到其他业务系统。更可怕的是，开发人员不得不在一个处于病态的系统上升级、维护和加入新的业务系统，导致开发及维护成本居高不下。另外网上出售的商城系统只是满足了一般的产品展示及订购功能，缺少对网站运营的支持。作者主持研发的商城系统于 2007 年底完成，已经被运用到巴巴运动网和一个对外贸易的商务网站。如果您公司想开展网上商城业务，需要这样一个平台，请与作者联系。

本书的官方网站及 MSN 群

官方网站：http://www.foshanshop.net。
MSN 群账号：group22723@xiaoi.com ，加入该群即可与大家一起交流 ejb3.x 的学习经验，了解 ejb 技术的最新发展情况等。
EJB 培训：作者作为传智播客老师，会不定期到传智播客讲授 EJB3.0 的使用，如果您想参加相关面授培训，请与传智播客公司联系，网址：http://www.itcast.cn。对于企业培训也可以直接与作者联系：lihuoming@sohu.com。

第一章　EJB 知识与运行环境配置

1.1 什么是 Enterprice JavaBeans(EJB)

Enterprice JavaBeans 是一个用于分布式业务应用的标准服务端组件模型。采用 Enterprice JavaBeans 架构编写的应用是可伸的、事务性的、多用户安全的。可以一次编写这些应用，然后部署在任何支持 Enterprice JavaBeans 规范的服务器平台，如 jboss、weblogic 等。

Enterprise JavaBean（EJB）定义了三种企业 Bean，分别是会话 Bean（Session Bean），实体 Bean（Entity Bean）和消息驱动 Bean（MessageDriven Bean）。

会话 Bean

会话 Bean 用于实现业务逻辑，它分为有状态 bean 和无状态 bean。每当客户端发出 EJB 调用请求时，容器就会选择一个 Session Bean 来为客户端服务。会话 Bean 可以直接访问数据库，但更多时候，它是通过实体 Bean 实现数据访问。

实体 Bean：

从名字上我们就能猜到，实体 bean 代表真实物体的数据。在 EJB3.0 中，实体 bean 仅作为普通 Java 对象来使用，它负责跟数据库表进行对象与关系映射（O/R Mapping）。

消息驱动 Bean(MDB)：

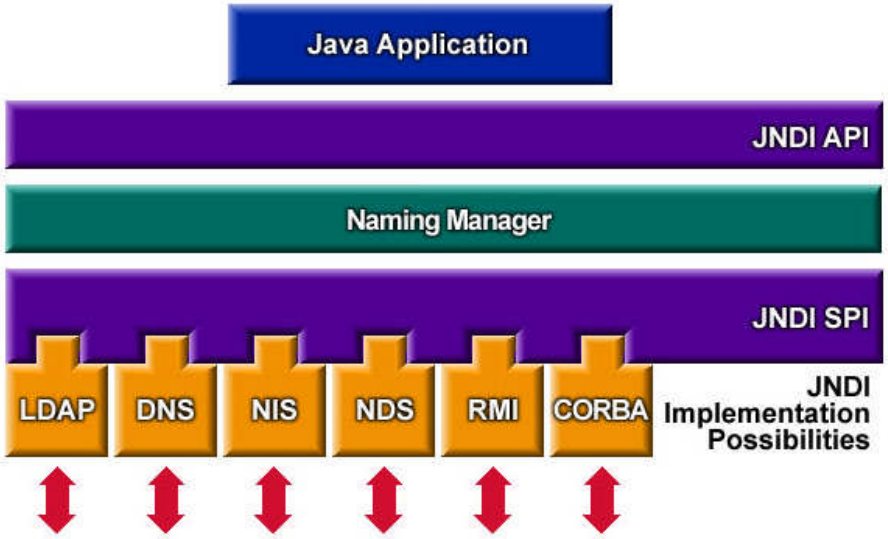
MDB 是设计用来专门处理基于消息请求的组件。它能够收发异步 JMS 消息，并能够轻易地与其他 EJB 交互。它特别适合用于当一个业务执行的时间很长，而执行结果无需实时向用户反馈的这样一个场合。

1.2 EJB 的运行环境

EJB 需要运行在 EJB 容器，每个 JavaEE 应用服务器都含有 EJB 容器和 Web 容器，所以既可以运行 EJB，也可以运行 Web 应用。目前支持 EJB3.0 的应用服务器有 Jboss（4.2.x 以上版本）、Glassfish、Weblogic（10 以上版本）、Sun Application Server（9.0 以上版本）、Oracle Application Server（10g 以上版本）和我们国内的 apusic 应用服务器。本书将介绍 Jboss 和 Weblogic，前者是使用者最多的开源应用服务器，后者是市场占有率最高的商业应用服务器。
注意：Tomcat 目前只是 Web 容器，它不能运行 EJB 应用。

1.3 什么是 JNDI

JNDI 是自 JDK1.3 版本开始就绑定的标准 Java API。它为各种现有的命名和目录服务提供了通用接口：DNS、LDAP、活动目录（Active Directory）、RMI 注册器、COS 注册器、NIS 及文件系统。在结构上，JNDI 由两部分组成：客户 API 和服务提供商接口（Service Provider Intergace，SPI），应用程序通过客户 API 访问命名和目录服务；服务提供商接口用于供厂商创建命名和目录服务的 JNDI 实现。下面是 JNDI 的结构图：

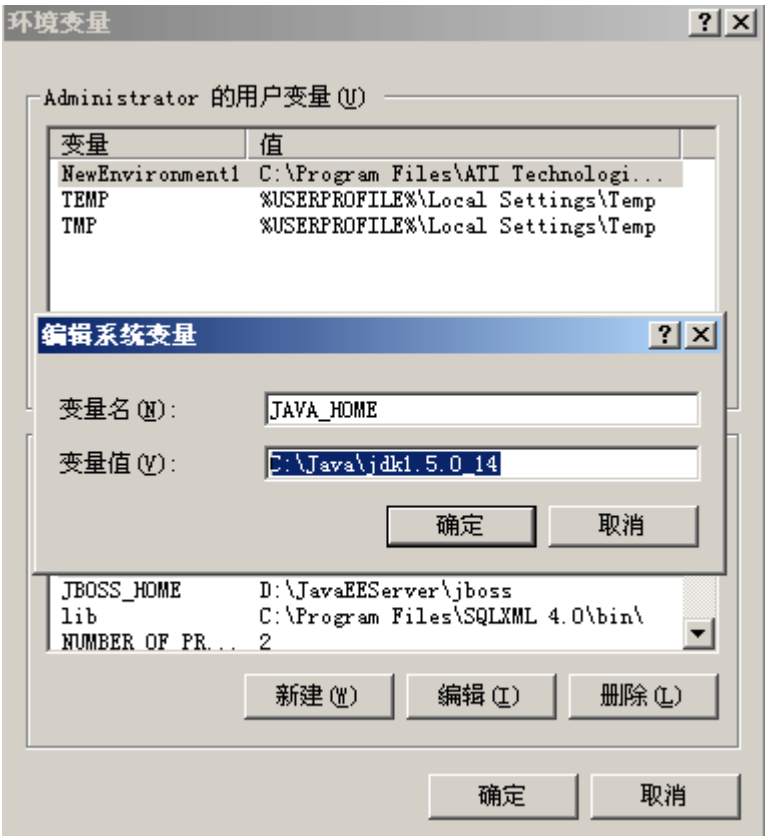


对于 EJB 开发者来说，我们只需要知道使用客户 API 如何访问命名和目录服务即可，而不需要知道 JNDI SPI 的使用，因为我们不需要使用 JNDI SPI 开发 JNDI 实现产品，这就好比通过 JDBC 访问数据库，我们只需要知道使用 JDBC API 如何访问数据库，而不需要知道数据库的 JDBC 驱动如何实现。使用客户 API 访问 EJB3.0，我们需要编写的 JNDI 代码不过几句，所以大家不要被吓倒了。

命名服务用于将名称和对象联系起来，使得我们可以用名称访问对象。例如，当你在 web 浏览器输入 URL：http://www.foshanshop.net 时，DNS(Domain Name System，域名系统)将这个域名转换成 IP 地址。
目录服务是命名服务的自然扩展，在这种服务里，对象不但有名称，还有属性。与命名服务的关键差别是：目录服务中对象可以有属性（例如，用户有 email 地址），而命名服务中对象没有属性。

1.4 下载与安装 JDK

进入 http://java.sun.com/javase/downloads/index_jdk5.jsp 下载 JDK。在页面中找到 JDK 5.0 Update 14（版本在不断更新中，有可能大于 14），点击右边的 Download，注意中间有 Accept 和 Decline 两选项，点选 Accept。在 Windows Platform 一栏找到 Windows Offline Installation, Multi-language 这个链接，点击下载。（注：本书光盘“软件”文件夹中带有 JDK 5.0）
按照安装向导提示安装，安装路径选择 C:\Java\jdk1.5.0_14。Jdk 安装完后，接着问你是否安装 jre，也一起安装上。右键点击“我的电脑”->“属性”->“高级”->“环境变量”，在“系统变量”里添加 JAVA_HOME 变量，值为 JDK 的安装路径，如：C:\ Java\jdk1.5.0_14。



在“系统变量”里再添加 CLASSPATH 变量，值为：**.;**%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;
在系统变量栏找到变量名为 Path 的选项，点“编辑”在变量值的末尾添加;%JAVA_HOME%\bin;

1.5 下载与安装 Eclipse

下载地址：http://www.eclipse.org/downloads
在页面上选择下载：Eclipse IDE for Java EE Developers
下载后直接解压缩即可完成安装。

注：本书配套光盘“软件”文件夹中已经带有 eclipse 安装包。

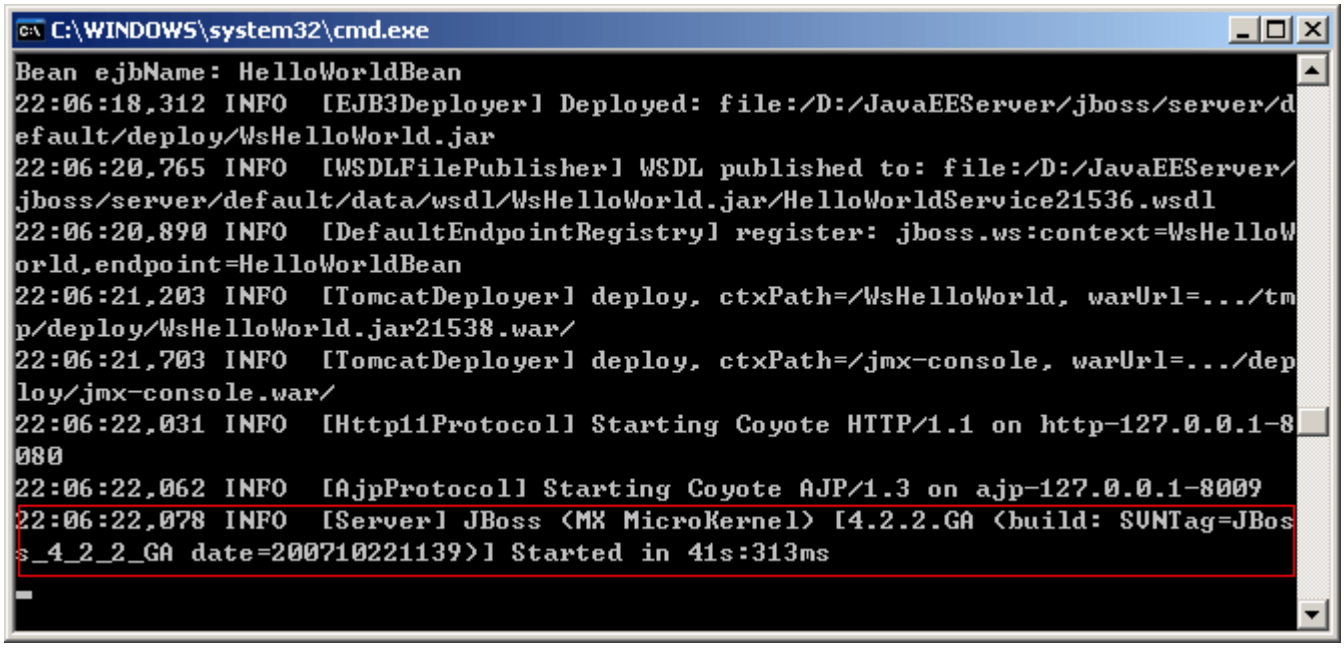
1.6 下载与安装 jboss

进入 http://labs.jboss.com/jbossas/downloads/下载页面，选择 jboss4.2.2.GA 文件下载（大小为 92MB）。下载后直接解压缩文件即可完成安装，为了避免应用出现莫名的错误，解压缩的路径最好不要带有空格，如“Program Files”。
注：本书配套光盘“软件”文件夹中已经带有 Jboss4.2.2.GA 安装包。

安装完后请右键点击“我的电脑”->“属性”->“高级”->“环境变量”，在“系统变量”里添加 JBOSS_HOME 变量，值为 Jboss 的安装路径，如：D:\JavaEEServer\jboss。

在系统变量一栏找到变量名为 Path 的选项，点“编辑”在变量值的末尾添加：;%JBOSS_HOME%\bin;

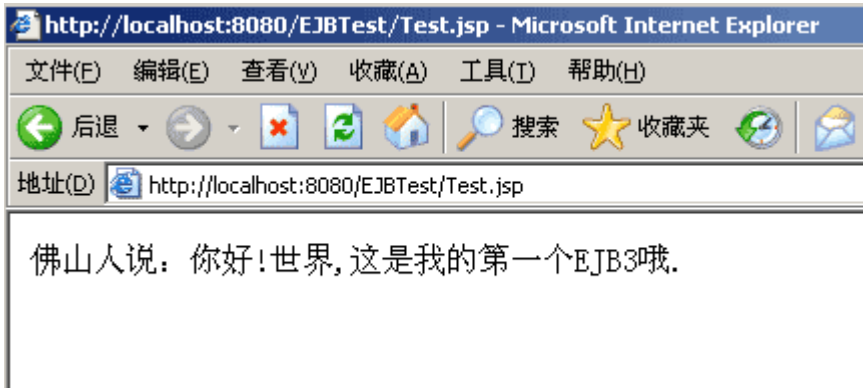
现在验证 Jboss 安装是否成功， 进入[jboss 安装目录]\bin 目录，双击 run.bat 启动 jboss。观察控制台有没有 Java 例外抛出，如果没有例外并看到下图，恭喜你，安装成功了。



你可以输入 http://localhost:8080 进入 Jboss 的欢迎主页。在 JBoss Management 栏点击“JMX Console”进入 Jboss 的管理平台。如果需要输入用户名及密码，默认的用户名及密码都是 admin。
如果 jboss 启动出错，应检查打印在 Jboss 控制台的 JDK 版本是否 5.0 以上，jboss 所用的端口是否被占用（如 1099, 8009 ,8080 等端口）。你可以使用本书提供的端口查看器 ActivePort.exe（在光盘“软件”文件夹下）查看端口使用情况，如果端口被占用就关闭占用此端口的进程。
如果启动时出现这样的错误：“findstr 不是内部或外部命令，也不是可运行的程序或批处理文件”。那么应该在系统变量 Path 中追加“**%SystemRoot%\system32;%SystemRoot%;**”。
最后的办法是重装机器上的 JDK，祝你好运。

1.7 运行第一个 EJB3 例子

Jboss 安装成功后，得来一个真家伙试试。在本书配套光盘的 HelloWorld 文件夹下，找到 HelloWorld.jar，把该文件拷贝到“jboss 安装目录/server/default/deploy/”目录下，jboss 会对 HelloWorld 进行热部署。接下来继续把 EJBTest 文件夹下的 EJBTest.war 拷贝到“jboss 安装目录/server/default/deploy/”。
在浏览器上输入：http://localhost:8080/EJBTest/Test.jsp（注意大小写）。将会看见下图所示。



1.8 熟悉 JBoss 的目录结构

安装 JBoss 会创建下列目录结构：

目录	描述
bin	启动和关闭 JBoss 的脚本
client	客户端与 JBoss 通信所需的 Java 库（JAR）
docs	配置文件的例子（数据库配置等）
docs/dtd	在 JBoss 中使用的各种 XML 文件的 DTD。
lib	JBoss 启动时使用到的 JAR，这些库为所有 JBoss 配置所共享。（不要把你的库放在这里）
server	各种 JBoss 配置。每个配置必须放在不同的子目录。子目录的名字表示配置的名字。JBoss 包含 3 个默认的配置：minimal，default 和 all。
server/all	JBoss 的完全配置，启动所有服务，包括集群和 IIOP。
server/minimal	这是启动 JBoss 服务器所要求的最低配置。minimal 配置将启动日志服务、JNDI 服务器以及 URL 部署扫描器，以找到待部署的（新）应用。对于那些不需要使用任何其他 J2EE 技术，而只是使用自定义服务的场合而言，则这种配置最适合。它仅仅是服务器，而不包含 Web 容器、不提供 EJB 和 JMS 支持。

server/default	默认配置，它含有大部分 J2EE 应用所需的标准服务。但是，它不含有 JAXR 服务、IIOP 服务、或者其他任何群集服务。如果在 JBoss 命令行中没有指定配置名称，则默认使用此配置。(本教程就采用此配置)
server/default/conf	JBoss 的配置文件。如：log4j.xml 是 Log4j 日志配置文件，login-config.xml 是 Jboss 安全配置文件，jboss-service.xml 配置在 jboss 启动时开启的 Jboss 服务（像类加载器，JNDI，部署工具等），jbossmq-state.xml 是 JbossMQ（JMS 实现）的用户配置文件。
server/default /deploy	JBoss 的热部署目录。任何位于此目录下的文件或目录都会被自动部署。如：EJB、WAR、EAR，甚至服务。
server/default/lib	一些 JAR，JBoss 启动 default 配置时会加载它们。
server/default/log	日志信息将存储到该目录。JBoss 使用 Jakarta Log4j 包作为其日志功能。同时，用户可以在应用中直接使用 Log4j 日志记录功能。
server/default/data	这一目录存储持久化数据，即使服务器发生重启其中的数据也不会丢失。许多 JBoss 服务将数据存储在这里，比如 Hypersonic 数据库实例。
server/default/tmp	供部署器临时存储未打包应用使用，也可以作为其他用途。
server/default/work	供 Tomcat 编译 JSP 使用。

其中，log、data、tmp、work 目录是 JBoss 创建的。如果用户没有启动过 JBoss 服务器，则这些目录不会被创建。

1.9 在 JBoss 部署应用

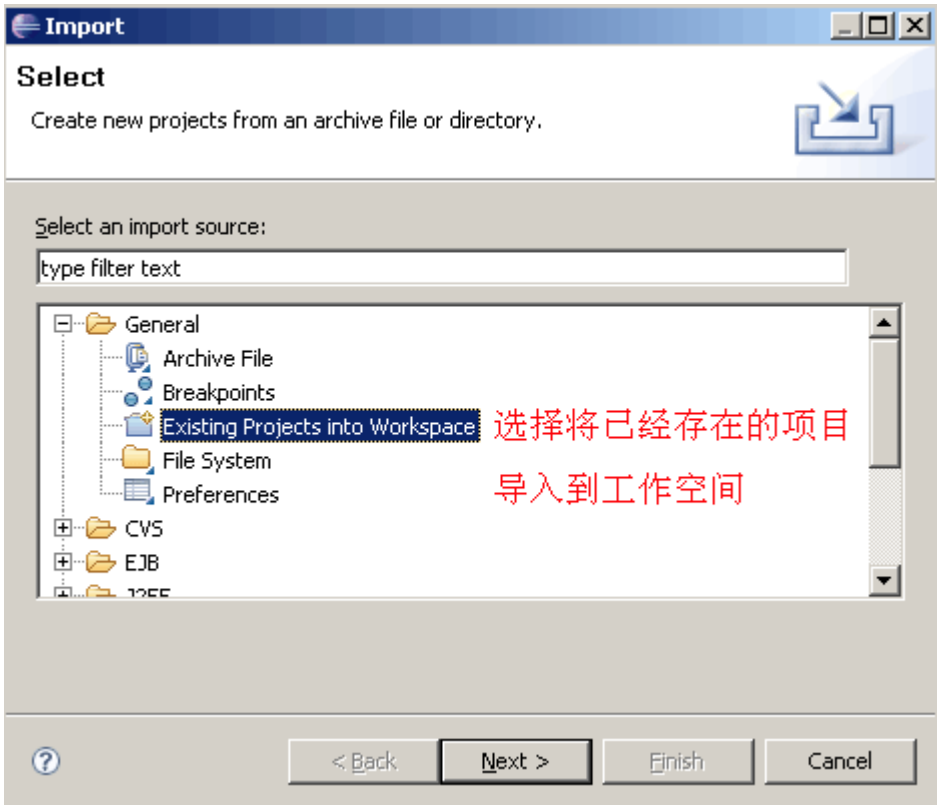
在 JBoss 部署应用的过程非常简单、直接。在每一个配置中，Jboss 会不断扫描一个名为[jboss 安装目录]/server/config-name/deploy 的特定目录，查看是否有任何更新，此目录一般被称为“部署目录”。
你可以把下列文件拷贝到部署目录中：

- 任何 Java 库（其中的类将被自动添加到 JBoss 的 classpath 中）
- EJB-JAR
- WAR (Web Appliction aRrchive)
- EAR (Enterprise Application aRchive)
- 包含 JBoss MBean 定义的 XML 文件
- 以.jar、.war 或者.ear 结尾的目录，分别包含了 EJB-JAR、WAR 或者 EAR 的解压缩内容。

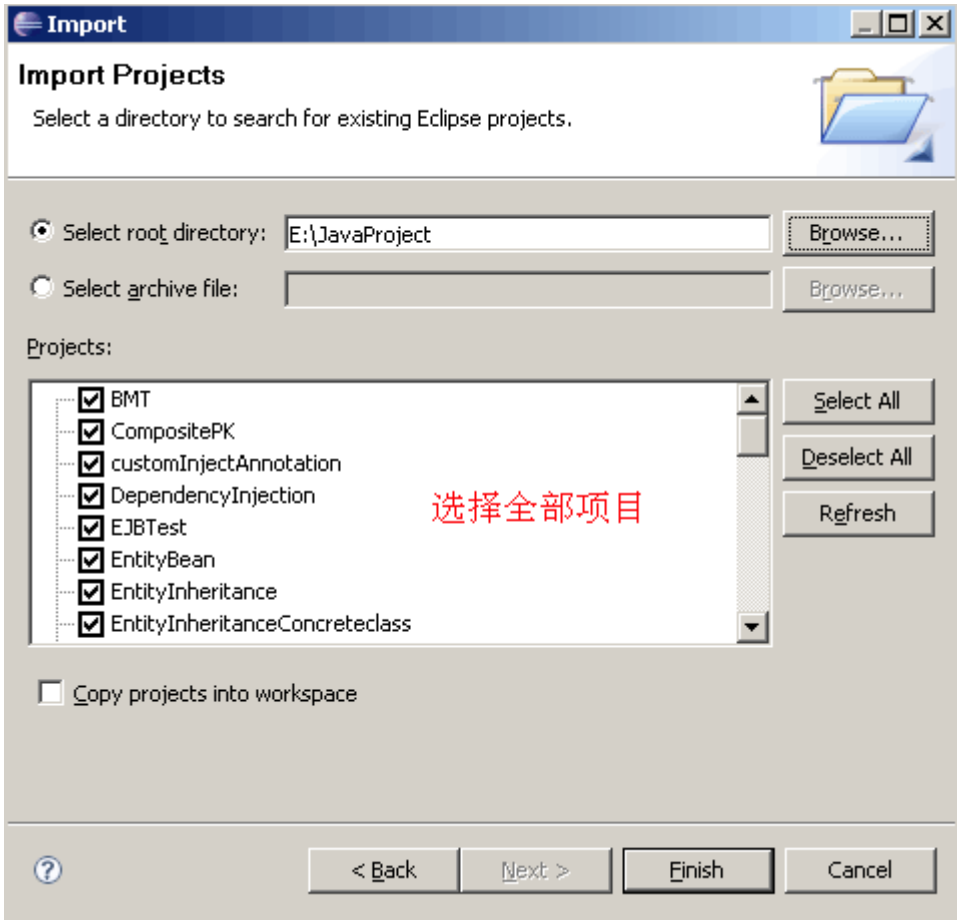
要重新部署上述文件（JAR、WAR、EAR、XML 等），只需用更新后的版本覆盖即可。Jboss 将通过对比时间戳来检测变化，卸载旧文件，部署相应的新文件。如果要重新部署目录，只需更新其时间戳即可。从部署目录中删除文件会导致相应文件从部署中卸载。

1.10 如何恢复本书配套例子的开发环境

如果你尚未安装 Eclipse，请先安装。然后打开本书配套光盘的“sourcecode”文件夹，把文件夹下的所有项目拷贝到工作区，如：E:\JavaProject，去掉文件的只读属性。在 Eclipse 上点击“File”->“import”，出现下面窗口：

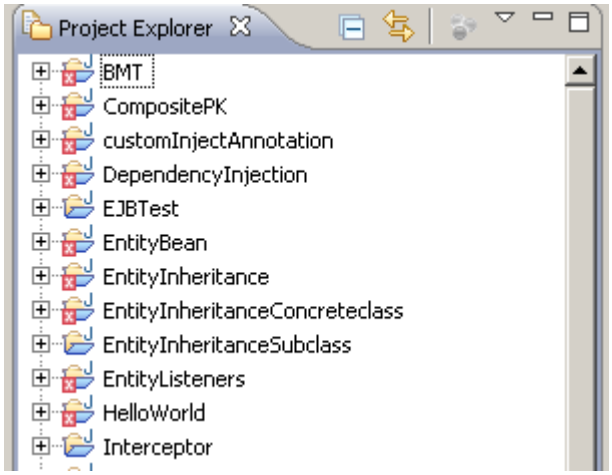


在上面窗口中展开“General”文件夹，选择“Existing Projects into Workspace”，点击“Next”，出现下面窗口：



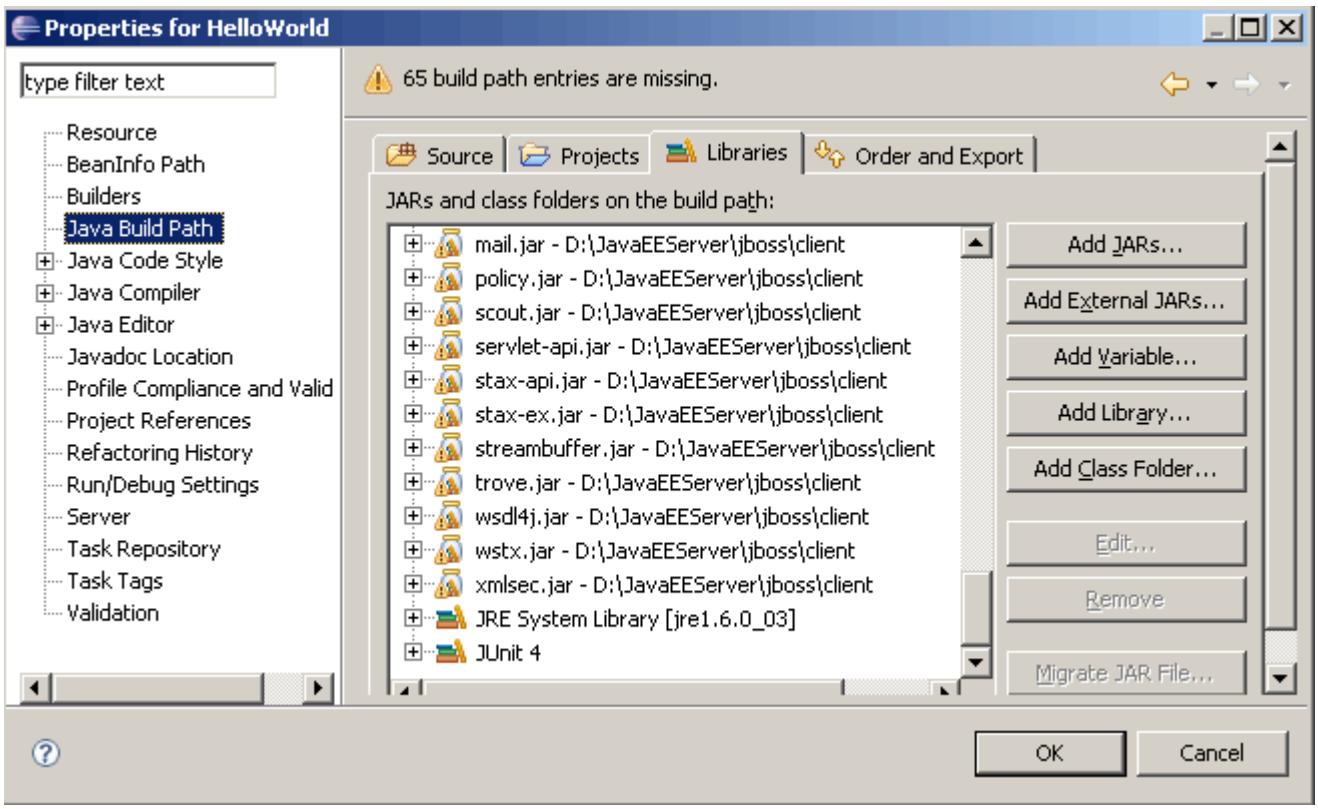
在上面窗口中定位到项目所在目录，Eclipse 能检测到目录下的所有项目，全选所有项目，然后点击“Finish”，所

选项目将出现在 Eclipse 的项目面板中，如下图：



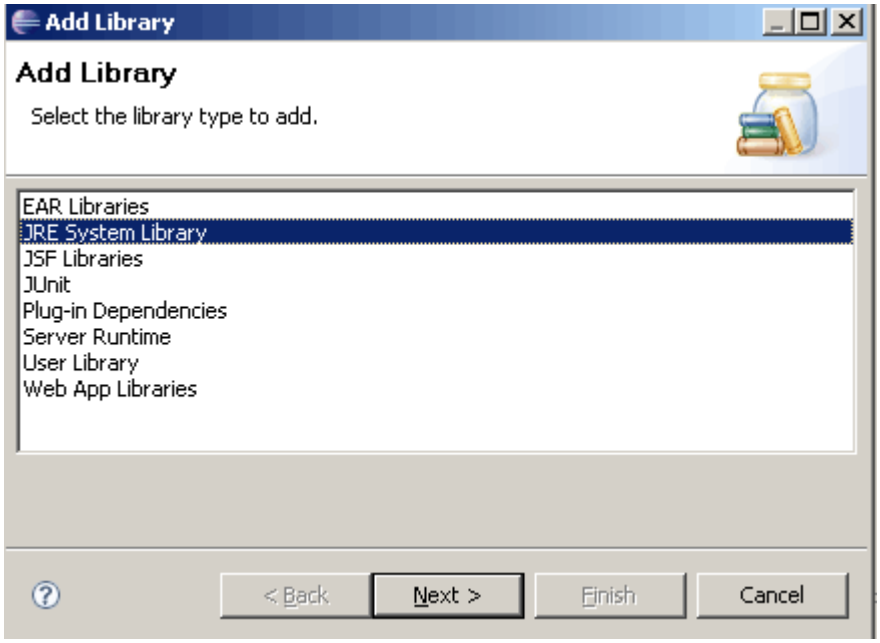
在上图，我们看到，很多项目都打了红叉，这是因为项目使用到的 JRE、Junit 和 Jave 库文件路径不正确。只要我们把它们删除，并重新添加就可以解决。项目使用到的 JAR 可以在[Jboss 安装目录]\client 下得到，个别项目使用到了本书提供的 JAR，这些 JAR 可以在光盘的 sourcecode\lib 目录下得到。下面我们以 HelloWorld 项目为例，介绍如何更正这些错误：

在“HelloWorld” 项目上点击鼠标右键，在属性菜单中点击“Properties”，在出现的窗口中点击框架左边的“Java Build Path”， 点击框架右边的“Libraries”，如下图：

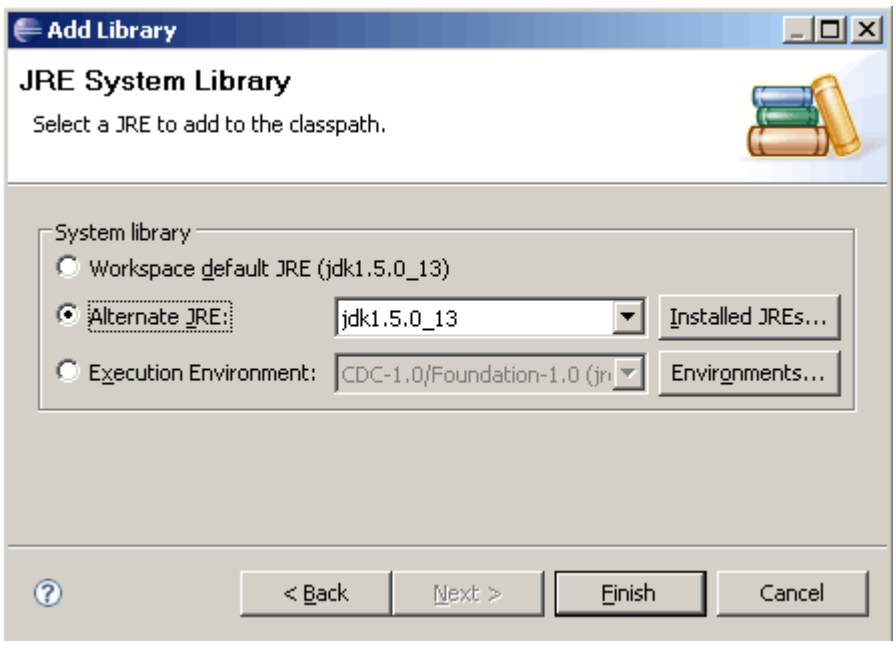


你需要把上图出现的 Java 库、JRE System Library 和 Junit4 全部删除，然后重新添加。首先添加“JRE System Library”，操作步骤如下：

点击“Add Library”，出现下面窗口：

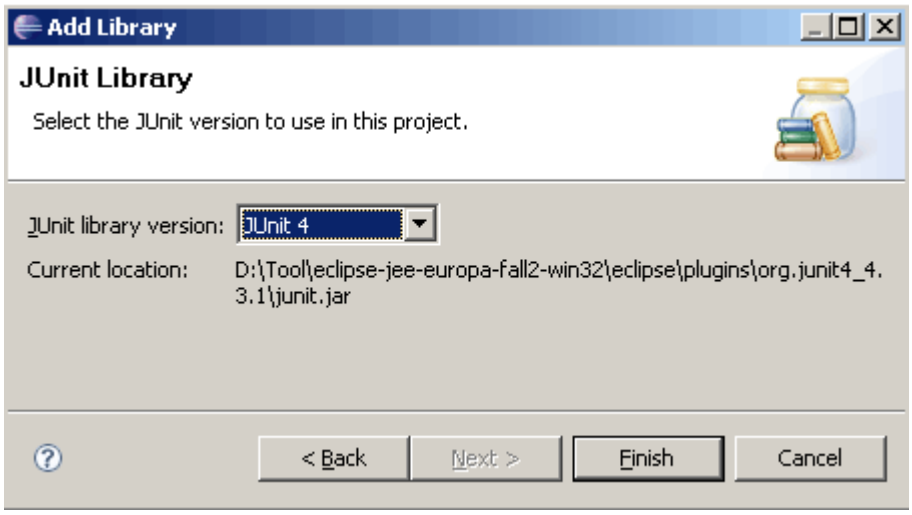


在上面窗口，选择“JRE System Library”，点击“Next”，出现下面窗口：



在上面窗口，选择你要使用的 JRE，点击“Finish”完成 JRE System Library 的添加。窗口回到了“Libraries”界面，接着添加 Jboss 客户端 JAR，操作如下：
点击“Add External JARs”，在出现的窗口中定位到[Jboss 安装目录]\client 目录下，然后 Ctrl+A 全选所有 JAR 文件。点击“打开”便完成了 Jboss 客户端 JAR 的添加。

如果项目中使用了 Junit，你还需要添加 Junit 库，操作如下：
点击“Add Library”，在出现的窗口中选择“Junit”，点击“Next”，出现下面窗口：

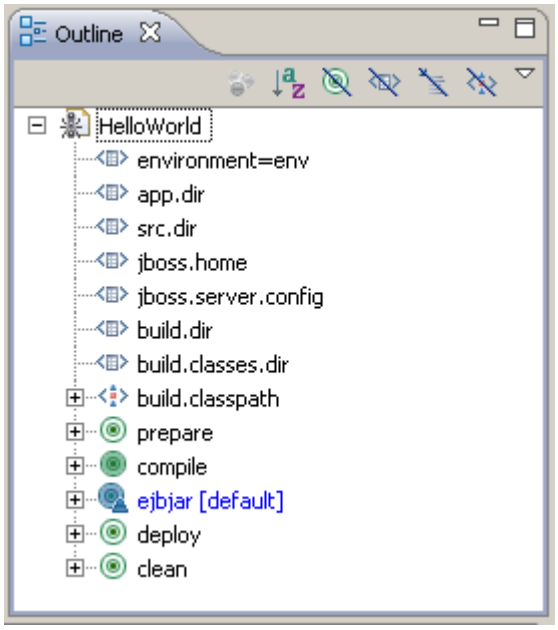


在上面窗口中，选择“Junit4”，点击“Finish”完成 Junit 库的添加。
点击“OK”回到项目面板，此时，“HelloWorld” 项目的红叉已经不见了。你可以按照上面方法更正其它项目。

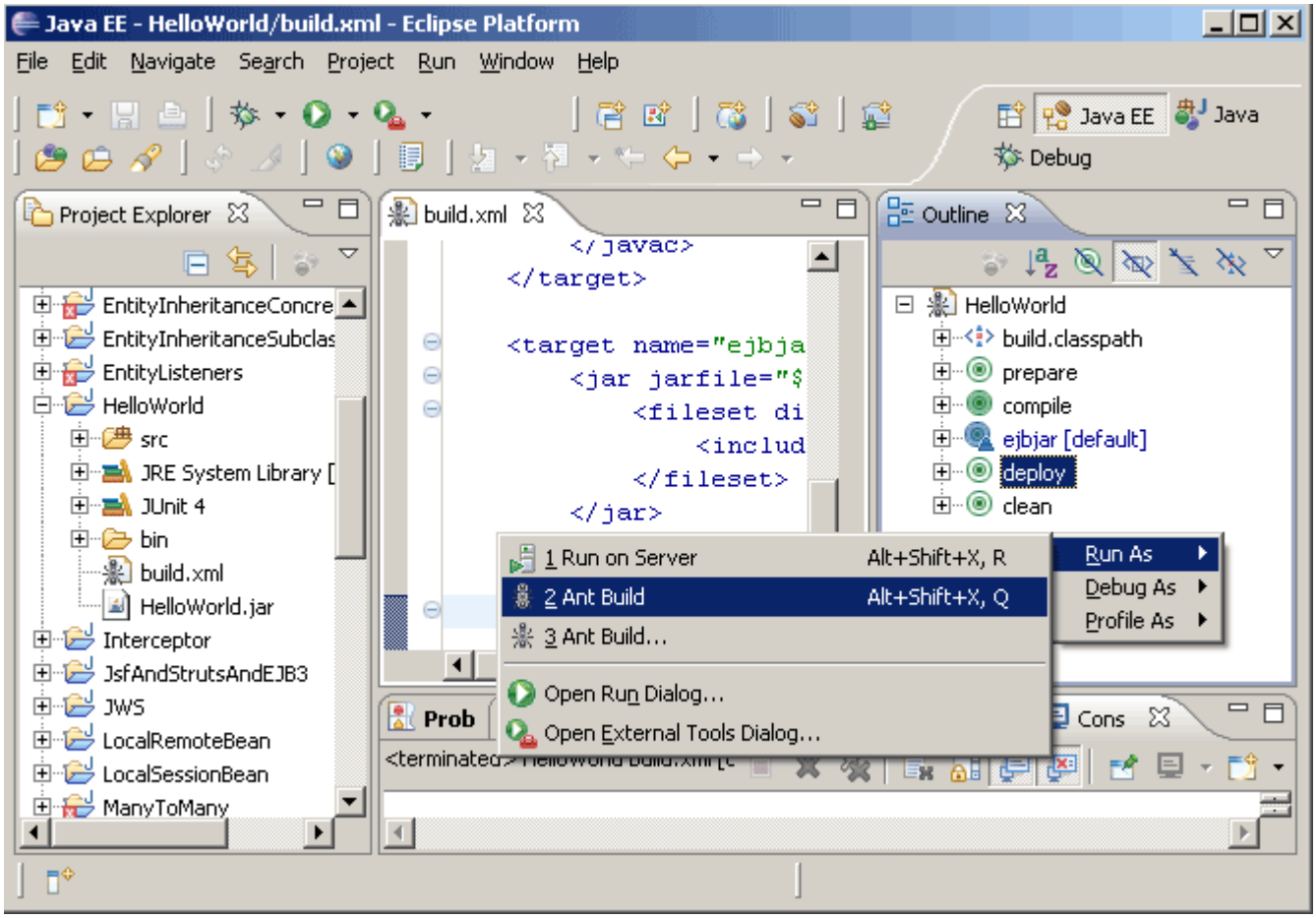
注意： weblogic 的例子需要用到 JRE1.6，使用到的 JAR 分别是：光盘\sourcecode\lib\javaee\javaee.jar 和[weblogic 安装目录]\server\lib\weblogic.jar。你不需要放入 Jboss 的客户端 JAR。

执行项目中的 Ant 任务

每个项目下都有一个 build.xml 文件，该文件是 Ant 的配置文件，打开该文件，在“Outline”视图中显示了 Ant 的每一项任务（前面带圆圈图标），如下图：

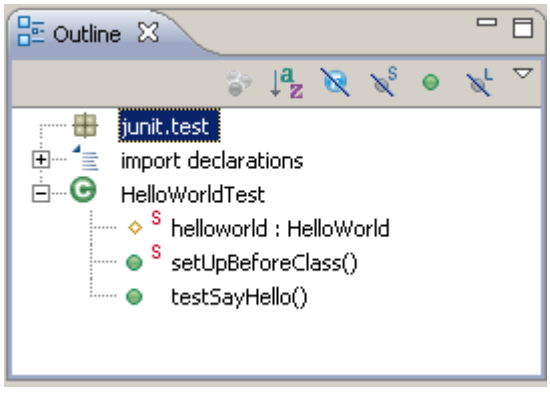


如果你的 Eclipse 没有 “Outline”视图，你可以点击“Window”-“Show View”-“ Outline”。
在你要执行的任务上点击鼠标右键，在出现的属性菜单中点击“Run As”-“Ant Build”运行该任务，如下图：

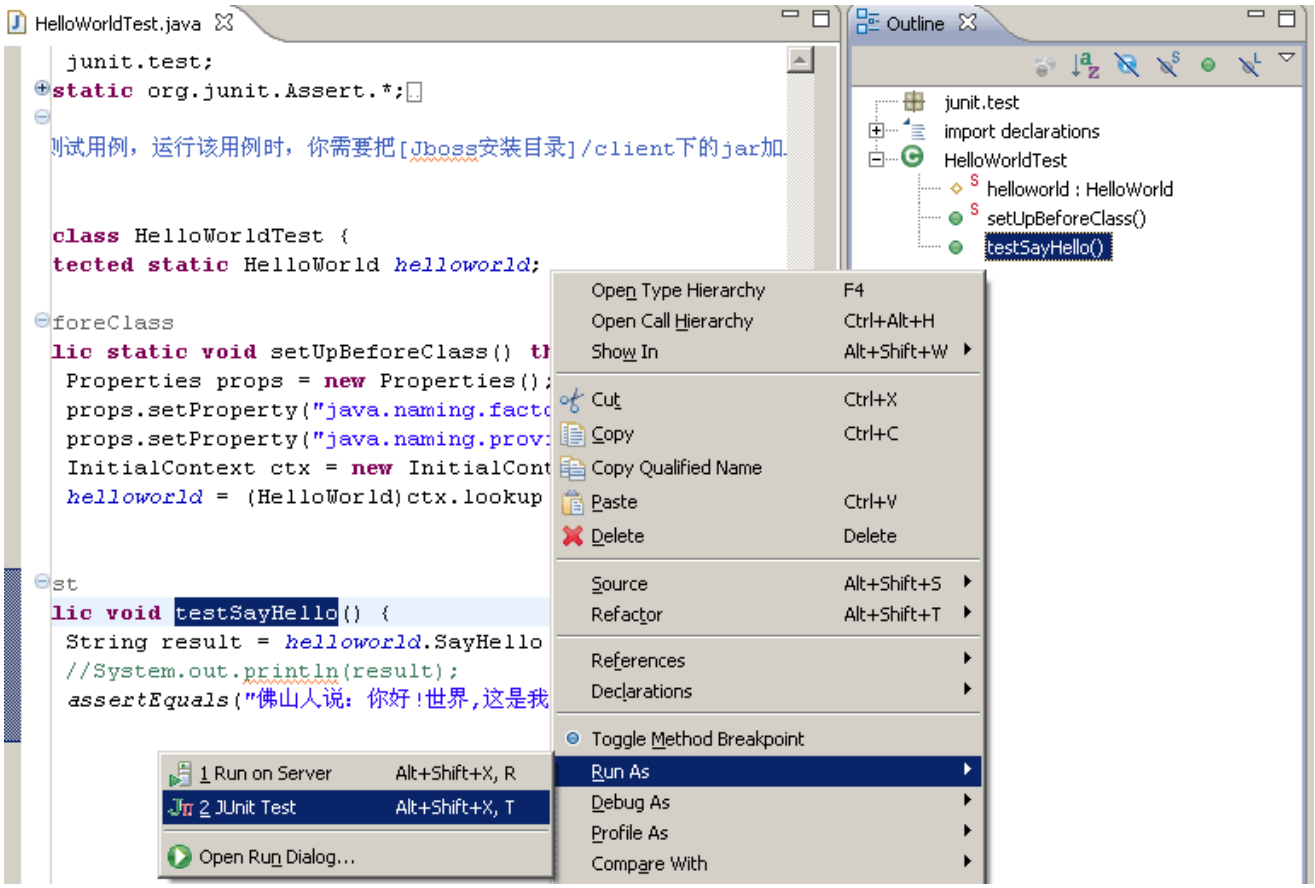


运行单元测试用例

有些项目带有单元测试用例，文件在以“junit”开头的包下。打开这些单元测试用例，在“Outline”视图中显示了单元测试的每一个测试方法，如下图：



你可以在需要测试的方法上面点击鼠标右键，在出现的属性菜单中点击“Run As”-“Junit Test”，如下图：



第二章　会话 Bean(Session Bean)

Session Bean 是实现业务逻辑的地方。简单地说，像我们要实现两数相加或是从数据库中读取数据，都是通过 Session Bean 来实现。根据是否可以维护会话状态，Session Bean 分为有状态 bean 和无状态 bean。有状态 bean 可以维护会话状态，无状态 bean 不维护会话状态。要维护会话状态，意味着 EJB 容器要为每个用户创建一个 bean 实例，并通过该实例保存着与用户的会话状态。不维护会话状态，意味着一个 bean 实例不需要保存与某个用户的会话状态，这时一个 bean 实例可以为多个用户服务。

要开发一个 Session Bean，我们需要定义接口和 Bean class。其中接口分为远程(remote)和本地(local)接口。在 EJB3.0 中，不要求你同时实现 remote 和 local 接口，但实现两者是比较好的做法。

- 远程接口（remote interface）：定义了 session bean 的业务方法，这些方法可以被来自 EJB 容器之外的应用访问到。
- 本地接口（local interface）：同样定义了 session bean 的业务方法，这些方法可以被同处于 EJB 容器内的其它应用使用。因为 local 接口允许 bean 之间直接通过内存交互，没有分布式对象协议的开销，从而改善了性能。
- Bean 类(bean class)：bean class 包含了业务逻辑，它必须具备一个远程或本地接口。在 Bean 类，我们应该实现接口的业务方法，尽管这并不是必须的，但我们没理由不这样做。

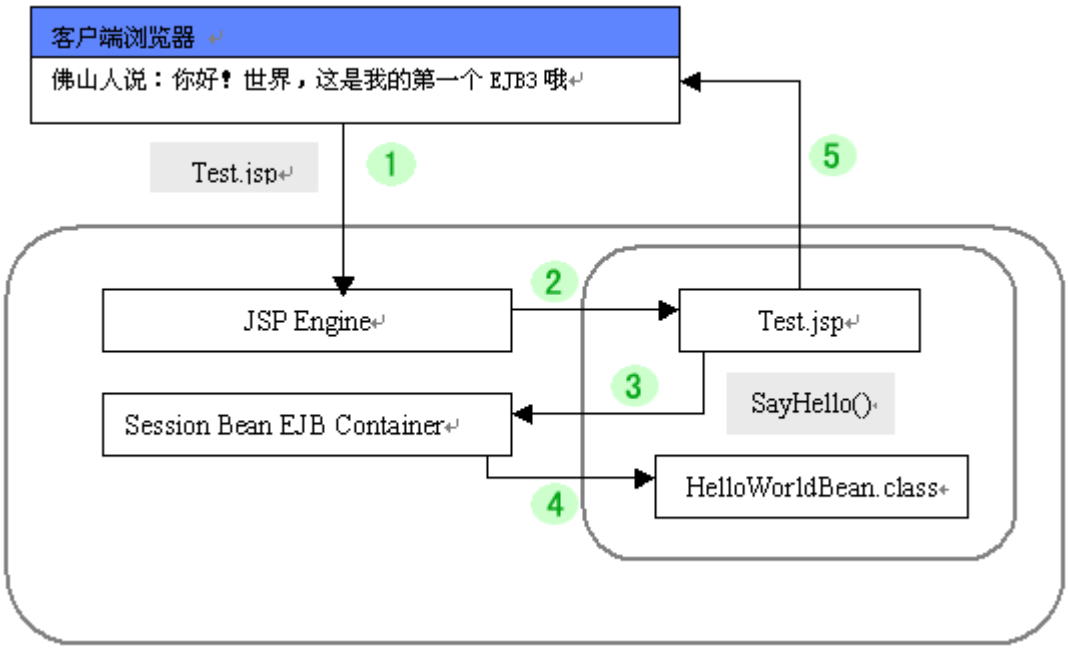
2.1 Stateless Session Beans（无状态 bean）开发

由于无状态会话 Bean 不维护会话状态，意味着一个 bean 实例可以为多个用户服务。因此 EJB 容器使用实例池化技术管理无状态会话 Bean。简单的说就是：当无状态会话 Bean 部署到应用服务器时，EJB 容器会为它预先创建一些 bean 实例放在对象池。当有用户访问 EJB 方法时，EJB 容器会从对象池中取出一个实例为之服务，服务完了就回到对象池。当下一个用户再访问 EJB 方法时，EJB 容器有可能再次把该实例取出来为之服务。正因如此，无状态会话 Bean 只需要少量的实例就可以为成百上千的用户服务，大大提高了系统性能。

由于无状态会话 Bean 能够支持多个用户，并且通常在 EJB 容器中共享，可以为需要大量客户的应用提供更好的扩充能力。无状态会话 Bean 比有状态会话 Bean 更具性能优势，在条件允许的情况下开发人员应该首先考虑使用无状态会话 Bean。

2.1.1 开发只实现 Remote 接口的无状态 Session Bean

在开发前，先熟悉一下本例子的调用流程图：



1. 浏览器请求 Test.jsp 文件
2. 应用服务器的 JSP 引擎编绎 Test.jsp
3. Test.jsp 通过 JNDI 查找获得 HelloWorld EJB 的存根对象，然后调用 SayHello()方法，EJB 容器截获到方法调用。
4. EJB 容器调用 HelloWorld 实例的 SayHello()方法.

现在我们就开始本例子的开发。首先在 Eclipse 中新建一个普通的 java 项目，然后把[Jboss 安装目录]/client 下的所有 jar 文件加入到项目的构建路径中。如果不需要在项目中使用单元测试用例或普通 J2SE 调用 EJB，你只需要加入 javaee.jar，该文件在本书源代码 lib/javaee 目录下。接下来开始代码编写。

开发步骤如下：

第一步：定义一个包含业务方法的接口。这个接口不需要包含任何注释，它是一个普通的 java 接口。调用 EJB 的客户端使用这个接口引用从 EJB 容器返回的存根(stub)。代码如下：

HelloWorld.java

```
package com.foshanshop.ejb3;
public interface HelloWorld {
    public String SayHello(String name);
}
```

第二步：编写 Bean class。

HelloWorldBean.java 。Bean 类推荐的命名方式是：接口+Bean ，如： HelloWorldBean 。

```
package com.foshanshop.ejb3.impl;
import com.foshanshop.ejb3.HelloWorld;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote ({HelloWorld.class})
public class HelloWorldBean implements HelloWorld {

    public String SayHello(String name) {
        return name +"说：你好!世界,这是我的第一个EJB3哦.";
    }
}
```

在 Bean 类上面有两个注释@Stateless 和@Remote，@Stateless 注释指明这是一个无状态会话 Bean。@Stateless 注释的定义如下：

```
Package javax.ejb;
@Target(TYPE) @Retention(RUNTIME)
public @interface Stateless {
    String name( ) default "";
    String mappedName() default "";
}
```

name()属性用于指定 session bean 的 EJB 名称。该名称在 EJB Jar 包中必须是全局唯一的，而在 EAR 中却可以重复(因为 EAR 可以包含多个 EJB Jar，而每个 jar 可以存在一个同名的 EJB，在 EAR 中要定位某个 EJB，可以这样使用：xxx.jar#HelloWorldBean)。如果不指定该属性，默认就是 bean class 的非限定名称。对本例而言，EJB 名称默认为 HelloWorldBean。

mappedName()属性指定 Bean 的全局 JNDI 名称，这个属性在 weblogic，Sun 应用服务器和 glassfish 起作用。

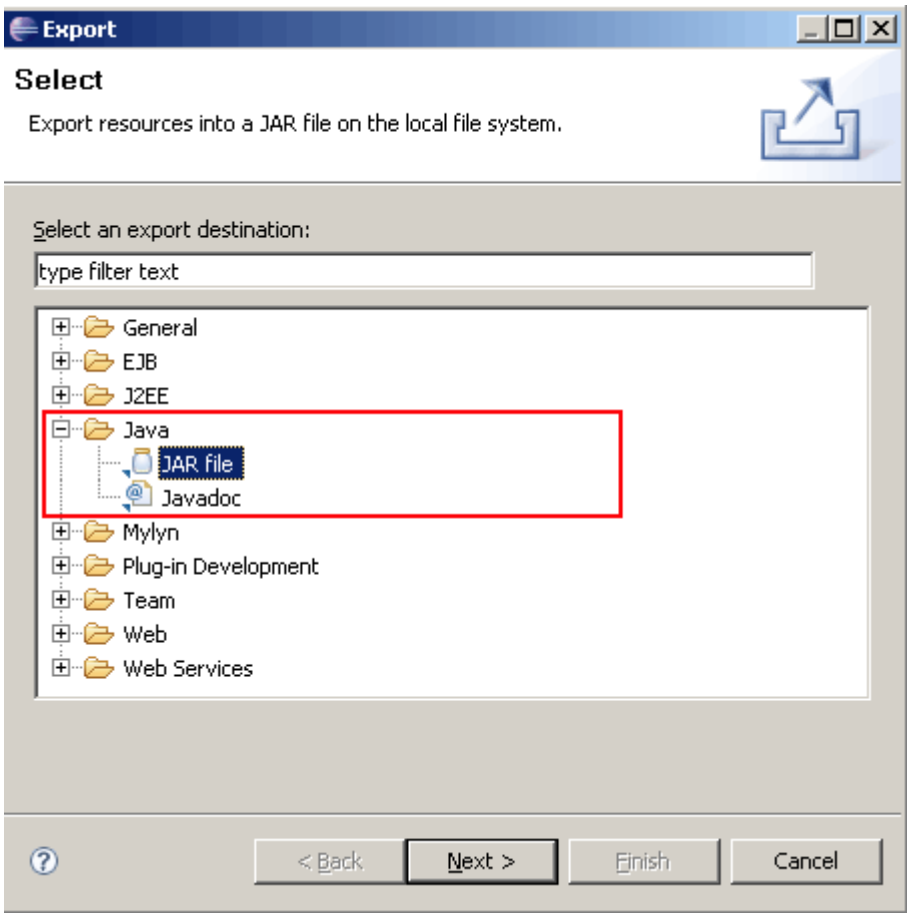
@Remote 注释指定这个无状态 Bean 的 remote 接口。Bean 类可以具有多个 remote 接口，每个接口之间用逗号分隔，如：@Remote ({HelloWorld.class,Hello.class,World.class})。如果你只有一个接口，你可以省略大括号，对于本例而言，可以写成这样：@Remote (HelloWorld.class)。

经过上面两步，一个 HelloWorld EJB 就开发完了。现在我们把它发布到 Jboss 中。在发布前我们需要把它打成 Jar 包。打 JAR 包的方法有很多，如使用 jar 命令、集成开发工具或者 Ant。下面为你介绍两种常用的打包方式：Eclipse

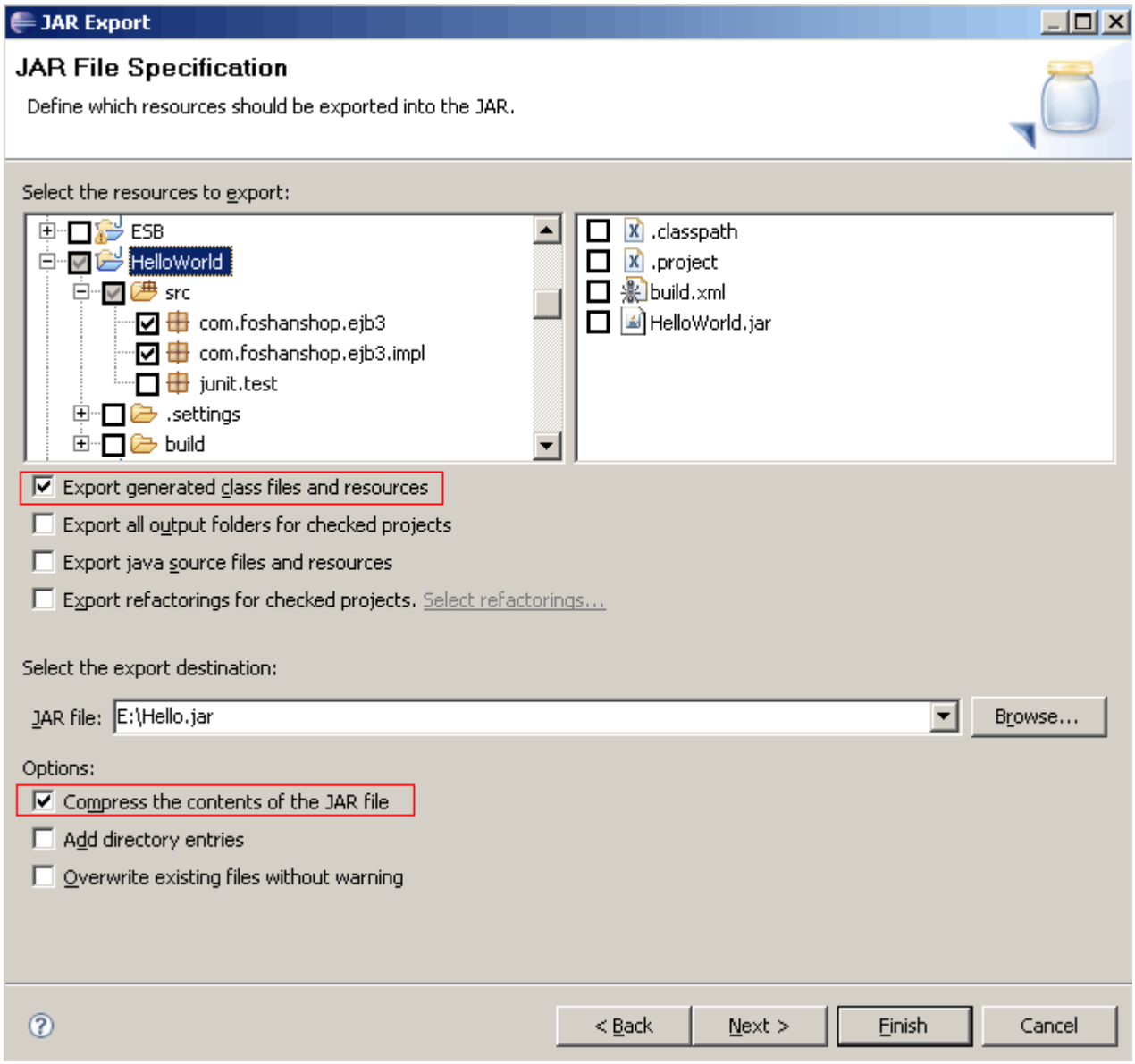
打包向导和 Ant 打包。

Eclipse 打包向导

在 Eclipse 开发环境下，可以通过导出向导进行打包。在项目名称上点击右键，在跳出的菜单中选择“Export（导出）”，在“Export”对话框选择“JAR file”，如下图：



点“Next”，在“select the resources to export（选择要导出的资源）”一栏，展开你的项目并选择需要打包的文件。然后选择一个存放目录及文件名。点“Finish”结束打包。如下图：



Ant 打包任务

使用 Ant 打包是比较方便的，也是作者推荐的打包方式。我们可以在项目根目录下建立一个名为 build.xml 的 xml 文件，然后在 xml 文件里面定义我们的打包任务。如下：

```
<?xml version="1.0"?>
<project name="HelloWorld" default="ejbjar" basedir=".">
    <property environment="env" />
    <property name="src.dir" value="${basedir}/src" />
    <property name="jboss.home" value="${env.JBOSS_HOME}" />
    <property name="build.dir" value="${basedir}/build" />
    <property name="build.classes.dir" value="${build.dir}/classes" />

    <!-- Build classpath -->
    <path id="build.classpath">
        <fileset dir="${jboss.home}/client">
```

```
            <include name="*.jar" />
        </fileset>
        <pathelement location="${build.classes.dir}" />
    </path>
    <target name="prepare" depends="clean">
        <mkdir dir="${build.dir}" />
        <mkdir dir="${build.classes.dir}" />
    </target>
    <target name="compile" depends="prepare" description="编绎">
        <javac srcdir="${src.dir}" destdir="${build.classes.dir}" debug="on"
deprecation="on" optimize="off" includes="**">
            <classpath refid="build.classpath" />
        </javac>
    </target>
    <target name="ejbjar" depends="compile" description="创建EJB发布包">
        <jar jarfile="${basedir}/HelloWorld.jar">
            <fileset dir="${build.classes.dir}">
                <include name="**/*.class" />
            </fileset>
            <metainf dir="${src.dir}/META-INF">
                <include name="*.xml" />
            </metainf>
        </jar>
    </target>
    <target name="clean">
        <delete dir="${build.dir}" />
    </target>
</project>
```

上面建立了一个名为 HelloWorld 的 Ant 项目，default="ejbjar"指定运行 Ant 时，如果没有给定任务名称，则默认执行 ejbjar 任务。basedir="."指定项目的路径为 build.xml 文件所在目录。

<property environment="env" />用于引用操作系统的环境变量。

<property name="jboss.home" value="${env.JBOSS_HOME}" />定义了一个名为 jboss.home 的属性，它的值引用名为 JBOSS_HOME 的环境变量(环境变量 JBOSS_HOME 是在安装 jboss 时让大家设置的)。

<property name="build.dir" value="${basedir}/build" />定义一个名为 build.dir 的属性，它的值指向项目路径下 build 目录，该目录用于存放编绎后的临时文件。

<property name="build.classes.dir" value="${build.dir}/classes" />定义一个名为 build.dir 的属性，它的值指向项目路径下 build/ classes 目录，该目录用于存放编绎后的 class 文件。

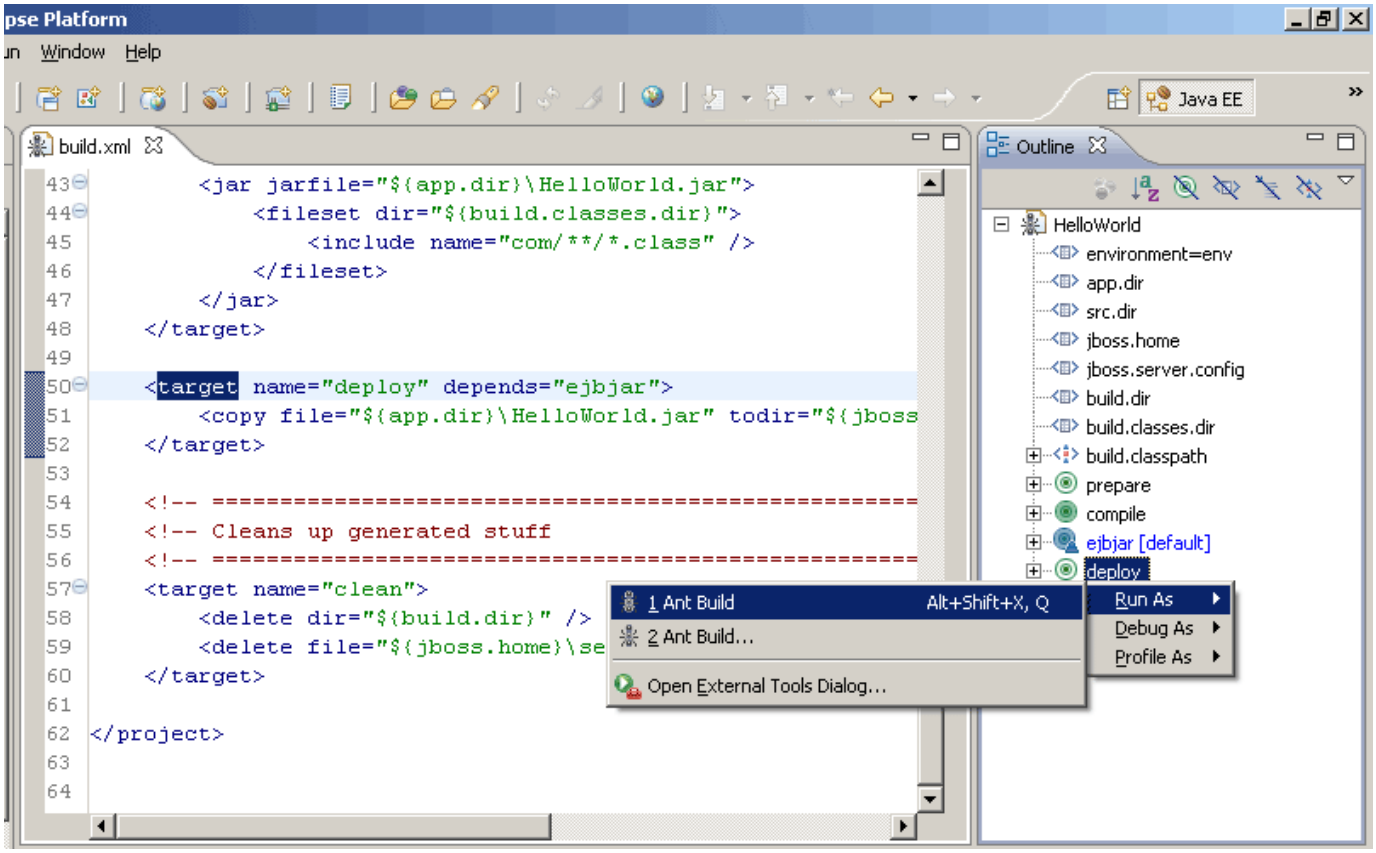
<path id="build.classpath">节点定义了一个 id 为 build.classpath 的类路径，类路径包含[jboss 安装目录]\client 下的所有 jar 文件及/build/ classes 下的所有类文件。

<target name="prepare" depends="clean">节点用于在项目路径下创建 build 和/build/ classes 文件夹。该任务依赖 clean 任务，执行 prepare 任务前会先执行 clean 任务。

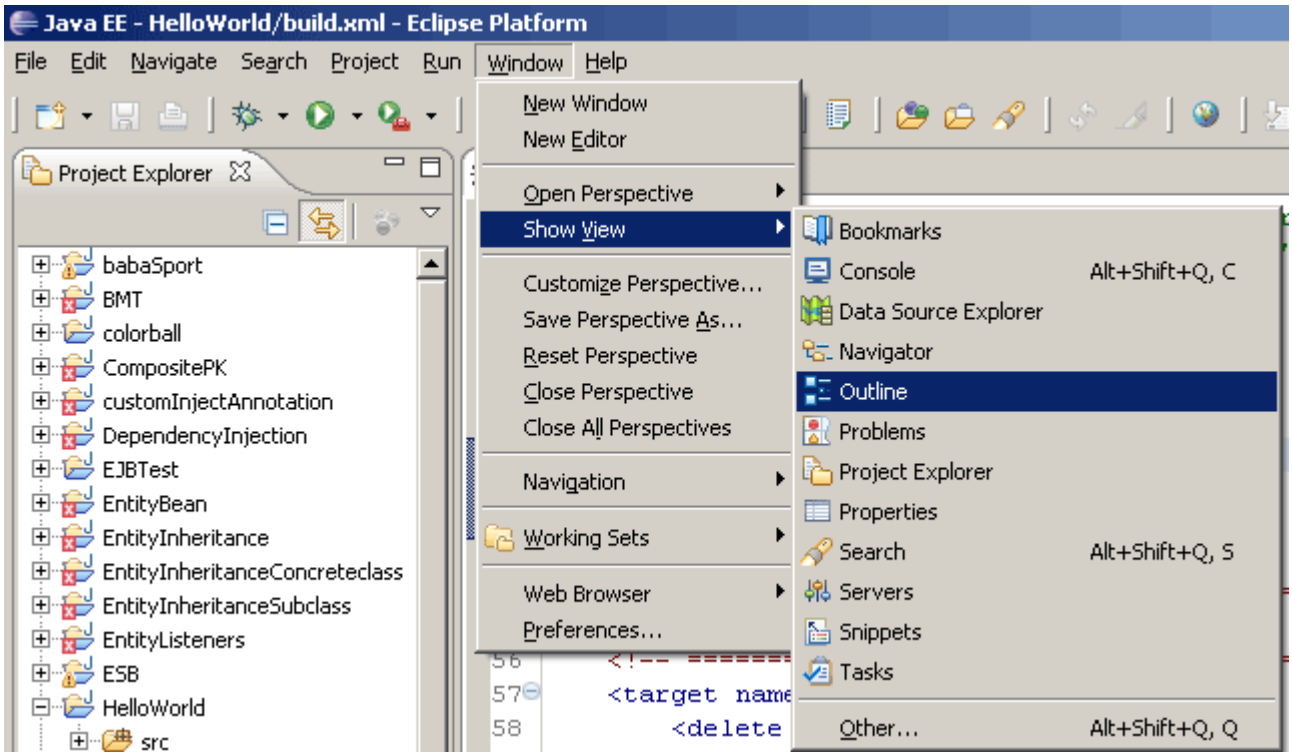
<target name="compile" depends="prepare" description="编绎">节点定义了一个编绎任务，该任务调用 javac 对 src 目录下的源文件进行编绎。classpath 引用 id 为 build.classpath 的类路径。编绎后的 class 文件存放在/build/classes 目录，在任务执行前会先执行 prepare 任务。

<target name="ejbjar" depends="compile" description="创建 EJB 发布包">节点定义了一个打包任务，该任务调用 jar 命令对/build/classes 目录下的所有 class 文件进行打包，并且把 src/META-INF 目录下的所有 xml 文件打进 jar 文件的 META-INF 目录。生成后的 jar 文件存放在项目的根目录下，名为 HelloWorld.jar。在任务执行前会先执行 compile 任务。

当 build.xml 编写完成后，我们可以在 eclipse 中执行 Ant 任务。方法是：打开 build.xml 文件，在窗口右边的 Outline（大纲）中右键点击任务名称，在出现的菜单中点击“Run As（运行方式）”-“Ant Build（Ant 构建）”如下图：



如果你的 eclipse 中没有大纲窗口，你可以点击“window”-“show view（显示视图）”-“Outline”，如下图：



不管使用何种打包方式，一个 EJB 打包后应具有以下目录结构：

```
EJB 应用根目录
| -- **/*.class   (你的.class 文件)
| -- META-INF
        | -- MANIFEST.MF (如果使用工具打包，该文件由工具自动生成)
```

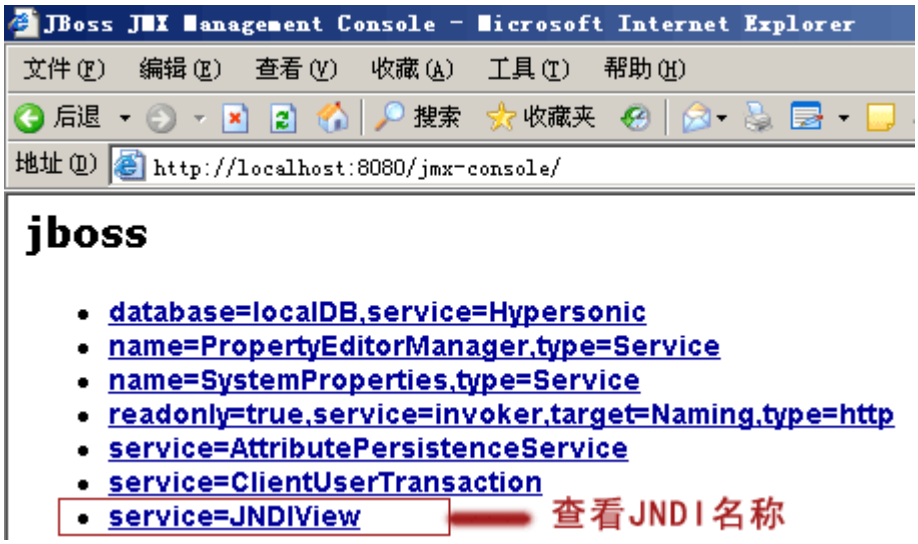
当 HelloWorld 打成 jar 文件后，我们把它发布到 Jboss。发布前先检查 jboss 是否已经启动，如果没有启动，我们可以进入[jboss 安装目录]/bin，双击 run.bat 启动 Jboss。不指定启动参数的情况下，Jboss 默认使用 default 配置项。把 jar 文件拷贝到[jboss 安装目录]\server\default\deploy\目录。观察 Jboss 控制台输出，如果没有抛出例外并看到下面的输出界面，发布就算成功了。

```
11:09:51,671 INFO  [AjpProtocol] Starting Coyote AJP/1.3 on ajp-127.0.0.1-8009
11:09:51,671 INFO  [Server] JBoss (MX MicroKernel) [4.2.1.GA (build: SVNTag=JBos
s_4_2_1_GA date=200707131605)] Started in 21s:250ms
11:10:11,921 INFO  [JmxKernelAbstraction] creating wrapper delegate for: org.jbo
ss.ejb3.stateless.StatelessContainer
11:10:11,937 INFO  [JmxKernelAbstraction] installing MBean: jboss.j2ee:jar=Hello
World.jar,name=HelloWorldBean,service=EJB3 with dependencies:
11:10:12,093 INFO  [EJBContainer] STARTED EJB: com.foshanshop.ejb3.impl.HelloWor
ldBean ejbName: HelloWorldBean
11:10:12,140 INFO  [EJB3Deployer] Deployed: file:/D:/JavaEEServer/jboss/server/d
efault/deploy/HelloWorld.jar
```

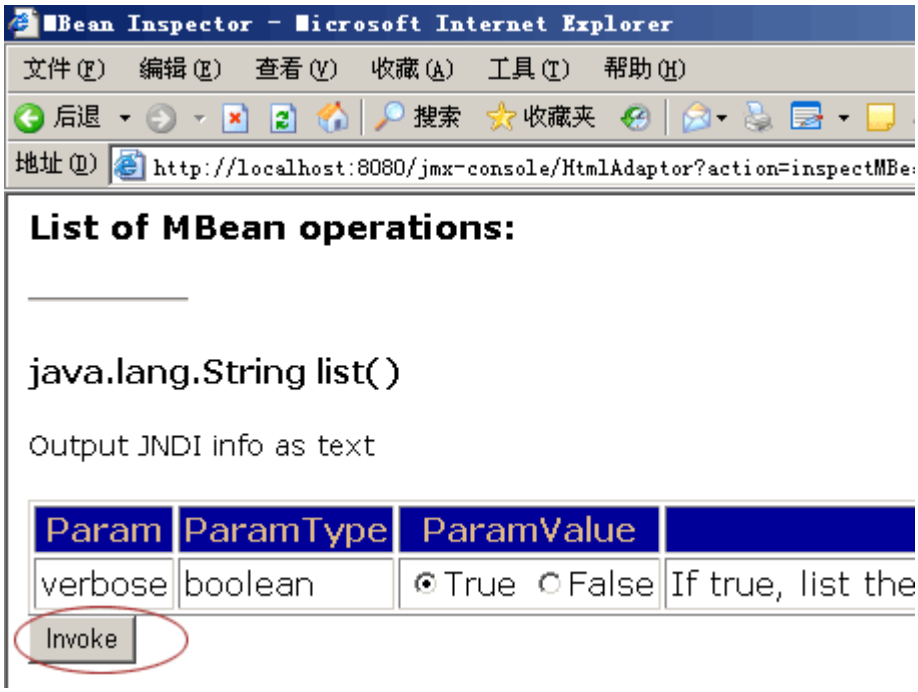
当 EJB 发布成功后，Jboss 容器会为它生成一个全局 JNDI 名称，我们可以利用这一点进一步判断 EJB 发布是否成功。我们进入 Jboss 的管理台查看它的 JNDI 名称，输入下面 URL

http://localhost:8080/jmx-console/

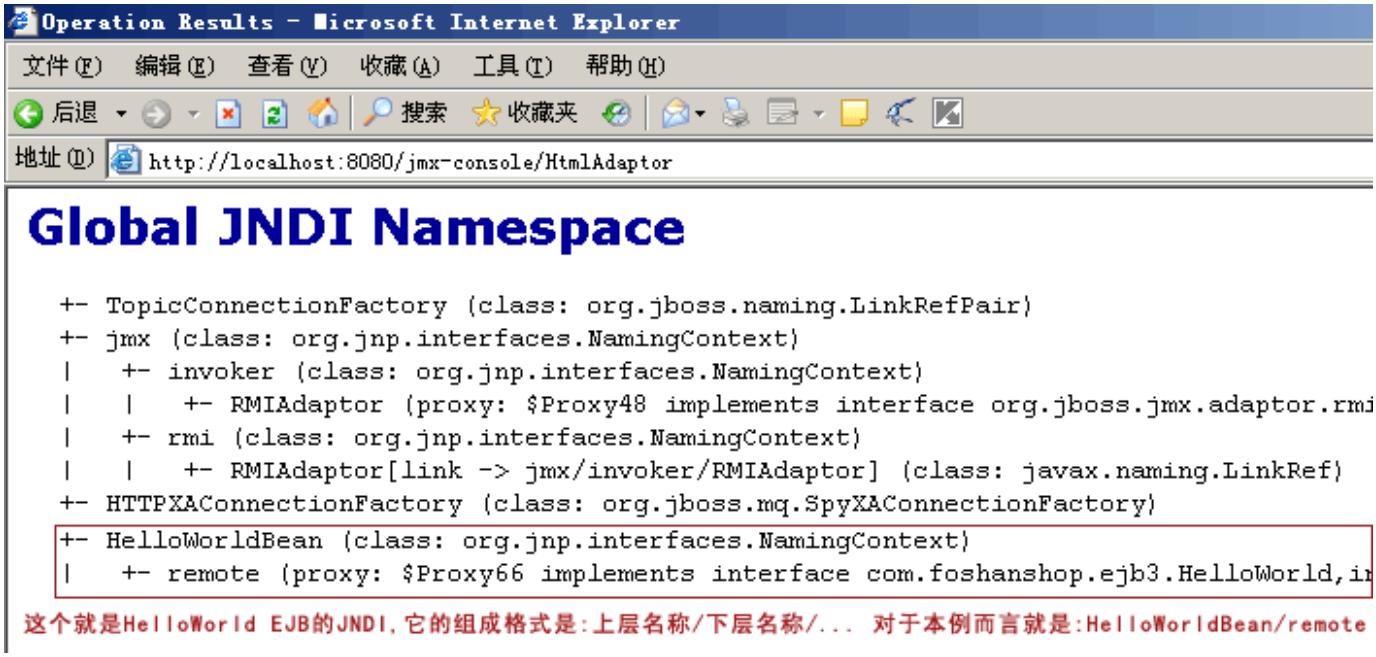
点击 service=JNDIView，查看 EJB 的 JNDI 名称。（如下图）



在出现的页面中，找到“List of MBean operations:”栏。（如下图）



点击“Invoke”按钮，出现如下界面:



在出现的页面中，我们可以看到 JBOSS 的 JNDI 树，它的命名约定如下：

（1）java:comp (java:comp namespace)

这个上下文环境和其子上下文环境仅能被应用组件内部访问和使用

（2）java：(java: Namespace)

子上下文环境和绑定的对象只能被处在同一个 JVM 内的客户访问

（3）Global JNDI Namespace

上下文环境能被所有客户访问，不管它们是否处在同一个 JVM 内。

当 EJB 发布到 Jboss 时，如果我们没有为它指定全局 JNDI 名称或修改过其默认 EJB 名称，Jboss 就会按照默认的命名规则为 EJB 生成全局 JNDI 名称，默认的命名规则如下：

如果把 EJB 作为模块打包进后缀为*.ear 的 JAVA EE 企业应用文件，默认的全局 JNDI 名称是

本地接口：EAR-FILE-BASE-NAME/EJB-CLASS-NAME/local

远程接口：EAR-FILE-BASE-NAME/EJB-CLASS-NAME/remote

EAR-FILE-BASE-NAME 为 ear 文件的名称，EJB-CLASS-NAME 为 EJB 的非限定类名。

例:把 HelloWorld 应用作为 EJB 模块打包进名为 HelloWorld.ear 的企业应用文件,它的远程接口的 JNDI 名称是:HelloWorld/HelloWorldBean/remote

如果把 EJB 应用打包成后缀为*.jar 的模块文件，默认的全局 JNDI 名称是

本地接口：EJB-CLASS-NAME/local

远程接口：EJB-CLASS-NAME/remote

例：把 HelloWorld 应用打包成 HelloWorld.jar 文件，它的远程接口的 JNDI 名称是：HelloWorldBean/remote

注意：EJB-CLASS-NAME 是不带包名的，如 com.foshanshop.ejb3.impl.HelloWorldBean 只需取 HelloWorldBean。

如果你通过@Stateless.name()、@Stateful.name()及其等价的 XML 指定了 EJB 名称，那么上面的 EJB-CLASS-NAME 应该换为 EJB 名称，此时的 JNDI 名称格式如：EJB 名称/remote、EAR 文件名/EJB 名称/remote。

在 Global JNDI Namespace 一栏,我们看到了 HelloWorldBean 的远程接口的 JNDI 名称为 HelloWorldBean/remote。意味着 EJB 已经发布成功。接下来我们看看客户端如何访问它。

Test.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
```

```
<%@ page import="com.foshanshop.ejb3.HelloWorld, javax.naming.*,
java.util.Properties"%>
<%
        Properties props = new Properties();
        props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
        props.setProperty("java.naming.provider.url", "localhost:1099");
        try {
            InitialContext ctx = new InitialContext(props);
            HelloWorld helloworld = (HelloWorld)
ctx.lookup("HelloWorldBean/remote");
            out.println(helloworld.SayHello("佛山人"));
        } catch (NamingException e) {
            out.println(e.getMessage());
        }
%>
```

在进行 JNDI 查找前，我们必须设置应用服务器的上下文信息，主要是设置 JNDI 驱动的类名（java.naming.factory.initial）和命名服务提供者的 URL （java.naming.provider.url） 。

- java.naming.factory.initial 或 Context.INITIAL_CONTEXT_FACTORY：环境属性名，用于指定 InitialContext 工厂（作者称它为 JNDI 驱动更容易理解），它类似于 JDBC 指定数据库驱动类。
  因为本例子连接的是 JbossNS（命名服务的实现者），所以使用 Jboss 提供的驱动类：org.jnp.interfaces.NamingContextFactory
- java.naming.provider.url 或 Context.PROVIDER_URL：环境属性名，包含提供命名服务的主机地址和端口号。它类似于 JDBC 指定数据库的连接 URL。
  连接到 JbossNS 的 URL 格式为：jnp://host:port，该 URL 的“jnp:”部分是指使用的协议，JBoss 使用的是基于 Socket/RMI 的协议。host 为主机的地址，port 为 JNDI 服务的端口。除了 host 之外，其他部分都是可以不写的。
- 除了上述两个环境属性外，还有两个环境属性是我们经常使用到的：java.naming.security.principal（或 Context.SECURITY_PRINCIPAL）和 java.naming.security.credentials（或 Context.SECURITY_CREDENTIALS），这两个环境属性用于指定用户标识（如用户名）及凭证（如密码），当 EJB 使用了安全服务时，你必须提供这两个属性。它类似于 JDBC 指定连接到数据库的用户名及密码。

如同数据库一样，根据访问命名服务器的不同，为上下文设置的驱动类和 URL 也是不同的，如下面是访问 Sun 应用服务器的上下文信息：

```
Properties props = new Properties();
props.setProperty("java.naming.factory.initial",
"com.sun.enterprise.naming.SerialInitContextFactory");
props.setProperty("java.naming.provider.url", "localhost:3700");
InitialContext = new InitialContext(props);
HelloWorld helloworld = (HelloWorld)
ctx.lookup("com.foshanshop.ejb3.HelloWorld");
```

如果客户端运行在应用服务器内，我们不需要为 InitialContext 设置应用服务器的上下文信息，也不建议设置。因为应用服务器启动时会把 JNDI 驱动类等上下文信息添加进系统属性，创建 InitialContext 对象时如果没有指定

Properties 参数，InitialContext 内部会调用 System.getProperty()方法从系统属性里获取必要的上下文信息。对本例子而言，你可以省略传入 props 参数，之所以给 InitialContext 设置参数，目的是引出相关知识点，便于教学。在实际应用中，如果给 InitialContext 设置了参数，反而会带来不可移植的问题。

注：创建 InitialContext 对象时如果没有指定 Properties 参数，InitialContext 还会在 classpath 下寻找 jndi.properties 文件，并从该文件中加载应用服务器的上下文信息。这样避免了硬编码为 InitialContext 设置 Properties 参数。jndi.properties 的配置如下：

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

当 InitialContext 初始化后，我们使用 EJB 的 Jndi 名称通过 lookup()方法查找 EJB。lookup()方法返回一个存根对象，该存根实现了 HelloWorld 接口。它负责将方法调用路由到应用服务器，应用服务器再把方法调用请求路由到 HelloWorldBean 实例。

本例子的客户端是个 web 应用，我们打算把它发布到 Jboss 中，这时 web 应用与 EJB 应用发布在同一个 Jboss 中。因为 EJB 的类在发布时已经被 EJB 类加载器加载，所以我们的 web 应用不需要加入任何 EJB 的类文件。有时候如果加入了 EJB 的类文件，反而会导致程序出错。如在调用 Stateful Bean 时会发生类型冲突，引发下面的例外。

```
java.lang.ClassCastException: $Proxy84
    org.apache.jsp.StatefulBeanTest_jsp._jspService(org.apache.jsp.StatefulBeanTest_jsp:55)
```

发布一个 web 应用到 Jboss 前，我们需要把它打成 war 文件。作者发现很多同学在打 war 文件的时候经常出现些小错误。首先，我们必须了解一个 web 应用应该具有那些目录及文件，下面是一个普通的 web 应用，其中 WEB-INF 目录和 web.xml 文件是必须提供的，其它的目录和文件可以不存在。当你打成 war 文件后应检查 war 文件内是否具有以下文件结构，如果根目录下没有 WEB-INF 或 WEB-INF 下没有 web.xml 文件，那么在发布 web 应用时将会失败。

```
War 文件根目录
| -- **/*. jsp (可选的)
| -- WEB-INF (必需的)
        | -- web.xml (必需的)
        | -- lib (可选的)
            | -- *.*
        | -- classes (可选的)
            | -- **/*.class
```

现在我们已经了解了 web 应用的目录结构，接下来我们就开始学习使用 jar 命令打 war 包。我们按照 web 目录结构的要求，组织我们的客户端文件，如下：

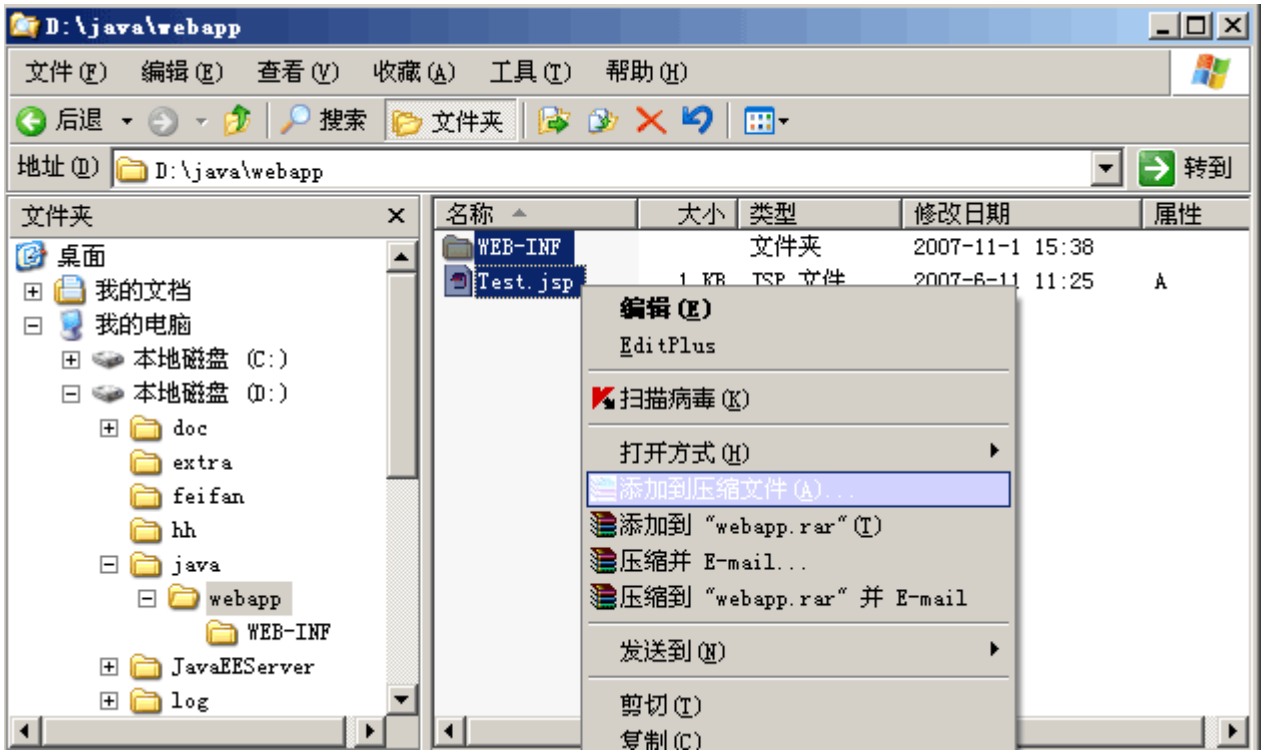
```
Web 应用根目录
| -- Test. jsp
| -- WEB-INF
        | -- web.xml
```

Test.jsp 是本例子的客户端代码，web.xml 文件内容如下：

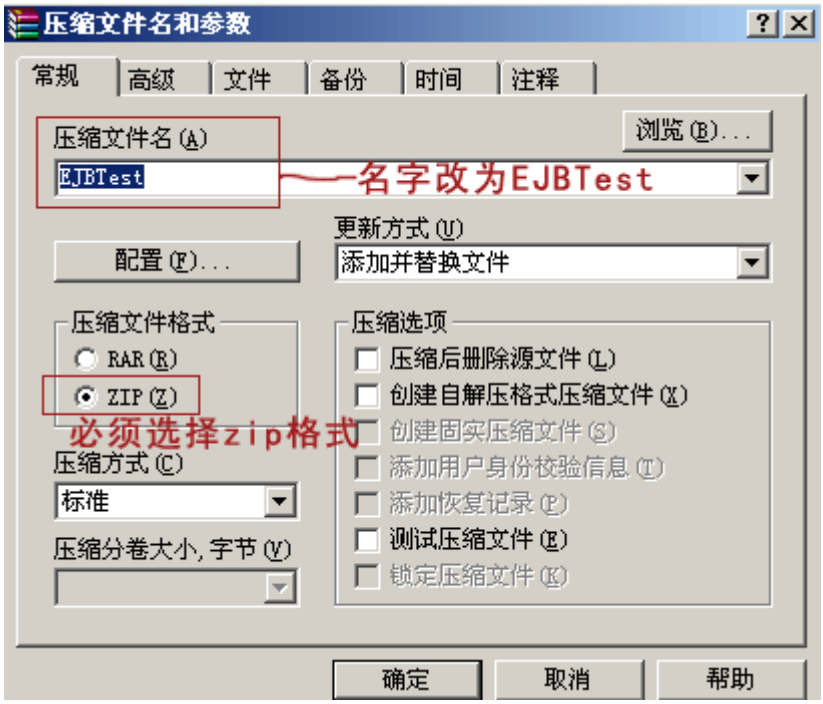
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
   <description>ejb3.0 test web site</description>
</web-app>
```

假设你的 web 应用在 D:\java\webapp，我们在 Dos 命令行下进入到 WEB 应用的根目录下，执行如下命令
D:\java\webapp> jar cvf EJBTest.war *
此命令把 WEB 应用根目录下的所有文件及文件夹打包成 EJBTest.war 文件

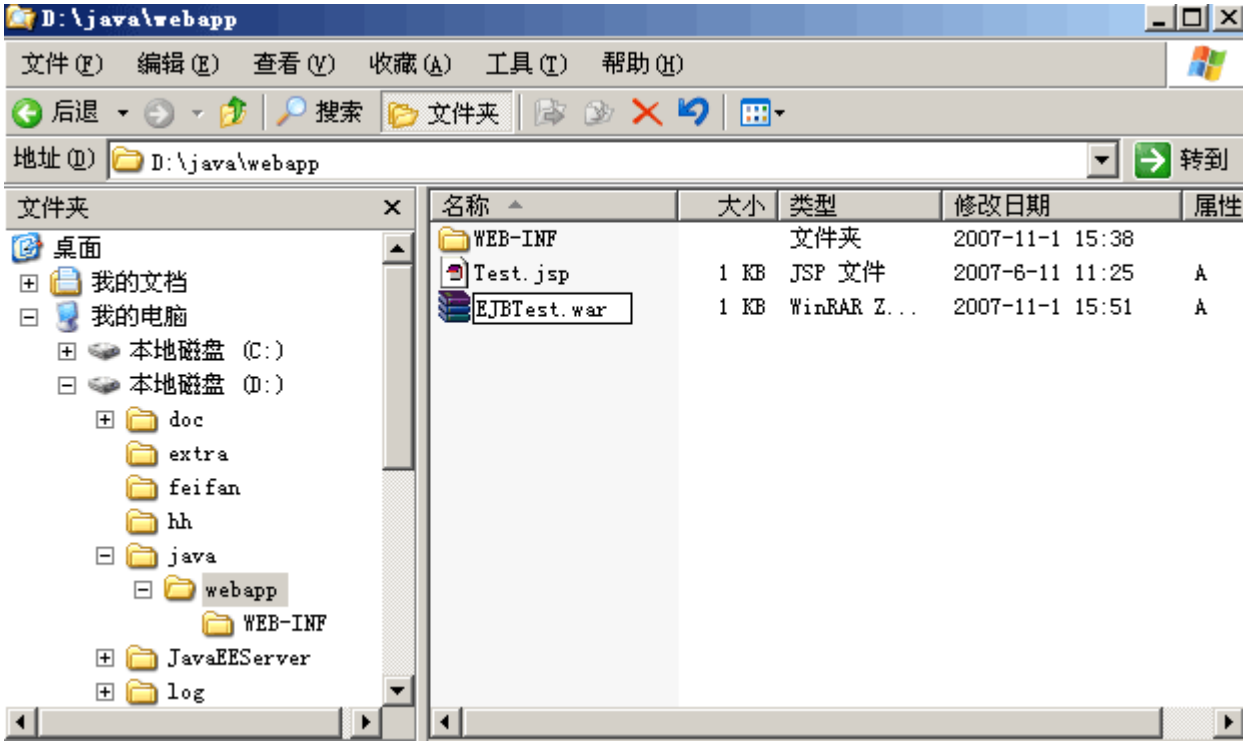
如果你觉得 jar 命令使用起来比较麻烦，你也可以通过 winrar 软件进行打包。打包前，必须把文件组织成上面的目录结构，然后全选 D:\java\webapp 目录下的所有文件，在被选文件上点击鼠标右键，在出现的菜单中选择“添加到压缩文件”，如下图



在出现的对话框中，输入文件名：EJBTest，压缩文件的格式必须是 zip，如下图：



点“确定”后，生成 EJBTest.zip 文件，把文件的后缀改为 war 就完成了 web 打包。如下图：



虽然采用上面两种方式都可以进行 web 打包，但显的不太专业，作者建议使用 Ant 进行 war 文件打包，下面是打包片断：

```
<project name="EJBTest" default="war" basedir=".">
    <target name="war" description="创建WEB发布包">
        <war warfile="${basedir}/EJBTest.war"
webxml="${basedir}/WEB-INF/web.xml">
```

```
        <fileset dir="${basedir}">
            <include name="*.jsp"/>
        </fileset>
        <classes dir="${basedir}/WEB-INF/classes">
            <include name="**/*.class"/>
        </classes>
        <lib dir="${basedir}/WEB-INF/lib"></lib>
        </war>
    </target>
</project>
```

当客户端应用打完包后，我们把它拷贝到“[jboss 安装目录]\server\default\deploy”。如果 war 文件的名称为 EJBTest.war ，我们可以通过 http://localhost:8080/EJBTest/Test.jsp 访问客户端。

本例子的源代码在配套光盘的 HelloWorld 文件夹。要恢复 HelloWorld 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。
本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用到 Jboss，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/Test.jsp 访问客户端。

小知识点：

对于 HelloWorld 的 Bean class，并不要求你必须实现 HelloWorld 接口，你可以去掉接口实现的代码，如：

```
package com.foshanshop.ejb3.impl;
import com.foshanshop.ejb3.HelloWorld;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote ({HelloWorld.class})
public class HelloWorldBean {

    public String SayHello(String name) {
        return name +"说：你好!世界,这是我的第一个EJB3哦.";
    }
}
```

虽然没有规定你必须实现业务接口，但实现业务接口是绝对有利的，它能在语法上约定 Bean class 的方法与接口的方法声明一致。或许你对上面的代码有些疑惑，按照接口编程的约束，接口要引用目标类，那么目标类必须实现接口才可以，否则接口是引用不了实现类的。事实上，EJB 客户端接口引用的对象并非 Bean 类，跟 Bean 类对象的交互也不是直接进行的，这些疑问你都可以在后面“EJB 调用机制”中得到答案。

2.1.2 开发只实现 Local 接口的无状态 Session Bean

开发只有 Local 接口的无状态 Session Bean 的步骤和上节开发只有 Remote 接口的无状态会话 Bean 的步骤相同，

两者唯一不同之处是，前者使用@Remote 注释声明接口是远程接口，后者使用@Local 注释声明接口是本地接口。当@Local 和@Remote 注释都不存在时，容器会将 Bean class 实现的接口默认为 Local 接口。如果 EJB 与客户端部署在同一个应用服务器，采用 Local 接口访问 EJB 优于 Remote 接口。因为通过 Remote 接口访问 EJB 需要在 tcp/ip 协议基础上转换和解释 Corba IIOP 协议消息，在调用 EJB 的这一过程中存在对象序列化，协议解释、tcp/ip 通信等开销。而通过 Local 接口访问 EJB 是在内存中与 bean 彼此交互的，没有了分布式对象协议的开销，大大改善了性能。下面是只有 Local 接口的无状态会话 Bean。

业务接口：LocalHelloWorld.java

```
package com.foshanshop.ejb3;
public interface LocalHelloWorld {
    public String SayHello(String name);
}
```

Bean 类：LocalHelloWorldBean.java

```
package com.foshanshop.ejb3.impl;
import javax.ejb.Local;
import javax.ejb.Stateless;
import com.foshanshop.ejb3.LocalHelloWorld;

@Stateless
@Local ({LocalHelloWorld.class})
public class LocalHelloWorldBean implements LocalHelloWorld {

    public String SayHello(String name) {
        return name +"说：你好!世界,这是一个只具有Local接口的无状态Bean";
    }

}
```

和@Remote 注释一样，@Local 注释也可以定义多个本地接口。
如：@Local ({LocalHelloWorld.class,Hello.class,World.class})。
如果只有一个本地接口，可以省略大括号，对于本例而言，可以写成这样：@Remote (LocalHelloWorld.class)。

把上面的 EJB 打成 jar 文件后，发布到[jboss 安装目录]\server\default\deploy 目录中。接下来我们编写客户端调用代码。

LocalSessionBeanTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.LocalHelloWorld, javax.naming.*"%>
<%
        try {
            InitialContext ctx = new InitialContext();
            LocalHelloWorld helloworld = (LocalHelloWorld)
ctx.lookup("LocalHelloWorldBean/local");
            out.println(helloworld.SayHello("佛山人"));
        } catch (NamingException e) {
```

```
            out.println(e.getMessage());
        }
%>
```

把上面的 jsp 打包成 war 文件并发布到 jboss 中。我们不能在应用服务器外调用 Local 接口，如果你在独立的 Tomcat 或 J2SE 中调用 Local 接口，将获得以下例外：

```
java.lang.NullPointerException
    org.jboss.ejb3.stateless.StatelessLocalProxy.invoke(StatelessLocalProxy.java:74)
```

产生此例外的原因是，调用 Local 接口的客户端与 EJB 容器不在同一个 JVM。为了确保客户端与应用服务器处于同一个 JVM，请把客户端应用部署在应用服务器下。如果客户端应用与 EJB 容器不在同一个 JVM，只能通过其 Remote 接口进行访问。

本例子的源代码在配套光盘的 LocalSessionBean 文件夹。要恢复 LocalSessionBean 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。

本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用到 Jboss，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/LocalSessionBeanTest.jsp 访问客户端。

2.1.3 开发实现了 Remote 与 Local 接口的无状态 Session Bean

在实际应用中，同时实现 Remote 与 Local 接口是一种比较好的做法。这样你既可以在远程访问 EJB，也可以在本地访问 EJB。

远程接口：Operation.java

```
package com.foshanshop.ejb3;
public interface Operation {
   public int Addup();
}
```

本地接口：LocalOperation.java，本地接口继承了远程接口的所有方法

```
package com.foshanshop.ejb3;
public interface LocalOperation extends Operation {
}
```

会话 Bean：OperationBean.java

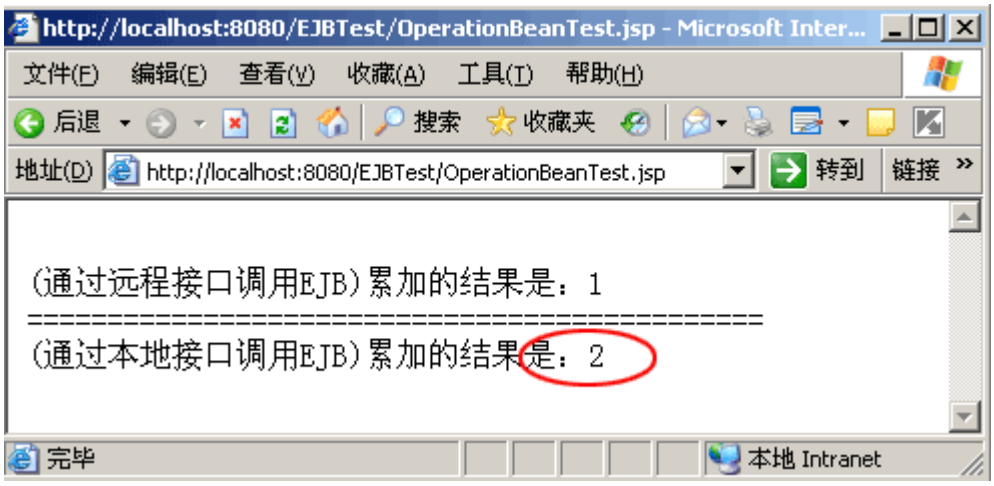
```
package com.foshanshop.ejb3.impl;
import javax.ejb.Local;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import com.foshanshop.ejb3.LocalOperation;
import com.foshanshop.ejb3.Operation;

@Stateless
@Remote (Operation.class)
```

```
@Local (LocalOperation.class)
public class OperationBean implements Operation, LocalOperation {
    private int total = 0;

    public int Addup() {
       total++;
       return total;
    }
}
```

JSP 客户端：OperationBeanTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.*, javax.naming.*, java.util.Properties"%>
<%
        InitialContext ctx = new InitialContext();
        try {
            //通过远程接口调用EJB
            Operation remote = (Operation) ctx.lookup("OperationBean/remote");
            out.println("<br>(通过远程接口调用EJB)累加的结果是："+ remote.Addup());
        } catch (Exception e) {
            out.println("<br>远程接口调用失败");
        }
        out.println("<br>================================================");
        try {
            //通过本地接口调用EJB
            LocalOperation local = (LocalOperation)
ctx.lookup("OperationBean/local");
            out.println("<br>(通过本地接口调用EJB)累加的结果是："+ local.Addup());
        } catch (Exception e) {
            out.println("<br>本地接口调用失败");
        }
%>
```

把 EJB 及客户端应用打包后发布到 jboss 中，第一次执行客户端的结果如下：



细心的读者可能会发现在通过 local 接口调用 EJB 的地方，累加的结果为 2。怎么会为 2，明明是第一次调用嘛，应该是 1 才对呀？
这是因为 Stateless Session Bean 不负责维护会话状态，Stateless Session Bean 一旦实例化就被加进实例池中，各个用户都可以共用。即使用户已经消亡，Stateless Session Bean 的生命期也不一定结束，它可能依然存在于实例池中，供其他用户使用。如果它有自己的属性（变量），那么这些变量就会受到所有使用它的用户影响。

下图说明了这一执行过程：

本例子的源代码在配套光盘的 LocalRemoteBean 文件夹。要恢复 LocalRemoteBean 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss（确保配置了环境变量 JBOSS_HOME 及启动了 Jboss），你可以执行 Ant 的 deploy 任务。
本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用到 Jboss，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/OperationBeanTest.jsp 访问客户端。

2.2 实例池化(Instance Pooling)

上节我们提到过实例池，这一节要告诉大家为什么要使用实例池。在一个大型业务系统中，存在着大量的用户，这时可能需要同时使用数以万计的 Bean 实例对象。如果不使用实例池化，容器会为客户每次方法调用进行实例的创建和销毁。这样必然会影响到系统的性能，使系统的吞吐量减少。因此 EJB 提供了实例池化这种管理大量 bean 实例的机制。

2.3 stateless session bean 的生命周期

stateless session bean 的生命周期相当简单。它只有两个状态：does not exist 和 method-ready pool。下图展示了一个 stateless session bean 实例在其生命周期中经历的状态及其状态迁移。

2.4 Stateful Session Bean（有状态 bean）开发

前面我们已经了解到，stateless session bean 创建在对象池中，提供给众多用户使用，如果 Bean class 有自己的成员属性（变量），那么这些变量就会受到所有调用它的用户影响。在一些应用场合中，有时我们需要每个用户都有自己的一个实例，这个实例不受其他用户影响。就好比购物车对象，每个用户都应有自己的购物车，你不希望有人往你的购物车里添加或拿掉商品，而有状态 Bean 正好满足你的这种需求。每个有状态 Bean 在 bean 实例的生命周期内都只服务于一个用户，bean class 的成员变量可以在不同的方法调用间维护特定于某个用户的数据。正因为在 bean 实例的生命周期内，Stateful Session Bean 保持了用户的信息，所以叫有状态 Bean。

对于 Stateful Session Bean，用户每调用一次 lookup()都将创建一个新的 bean 实例，如果你希望一直使用某个 Bean 实例，你必须在客户端缓存存根。如：你使用 Stateful Session Bean 做购物车，在 A 页面中通过 lookup()得到购物车存根，添加一个商品到购物车，接着浏览商品。在 B 页面发现了一个最牛 B 的商品，你想把它加入购物车，如果你企图通过 lookup()方法获取刚才的购物车，这时你将会得到一个没有任何商品的新购物车。要想获取到刚才的购物车，你需要在第一次 lookup()购物车时，也就是在 A 页面，使用 session（或别的缓存方案）缓存购物车存根，这样在后续的页面（如 B 页面）就可以通过 session 里的存根对象获取到原先的购物车。

Stateful Session Bean 的开发步骤与 Stateless Session Bean 的开发步骤相同。注：在 EJB3.0 最终规范中，stateful session bean 不要求必须实现 Serializable 接口。

业务接口：Cart.java

```
package com.foshanshop.ejb3;
import java.io.Serializable;
import java.util.List;

public interface Cart extends Serializable {
    public void AddBuyItem(String productName);
    public List<String> getBuyItem();
}
```

Bean 类：CartBean.java

```
package com.foshanshop.ejb3.impl;
import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remote;
import javax.ejb.Stateful;
import com.foshanshop.ejb3.Cart;

@SuppressWarnings("serial")
@Stateful
@Remote(Cart.class)
public class CartBean implements Cart{
    private List<String> buyitem = new ArrayList<String>();

    public void AddBuyItem(String productName) {
        buyitem.add(productName);
    }

    public List<String> getBuyItem() {
        return buyitem;
    }
}
```

@Stateful 注释指明这是一个有状态会话 Bean，@Remote 注释指定有状态 Bean 的 remote 接口。
@SuppressWarnings("serial") 注释屏蔽缺少 serialVersionUID 定义警告。
下面是有状态 Bean 的 JSP 客户端代码：StatefulBeanTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.*,java.util.*,javax.naming.*"%>
<%
    try {
        InitialContext ctx = new InitialContext();
        Cart cat = (Cart)session.getAttribute("cat");
        if(cat==null){//创建一个购物车
            cat = (Cart) ctx.lookup("CartBean/remote");
            session.setAttribute("cat", cat);
        }
        cat.AddBuyItem("《EJB3.0实例教程》");
        List<String> buyitem = cat.getBuyItem();
        out.println("购物车的商品列表：<br>");
        for(String name : buyitem){
            out.println("  "+ name+ "<br>");
        }
    } catch (Exception e) {
        out.println(e.getMessage());
    }
%>
```

在 JSP 代码中，先试图从 session 中获取购物车的存根。如果当前 session 不存在购物车，就创建一个新的购物车放入 session 中。后面每执行一次页面都会添加一个相同的商品。

本例子的源代码在配套光盘的 StatefulBean 文件夹。要恢复 StatefulBean 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。
本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用到 Jboss，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/StatefulBeanTest.jsp 访问客户端。

2.5 激活机制(Activation mechanism)

在大量用户使用系统的时候，为了提高系统性能，增加系统的吞吐量。EJB 容器使用了实例池化机制，使的系统可以用最少的 bean 实例为用户服务。这种机制建立在一个 bean 实例可以为多个用户服务的基础上。对于有状态 Bean 而言，每个有状态 Bean 在其生命周期内只服务于一个用户，这样对有状态 Bean 使用实例池化机制是毫无意义的。

2.6 Stateful Session Bean 的生命周期

stateful Session Bean 的生命周期包含三种状态：does not exist，method-ready 和 passivated。看起来与 Stateless Session Bean 有些相似，但实际上 method-ready 状态和 Stateless Session Bean 的 method-ready pool 之间有着显著的不同。
下图显示了 stateful Session Bean 的状态图：

2.7 EJB 调用机制

由于 EJB 的调用过程对开发者来说是透明的，以至于我们错误地认为：lookup()方法返回的对象就是 bean 实例。实际上，客户端与 Session bean 交互，它并不直接与 Bean 实例打交道，而是经由 bean 的远程或本地接口。当你调用远程或本地接口的方法时，接口使用的是存根（stub）对象。该存根实现了 session bean 的远程或本地接口。它负责将方法调用经过网络发送到远程 EJB 容器，或将请求路由到位于本地 JVM 内的 EJB 容器。存根是在部署

期间使用 JDK 所带的 java.lang.reflect.Proxy 动态生成。

下面是通过远程接口调用 EJB 的过程（以 HelloWorld 为例）：

2.8 如何改变 Session Bean 的 JNDI 名称

如果我们没有指定 EJB 的 JNDI 名称，当 EJB 发布到应用服务器时，应用服务器会按默认规则为 EJB 生成全局 JNDI 名称。当我们需要自定义 JNDI 名称时，可以这样做
如果 EJB 在 Jboss 中使用，可以使用 Jboss 提供的 @LocalBinding 和 @RemoteBinding 注释，@LocalBinding 注释指定 Session Bean 的 Local 接口的 JNDI 名称，@RemoteBinding 注释指定 Session Bean 的 Remote 接口的 JNDI 名称，如下：

```
package com.foshanshop.ejb3.impl;
import javax.ejb.Local;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import com.foshanshop.ejb3.LocalOperation;
import com.foshanshop.ejb3.Operation;
import org.jboss.annotation.ejb.LocalBinding;
import org.jboss.annotation.ejb.RemoteBinding;

@Stateless
@Remote ({Operation.class})
@RemoteBinding (jndiBinding="foshanshop/RemoteOperation")
@Local ({LocalOperation.class})
@LocalBinding (jndiBinding="foshanshop/LocalOperation")
public class OperationBean implements Operation, LocalOperation {
	private int total = 0;
	private int addresult = 0;
	public int Add(int a, int b) {
		addresult = a + b;
		return addresult;
	}
	public int getResult() {
	    total += addresult;
 return total;
	}
```

```
}
```

访问上面 EJB 的客户端代码片断如下：

```
InitialContext ctx = new InitialContext(props);
Operation operation = (Operation) ctx.lookup("foshanshop/RemoteOperation");
```

@LocalBinding 和 @RemoteBinding 注释只是针对 jboss，如果你使用的是 weblogic10/Sun Application Server/Glassfish，你可以使用@Stateless.mappedName()设置 bean 的全局 JNDI 名称，如：

```
@Stateless(mappedName="OperationBeanRemote")
public class OperationBean implements Operation, LocalOperation {
```

客户端访问 EJB 的代码片断如下：

```
InitialContext ctx = new InitialContext(props);
Operation operation = (Operation) ctx.lookup("OperationBeanRemote#com.foshanshop.ejb3.Operation");
```

由于JNDI名称与厂商有关，如果使用注释定义JNDI名称会带来移植问题，因此建议使用ejb-jar.xml部署描述文件进行定义，该文件必须放置在jar的META-INF目录下

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                      http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
      version="3.0">
<enterprise-beans>
  <session>
    <ejb-name>HelloWorldBean</ejb-name>
    <mapped-name>HelloWorldBean</mapped-name>
  </session>
</enterprise-beans>
</ejb-jar>
```

ejb-name为EJB名称，mapped-name为bean的JNDI名称。目前jboss不支持在ejb-jar.xml通过mapped-name指定JNDI名称，但我们可以使用他专属的部署描述文件jboss.xml进行定义，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jboss PUBLIC
     "-//JBoss//DTD JBOSS 4.2//EN"
     "http://www.jboss.org/j2ee/dtd/jboss_4_2.dtd">
<jboss>
    <enterprise-beans>
       <session>
          <ejb-name>HelloWorldBean</ejb-name>
          <jndi-name>hello/remote</jndi-name>
```

```
        </session>
    </enterprise-beans>
</jboss>
```

该文件必须放置在 jar 的 META-INF 目录下。

2.9 Session Bean 的生命周期事件

前面我们已经对 Stateless Session Bean 和 stateful session bean 的生命周期有所了解。在 Session bean 的生命周期里，状态变化会触发生命周期事件的发生。如从 does not exist 状态进入 method-ready pool 状态会触发@PostConstruct 事件。同样从 method-ready pool 状态进入 does not exist 状态也会触发@PreDestroy 事件。

有些时候，你需要定制 session bean 的管理过程。例如，你可能想在创建 session bean 实例的时候初始化字段变量，或在 bean 实例被销毁的时候关掉外部资源。上述这些，你都可能通过在 bean 类定义生命周期的回调方法来实现。这些方法将会被容器在生命周期的不同阶段调用（如：创建或销毁）。通过使用下面的注释，EJB 3.0 允许你将任何方法指定为回调方法。这不同于 EJB 2.1，所有的回调方法必须实现，即使是空的。在 EJB 3.0，bean 可以有任意数量，任意名字的回调方法。

- @PostConstruct：当 bean 对象完成实例化后，标注了这个注释的方法会被立即调用，每个 bean class 只能定义一个@PostConstruct 方法。这个注释同时适用于有状态和无状态会话 bean。
- @PreDestroy：标注了这个注释的方法会在容器销毁一个无用的或者过期的 bean 实例之前调用。这个注释同时适用于有状态和无状态会话 bean。
- @PrePassivate：当一个有状态的 bean 实例空闲时间过长，就会发生钝化(passivate)。标注了这个注释的方法会在钝化之前被调用。bean 实例被钝化后，在一段时间内，如果仍然没有用户对 bean 实例进行操作，容器将会从硬盘中删除它。以后，任何针对该 bean 方法的调用，容器都会抛出例外。这个注释适用于有状态会话 bean。
- @PostActivate：当客户端再次使用已经被钝化的有状态 bean 时，EJB 容器会重新实例化一个 Bean 实例，并从硬盘中将之前的状态恢复。标注了这个注释的方法会在激活完成时被调用。这个注释只适用于有状态会话 bean。
- @Init：这个注释指定了有状态 bean 初始化的方法。它区别于@PostConstruct 注释在于：多个@Init 注释方法可以同时存在于有状态 session bean 中，但每个 bean 实例只会有一个@Init 注释的方法会被调用。@PostConstruct 在@Init 之后被调用。
- @Remove：当客户端调用标注了@Remove 注释的方法时，容器将在方法执行结束后把 bean 实例删除。

要处理这些生命周期事件，我们可以为这些事件注册一个回调方法，回调方法可以使用任何名称，但是它必须返回 void，不带参数，且不能抛出任何 checked exception。事件注册的过程很简单，只需在回调方法上面加上事件的注释就可以。

LifeCycleBean.java

```
package com.foshanshop.ejb3.impl;
import com.foshanshop.ejb3.LifeCycle;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Init;
```

```
import javax.ejb.PostActivate;
import javax.ejb.PrePassivate;
import javax.ejb.Remote;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful
@Remote (LifeCycle.class)
public class LifeCycleBean implements LifeCycle {

    public String Say() {
        try {
            Thread.sleep(1000*10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "这是会话Bean生命周期应用例子";
    }

    @Init
    public void initialize () {
      System.out.println("@Init事件触发");
    }

    @PostConstruct
    public void Construct () {
      System.out.println("@PostConstruct事件触发");
    }

    @PreDestroy
    public void exit () {
      System.out.println("@PreDestroy事件触发");
    }

    @PrePassivate
    public void serialize () {
      System.out.println("@PrePassivate事件触发");
    }

    @PostActivate
    public void activate () {
      System.out.println("@PostActivate事件触发");
    }
```

```
    @Remove
    public void stopSession () {
      System.out.println("@Remove事件触发");
      //调用该方法以通知容器，移除该bean实例、终止会话。方法体可以是空的。
    }
}
```

业务接口：LifeCycle.java

```
package com.foshanshop.ejb3;
public interface LifeCycle {
    public String Say();
    public void stopSession ();
}
```

JSP 客户端代码：LifeCycleTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.*, javax.naming.*"%>
<%
    try {
        LifeCycle lifecycle = (LifeCycle) session.getAttribute("lifecycle");
        if (lifecycle == null) {
          InitialContext ctx = new InitialContext();
          lifecycle = (LifeCycle) ctx.lookup("LifeCycleBean/remote");
          session.setAttribute ("lifecycle", lifecycle);
        }
        out.println(lifecycle.Say());
        out.println("<BR>请注意观察Jboss控制台输出.等待10分钟,容器将会钝化此会话
Bean,@PrePassivate注释的方法将会执行<BR>");
        out.println("<font color=red>你可以调用stopSession方法把会话Bean实例删除。在删
除会话Bean时，将触发@PreDestroy事件<BR></font>");
        /*
        lifecycle.stopSession();
        */
    } catch (Exception e) {
        out.println(e.getMessage());
    }
%>
```

例子的运行结果输出在 Jboss 控制台，请仔细观察。

本例子的源代码在配套光盘的 SessionBeanLifeCycle 文件夹。要恢复 SessionBeanLifeCycle 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。

本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用到 Jboss，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/LifeCycleTest.jsp 访问客户端。

2.10 拦截器(Interceptor)

拦截器可以拦截 Session bean 和 message-driven bean 的方法调用或生命周期事件。拦截器用于封装应用的公用行为，使这些行为与业务逻辑分离，一旦这些公用行为发生改变，而不必修改很多业务类。拦截器可以是同一 bean 类中的方法或是一个外部类。

下面介绍如何在 Session Bean 类中使用外部拦截器类。

HelloChinaBean.java

```
package com.foshanshop.ejb3.impl;
import com.foshanshop.ejb3.HelloChina;
import com.foshanshop.ejb3.HelloChinaRemote;
import javax.ejb.Local;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.interceptor.Interceptors;

@Stateless
@Remote (HelloChinaRemote.class)
@Local(HelloChina.class)
@Interceptors(HelloInterceptor.class)
public class HelloChinaBean implements HelloChina,HelloChinaRemote {

    public String SayHello(String name) {
        return name +"说：你好!中国.";
    }

    public String Myname() {
        return "我是佛山人";
    }
}
```

@Interceptors 注释指定一个或多个在外部类中定义的拦截器，多个拦截器类之间用逗号分隔，如：@Interceptors({A.class,A.class,B.class})，如果只有一个拦截器可以省略大括号。上面拦截器 HelloInterceptor 对 HelloChinaBean 的所有方法进行拦截。如果你只需对某一方法进行拦截，你可以在方法上面定义拦截器，如：

```
public class HelloChinaBean implements HelloChina,HelloChinaRemote {
    @Interceptors(HelloInterceptor.class)
    public String SayHello(String name) {
        return name +"说：你好!中国.";
    }
}
```

拦截器 HelloInterceptor.java

```
package com.foshanshop.ejb3.impl;
import javax.interceptor.AroundInvoke;
```

```
import javax.interceptor.InvocationContext;

public class HelloInterceptor {

    @AroundInvoke
    public Object log(InvocationContext ctx) throws Exception {
        System.out.println("*** HelloInterceptor intercepting");
        long start = System.currentTimeMillis();
        try{
            if (ctx.getMethod().getName().equals("SayHello")){
                System.out.println("*** SayHello 已经被调用! *** " );
            }
            if (ctx.getMethod().getName().equals("Myname")){
                System.out.println("*** Myname 已经被调用! *** " );
            }
            return ctx.proceed();
        }catch (Exception e) {
            throw e;
        }finally {
            long time = System.currentTimeMillis() - start;
            System.out.println("用时:"+ time + "ms");
        }
    }
}
```

@AroundInvoke 注释指定了要用作拦截器的方法，拦截器方法与被拦截的业务方法执行在同一个 java 调用堆栈、同一个事务和安全上下文中。用@AroundInvoke 注释指定的方法必须遵守以下格式：

public Object XXX(javax.interceptor.InvocationContext ctx) throws Exception

XXX 代表方法名可以任意。javax.interceptor.InvocationContext 封装了客户端所调用业务方法的一些信息。下面是 InvocationContext 的定义：

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

getTarget( ) 指向被调用的 bean 实例

getMethod( ) 指向被拦截的业务方法

getParameters( ) 获取被拦截业务方法的参数

setParameters() 设置被拦截业务方法的参数

getContextData( ) 方法返回一个 Map 对象，它在整个方法调用期间都可以被访问到。位于同一方法调用内的不同拦截器之间可以利用它来传递上下文相关的数据。

在 HelloInterceptor 代码中，我们调用了 ctx.proceed()方法。如果还有其它拦截器未执行，ctx.proceed()方法内部会调用后面拦截器的@AroundInvoke 方法，直到后面的拦截器全部执行结束，EJB 容器才会执行被拦截的业务方法。ctx.proceed()方法必须在拦截器代码中被调用，否则被拦截的业务方法就根本不会被执行。另外如果我们想在被拦截的业务方法执行结束后再执行一些自定义代码，我们可以在 ctx.proceed()执行后方法返回前加入自己的代码，如：

```
    @AroundInvoke
    public Object log(InvocationContext ctx) throws Exception {
        System.out.println("*** HelloInterceptor intercepting");
        long start = System.currentTimeMillis();
        try{
            Object o = ctx.proceed();
            //这里加入你需要执行的代码
            return o;
        }catch (Exception e) {
            throw e;
        }finally {
            long time = System.currentTimeMillis() - start;
            System.out.println("用时:"+ time + "ms");
        }
    }
```

由于拦截器和被拦截的 bean 方法同处于一个 java 调用堆栈中，如果你希望终止方法的执行，你可以抛出异常。现在有这么个需求，如果发现 SayHello()的输入参数为“zhangming”，就终止业务方法的执行，代码如下：

```
public class HelloInterceptor {
    @AroundInvoke
    public Object log(InvocationContext ctx) throws Exception {
        if (ctx.getMethod().getName().equals("SayHello")){
            System.out.println("*** SayHello 已经被调用！ *** " );
            String name = (String)ctx.getParameters( )[0];
            if("zhangming".equals(name)){
             throw new Exception ("就不给zhangming调用,怎么样嘛");
            }
        }
        return ctx.proceed();
    }
}
```

下面是 HelloChinaBean 的本地及远程业务接口

本地接口：HelloChina.java

```
package com.foshanshop.ejb3;
public interface HelloChina extends HelloChinaRemote{
}
```

远程接口：HelloChinaRemote.java

```
package com.foshanshop.ejb3;
public interface HelloChinaRemote {
    public String SayHello(String name);
    public String Myname();
}
```

JSP 客户端代码：InterceptorTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.*, javax.naming.*"%>
<%
    try {
        InitialContext ctx = new InitialContext();
        HelloChinaRemote hellochinaremote = (HelloChinaRemote)
ctx.lookup("HelloChinaBean/remote");
        out.println(hellochinaremote.SayHello("佛山人"));
        out.println("<br>"+ hellochinaremote.Myname());
    } catch (NamingException e) {
        out.println(e.getMessage());
    }
%>
```

除了可以在外部类定义拦截器之外，也可以在 Session Bean 内部，你可以将 bean 的一个或多个方法定义为拦截器。下面以前面的 HelloChinaBean 为例，介绍在 Session Bean 内部如何定义拦截器。

```
package com.foshanshop.ejb3.impl;
import com.foshanshop.ejb3.HelloChina;
import com.foshanshop.ejb3.HelloChinaRemote;
import javax.ejb.Local;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

@Stateless
@Remote (HelloChinaRemote.class)
@Local(HelloChina.class)
public class HelloChinaBean implements HelloChina,HelloChinaRemote {
    public String SayHello(String name) {
        return name +"说：你好!中国.";
    }

    public String Myname() {
        return "我是佛山人";
    }
```

```
    @AroundInvoke
    public Object log(InvocationContext ctx) throws Exception {
        try{
            if (ctx.getMethod().getName().equals("SayHello")){
                System.out.println("*** HelloChinaBean.SayHello() 已经被调用! *** " );
            }
            if (ctx.getMethod().getName().equals("Myname")){
                System.out.println("*** HelloChinaBean.Myname() 已经被调用! *** " );
            }
            return ctx.proceed();
        }catch (Exception e) {
            throw e;
        }
    }
}
```

上面只需一个@AroundInvoke 注释就指定了要用作拦截器的方法。

本例子的源代码在配套光盘的 Interceptor 文件夹。要恢复 Interceptor 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。
本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用到 Jboss，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/InterceptorTest.jsp 访问客户端。

默认拦截器

在一个 EJB 模块中通常有很多 EJB，如果你需要对所有 EJB 的方法进行拦截，为每个 Bean class 加入@Interceptors 注释，显然很烦琐。EJB 为我们提供了使用 ejb-jar.xml 定义拦截器的方式，这种方式支持通配符。下面我们定义对所有 EJB 的方法进行拦截：

禁用拦截器

上面通过<ejb-name>*</ejb-name>拦截所有 EJB，在实际应用中，我们不希望在某些 EJB 上面使用默认拦截器。EJB3.0 为我们提供了@ExcludeDefaultInterceptors 注释，该注释可以禁用默认拦截器。假设，我们已经定义了拦截所有 EJB，但不希望在 HelloChinaBean 上面使用默认拦截器，因此，我们需要在 HelloChinaBean 上面加入@ExcludeDefaultInterceptors 注释，如：

```
@Stateless
@Remote (HelloChinaRemote.class)
@Local(HelloChina.class)
@ExcludeDefaultInterceptors
public class HelloChinaBean implements HelloChina,HelloChinaRemote {
```

```
```

不执行默认的拦截器并不代表不执行自身通过@Interceptors 注释定义的拦截器，如下面代码仍然会执行 ABCInterceptor.class 拦截器。

```
@Stateless
@Remote (HelloChinaRemote.class)
@Local(HelloChina.class)
@ExcludeDefaultInterceptors
@Interceptors({ABCInterceptor.class})
public class HelloChinaBean implements HelloChina,HelloChinaRemote {
```

拦截生命周期事件

拦截器除拦截 EJB 方法调用，还可以拦截 EJB 的生命周期事件。我们可以利用这些事件回调来初始化 bean 类的状态，后面依赖注入章节就利用这一特点定义了一个注入注释。

定义生命周期的拦截方法与@AroundInvoke 拦截方法非常相似：

@<callback-annotation> void <method-name>(InvocationContext ctx)

因为 EJB 的生命周期事件回调方法是没有返回值的，所以拦截方法的返回值必定是 void，方法名可以任意指定，但不能在方法后声明 throws 子句。其它的特性就和@AroundInvoke 拦截方法一样。

下面是例子代码：

```
package com.foshanshop.ejb3.impl;
import javax.annotation.PostConstruct;
import javax.ejb.EJBException;
import javax.interceptor.InvocationContext;

public class HelloInterceptor {
    @PostConstruct
    public void initdata(InvocationContext ctx){
        try{
            ctx.proceed();
        }catch (Exception e) {
            throw new EJBException("Failed to execute @JndiInjected", e);
        }
    }
}
```

拦截器里使用注入

另外在拦截器里面同样可以使用注入(依赖注入在后面章节有详细介绍)，如：

这部分内容在《EJB3.0入门经典》中

2.11 依赖注入(dependency injection)

前面，你学会了如何开发藕合松散的服务组件。在实际应用中，EJB 可能会使用到其它 EJB 或资源。在传统的开发中，我们要使用某个类对象，可以通过 new object 的方式来使用它。但在 EJB 中，你不能这样做，因为 EJB 实例的创建及销毁是由容器管理的。要在 bean 中要用其它 EJB 或资源，你必须通过 JNDI 查找或注入注释。如在 InjectionBean 中使用 HelloBean EJB，你需要在 InjectionBean 中通过 JNDI 查找 HelloBean 的引用：

```
public class InjectionBean implements Injection {
    public String SayHello() {
       InitialContext ctx = new InitialContext();
       LocalHello helloworld =
(LocalHello)ctx.lookup("java:comp/env/ejb/HelloWorld")
        return helloworld.SayHello("注入者");
    }
}
```

或者通过注入注释:

```
@Stateless
@Remote (Injection.class)
public class InjectionBean implements Injection {
    @EJB (beanName="HelloBean") LocalHello helloworld;
    public String SayHello() {
        return helloworld.SayHello("注入者");
    }
}
```

而千万不能这样做：

```
@Stateless
public class InjectionBean implements Injection {
    public String SayHello() {
        LocalHello helloworld = new HelloBean();
        return helloworld.SayHello("注入者");
    }
}
```

上面通过查询 java:comp/env/ejb/HelloWorld 获取 HelloBean 的本地引用，这种方式需要在查询执行前先往 JNDI ENC 添加一个名为“ejb/HelloWorld”的指向 HelloBean 本地引用的注册项。虽然通过 ctx.lookup("HelloBean/local") 也能获得 HelloBean 的本地引用，但这种方式只能在 jboss 中使用，其它应用服务器并不支持。

什么是 ENC？怎样向 ENC 添加一个注册项？
应用服务器中的 EJB 容器有一个属于它自己的内部注册表（internal registry），这个内部注册表被称为 Enterprise Naming Context(ENC)，EJB 容器可以在其中维护某些指向外部环境的引用，我们可以把它想象是 EJB 容器的私人地址簿。先在 ENC 登记资源，然后再从 ENC 中获取资源的引用。要往 ENC 添加一个指向资源引用的注册项，我们可以通过 ejb-jar.xml（用于 EJB 模块）、web.xml（用于 WEB 模块）或注入注释。

通过 ejb-jar.xml 添加注册项，(该文件需放在 jar 的 META-INF)：

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                          http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
      version="3.0">
   <enterprise-beans>
      <session>
        <ejb-name>InjectionBean</ejb-name>
        <ejb-local-ref>
           <ejb-ref-name>ejb/HelloWorld</ejb-ref-name>
           <ejb-ref-type>Session</ejb-ref-type>
           <local>com.foshanshop.ejb3.LocalHello</local>
           <ejb-link>HelloBean</ejb-link>
        </ejb-local-ref>
       <!-- 以下是配置远程引用例子
        <ejb-ref>
            <ejb-ref-name>ejb/HelloWorldBean</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <remote>com.foshanshop.ejb3.RemoteHello</remote>
            <ejb-link>HelloBean</ejb-link>
        </ejb-ref>
        -->
      </session>
   </enterprise-beans>
</ejb-jar>
```

<ejb-local-ref>元素在 InjectionBean 的 JNDI ENC 中注册指向 HelloBean 的 EJB 引用，注册名称为 ejb/HelloWorld。

通过注释添加注册项：

```
@Stateless
@EJB (name="ejb/HelloWorld",
beanName="HelloBean",beanInterface=LocalHello.class)
public class InjectionBean implements Injection {
}
```

下面是注释与 ejb-jar.xml 的对应图

ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
       xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                           http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
       version="3.0">
<ejb-jar>
   <enterprise-beans>
      <session>
         <ejb-name>InjectionBean</ejb-name>
          <ejb-ref>
            <ejb-ref-name>ejb/HelloWorld</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <remote>com.foshanshop.ejb3.HelloWorld</remote>
            <ejb-link>HelloWorldBean</ejb-link>
          </ejb-ref>
      </session>
   </enterprise-beans>
</ejb-jar>
```

```
@EJB(
     name = "ejb/HelloWorld",//ejb引用名称
     beanName = "HelloWorldBean", //被调用EJB的名称
     beanInterface = HelloWorld.class, //接口名称
     mappedName="HelloWorldBean/remote"
)
```

如何从 ENC 中获取资源的引用？

要从 ENC 中获取资源的引用，我们可以通过注册项名称进行 JDNI 查找，如：LocalHello helloworld = (LocalHello)ctx.lookup("java:comp/env/ejb/HelloWorld")，comp 代表组件，java:comp/env 指向该 EJB 的 ENC，ejb/HelloWorld 是在 ENC 中定义的注册项名称。大家需要注意，根据调用 lookup()位置的不同，“java:comp/env”会被解析成不同的上下文环境，例如，在 EJB1 调用 ctx.lookup("java:comp/env")获取的是 EJB1 的 ENC，在 EJB2 调用 ctx.lookup("java:comp/env")获取的是 EJB2 的 ENC。

为了使用一个 EJB，我们需要往 ENC 添加一个注册项，然后使用 lookup()进行查找，这显然太麻烦了。而且使用一个字符串进行 JNDI 查找并不优雅。是否还有更简单的方式使用 EJB 或资源？答案是有的，EJB 3.0 提供了环境用注释(environment annotation)，只要把环境用注释标注到字段或属性的 setter 方法上，EJB 容器就会在 bean 实例被创建时，自动往 bean 的 ENC 中添加一个注册项，同时把 EJB 的引用直接注入进字段或属性的 setter 方法。依赖注入只工作在本地命名服务中，因此你不能注入远程服务器的对象。

下面例子显示了怎样把 HelloBean EJB 的本地引用注入到 InjectionBean 中。

Remote 业务接口：Injection.java

```
package com.foshanshop.ejb3;
public interface Injection {
    public String SayHello();
}
```

Bean 类：InjectionBean.java

```
package com.foshanshop.ejb3.impl;
import com.foshanshop.ejb3.Injection;
```

```
import com.foshanshop.ejb3.LocalHello;
import javax.ejb.EJB;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote (Injection.class)
public class InjectionBean implements Injection {
    @EJB (beanName="HelloBean") LocalHello helloworld;
    public String SayHello() {
        return helloworld.SayHello("注入者");
    }
}
```

JSP 客户端代码：InjectionTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.*, javax.naming.*"%>
<%
    try {
        InitialContext ctx = new InitialContext();
        Injection injection = (Injection) ctx.lookup("InjectionBean/remote");
        out.println(injection.SayHello());
    } catch (NamingException e) {
        out.println(e.getMessage());
    }
%>
```

通过为 helloworld 成员字段标注@EJB 注释，EJB 容器会在 bean 实例被创建时，把 HelloBean 的本地引用注入进 helloworld 成员字段。下面是@EJB 注释的定义:

```
package javax.ejb;
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface EJB {
    String name( ) default "";
    Class beanInterface( ) default Object.class;
    String beanName( ) default "";
    String mappedName( ) default "";
}
```

name() 属性表示被引用的 EJB 的 JNDI ENC 注册项名称。该名称相对于 java:comp/env 上下文。

beanInterface( )属性指定使用的 bean 接口。该属性通常被容器用来区分所使用的 EJB 引用是远程还是本地。如果@EJB 用在 bean class 上面，必须指定该属性值，如果@EJB 用在成员字段或属性的 setter 方法上，可以省略。

beanName( )属性指定被引用 EJB 的名称，其值与@Stateless.name()、@Stateful.name()或 ejb-jar.xml 中<ejb-name>元素所指定的值相等。

mappedName( )属性表示 EJB 的全局 JNDI 名，而全局 JNDI 名是与容器厂商有关的，设置该属性值将不利于移植。

@EJB 注释工作时，会自动引发容器在 JNDI ENC 中为被注入的元素创建一个注册项，这不仅对@EJB 注释适用，对其它环境用注释也同样适用。注册项的名称由 name() 属性指定，如果没有指定名称，则容器会根据被标注的成员字段或属性的 setter 方法的名称提取出 ENC 名。在这种情况下，默认的 ENC 名称由成员字段或属性的 setter 方法所在类的全限定名和字段名或 setter 方法的基础名组成。例如，对前面例子中的 helloworld 而言，默认的 ENC 名称为 com.foshanshop.ejb3.impl.InjectionBean/helloworld。注册项引用的接口由 beanInterface( )属性指定，如果没有指定接口，容器会使用字段类型或 setter 方法的参数类型。例如，对前面例子中的 helloworld 而言，引用的接口为 com.foshanshop.ejb3.LocalHelloWorld。注册项引用的 EJB 名称由 beanName( )属性指定。

@EJB 注释的 name()、beanName( )、mappedName( )属性使用场合：

这部分内容在《EJB3.0入门经典》中

@EJB 注释除了可以标注字段，还可以标识属性的 setter 方法。容器在 bean 实例创建时，把 EJB 引用作为参数注入进去。如：

```
@Stateless
@Remote (Injection.class)
public class InjectionBean implements Injection {
    private LocalHello hello;
    @EJB (beanName="HelloManBean")
    public void setHello(LocalHello hello) {
        this.hello = hello;
    }
```

本例子的源代码在配套光盘的 DependencyInjection 文件夹。要恢复 DependencyInjection 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。
本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用到 Jboss，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/InjectionTest.jsp 访问客户端。

小知识点：
如果被注入的 EJB 与使用者不在同一个 jar，如使用者在 DependencyInjection.jar，被注入 EJB 在 HelloWorld.jar。Jboss 启动时，会按默认的发布顺序先发布 DependencyInjection.jar，后发布 HelloWorld.jar，因为 DependencyInjection.jar 中的文件使用到了 HelloWorld.jar 中的 EJB，所以会抛出找不到类的例外，导致 DependencyInjection.jar 发布失败。要解决这个问题，大家可以这样做：
在[Jboss 安装目录]/ server/default/conf 文件夹中找到 jboss-service.xml 文件，打开文件并找到：

```
<attribute name="URLComparator">org.jboss.deployment.DeploymentSorter</attribute>
<!--
<attribute name="URLComparator">org.jboss.deployment.scanner.PrefixDeploymentSorter</attribute>
-->
```

修改成：

```
<!—
```

```
<attribute name="URLComparator"> org.jboss.deployment.DeploymentSorter</attribute>
-->
<attribute name="URLComparator">org.jboss.deployment.scanner.PrefixDeploymentSorter</attribute>
```

给 jar 文件编个号，格式为：01_XXX.jar
如：01_HelloWorld.jar ，02_DependencyInjection.jar
Jboss 将根据编号按从小到大的顺序发布 jar 文件。

2.11.1 资源类型的注入

前面我们知道了怎样通过@EJB 注释注入 EJB 引用。很多时候我们还需注入其它外部资源，外部资源的类型可以是 javax.sql.DataSource, javax.jms.ConnectionFactory, javax.jms.QueueConnectionFactory, javax.jms.TopicConnectionFactory, javax.mail.Session, java.net.URL 或者 javax.resource.cci.ConnectionFactory，以及任何由 JCA 资源适配器定义的其他类型。要注入外部资源，我们可以使用@javax.annotation.Resource 注释。其工作过程与@EJB 相同，先往 JNDI ENC 添加一个指向资源引用的注册项，然后把资源引用注入到成员字段或属性的 setter 方法中。

下面例子显示了如何注入数据源。“DefaultMySqlDS”是数据源的局部 JNDI 名称，只供 Jboss 容器内的应用访问。查找该局部 JNDI 名称时，需要带有前缀“java:/”。关于数据源的配置请参考后面章节内容。

```
package com.foshanshop.ejb3.impl;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import com.foshanshop.ejb3.Injection;
import javax.annotation.Resource;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.sql.DataSource;

@Stateless
@Remote (Injection.class)
public class InjectionBean implements Injection {
    @Resource(mappedName = "java:/DefaultMySqlDS") DataSource myDb;

    public String SayHello() {
        String str = "";
        Connection conn = null;
        try {
            conn = myDb.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT studentName FROM student");
            if (rs.next()) {
                str = rs.getString(1);
```

```
            }
            rs.close();
            stmt.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
            try {
                if(null!=conn && !conn.isClosed()) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return str;
    }
}
```

EJB 容器使用@Resource.mappedName()属性值进行 JNDI 查找，并把查找到的数据源引用注入进 myDb。@Resource 注释的定义如下：

```
package javax.annotation;
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface Resource {
    public enum AuthenticationType {
        CONTAINER,
        APPLICATION
    }
    String name() default "";
    Class type() default Object.class;
    AuthenticationType authenticationType() default AuthenticationType.CONTAINER;
    boolean shareable() default true;
    String description() default "";
    String mappedName() default "";
}
```

name ()属性指定被引用的外部资源的 JNDI ENC 名。该名称是相对于 java:comp/env 上下文的。

type()属性指定资源对应的 Java 全限定类名。如：@Resource(mappedName="java:/DefaultMySqlDS", type=DataSource.class)

mappedName()属性指定资源的 JNDI 名，该值与厂商有关。

shareable()属性指定资源是否共享，默认为 true。当多个 EJB 要在同一事务中使用相同资源时，该属性决定了 EJB 是否通过同样的连接访问同一资源，如：对于数据库而言，引用同一数据库的多个 EJB 希望在同一个事务中使用相同的数据库连接，那么 shareable()属性应设为 true。

authenticationType()属性告知容器，在对资源进行访问时由谁负责权限认证。其取值可以是：CONTAINER 或 APPLICATION，默认值为 CONTAINER。如果指定了 CONTAINER，则容器将自动执行访问资源所需的认证操作。如果指定了 APPLICATION，则 bean class 需要在使用资源之前自行认证。如：

```
@Stateless
@Remote (Injection.class)
```

```
public class InjectionBean implements Injection {
    @Resource(mappedName = "java:/DefaultMySqlDS",
authenticationType=AuthenticationType.APPLICATION) DataSource myDb;

    public String SayHello() {
        String str = "";
        Connection conn = null;
        try {
            conn = myDb.getConnection("root","123456");
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT studentName FROM student");
            if (rs.next()) {
                str = rs.getString(1);
            }
            rs.close();
            stmt.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
            try {
                if(null!=conn && !conn.isClosed()) conn.close();
            } catch (SQLException e) {
             e.printStackTrace();
            }
        }
        return str;
    }
}
```

除了可以注入 javax.sql.DataSource 类型的资源之外，@Resource 注释还可以注入资源环境注册项(Resource environment entry)所引用的对象，如：javax.transaction.UserTransaction 和 javax.transaction.TransactionSynchronizationRegistry。如：

```
@Stateless
@Remote (Injection.class)
public class InjectionBean implements Injection {
    @Resource private javax.transaction.UserTransaction utx;
    @Resource private javax.ejb.TimerService tms;
    @Resource private javax.ejb.SessionContext ctx;
    …
}
```

当注入这些服务类型的引用时，无需指定 shareable()和 authenticationType()属性，如果你指定了这些属性，会被视为非法。

@Resource 注释还可以注入消息连接工厂和消息目标地址，如：

```
@Resource(mappedName="QueueConnectionFactory")
private javax.jms.QueueConnectionFactory factory;

@Resource(mappedName="queue/mail")
private javax.jms.Queue queue;
```

2.11.2 注入与继承关系

Bean class 有可能处于某个类继承结构中。如果父类的字段或属性的 setter 方法标有注入注释，则这些字段或属性会在子类中依据一定的注入规则接受注入：

```
@Stateless
@Remote (Injection.class)
public class InjectionBean implements Injection {
    private LocalHello helloworld;

    @EJB (beanName="HelloBean")
    public void setHelloworld(LocalHello helloworld) {
        this.helloworld = helloworld;
    }
}
```

```
@Stateless
@Remote (Injection.class)
public class SubInjectionBean extends InjectionBean {
}
```

被注入到基类 setHelloworld()方法的 HelloBean EJB 会被 SubInjectionBean 的实例所继承。我们可以通过在子类重载 setHelloworld()方法，来更改注入的资源。如：

```
@Stateless
@Remote (Injection.class)
public class SubInjectionBean extends InjectionBean {
    @Override
    @EJB (beanName="HelloManBean")
    public void setHelloworld(LocalHello helloworld) {
        super.setHelloworld(helloworld);
    }
}
```

HelloBean 将不再被注入到子类 SubInjectionBean 的 setHelloworld()方法中，取而代之的是 HelloManBean，如果基类 InjectionBean 中的 setHelloworld()方法是 private，而非 protected 或 public，则 HelloBean 仍会被注入到基类中。

2.11.3 自定义注入注释

如果你觉的系统提供的注入注释不能满足你的需要，或者你想玩玩花样，那么你可以自定义注释。

2.12 定时服务(Timer Service)

定时服务用作在一段特定的时间后执行某段程序，估计大家在不同的场合中已经使用过。定时服务可以用在 stateless session bean 和 message-driven bean，当某个 stateless session bean 或 message-driven bean 的定时器启动时，容器会从实例池中选择 bean 的一个实例，然后调用其 timeout 回调方法。你可以使用@Resource 注释注入定时服务，或者使用容器对象 SessionContext 创建定时器。

下面例子在 session bean 中定义了一个定时器，定时器每隔 3 秒钟触发一次事件，当定时事件触发次数超过 5 次的时候便终止定时器的执行。

TimerServiceBean.java

```
package com.foshanshop.ejb3.impl;
import java.util.Date;
import com.foshanshop.ejb3.TimerServiceDAO;
import javax.annotation.Resource;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerService;

@Stateless
@Remote (TimerServiceDAO.class)
public class TimerServiceBean implements TimerServiceDAO {
    private static int count = 0;
    @Resource private TimerService timerService; //直接注入定时服务

    public void scheduleTimer(long milliseconds){
        if(count ==0 ){
            count = 1;
            timerService.createTimer(new Date(new Date().getTime() + milliseconds),
                milliseconds, "大家好，这是我的第一个定时器");
        }
    }
```

```
    @Timeout
    public void timeoutHandler(Timer timer){
       System.out.println("---------第 "+ count +" 次------------");
       System.out.println("定时器事件发生,传进的参数为: " + timer.getInfo());

       if (count>=5){
           timer.cancel();//如果定时器触发5次，便终止定时器
           count = 0;
       }else{
          count++;
       }
    }
}
```

通过注入@Resource private TimerService timerService，我们获得了 TimerService 对象，调用 TimerService.createTimer (Date arg0, long arg1, Serializable arg2)方法创建定时器，三个参数的含义如下：
Date arg0 定时器启动时间，如果传入时间小于现在时间，定时器会立刻启动。
long arg1 间隔多长时间后再次触发定时事件。单位：毫秒
Serializable arg2 传给定时器的参数，该参数必须实现 Serializable 接口。

当定时器创建完成后，我们还需要添加定时事件的回调方法。回调方法使用@javax.ejb.Timeout 注释标注，必须返回 void，并接受一个 javax.ejb.Timer 类型的参数，回调方法声明的格式如下：
void XXX(Timer timer)
在定时事件发生时，此方法将被执行。

下面是 TimerServiceBean 的 Remote 业务接口：TimerServiceDAO.java

```
package com.foshanshop.ejb3;
public interface TimerServiceDAO {
    public void scheduleTimer(long milliseconds);
}
```

JSP 客户端代码：TimerServiceTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.*, javax.naming.*"%>
<%
    try {
        InitialContext ctx = new InitialContext();
        TimerServiceDAO timer = (TimerServiceDAO)
ctx.lookup("TimerServiceBean/remote");
        timer.scheduleTimer((long)3000);
        out.println("定时器已经启动，请观察Jboss控制台输出，如果定时器触发5次，便终止定时器
");
    } catch (Exception e) {
        out.println(e.getMessage());
```

```
    }
%>
```

本例子的源代码在配套光盘的 TimerService 文件夹。要恢复 TimerService 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。
本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用到 Jboss，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/TimerServiceTest.jsp 访问客户端。

2.13 安全服务(Security service)

当公司网络外部的用户需要访问你编写的应用程序或受安全保护的资源时，很多 Java 程序员为他们的应用程序创建自己的安全模块。大多数这些模块都是针对具体的应用程序，这样在下一个应用程序需要安全保护时，程序员又重新开始所有工作。构建自己的安全模块的另外一个缺点是，在应用程序变得越来越复杂时，安全需求也会变得很复杂。

使用 Java 验证和授权服务（JAAS）可以很好地解决上面的问题，你可以用它来管理应用程序的安全性。JAAS 具有两个特性：验证（Authentication）和授权（authorization）。

验证（Authentication）
认证是完成用户名和密码的匹配校验；其校验的对象是试图访问受保护系统的用户。在进行校验时，应用服务器会检查用户是否存在于系统之中，以及是否提供了凭证（通常指密码）。

授权（authorization）
用户一旦通过了系统验证，需要与系统进行某种形式的交互。授权就是决定用户是否有权执行某项操作的过程，授权是基于角色的。

Jboss 服务器提供了安全服务来进行用户认证和根据用户规则来限制对 POJO 的访问。对每一个 POJO 来说，你可以使用@SecurityDomain 注释为它指定一个安全域，安全域告诉容器到哪里去找密码和用户角色列表。JBoss 中的 other 域指明要到 classpath 下寻找 users.propertes 和 roles.properties。这样，对每一个方法来说，我们可以使用一个安全限制注释来指定谁可以运行这个方法。比如，下面的例子，容器对所有试图调用 AdminUserMethod()的用户进行认证，只允许拥有 AdminUser 角色的用户运行它。如果你没有登录或者没有以管理员的身份登录，一个安全意外将会抛出。

本例定义了三种角色，各角色的含义如下：

角色名称	说明
AdminUser	系统管理员角色
DepartmentUser	各事业部用户角色
CooperateUser	公司合作伙伴角色

本例设置了三个用户，各用户名及密码如下：

用户名	密码
lihuoming	123456
zhangfeng	111111

wuxiao	123

本例使用 Jboss 默认的安全域“other”，“other”安全域告诉容器到 classpath 下的 users.properties 和 roles.properties 中寻找密码和用户角色列表。“other”安全域定义在[jboss 安装目录]/server/default/conf/login-config.xml 文件中。内容如下：

```
<application-policy name = "other">
    <!-- A simple server login module, which can be used when the number
        of users is relatively small. It uses two properties files:
        users.properties, which holds users (key) and their password (value).
       roles.properties, which holds users (key) and a comma-separated list of
        their roles (value).
        The unauthenticatedIdentity property defines the name of the principal
        that will be used when a null username and password are presented as is
        the case for an unuathenticated web client or MDB. If you want to
        allow such users to be authenticated add the property, e.g.,
        unauthenticatedIdentity="nobody"
    -->
     <authentication>
          <login-module code = "org.jboss.security.auth.spi.UsersRolesLoginModule"
                flag = "required" />
     </authentication>
</application-policy>
```

“other”安全域默认情况下是不允许匿名用户访问的(即必须进行登录，才可访问)，如果你想让匿名用户能够访问@PermitAll 注释定义的资源，需要修改“other”安全域配置，修改片断如下(粗体部分)：

```
<application-policy name = "other">
    <authentication>
          <login-module code = "org.jboss.security.auth.spi.UsersRolesLoginModule"
                flag = "required" />
                <module-option name = "unauthenticatedIdentity">AnonymousUser</module-option>
    </authentication>
</application-policy>
```

下面我们开始安全服务的具体开发：
第一步，定义安全域，安全域的定义有两种方法：
第一种方法：通过 Jboss 部署描述文件 jboss.xml 进行定义(本例采用的方法)，定义内容如下：
jboss.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <!--  Bug in EJB3 of JBoss 4.0.4 GA
  <security-domain>java:/jaas/other</security-domain>
  -->
  <security-domain>other</security-domain>
  <unauthenticated-principal>AnonymousUser</unauthenticated-principal>
```

```
</jboss>
```

<security-domain>指定我们使用的安全域是“other”，
<unauthenticated-principal>AnonymousUser</unauthenticated-principal>节点指定允许匿名用户访问。jboss.xml 必须放进 Jar 文件的 META-INF 目录。

第二种方法：通过@SecurityDomain 注释进行定义，注释代码片断如下：

```
import org.jboss.annotation.security.SecurityDomain;

@Stateless
@Remote ({SecurityAccess.class})
@SecurityDomain("other")
public class SecurityAccessBean implements SecurityAccess{
}
```

由于程序通过硬编码方式使用 Jboss 安全注释，不利于移植，所以不建议使用这种方法定义安全域。

第二步，定义用户名，密码及用户的角色。用户名和密码定义在 users.properties 文件，用户所属角色定义在 roles.properties 文件。下面是这两个文件的具体配置：
users.properties（定义了本例使用的三个用户）

```
lihuoming=123456
zhangfeng=111111
wuxiao=123
```

roles.properties（定义了三个用户所具有的角色，其中用户 lihuoming 具有三种角色）

```
lihuoming=AdminUser,DepartmentUser,CooperateUser
zhangfeng=DepartmentUser
wuxiao=CooperateUser
```

按照“other”域的要求，以上这两个文件需放在类路径下。在进行用户验证时，Jboss 容器会自动在类路径下寻找这两个文件。

第三步，为业务方法定义访问角色。本例定义了三个方法：AdminUserMethod()、 DepartmentUserMethod()、AnonymousUserMethod()，第一个方法允许 AdminUser 角色的用户访问，第二个方法允许 DepartmentUser 角色的用户访问，第三个方法允许任何角色的用户访问。下面是 Session Bean 代码。
SecurityAccessBean.java

```
package com.foshanshop.ejb3.impl;
import com.foshanshop.ejb3.SecurityAccess;
import javax.annotation.security.PermitAll;
import javax.annotation.security.RolesAllowed;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote ({SecurityAccess.class})
public class SecurityAccessBean implements SecurityAccess{
```

```
    @RolesAllowed({"AdminUser"})
    public String AdminUserMethod() {
        return "具有管理员角色的用户才可以访问AdminUserMethod()方法";
    }

    @RolesAllowed({"DepartmentUser"})
    public String DepartmentUserMethod() {
        return "具有事业部门角色的用户才可以访问DepartmentUserMethod()方法";
    }

    @PermitAll
    public String AnonymousUserMethod() {
        return "任何角色的用户都可以访问AnonymousUserMethod()方法";
    }
}
```

@RolesAllowed 注释指定允许访问方法的角色列表，如果角色存在多个，可以用逗号分隔。@PermitAll 注释指定任何角色都可以访问此方法。

下面是 SecurityAccessBean 的 Remote 接口：SecurityAccess.java

```
package com.foshanshop.ejb3;
public interface SecurityAccess {
    public String AdminUserMethod();
    public String DepartmentUserMethod();
    public String AnonymousUserMethod();
}
```

经过上面的步骤，一个安全服务例子就开发完成。下面是例子打包后的目录结构：

```
SecurityAccess.jar
      |-- com/**/*.class
      +- META-INF
      |     |-- jboss.xml
      |-- users.properties
      |-- roles.properties
```

下面我们看看客户端如何访问上面具有安全服务的 EJB。

SecurityAccessTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.SecurityAccess,
                javax.naming.*,
                java.util.*"%>
<%
    Properties props = new Properties();
```

```
    props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.security.jndi.JndiLoginInitialContextFactory");
    props.setProperty(Context.PROVIDER_URL, "localhost:1099");
    props.setProperty(Context.SECURITY_PRINCIPAL, "");
    props.setProperty(Context.SECURITY_CREDENTIALS, "");
    String user = request.getParameter("user");
    String pwd = request.getParameter("pwd");
    if (user!=null && !"".equals(user.trim())){
        props.setProperty(Context.SECURITY_PRINCIPAL, user);
        props.setProperty(Context.SECURITY_CREDENTIALS, pwd);
    }
    InitialContext ctx = new InitialContext(props);
    SecurityAccess securityaccess = (SecurityAccess)
ctx.lookup("SecurityAccessBean/remote");
    try{
        out.println("<font color=green>调用结果:</font>"+
securityaccess.AdminUserMethod()+ "<br>");
    }catch(Exception e){
        out.println(user+ "没有权限访问AdminUserMethod方法<BR>");
    }

    out.println("============================<BR>");
    try{
        out.println("<font color=green>调用结果:</font>"+
securityaccess.DepartmentUserMethod()+ "<br>");
    }catch(Exception e){
        out.println(user+ "没有权限访问DepartmentUserMethod方法<BR>");
    }

    out.println("============================<BR>");
    try{
        out.println("<font color=green>调用结果:</font>"+
securityaccess.AnonymousUserMethod()+ "<br>");
    }catch(Exception e){
        out.println(user+ "没有权限访问AnonymousUserMethod方法<BR>");
    }
%>

<html>
  <head>
    <title>安全访问测试</title>
  </head>

  <body>
```

```
    <center><h2>安全访问测试</h2></center>
    <br />
    请输入你的用户名及密码
    <br />
    <form method="POST" action="SecurityAccessTest.jsp">
       Username: <input type="text" name="user"/>
       <br />
       Password: <input type="password" name="pwd"/>
       <br />
       <input type="submit" value="身份验证"/>
    </form>
    <p>
      管理员  用户名: <STRONG>lihuoming</STRONG>  密码:
<STRONG>123456</STRONG>
    </p>
    <p>事业部  用户名: <STRONG>zhangfeng</STRONG>  密码
<STRONG>111111</STRONG></p>
    <p>合作伙伴  用户名: <STRONG>wuxiao</STRONG>  密码
<STRONG>123</STRONG></p>
  </body>
</html>
```

因为 Jboss 是通过 JNDI API 来完成用户身份的验证，所以需要把用户名设置在 Context.SECURITY_PRINCIPAL，密码设置在 Context.SECURITY_CREDENTIALS。这些验证信息是在客户端与 EJB 服务器建立 JNDI 连接时传递的。这和通过 JDBC API 访问数据库，设置用户名及密码的道理一样。如果在 J2SE 中访问具有安全服务的 EJB，你同样需要这样做：

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.security.jndi.JndiLoginInitialContextFactory");
props.setProperty(Context.PROVIDER_URL, "localhost:1099");
props.setProperty(Context.SECURITY_PRINCIPAL, "lihuoming");
props.setProperty(Context.SECURITY_CREDENTIALS, "123456");
InitialContext ctx = new InitialContext(props);
SecurityAccess securityaccess = (SecurityAccess)
ctx.lookup("SecurityAccessBean/remote");
securityaccess.AdminUserMethod();
```

通常在实际项目中，大部分客户端都是 web 应用。web 应用可以使用安全域对资源进行访问控制。下面的 web 应用与本例 EJB 共同使用一个安全域，如果用户在 web 端验证通过，将使用 web 端的角色访问 EJB。

因为使用的 web 文件较多，下面列出本例 web 客户端打包后的目录结构

```
JaasTest.war
    |
    | -- index.jsp (主页)
```

```
| -- login.html (登录页)
| -- loginFailed.html  (登录失败页)
| -- logout.jsp (登录退出页)
| -- notAuthenticated.html  (角色验证失败页)
+-- admin
|    -- adminPage.jsp  (管理员角色操作页)
+-- anon
|    -- anonymousPage.jsp  （任何用户角色操作页)
+-- includes
|    -- menubar.jsp  (菜单)
+-user
|    -- departmentUser.jsp (部门角色操作页)
+-WEB-INF
|    +--classes
|         |-- users.properties
|         |-- roles.properties
|    -- jboss-web.xml
|    -- web.xml
```

首先把前面在 EJB 中定义的 users.properties 和 roles.properties 文件拷贝到 Web 类路径下（WEB-INF/classes 目录）。如果这两个文件没有放在 web 类路径下，web 的 ClassLoader 找不到文件后将交由 EJB 的 ClassLoader 负责寻找，EJB 的 ClassLoader 有可能找到其它 EJB-JAR 中的文件，所以最好在 WEB-INF/classes 目录下放置这两个文件。

为了使用容器的安全服务，我们需要在 jboss-web.xml 中定义使用的安全域(例子使用 other 域)，该文件放置在 WEB-INF 目录下

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC
"-//JBoss//DTD Web Application 2.3V2//EN"
"http://www.jboss.org/j2ee/dtd/jboss-web_3_2.dtd">
<jboss-web>
    <security-domain>java:/jaas/other</security-domain>
</jboss-web>
```

然后在 web 应用的 web.xml 里定义验证模块及对某些 URL 进行权限设置

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
   xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
      "http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

   <display-name>JaasTests</display-name>
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
```

```
</welcome-file-list>
  <!—角色验证不通过时的处理 -->
 <error-page>
    <error-code>403</error-code>
    <location>/notAuthenticated.html</location>
 </error-page>

 <!--  下面设置以/user/开头的路径只允许DepartmentUser角色访问-->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Protected Pages</web-resource-name>
    <url-pattern>/user/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
    <auth-constraint>
        <role-name>DepartmentUser</role-name>
    </auth-constraint>

    <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
<!--  End DepartmentUser user allowed URL's -->

<!--  下面设置以/admin/开头的路径只允许AdminUser角色访问-->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Protected Pages</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
    <auth-constraint>
        <role-name>AdminUser</role-name>
    </auth-constraint>

    <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
<!--  End Admin user allowed URL's -->

 <security-role>
```

```
        <description>Authorized to access everything.</description>
        <role-name>AdminUser</role-name>
    </security-role>
    <security-role>
        <description>Authorized to limited access.</description>
        <role-name>DepartmentUser</role-name>
    </security-role>
     <!-- 下面设置登录配置，登录验证由容器负责处理 -->
    <login-config>
        <auth-method>FORM</auth-method>
        <form-login-config>
            <form-login-page>/login.html</form-login-page>
            <form-error-page>/loginFailed.html</form-error-page>
        </form-login-config>
    </login-config>
</web-app>
```

配置工作完成了，接下来我们开始页面开发。根据 web.xml 的配置，我们要开发 login.html，loginFailed.html，notAuthenticated.html 这三个文件，三个文件的代码如下：

login.html

```
  <body>
    <center><h2>Jaas Tests Login</h2></center>
    <br />
    请输入你的用户名及密码
    <br />
    <form method="POST" action="j_security_check">
        用户名: <input type="text" name="j_username"/>
        <br />
        密码: <input type="password" name="j_password"/>
        <br />
        <input type="submit" value="登录"/>
    </form>
    <p>管理员角色  用户名: <STRONG>lihuoming</STRONG>  密码: <STRONG>123456</STRONG>
</p>
    <p>事业部角色  用户名: <STRONG>zhangfeng</STRONG>  密码
<STRONG>111111</STRONG></p>
    <p>合作伙伴角色  用户名: <STRONG>wuxiao</STRONG>  密码 <STRONG>123</STRONG></p>
  </body>
```

上面粗体部分是 Jaas 规范定义的，表单的 action 必须为 j_security_check，用户名字段为 j_username，密码字段为 j_password，表单提交后 Jaas 服务接受请求并处理，如果用户验证失败，页面导向<form-error-page>指定的页面。

loginFailed.html 页面代码

```
<html>
  <head>
```

```
  <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
  <title>Jaas tests 用户登录失败</title>
  </head>

  <body>
    <center><h2><FONT COLOR="red">登录失败</FONT></h2></center>
    <br />
    请输入你的用户名及密码
    <br />
    <form method="POST" action="j_security_check">
       用户名: <input type="text" name="j_username"/>
       <br />
       密码: <input type="password" name="j_password"/>
       <br />
       <input type="submit" value="Log me in"/>
    </form>

    <p>管理员角色  用户名: <STRONG>lihuoming</STRONG>  密码:
<STRONG>123456</STRONG>    </p>
    <p>事业部角色  用户名: <STRONG>zhangfeng</STRONG>  密码
<STRONG>111111</STRONG></p>
    <p>合作伙伴角色  用户名: <STRONG>wuxiao</STRONG>  密码
<STRONG>123</STRONG></p>
  </body>
</html>
```

notAuthenticated.html 页面代码

```
<html>
  <head>
  <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
  <title>Jaas tests 你没有访问权限呀!</title>
  </head>

  <body>
    <center><h2><FONT COLOR="red">你没有访问权限呀!</FONT></h2></center>
  </body>
</html>
```

为了测试的需要，我们定义 3 种角色访问的页面(anonymousPage.jsp, departmentUser.jsp, adminPage.jsp)。下面是 adminPage.jsp 文件的代码：

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.SecurityAccess,
                 javax.naming.*,
                 org.jboss.security.*,
```

```
                java.util.*"%>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
  <title>Jaas tests</title>
</head>

<body>
<H3>Jaas Tests</H3>
<P>
    这个页面只允许管理员角色访问:<br>
<%
  InitialContext ctx = new InitialContext();
  SecurityAccess securityaccess = (SecurityAccess) ctx.lookup("SecurityAccessBean/remote");
  try{
    out.println("<font color=green>调用结果:</font>"+ securityaccess.AdminUserMethod()+ "<br>");
  }catch(Exception e){
        out.println("访问AdminUserMethod方法出错");
  }
%>
</P>
</body>
</html>
```

下面是 departmentUser.jsp 文件的代码：

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.SecurityAccess,
               javax.naming.*,
               org.jboss.security.*,
               java.util.*"%>

<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
  <title>Jaas tests</title>
</head>

<body>
<H3>Jaas Tests</H3>
<P>
    这个页面只允许部门角色访问:<br>
<%
  InitialContext ctx = new InitialContext();
  SecurityAccess securityaccess = (SecurityAccess)
```

```
ctx.lookup("SecurityAccessBean/remote");
   try{
    out.println("<font color=green>调用结果:</font>"+
securityaccess.DepartmentUserMethod()+ "<br>");
   }catch(Exception e){
    out.println("访问DepartmentUserMethod方法出错");
   }
%>
</P>
</body>
</html>
```

下面是 anonymousPage.jsp 文件的代码：

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.SecurityAccess,
              javax.naming.*,
              org.jboss.security.*,
              java.util.*"%>

<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
  <title>Jaas tests</title>
</head>

<body>
<H3>Jaas Tests</H3>
<P>
    这个页面允许匿名用户访问:<br>
<%
  InitialContext ctx = new InitialContext();
  SecurityAccess securityaccess = (SecurityAccess)
ctx.lookup("SecurityAccessBean/remote");
   try{
    out.println("<font color=green>调用结果:</font>"+
securityaccess.AnonymousUserMethod()+ "<br>");
   }catch(Exception e){
    out.println("访问AnonymousUserMethod方法出错");
   }
%>
</P>
</body>
</html>
```

为了便于更换用户角色，我们需要一个登录退出页面。代码如下：

logout.jsp

```
<%@ page contentType="text/html; charset=GBK"
    import="java.util.*,javax.naming.*, javax.servlet.http.*, org.jboss.security.*"%>
<%
      ((HttpSession) request.getSession()).invalidate ();
      SecurityAssociation.clear ();
%>
<html>
<head>
<META HTTP-EQUIV="Refresh" CONTENT="1;URL=http:index.jsp">
</head>
  <body>
    正在退出登录...<br>
  </body>
</html>
```

下面是例子的主页 index.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
  <title>Jaas tests</title>
</head><body>
<jsp:include page="/includes/menubar.jsp"/>
<H3>Jaas Tests</H3>
<P>
   <A href="anon/anonymousPage.jsp">这个页面允许匿名用户访问</A>
</P>
<P>
   <A href="user/departmentUser.jsp">这个页面允许部门角色访问</A>
</P>
<P>
    <A href="admin/adminPage.jsp">这个页面允许管理员角色访问</A>
</P>
</body>
</html>
```

下面是菜单包含页 menubar.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<A href="logout.jsp">退出</A>   
<A href="anon/anonymousPage.jsp">合伙用户</A>    
<A href="user/departmentUser.jsp">部门用户</A>   
<A href="admin/adminPage.jsp">管理员</A>   
```

本例子的源代码在配套光盘的 SecurityWithPropertiesFile 文件夹。要恢复 SecurityWithPropertiesFile 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。
本例子的客户端代码也在 SecurityWithPropertiesFile 文件夹，客户端应用随 Ant 的 deploy 任务一起发布。通过 http://localhost:8080/JaasTest 访问客户端。

2.13.1 自定义安全域

把用户名/密码及角色存放在 users.propertes 和 roles.properties 文件，不便于日后的管理。大多数情况下我们都希望把用户名/密码及角色存放在数据库中。因此，我们需要自定义安全域，下面的例子定义了一个名为 foshanshop 的安全域，它采用数据库存储用户名及角色。

安全域需要在[jboss 安装目录]/server/default/conf/login-config.xml 文件中配置，本例配置如下：

```
<!-- ..................... foshanshop login configuration ..................-->
<application-policy name="foshanshop">
   <authentication>
      <login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
         flag="required">
        <module-option name="dsJndiName">java:/DefaultMySqlDS</module-option>
        <module-option name="principalsQuery">
          select password from sys_user where name=?
        </module-option>
        <module-option name="rolesQuery">
          select rolename,'Roles' from sys_userrole where username=?
        </module-option>
        <module-option name = "unauthenticatedIdentity">AnonymousUser</module-option>
      </login-module>
    </authentication>
</application-policy>
```

上面使用了 Jboss 数据库登录模块(org.jboss.security.auth.spi.DatabaseServerLoginModule)，它的 dsJndiName 属性用于指定数据源的 JNDI 名称，“DefaultMySqlDS”为本书配置的数据源（关于 jboss 数据源的配置请参考后面章节内容）。principalsQuery 属性指定如何通过给定的用户名获取密码，rolesQuery 属性指定如何通过给定的用户名获取角色列表，注意：SQL 中的'Roles'常量字段不能去掉。unauthenticatedIdentity 属性指定允许匿名用户访问。

“foshanshop”安全域使用的 sys_user 和 sys_userrole 表是自定义表，实际应用中，你可以使用别的表名。
sys_user 表必须含有用户名及密码两个字段，字段类型为字符型，至于字符长度视你的应用而定。
sys_userrole 表必须含有用户名及角色两个字段，字段类型为字符型，字符长度也视你的应用而定。
下面是 sys_user，sys_userrole 表的具体定义(实际应用中字段名也可以自定义)
sys_user 表：

字段名	属性
name(主键)	varchar(45) NOT NULL

password	varchar(45) NOT NULL

在 Mysql 中的 DDL：

```
CREATE TABLE `sys_user` (
  `name` varchar(45) NOT NULL,
  `password` varchar(45) NOT NULL,
  PRIMARY KEY  (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=gb2312;
```

sys_userrole 表：

字段名	属性
username(主键)	varchar(45) NOT NULL
rolename(主键)	varchar(45) NOT NULL

在 Mysql 中的 DDL:

```
CREATE TABLE `sys_userrole` (
  `username` varchar(45) NOT NULL,
  `rolename` varchar(45) NOT NULL,
  PRIMARY KEY  (`username`,`rolename`)
) ENGINE=InnoDB DEFAULT CHARSET=gb2312;
```

为了使用本例子，你需要在数据库中创建上面两个表，然后往表中添加下列数据：

name	password
lihuoming	123456
zhangfeng	111111
wuxiao	123

username	rolename
lihuoming	AdminUser
lihuoming	DepartmentUser
lihuoming	CooperateUser
wuxiao	CooperateUser
zhangfeng	DepartmentUser

完成上面的配置后，我们就可以使用“foshanshop”安全域了，其开发步骤与前面章节相同。唯一不同的是安全域，我们将 jboss.xml 里的安全域指定为“foshanshop”，如下：

jboss.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <!--  Bug in EJB3 of JBoss 4.0.4 GA
  <security-domain>java:/jaas/foshanshop</security-domain>
  -->
  <security-domain>foshanshop</security-domain>
  <unauthenticated-principal>AnonymousUser</unauthenticated-principal>
</jboss>
```

jboss.xml 必须打进 Jar 文件的 META-INF 目录。

本例的客户端代码与上面章节相同，这里不再列出。你只需要把 jboss-web.xml 使用的安全域指定为 foshanshop ，如下：

jboss-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC
        "-//JBoss//DTD Web Application 2.3V2//EN"
        "http://www.jboss.org/j2ee/dtd/jboss-web_3_2.dtd">
<jboss-web>
  <security-domain>java:/jaas/foshanshop</security-domain>
</jboss-web>
```

本例子的源代码在配套光盘的 SecurityWithDB 文件夹。要恢复 SecurityWithDB 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。

本例子的客户端代码也在 SecurityWithDB 文件夹，客户端应用随 Ant 的 deploy 任务一起发布。通过 http://localhost:8080/JaasTest 访问客户端。

注意：运行本例子时，别忘了创建数据库表，数据源及“foshanshop”安全域。

第三章　实体 Bean(Entity Bean)

持久化是位于 JDBC 之上的一个更高层抽象。持久层将对象映射到数据库，以便在查询、装载、更新，或删除对象的时候，无须使用像 JDBC 那样繁琐的 API。在 EJB 的早期版本中，持久化是 EJB 平台的一部分。从 EJB 3.0 开始，持久化已经自成规范，被称为 Java Persistence API。

Java Persistence API 定义了一种方法，可以将常规的普通 Java 对象（有时被称作 POJO）映射到数据库。这些普通 Java 对象被称作 entity bean。除了是用 Java Persistence 元数据将其映射到数据库外，entity bean 与其他 Java 类没有任何区别。事实上，创建一个 Entity Bean 对象相当于新建一条记录，删除一个 Entity Bean 会同时从数据库中删除对应记录，修改一个 Entity Bean，容器会自动将 Entity Bean 的状态同步到数据库。

Java Persistence API 还定义了一种查询语言（JPQL），具有与 SQL 相类似的特征，只不过做了裁减，以便处理 Java 对象而非原始的关系 schema。

3.1 JBoss 数据源的配置

数据源用于配置数据库的连接信息，每个数据源必须指定一个唯一的 JNDI 名称。应用通过 JNDI 名称找到数据源。在 Jboss 中，有一个默认的数据源 DefaultDS，它使用 Jboss 内置的 HSQLDB 数据库。实际项目中，你可能使用不同的数据库，如 MySql、SqlServer、Oracle 等。每种数据库的数据源配置模版可以在[Jboss 安装目录]\docs\examples\jca 目录中找到，名称为：数据库名+ -ds.xml。

数据源配置文件的取名格式必须为 xxx–ds.xml，其中 xxx 代表任意名称 ，如: mysql-ds.xml ，mssqlserver-ds.xml，

oracle-ds.xml。数据源部署前，必须把数据库驱动 Jar 拷贝到[jboss 安装目录]/server/配置名/lib 目录，本书采用的配置名为 default，因此你需要把数据库驱动拷贝到 [jboss 安装目录]/server/default/lib。完成拷贝后，你必须重启 Jboss 服务器。本书使用的数据库是 mysql-5.0.22 ，其驱动为 mysql-connector-java-3.1.13-bin.jar。你可以在本书配套光盘的 lib 文件夹下得到。

数据源部署的过程很简单，直接把它拷贝到 jboss 的 deploy 目录即可，本书为[jboss 安装目录]/server/default/deploy 目录。容器遇到以*-ds.xml 结尾的文件时，会进行动态发布。发布完成后，你可以在 http://localhost:8080/jmx-console/ 查看到数据源的信息，如下图：

你可以点击 name=DefaultMySqlDS，service=ManagedConnectionPool 进入连接池属性修改界面。在连接池属性界面中，MaxSize 属性指定了最大连接数，InUseConnectionCount 代表目前正在使用的连接数。一旦 InUseConnectionCount 大于 MaxSize，后面发起的数据库连接将会报错。一般导致这种错误发生的原因是：在代码中通过 JDBC 操作数据库连接，可是在使用完后没有释放掉连接。

3.1.1 MySql 数据源的配置

下面定义了一个 JNDI 名称为 DefaultMySqlDS 的 Mysql 数据源。连接的数据库为 foshanshop，数据库登录用户名为 root，密码为 123456，数据库驱动类为 org.gjt.mm.mysql.Driver。

mysql-ds.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>DefaultMySqlDS</jndi-name>
<connection-url>jdbc:mysql://localhost:3306/foshanshop?useUnicode=true&amp;character
Encoding=GBK</connection-url>
    <driver-class>org.gjt.mm.mysql.Driver</driver-class>
    <user-name>root</user-name>
    <password>123456</password>
    <!-- 最小连接数 -->
```

```
    <min-pool-size>3</min-pool-size>
    <!-- 最大连接数 -->
    <max-pool-size>32</max-pool-size>
    <!-- 抛出异常前最大的等待连接时间 -->
    <blocking-timeout-millis>30000</blocking-timeout-millis>
    <!-- 关闭连接前连接空闲的最大时间 -->
    <idel-timeout-minutes>5</idel-timeout-minutes>
    <exception-sorter-class-name>org.jboss.resource.adapter.jdbc.vendor.MySQLExceptio
nSorter</exception-sorter-class-name>
    <metadata>
       <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

3.1.2 Ms Sql Server2000 数据源的配置

下面定义了一个 JNDI 名称为 MSSQLDS 的 SqlServer 数据源。连接的数据库为 foshanshop，数据库登录用户名为 sa，密码为 123456，数据库驱动类为 com.microsoft.jdbc.sqlserver.SQLServerDriver。

mssqlserver-ds.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>MSSQLDS</jndi-name>
    <connection-url>jdbc:microsoft:sqlserver://
localhost:1433;DatabaseName=foshanshop </connection-url>
    <driver-class>com.microsoft.jdbc.sqlserver.SQLServerDriver</driver-class>
    <user-name>sa</user-name>
    <password>123456</password>
      <metadata>
         <type-mapping>MS SQLSERVER2000</type-mapping>
      </metadata>
  </local-tx-datasource>
</datasources>
```

3.1.3 Oralce9i 数据源的配置

下面定义了一个 JNDI 名称为 OracleDS 的 Oracle9i 数据源。连接的数据库为 FS，数据库登录用户名为 root，密码为 123456，数据库驱动类为 oracle.jdbc.driver.OracleDriver。

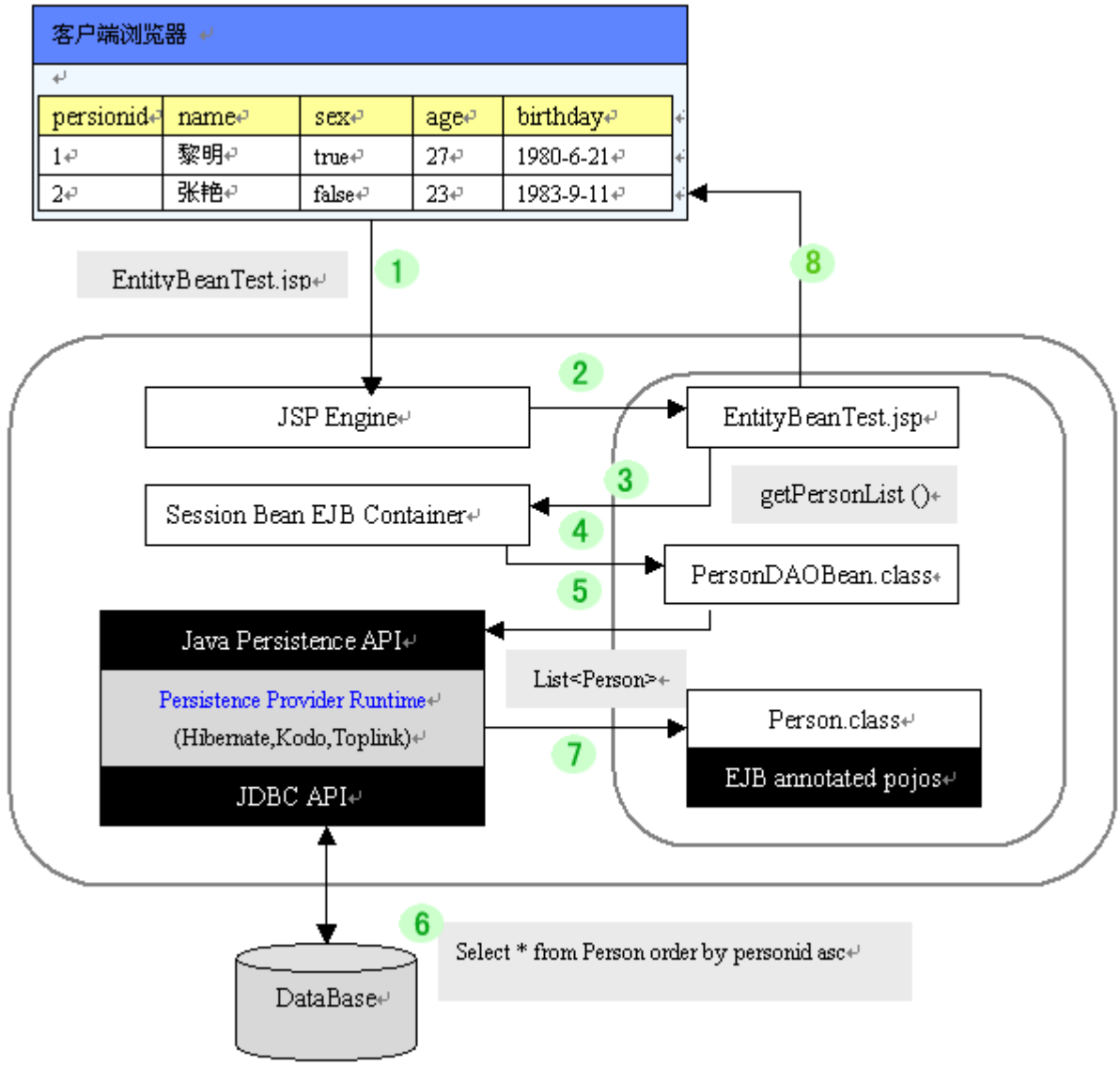
oracle-ds.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<datasources>
  <local-tx-datasource>
    <jndi-name>OracleDS</jndi-name>
    <connection-url>jdbc:oracle:thin:@192.168.1.2:1521:FS</connection-url>
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
    <user-name>root</user-name>
    <password>123456</password>
  <exception-sorter-class-name>
    org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</exception-sor
ter-class-name>
     <metadata>
        <type-mapping>Oracle9i</type-mapping>
     </metadata>
  </local-tx-datasource>
</datasources>
```

3.2 单表映射的实体 Bean

本例子要建立一个跟数据表进行映射的实体 bean，在 Session bean 中对该实体 bean 进行添/删/改/查操作。首先看看例子的流程图，了解应用执行的整个过程：



1> 浏览器请求 EntityBeanTest.jsp 文件

2> 应用服务器的 JSP 引擎编绎 EntityBeanTest.jsp

3> EntityBeanTest.jsp 通过 JNDI 查找 EJB 的 stub（存根），调用存根的 getPersonList()方法，EJB 容器截获方法调用。

4> EJB 容器注入 EntityManager，调用 PersonDAOBean 实例的 getPersonList ()方法。

5> PersonDAOBean 调用 EntityManager.createQuery("select o from Person o order by o.personid asc")进行持久化查询

6> Persistence provider runtime 通过 O/R Mapping annotation 把上面 JPQL 查询语句转绎成 SQL 语句。

7> Persistence provider runtime 把 SQL 查询结果处理成 Person 类型的 List。

8> 历遍存放 Person 的 List，打印在页面上。

开发前先了解需要映射的数据库表。在本例子，该表由持久化驱动自动生成，不需要我们创建。

person

字段名称	字段类型属性	描述
personid (主键)	Int(11) not null	人员 ID
name	Varchar(32) not null	姓名
sex	Tinyint(1) not null	性别

age	Smallint(6) not null	年龄
birthday	datetime null	出生日期

下面我们建立映射 Person 表的实体 Bean，实体 Bean 的成员属性分别映射到 Person 表的对应字段。在 EJB3.0 中，每个实体 Bean 必须具有一个主键，主键可以是基本类型，也可以是一个类。主键既作为实体 bean 在内存中的标识符，也作为数据表中一行的标识符。它在实体 bean 中是不可缺少的，并且必须是唯一的。

Person.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.GenerationType;

@SuppressWarnings("serial")
@Entity
@Table(name = "Person")
public class Person implements Serializable{

    private Integer personid;
    private String name;
    private boolean sex;
    private Short age;
    private Date birthday;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Integer getPersonid() {
        return personid;
    }
    public void setPersonid(Integer personid) {
        this.personid = personid;
    }

    @Column(nullable=false,length=32)
    public String getName() {
        return name;
    }
    public void setName(String name) {
```

```
        this.name = name;
    }

    @Column(nullable=false)
    public boolean getSex() {
        return sex;
    }
    public void setSex(boolean sex) {
        this.sex = sex;
    }

    @Column(nullable=false)
    public Short getAge() {
        return age;
    }
    public void setAge(Short age) {
        this.age = age;
    }

    @Temporal(value=TemporalType.DATE)
    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
}
```

如果实体 bean 对象需要返回客户端，必须实现 Serializable 接口。Person 实体 bean 只需通过一些注释就完成了跟 Person 数据表的映射，是不是很简单？下面介绍代码中各个注释的含义。

@javax.persistence.Entity 注释指明这是一个实体 Bean。注释的定义如下：

```
package javax.persistence;
@Target(TYPE) @Retention(RUNTIME)
public @interface Entity
{
    String name( ) default "";
}
```

name( )属性指定实体 bean 的名称，如果没有为该属性提供取值，默认值为 bean class 的非限定类名。如果你提供了该属性值，如@Entity(name="MyPerson")，那么在 JPQL 表达式中，你应使用该名称，如：

select o from MyPerson o where o.personid=?1。

@javax.persistence.Table 注释指定了实体 Bean 所要映射的表，注释的定义如下：

```
package javax.persistence;
```

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Table
{
    String name( ) default "";
    String catalog( ) default "";
    String schema( ) default "";
    UniqueConstraint uniqueConstraints( ) default {};
}
```

name()属性指定映射表的名称。如果缺省@Table 注释，系统默认采用实体名称作为映射表的名称。对于本例而言，默认的实体名称为 Person，因此默认的映射表名称也为 Person，所以你可以省略@Table(name = "Person")。

catalog( ) 属性指明数据库的 catalog

schema( ) 属性指明映射表所属的 schema

uniqueConstraints( )属性允许你指定唯一性字段约束。如，下面为 personid 和 name 字段指定唯一性约束：

```
@Table(
name="Person",
uniqueConstraints={@UniqueConstraint(columnNames={"personid", "name"})}
)
```

其中@UniqueConstraint 注释的定义如下：

```
public @interface UniqueConstraint
{
    String[] columnNames( );
}
```

columnNames( ) 属性指定字段名，如果有多个，请用大括号括起来，每个字段名之间用逗号分隔。

@javax.persistence.Id 注释指定实体 Bean 的主键。每个实体 bean 必须有一个主键，并且必须是唯一的。下面是注释的定义：

```
package javax.persistence;
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Id
{
}
```

本例指定 personid 成员属性为实体 Bean 的主键。

@javax.persistence.GeneratedValue 注释指定主键值生成方式，该注释与@Id 注释结合使用在主键属性上。只有在使用持久化驱动生成数据表 schema 时才需指定该注释。如果你的数据表已经存在，那么该注释不需要指定。下面是该注释的定义：

```
package javax.persistence;
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface GeneratedValue
{
    GenerationType strategy( ) default AUTO;
    String generator( ) default "";
}
```

generator()属性定义主键值生成器的名称。如果实体的主键值生成策略不是 GenerationType.AUTO 或者

GenerationType.IDENTITY，就需要提供相应的 SequenceGenerator 或者 TableGenerator 注释，然后将 generator()属性值设置为注释的 name 属性值。

strategy( )属性指定字段值生成策略，它的属性值是个枚举类型，定义如下：

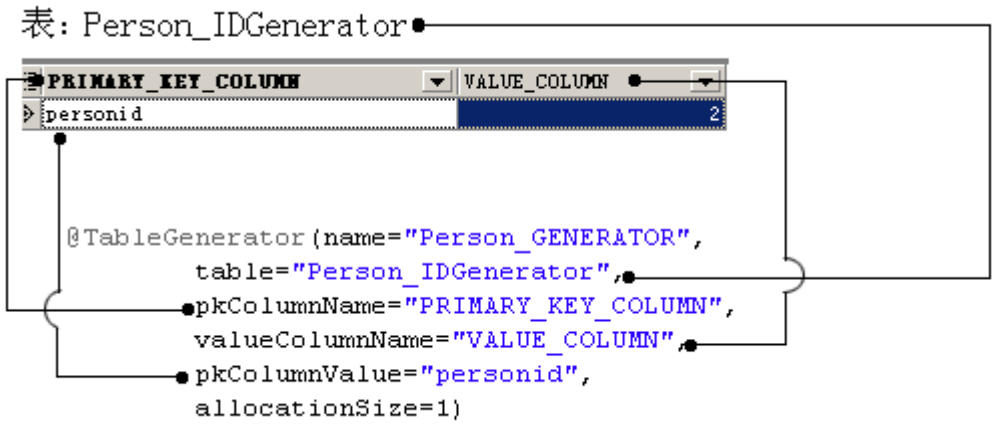
```
public enum GenerationType
{
    TABLE, SEQUENCE, IDENTITY, AUTO
}
```

各枚举值的含义如下：

- GenerationType.TABLE：指定使用一张表生成数值型序列号，并用序列号作为主键值。使用该策略更易于数据库移植。如果你使用了该策略，但没有设置@GeneratedValue.generator()属性值，持久化实现者将会自动生成一张表，不同的持久化实现厂商生成的表名是不同的，如 OpenJPA 生成 openjpa_sequence_table 表，Hibernate 生成一个 hibernate_sequences 表，而 TopLink 则生成 sequence 表。这些表都具有一个序列名和对应值两个字段，如 SEQ_NAME 和 SEQ_COUNT。当然我们也可以自定义生成表，如下：

```
    @TableGenerator(name="Person_GENERATOR",//为该生成器取个名称
            table="Person_IDGenerator",//生成ID的表
            pkColumnName="PRIMARY_KEY_COLUMN",//主键列的名称
            valueColumnName="VALUE_COLUMN",//存放生成ID值的列名称
            pkColumnValue="personid",//主键列的值(定位某条记录)
            allocationSize=1)//缓存主键值数量
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE,
generator="Person_GENERATOR")
    public Integer getPersonid() {
        return personid;
    }
```

与数据库生成表的映射关系如下：



@javax.persistence.TableGenerator 注释的定义：

```
@Target({TYPE,METHOD,FIELD})
@Retention(RUNTIME)
public @interface javax.persistence.TableGenerator{
   String name();
   String table() default "";
```

```
    String catalog() default "";
    String schema() default "";
    String pkColumnName() default "";
    String valueColumnName() default "";
    String pkColumnValue() default "";
    int allocationSize() default (int) 50;
    javax.persistence.UniqueConstraint[] uniqueConstraints() default {};
}
```

name()是必须设置的属性，它定义了表生成器在容器中的唯一名称，该名称将会被@GeneratedValue.generator()属性所使用。

table()属性指定生成序列号的表的名称。该属性是可选属性，如果开发者没有为该属性设置值，容器将会生成一个默认表名。

schema()和 catalog()属性描述了生成序列号的表的定义信息。

pkColumnName()该属性指定生成表主键字段的名称。该属性是可选属性，如果开发者没有为该属性设置值，容器将会生成一个默认值。

valueColumnName()属性指定生成表用作序列号累计的字段的名称。该属性是可选属性，如果开发者没有为该属性设置值，容器将会生成一个默认值。

pkColumnValue()属性指定生成表主键字段的标识值。该属性是可选属性，如果开发者没有为该属性设置值，容器将会生成一个默认值。可以为多个实体设置相同的 pkColumnValue 属性值，这些实体的主键值将通过同一序列的递增来实现。

allocationSize()为了降低主键值生成时频繁操作数据库造成性能上的影响，你可以让 persistence provider 缓存一批主键值，而不必在每次需要新的主键值时都去访问数据库，allocationSize 指定了缓存主键值的数量，注意：此时，生成表的序列号字段只在用完了缓存中的值而需要再次访问生成表时才会发生累计。该属性是可选属性，如果开发者没有为该属性设置值，容器将会使用默认值 50。

uniqueConstraints()属性指定一些约束。

- GenerationType.SEQUENCE：使用数据库的 SEQUENCE（序列）来生成主键值。这种情况下需要数据库提供对 SEQUENCE 的支持，常用的数据库中，Oracle、PostgreSQL 等数据库都能够提供这种支持。如：

```
@SequenceGenerator(name="Person_SEQUENCE", //为该生成器取个名称
    sequenceName="Person_SEQ")//sequence的名称
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE,
generator="Person_SEQUENCE")
public Integer getPersonid() {
    return personid;
}
```

@javax.persistence.SequenceGenerator 注释的定义：

```
@Target({TYPE,METHOD,FIELD})
@Retention(RUNTIME)
public @interface SequenceGenerator{
  String name();
  String sequenceName() default "";
  int initialValue() default (int) 1;
  int allocationSize() default (int) 50;
```

}

name()是必须设置的属性，它定义了 Sequence 生成器在容器中的唯一名称，该名称将会被@GeneratedValue.generator()属性所使用。

sequenceName()属性指定所使用 sequence 的名称。该属性是可选属性，如果开发者没有为该属性设置值，容器将会生成一个 sequence。

initialValue()属性设置所使用序列号的起始值。默认值为 1。

allocationSize()一些数据库的序列化机制允许预先分配序列号，比如 Oracle，这种预先分配机制可以一次性生成多个序列号，然后放在 cache 中，数据库用户获取的序列号是从 cache 中获取的，这样就避免了在每一次数据库用户获取序列号的时候都要重新生成序列号。allocationSize 属性设置的就是一次预先分配序列号的数目，默认情况下 allocationSize 属性的值是 50。

- GenerationType.IDENTITY：使用数据库 ID 自增长方式来生成主键值（像 HSQL、SQL Server、MySQL、DB2、Derby 等数据库提供了通过 ID 自增长方式生成唯一主键值）。
- GenerationType.AUTO：由容器根据数据库类型选择一种合适的生成方式，这种方式带有随机性，不同的 JPA 实现产品做法各有不同，对于本例而言，Hibernate 知道 Mysql 支持 ID 自增长，所以会选择 GenerationType.IDENTITY。

@javax.persistence.Column 注释指定实体 Bean 的成员属性映射到数据表中的哪一个字段和该字段的一些结构信息（如字段是否唯一，是否允许为空，是否允许更新等），该注释需标注在成员属性的 getter 方法上，@Column 注释的定义如下：

```
public @interface Column
{
    String name( ) default "";
    boolean unique( ) default false;
    boolean nullable( ) default true;
    boolean insertable( ) default true;
    boolean updatable( ) default true;
    String columnDefinition( ) default "";
    String table( ) default "";
    int length( ) default 255;
    int precision( ) default 0;
    int scale( ) default 0;
}
```

name( )属性指定成员属性（Property）映射到表中的哪个字段。例如：把成员属性 name 映射到 Person 表的 PersonName 字段，可以在成员属性的 getName()方法上面加入@Column(name = "PersonName")。如果不指定映射字段，EJB 容器会认为要映射的字段名称和属性名称相同，对于本例而言，属性 name 映射的是 name 字段。

unique( ) 属性指定字段是否唯一

nullable( ) 属性指定字段是否允许为空

length( ) 对于字符型字段，length 属性指定字段的最大字符长度。

insertable( ) 属性指定是否想让该字段出现在 SQL INSERT 语句中，注：在保存实体 Bean 的时候，持久化驱动默认会使用全部成员属性映射的字段来构造 INSERT 语句，如果你不想让某个字段出现在 Insert 语句中，可以设置 insertable=false。

updatable( ) 属性指定是否想让该字段出现在 SQL UPDATE 语句中。

columnDefinition( ) 属性定义生成表结构时，用于创建该字段的 DDL ，该属性特定于具体数据库，不建议使用。

table( ) 如果字段不在主表上（默认建在主表），该属性指明字段所在表的名称。

precision( ) 设置数字值的精度
scale( ) 属性设置数字值的小数位数。

@javax.persistence.Temporal 注释用来指定 java.util.Date 或 java.util.Calendar 类型成员属性与数据库类型 date，time 或 timestamp 中的那一种进行映射。缺省情况下，持久化驱动假定时间类型为 timestamp。它的定义如下：

```
package javax.persistence;
public enum TemporalType
{
    DATE, //代表 date 类型
    TIME, //代表时间类型
    TIMESTAMP //代表时间戳类型
}
```

发布到 Jboss 中的实体 bean 可以缺少@Temporal 注释，如果发布到使用了 TopLink 作为持久化产品实现的应用服务器，缺少该注释将会导致部署失败。

在 EJB3.0，实体 Bean 并不直接与客户端打交道。而是被 Session bean 或 Message-Driven Bean 使用。因此我们定义一个 Session Bean，在 Session Bean 中通过实体 bean 间接操作数据库。下面是 Session Bean 的业务接口。

PersonDAO.java

```java
package com.foshanshop.ejb3;
import java.util.List;
import com.foshanshop.ejb3.bean.Person;

public interface PersonDAO {
    /**
     * 添加一个Person
     * @param person 人员
     */
    public void insertPerson(Person person);
    /**
     * 更新姓名
     * @param newname 新姓名
     * @param personid 人员ID
     */
    public void updateName(String newname, int personid);
    /**
     * 更新person对象
     * @param person
     */
    public void mergePerson(Person person);
    /**
     * 删除指定Person
     * @param personid 人员ID
     */
    public void deletePerson(int personid);
```

```
    /**
     * 获取指定Person
     * @param personid 人员ID
     * @return
     */
    public Person getPersonByID(int personid);
    /**
     * 获取全部Person
     * @return
     */
    public List<Person> getPersonList();
}
```

下面是 Session Bean 的实现：PersonDAOBean.java

```
package com.foshanshop.ejb3.impl;
import java.util.List;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import com.foshanshop.ejb3.PersonDAO;
import com.foshanshop.ejb3.bean.Person;

@Stateless
@Remote (PersonDAO.class)
public class PersonDAOBean implements PersonDAO {
    @PersistenceContext(unitName="foshanshop") protected EntityManager em;

    public void insertPerson(Person person) {
        em.persist(person);
    }

    public Person getPersonByID(int personid) {
        return em.find(Person.class, personid);
    }

    public void mergePerson(Person person) {
       em.merge(person);
    }

    @SuppressWarnings("unchecked")//关闭unchecked警告
    public List<Person> getPersonList() {
        Query query = em.createQuery("select o from Person o order by o.personid
```

```
asc");
        return (List<Person>) query.getResultList();
    }

    public void deletePerson(int personid) {
        Person person = em.find(Person.class, personid);
        if(person!=null) em.remove(person);
    }

    public void updateName(String newname, int personid) {
        Person person = em.find(Person.class, personid);
        if(person!=null) person.setName(newname);
    }
}
```

上面我们通过@PersistenceContext 注释动态注入 EntityManager 对象。该注释和@EJB 注释一样，会在 EJB 的 JNDI ENC 中注册一个指向该资源的引用。EntityManager 是由 EJB 容器自动管理和配置的，这包括 EntityManager 的创建及清理工作。所以我们不需要调用它的 close()方法释放资源，如果你试图这样做，反而会得到 IllegalStateException 例外。借助 EntityManager，我们可以创建、更新、删除及查询实体 bean。EntityManager 负责将固定数量的一组类映射到数据库中，这组类被称作持久化单元(persistence unit)。persistence unit 是在 persistence.xml 中定义的。根据持久化规范的要求，该部署描述文件是必须提供的，如果不提供这一文件，则持久化单元也将不存在，因此应用也不能够获得和使用 EntityManager。我们需要将这一文件存放在下列各类文件的 META-INF 目录中。

- 出现于常规 JavaSE 程序的 classpath 中的普通 JAR 文件。
- EJB-JAR 文件。Persistence unit 可以包含在一个 EJB 部署中。
- Web 归档文件（.war）中，WEB-INF/lib 目录下的 JAR 文件。
- 企业归档文件（.ear）中，根目录下的 JAR 文件。
- EAR 文件中，lib 目录下的 JAR 文件。

persistence.xml 文件指定实体 Bean 使用的数据源及 EntityManager 的默认行为。本例子 persistence.xml 文件的配置如下：

```
<?xml version="1.0"?>
<persistence 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_1_0.xsd" version="1.0">
  <persistence-unit name="foshanshop" transaction-type="JTA">
    <jta-data-source>java:/DefaultMySqlDS</jta-data-source>
    <properties><!--下面属性只针对Jboss服务器 -->
     <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
     <!-- 调整JDBC抓取数量的大小: Statement.setFetchSize() -->
     <property name="hibernate.jdbc.fetch_size" value="18"/>
     <!-- 调整JDBC批量更新数量 -->
     <property name="hibernate.jdbc.batch_size" value="10"/>
```

```
    <!-- 显示最终执行的SQL -->
    <property name="hibernate.show_sql" value="true"/>
    <!-- 格式化显示的SQL -->
    <property name="hibernate.format_sql" value="true"/>
   </properties>
  </persistence-unit>
</persistence>
```

persistence-unit 节点可以有一个或多个，每个 persistence-unit 节点定义了持久化内容名称、使用的数据源及持久化产品专有属性。

name 属性定义了 persistence-unit 的名称，该属性是必需的，本例设置的名称为“foshanshop”。

transaction-type 属性指定 persistence unit 是受 JTA 事务管理并与之集成，还是使用 RESOURCE LOCAL 的 javax.persistence.EntityTransaction API 来管理 EntityManager 实例的事务完整性。此属性在 JavaEE 环境中的默认值是 JTA，而在 JavaSE 环境中则为 RESOURCE_LOCAL。如果 transaction-type="JTA"，你可以使用<jta-data-source>指定数据源的 JNDI 名称。如果 transaction-type="RESOURCE_LOCAL"，你可以使用<non-jta-data-source>指定数据源的 JNDI 名称。Jboss 数据源的 JNDI 名称在局部命名空间，因此数据源名称前必须带有 java:/前缀，数据源名称大小写敏感。

<properties>指定持久化产品的专有属性，各个应用服务器使用的持久化产品都不一样，如：Jboss 使用 Hibernate，weblogic 使用 Kodo（实际上是基于 OpenJPA 的封装），glassfish/sun application server/Oralce 使用 Toplink。对于 Hibernate 而言，它的 hibernate.hbm2ddl.auto 属性指定实体 Bean 发布时是否同步数据库结构，如果 hibernate.hbm2ddl.auto 的值设为 create-drop，实体 Bean 发布及卸载时将自动创建及删除相应数据库表（注意：Jboss 服务器启动或关闭时也会引发实体 Bean 的发布及卸载）。TopLink 产品的 toplink.ddl-generation 属性也起到同样的作用。关于 hibernate 的可用属性及默认值你可以在 [Jboss 安装目录]\server\default\deploy\ejb3.deployer\META-INF/persistence.properties 文件中找到。在开发阶段，Hibernate 的 hibernate.show_sql 和 hibernate.format_sql 属性特别有用，它们可以格式化显示 Hibernate 执行的 SQL 语句。

前面我们知道 persistence-unit 将固定数量的一组类映射到数据库，那么这组类都包含有那些？

上面我们了解了一个持久化单元(persistence unit)的内容，接下来我们看看@PersistenceContext 注释的定义：

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface PersistenceContext {
    String name( ) default "";
    String unitName( ) default "";
    PersistenceContextType type( ) default TRANSACTION;
    PersistenceProperty[] properties( ) default {};
}
```

name( )属性指定被引用的 EntityManager 在 JNDI ENC 中注册的名称。该名称是相对于 java:comp/env 上下文的。unitName( )属性指定了你想引用的 persistence unit，其取值与你在 persistence.xml 文件中声明的 persistence unit 名称一致。如果 persistence.xml 文件只有一个 persistence unit，可以不指定该属性值，此时容器默认使用这个唯一的 persistence unit。对于本例子而言，你可以省略 unitName="foshanshop"。当 persistence.xml 文件定义了多个 persistence

unit，你必须为该属性指定使用那个 persistence unit，否则容器会抛出一个部署例外。
type( )属于用于指定 persistence context 的类型，persistence context 的类型有 transaction-scoped persistence context 和 extended persistence context。分别用 PersistenceContextType.TRANSACTION 和 PersistenceContextType.EXTENDED 指定。默认为 transaction-scoped persistence context 类型。在 Stateless Session Bean 中你不能注入 PersistenceContextType.EXTENDED 类型的 EntityManager，该类型只能注入到 Stateful Session Bean。有关 persistence context 的知识将在后面章节介绍。
properties( )属性用于覆盖或添加在 persistence.xml 文件的<properties>元素中定义的厂商专有属性。

在 PersonDAOBean 业务代码中 em.find()方法用于查找特定主键的实体 bean。em.persist()方法用于保存实体 bean，即插入一条记录。em.merge()方法用于更新或保存实体(当实体不存在时，执行保存操作，当实体已经存在时，执行更新操作)。em.remove()方法用于删除实体，即删除一条记录。em.createQuery()用于执行 JPQL 语句。关于各方法的详细介绍请查看后面章节内容。

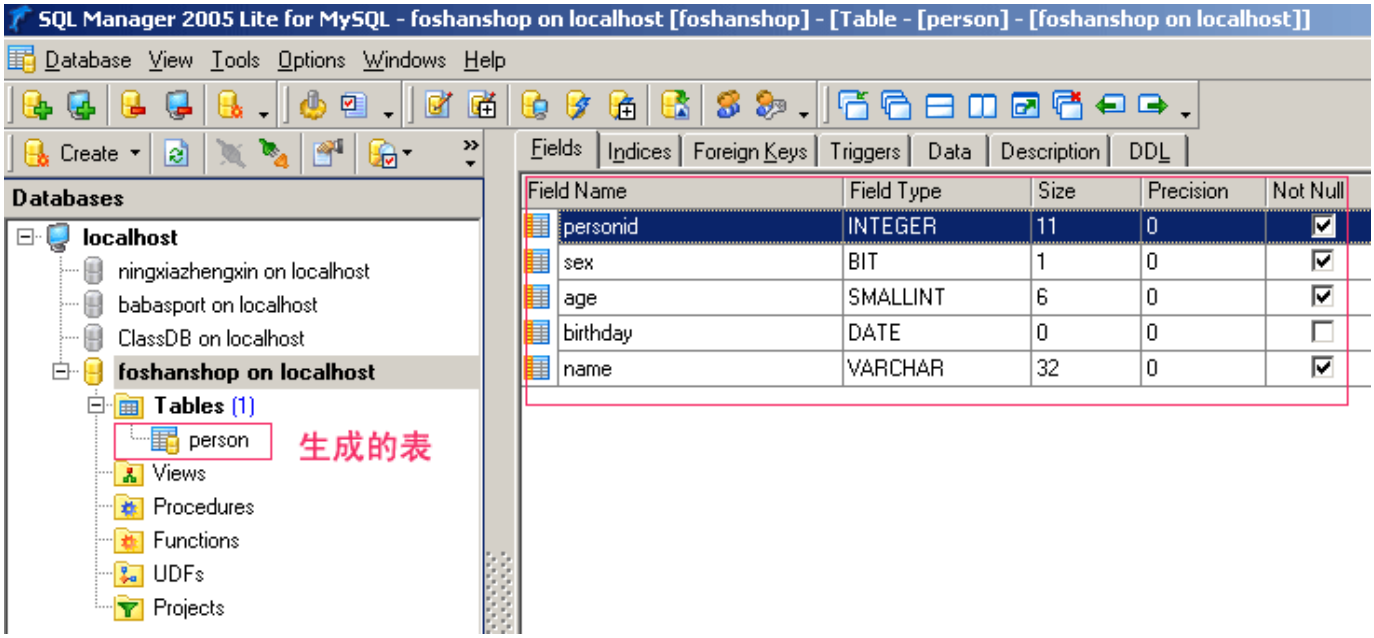
到目前为止，已经完成了实体 bean 应用的全部开发。接下来需要把它打成 JAR 文件，打完 JAR 后的文件应具有以下文件结构：

```
EntityBean.jar
      |
      | -- com/foshanshop/**/*.class
      +- META-INF
      |     -- persistence.xml
```

实体 bean 发布前，应检查下面几项工作是否完成：

1 数据源是否已经发布在 deploy 目录下，此时数据库是否已经启动。
2 数据源使用的数据库驱动 Jar 是否已经放置在[Jboss 安装目录]\server\配置名\lib 目录下，本书使用的配置名为 default，因此路径为：[Jboss 安装目录]\server\default\lib。(注：往 lib 目录放入 jar 后需重启 jboss)。
3 persistence.xml 文件是否存在于 jar 文件的 META-INF 目录，文件中是否指定了数据源。

当上面几项工作检查通过后。我们就可以把 jar 部署到 jboss 中。因为本例子在 persistence.xml 中指定了 hibernate.hbm2ddl.auto=create-drop，该属性值指定在实体 Bean 发布及卸载时将自动创建及删除表。当实体 bean 发布成功后，我们可以查看数据库中生成的 Person 表，如下图：



小提示：如果你的表已经存在，并且想保留数据，发布实体 bean 时可以把 hibernate.hbm2ddl.auto 的值设为 none 或 update，以后为了实体 bean 的改动能更新到数据库，建议使用 update，这样实体 Bean 添加一个属性时能在数据表中增加相应字段。

本例子的 JSP 客户端代码：EntityBeanTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.PersonDAO,com.foshanshop.ejb3.bean.Person,
                javax.naming.*,
                java.util.Properties,
                java.util.Date,
                java.util.List"%>

<TABLE width="80%" border="1">
<TR bgcolor="#DFDFDF">
    <TD>personid</TD>
    <TD>name</TD>
    <TD>sex</TD>
    <TD>age</TD>
    <TD>birthday</TD>
</TR>
<%
try {
        InitialContext ctx = new InitialContext();
        PersonDAO persondao = (PersonDAO) ctx.lookup("PersonDAOBean/remote");
        Person newperson = new Person();
        newperson.setName("张朗");
        newperson.setAge((short)27);
        newperson.setBirthday(new Date());
```

```
        newperson.setSex(true);
        persondao.insertPerson(newperson);

        List<Person> persons = persondao.getPersonList();
        for(Person person : persons){
            out.println("<TR><TD>"+ person.getPersonid()+"</TD><TD>"+
person.getName()+"</TD><TD>"+ person.getSex()+"</TD><TD>"+
person.getAge()+"</TD><TD>"+ person.getBirthday()+"</TD></TR>");
        }

} catch (Exception e) {
    out.println(e.getMessage());
}
%>
</TABLE>
```

上面代码往数据库添加一个 Person，然后获取全部记录打印出来。

本例子的源代码在配套光盘的 EntityBean 文件夹。例子使用的数据源配置文件是 mysql-ds.xml，你可以在配套光盘中找到。部署数据源到 Jboss 前，请先把数据库驱动拷贝到[jboss 安装目录]/server/default/lib 目录（请用你的配置名替换 default），拷贝完成后需重启 Jboss。要恢复 EntityBean 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。
本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/EntityBeanTest.jsp 访问客户端，另外你也可以执行 EntityBean 项目里的单元测试用例 PersonDAOTest.java。

在 Mysql 创建数据库时，你必须指定数据库的字符集编码为 GBK，否则当插入中文字符时会报: Data too long for column。你可以使用下面的 SQL 创建数据库：
create database `foshanshop` DEFAULT CHARSET=gbk

3.3 成员属性映射

@Transient 注释
默认情况下，实体 bean 的全部成员属性都会成为持久化字段。如果你不希望一些成员属性成为持久化字段，可以使用@Transient 注释标注，如：

```
public class Person implements Serializable{
   ......
   @Transient
   public String getFristName() {
      return "li";
   }
}
```

@Enumerated 注释

如果你需要将枚举类型成员属性映射到数据库，可以使用@Enumerated 注释进行标注。枚举类型成员属性可以被映射为字符串形式（用 EnumType.STRING 指定），也可以映射为枚举值的数据序号（用 EnumType.ORDINAL 指定）。如下:

CommentType.java

```
package com.foshanshop.site.bean.comment;
public enum CommentType {
    NEWS {public String getName(){return "资讯评论";}},
    PRODUCT {public String getName(){return "产品评论";}};

    public abstract String getName();
}
```

```
@Entity
public class CommentContent implements Serializable{
    ......
    private CommentType type;//评论类型

    @Enumerated(EnumType.STRING)
    public CommentType getType() {
        return type;
    }

    public void setType(CommentType type) {
        this.type = type;
    }
}
```

@Lob 注释

有些时候你需要存放一些文件或大文本数据进数据库，JDBC 使用 java.sql.Blob 类型存放二进制数据，java.sql.Clob 类型存放字符数据，这些数据都是非常占内存的，@Lob 注释用作映射这些大数据类型，当属性的类型为 byte[], Byte[]或 java.io.Serializable 时，@Lob 注释将映射为数据库的 Blob 类型，当属性的类型为 char[], Character[]或 java.lang.String 时，@Lob 注释将映射为数据库的 Clob 类型

在做内容管理系统时，我们通常会使用到大文本数据，这时就有必要在字段或属性的 getter 方法上标注@Lob。

```
@Entity
public class News implements Serializable {
    ......
    private String content;

    @Lob
    public String getContent() {
        return content;
    }
```

```
    public void setContent(String _content) {
        this.content = _content;
    }
}
```

@Basic 注释

对于加了@Lob 注释的大数据类型（有时存放的可能是 10M 以上的数据），为了避免每次加载实体时占用大量内存，我们有必要对该属性进行延时加载，这时我们需要用到@Basic 注释，@Basic 注释的定义如下：

```
public @interface Basic
{
    FetchType fetch( ) default EAGER;
    boolean optional( ) default true;
}
```

FetchType 属性指定是否延时加载，默认为立即加载。

optional 属性指定在生成数据库结构时字段是否允许为 null。

```
@Entity
public class News implements Serializable {
    ......
    private String content;

    @Lob
    @Basic(fetch=FetchType.LAZY)
    public String getContent() {
        return content;
    }

    public void setContent(String _content) {
        this.content = _content;
    }
}
```

@Temporal 注释

因为数据表对时间类型有更严格的划分，所以必须使用@Temporal 注释指明 java.util.Date 或 java.util.Calendar 类型的成员属性映射到数据库 date、time 和 timestamp 中的那种类型。注释的定义如下：

```
package javax.persistence;
public enum TemporalType
{
    DATE,
    TIME,
    TIMESTAMP
}
```

```
@Entity
public class Person implements Serializable {
```

```
    ......
    private Date birthday;

    @Temporal(value=TemporalType.DATE)
    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
}
```

成员属性中使用非实体 java 对象

除了一些基本数据类型及其对象类型可以用作成员属性之外，我们也可以使用可序列化的普通 java 对象。如下面代码，persistence provider 会假定 info 为@Lob 类型，并且会以字节流的方式将它序列化到 Person 表中的字段：

```
@Entity
public class Person implements Serializable{
    private static final long serialVersionUID = -3637041585395570481L;
    private Integer personid;
    private MyInfo info;

    @Id @GeneratedValue
    public Integer getPersonid() {
        return personid;
    }
    public void setPersonid(Integer personid) {
        this.personid = personid;
    }

    public MyInfo getInfo() {
        return info;
    }
    public void setInfo(MyInfo info) {
        this.info = info;
    }
}
```

```
public class MyInfo implements Serializable{
    private static final long serialVersionUID = 6143002359286196628L;
    private String content;
    public String getContent() {
        return content;
    }
```

```
    public void setContent(String content) {
        this.content = content;
    }
}
```

如果你不想将整个 MyInfo 对象映射进数据库，而只是想把 MyInfo 中的成员属性映射到数据库，你需要使用@Embedded 注释，如：

```
@Entity
public class Person implements Serializable{
    ...
    @Embedded
    public MyInfo getInfo() {
        return info;
    }
}
```

另外你可以通过@javax.persistence.AttributeOverride 注释重载 MyInfo 的成员属性的映射信息，如：

```
@Entity
public class Person implements Serializable{
    ...
    @Embedded
    @AttributeOverride(name="content",column=@Column(length=100))
    public MyInfo getInfo() {
        return info;
    }
}
```

如果需要重载多个成员属性，可以和@javax.persistence.AttributeOverrides 注释配合使用，如：

```
@Entity
public class Person implements Serializable{
    ...
    @Embedded
    @AttributeOverrides({
    @AttributeOverride(name="content",column=@Column(length=100)),
    @AttributeOverride(name="xxx",column=@Column(length=10))
    })
    public MyInfo getInfo() {
        return info;
    }
}
```

3.4 建议重载实体 Bean 的 equals()和 hashCode()方法

EJB3.0 并没有规定你必须重载 equals()和 hashCode()。但在实际应用中，重载这两个方法是值得提倡的做法。对于数据库，我们通常根据主键或唯一字段来判断记录是否同一条。对于实体 Bean，因为一个实体对象代表了某个客观的物体，当两个实体对象进行比较时，我们应该使用物体的特征点进行比较，这些特征点通常会被定义为主键、复合主键或唯一字段。如，航线是由出发地和到达地来决定的，我们应该比较其出发地和到达地是否都相等来判断两条航线是否相同：

```
@Entity
public class AirtLine implements Serializable {
    /** 出发城市 **/
    private String leavecity;
    /** 到达城市 **/
    private String arrivecity;
    ......
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.leavecity!=null && this.arrivecity!=null ? (this.leavecity+
"-"+ this.arrivecity).hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        if (!(object instanceof AirtLine)) {
            return false;
        }
        AirtLine other = (AirtLine)object;
        if (this.leavecity != other.leavecity && (this.leavecity == null
|| !this.leavecity.equalsIgnoreCase(other.leavecity))) return false;
        if (this.arrivecity != other.arrivecity && (this.arrivecity == null
|| !this.arrivecity.equalsIgnoreCase(other.arrivecity))) return false;
        return true;
    }

    @Override
    public String toString() {
        return this.getClass().getName()+"[leavecity="+ leavecity +
",arrivecity="+ arrivecity+ "]";
    }
}
```

另外也最好重载 toString()方法，让对象的输出信息更有意义些。

3.5 映射的表名或列名与数据库保留字同名时的处理

如果使用的数据库是 Mysql，当映射的表名或列名与数据库保留字同名时，Jboss 持久化引擎转绎后的 SQL 在执行时将会出错。(直到 Jboss4.2.2GA 版本仍存在这个问题)
如：

```
@Entity
@Table(name = "Order")
public class Order implements Serializable {
```

表名 Order 与排序保留字 “Order”相同，导致转绎后的 SQL 在执行时，出现 SQL 语法错误。
针对上面的情况，在 Jboss 未修正这个问题前，可以采用一种临时的解决方案。该方案特定于具体数据库，不利于数据库移植。建议大家为表或字段取名时，避免使用数据库保留字。
在 Mysql 中可以用``字符把 Order 括起来。如下：

```
@Entity
@Table(name = "`Order`")
public class Order implements Serializable {
```

列名与保留字同名的处理方法如上，如列名为 group

```
@Column(name = "`group`")
public String getGroup() {
    return group;
}
```

如果数据库是 Sqlserver 可以用 [] 把表名或列名括起来。

3.6 多表映射的实体 bean

前面介绍的实体 bean，其成员属性映射的字段都在一张表上。如果实体的成员属性映射的字段分布在多张表中，我们该如何处理？如：本例子的实体 MainTable 具有四个成员属性：id、name、address 和 postcode。其中与 id 和 name 映射的字段在 MainTable 表，与 address 和 postcode 属性映射的字段在 Address 表。在这种情况下，我们需要使用@javax.persistence.SecondaryTable 注释和@Column.table()属性

这部分内容在《EJB3.0入门经典》中

3.7 持久化实体管理器 EntityManager

在 Java persistence 规范中，EntityManager 是为所有持久化操作提供服务的中枢。实体作为普通 java 对象，只有在调用 EntityManager 将其持久化后，才会变成持久对象。EntityManager 在一组固定的实体类与底层数据源之间进行 O/R 映射的管理。它可以用来添加/删除/更新实体 Bean，通过主键查找实体 bean，也可以通过 JPQL 语言查找满足条件的实体 Bean。当 EntityManager 被注入到 EJB 时，EJB 容器会对 EntityManager 所依赖的持久上下文(persistence context)具有完全的控制权。

持久上下文(persistence context)

持久上下文是由一组受托管(attached)的实体对象实例所构成的集合。它受 EntityManager 的管理。EntityManager 跟踪 persistence context 中所有对象的修改和更新情况，并根据指定的 flush 模式将这些修改保存到数据库中。一旦 persistence context 被关闭，所有实体对象都会脱离 EntityManager 而成为游离（detached）对象。对象一旦从 persistence context 中脱离，就不再受 EntityManager 管理了，任何对此对象的状态变更也不会被同步到数据库。Java persistence 中有两种类型的 persistence context，分别是 transaction-scoped 和 extended persistence contexts。

3.7.1 实体的状态

实体对象拥有以下 4 个状态，这些状态通过调用 EntityManager 接口方法发生迁移：

- 新建状态：新创建的实体对象，尚未拥有持久化主键，没有和一个持久化上下文关联起来。
- 托管状态：已经拥有持久化主键并和持久化上下文建立了联系；
- 游离（detached）状态：拥有持久化主键，但尚未和持久化上下文建立联系；
- 删除状态：拥有持久化主键，已经和持久化上下文建立联系，但已经被安排从数据库中删除。

3.7.2 Entity 获取 find()或 getReference()

如果知道实体的主键，我们可以用 find()或 getReference()方法来获得实体。方法的第一个参数为实体类，第二个参数为实体 OID（OID 为对象标识，即标注了@Id 的属性）的值。

```
@PersistenceContext protected EntityManager em;
......
Person person = em.find(Person.class,1);

try {
   Person p = em.getReference (Person.class, 1);
} catch (EntityNotFoundException notFound) {
      // 找不到记录...
}
```

find()方法返回指定 OID 的实体。如果这个实体存在于当前的 persistence context 中，那么返回值是被缓存的对象；否则会创建一个新的实体，并从数据库中加载相关的持久状态。如果数据库不存在指定 OID 的记录，那么 find()方法返回 null。

getReference()方法与 find()相似。不同的是：如果缓存中没有指定的实体，EntityManager 会创建一个新的实体（实际上是实体的一个代理，Hibernate 采用 CGLIB 工具来生成实体类的代理类，关于 CGLIB 的更多知识，请参考：http://cglib.sourceforge.net/），但是不会立即访问数据库加载持久状态，而是在第一次访问某个持久属性的时候才加载相应的持久状态。此外，getReference()方法不返回 null，如果在数据库中找不到相应的实体，这个方法会抛

出 javax.persistence.EntityNotFoundException（注：执行 getReference()时并不会抛出该例外，该例外是在第一次访问实体属性时才会被抛出）。在某些场合下使用 getReference()方法可以避免从数据库加载持久状态的性能开销，如下面的情况：

操作	执行的 SQL
em.remove(em.getReference(Person.class, 1))	delete from Person where personid = 1
em.remove(em.find(Person.class,1))	select * from Person where personid = 1 delete from Person where personid = 1

上面 em.getReference()不会真正去做数据库的 select 操作，而是把托管的 Person 实例返回出来，这样做避免了从数据库加载持久状态的性能开销。

这两个方法都可以在事务范围之外调用，如果持久上下文是 Transaction-scoped persistence context，则两个方法都会返回游离对象，如果是 extended persistence context，则两个方法都会返回托管对象。如果传进这两个方法的参数不是实体 Bean，都会引发 IllegalArgumentException 例外。

3.7.3 持久化实体 persist()

persist()方法用于将新创建的实体纳入EntityManager的管理，当执行flush操作后或事务提交时，实体的数据将被保存到数据库中。该方法执行后，传入persist()方法的实体对象将转换为托管状态。如果传入persist()方法的实体对象已经处于托管状态，那么调用persist()操作不会发生任何事情。如果对删除状态的实体执行persist()操作，它将会转换为托管状态。如果对游离状态的实体执行persist()操作，有可能在persist()方法调用后抛出EntityExistsException，也有可能在flush操作或事务提交时抛出EntityExistsException或PersistenceException。

```
@PersistenceContext protected EntityManager em;
public void savePerson() {
    Person person = new Person();
    person.setName(name);
    em.persist(person);//把数据保存进数据库中
}
```

当 persist()是在事务范围之外被调用。如果持久上下文是 Transaction-scoped persistence context 时，则会引发 TransactionRequiredException 例外，如果持久上下文是 extended persistence context 时，则是合法的。如果传递进 persist()的参数不是实体 Bean，将会引发 IllegalArgumentException 例外。

值得注意：如果实体与其它实体之间存在关联关系，并且设置了级联保存策略（CascadeType.All 或 CascadeType.PERSIST），那么 persist 操作会传播到其它实体。

3.7.4 更新实体

如果实体对象当前正处于托管状态，而且持久化上下文(persistence context) 已经与事务绑定，在实体上调用 setter 方法进行任何的修改都会在容器决定 flush 时，把更新的数据同步到数据库中。Flush 一般是在“通过 javax.persistence.Query 对象进行的查询”或事务提交时发生。下面代码 Flush 会在事务提交时发生。

```
@PersistenceContext protected EntityManager em;
public void updatePerson() {
    Person person = em.find(Person.class, 1);
```

```
    person.setName("lihuoming"); //事务提交后即可更新数据
}
```

3.7.5 合并 Merge()

merge ()方法用于处理实体的同步，即数据库插入和更新操作，因为实体 Bean 可以脱离 EntityManager 管理并被序列化到远程客户端，实体对象可能在远程客户端的本地被更新，然后传回服务器。在服务器端，我们需要重新将这个游离的实体纳入到 EntityManager 的管理中，使它的数据能被同步回数据库。

在下面例子中，客户端执行了 getPersonByID() 方法，person 被发送到远程客户端，这时 person 已经脱离了容器的管理，成为游离对象。在客户端对 person 进行修改，然后把 person 传回服务器。updatePerson()内部调用 merge ()方法将 person 重新纳入到 EntityManager 的管理中（注：此时 person 引用的是游离对象， merge ()方法返回的才是托管对象），致使 person 的数据能被同步到数据库中。客户端调用代码如下：

```
PersonDAO persondao = (PersonDAO) ctx.lookup("PersonDAOBean/remote");
Person person = persondao.getPersonByID(1); //取personid为1的person,此时的person已经脱离容器的管理
person.setName("张小艳");
persondao.updatePerson(person);
```

Session Bean 代码：

```
@PersistenceContext
protected EntityManager em;
......
public Person getPersonByID(int personid) {
   return em.find(Person.class, personid);
}
public void updatePerson(Person person) {
   em.merge(person);
}
```

当 merge ()是在事务范围之外被调用。如果持久上下文是 Transaction-scoped persistence context 时，则会引发 TransactionRequiredException 例外，如果持久上下文是 extended persistence context 时，则是合法的。如果传进 merge ()的参数不是实体 Bean，将会引发 IllegalArgumentException 例外。

对处于不同状态的实体调用 merge ()方法具有以下行为：

- 如果是新建状态的实体，那么会创建该实体的一份拷贝，并将这份拷贝纳入 EntityManager 的管理。利用这点我们也可以保存一个新建实体，如下：

  ```
  Person person = new Person();
  person.setName("李明玉");
  EntityManager.merge(person);
  ```

- 如果是托管状态的实体，那么 merge()操作会被忽略。
- 如果是删除状态的实体，那么 merge()操作会导致 IllegalArgumentException。
- 如果是游离（detached）状态的实体，执行规则如下：

  1. 如果此时容器已经存在相同主键的托管对象 B，容器会把参数传进来的实体的内容拷贝进这个托管对象

B，merge()方法返回这个托管对象 B 的引用，但参数引用的仍然是游离对象。容器在决定 Flush 时会把托管对象 B 的状态同步到数据库中。

2. 容器中不存在相同主键的托管对象。容器会创建该实体的一份拷贝 B，并将这份拷贝纳入 EntityManager 的管理，同时 merge()返回这个托管对象 B 的引用，但参数引用的仍然是游离对象。容器在决定 Flush 时会把托管对象 B 的状态同步到数据库中。

值得注意：如果实体与其它实体之间存在关联关系，并且设置了级联合并策略（CascadeType.All 或 CascadeType.MERGE），那么 merge 操作会传播到其它实体。下面 Person 与 IDCard 存在级联合并策略，为 Person 实体添加了一个新建状态的 IDCard 实体。当对 Person 调用了 merge()方法后，merge 操作被传播到 IDCard。因为 IDCard 实体处于新建状态，所以新建的 IDCard 实体将被保存到数据库中：

```
PersonDAO persondao = (PersonDAO) ctx.lookup("PersonDAOBean/remote");
Person person = persondao.getPersonByID(1);
IDCard idCard = new IDCard("345345345");
person.setIdcard(idCard);
persondao.updatePerson(person);
```

3.7.6 删除 Remove()

需要删除某个实体 bean，也就是在数据库中删除某条记录，你可以执行该方法。

```
@PersistenceContext protected EntityManager em;
......
Person person = em.getReference(Person.class, 2);
//如果级联关系cascade=CascadeType.ALL，当删除person时，也会把级联对象删除。把cascade属性设为CascadeType.REMOVE有同样的效果。
em.remove (person);
```

当 remove()是在事务范围之外被调用。如果持久上下文是 Transaction-scoped persistence context 时，则会引发 TransactionRequiredException 例外，如果持久上下文是 extended persistence context 时，则是合法的。如果传入 remove()的参数不是实体 Bean，将会引发 IllegalArgumentException 例外。

对处于不同状态的实体调用 remove ()方法具有以下行为：

- 如果是新建状态的实体，那么操作会被忽略。
- 如果是托管状态的实体，那么实体将会被删除。
- 如果是删除状态的实体，那么操作会被忽略。
- 如果是游离（detached）状态的实体，那么操作会导致 IllegalArgumentException。

值得注意：如果实体与其它实体之间存在关联关系，并且设置了级联删除策略（CascadeType.All 或 CascadeType.REMOVE），那么 remove 操作会传播到其它实体。

3.7.7 执行 JPQL 操作 createQuery()

除了可以使用 find()或 getReference()获取实体 Bean 之外，你还可以使用 JPQL 查询获取实体 Bean。要执行 JPQL 语句，你可以通过 createQuery()方法创建一个 Query 对象。如：

```
@PersistenceContext protected EntityManager em;
.....
Query query = em.createQuery("select p from Person p where p. name='黎明'");
List result = query.getResultList();
Iterator iterator = result.iterator();
while( iterator.hasNext() ){
   //处理Person
}
......
// 执行更新语句
Query query = em.createQuery("update Person as p set p.name =?1 where p.
personid=?2");
query.setParameter(1, “黎明” );
query.setParameter(2, new Integer(1) );
int result = query.executeUpdate(); //影响的记录数
......
// 执行更新语句
Query query = em.createQuery("delete from Person");
int result = query.executeUpdate(); //影响的记录数
```

3.7.8 执行 SQL 操作 createNativeQuery()

如果你需要执行 SQL 语句，你可以使用此方法。注意，这里操作的是 SQL 语句，而非 JPQL，千万别搞晕了。

```
@PersistenceContext protected EntityManager em;
......
//我们可以让EJB3 Persistence运行环境将列值直接填充入Entity，并将Entity作为结果返回.
Query query = em.createNativeQuery("select * from person", Person.class);
List result = query.getResultList();
if (result!=null){
   Iterator iterator = result.iterator();
   while( iterator.hasNext() ){
       Person person= (Person)iterator.next();
       ......
   }
}
......
// 直接通过SQL执行更新语句
Query query = em.createNativeQuery("update person set age=age+2");
query.executeUpdate();
```

3.7.9 刷新实体 refresh()

如果你怀疑当前托管对象存放的并非数据库中最新的数据时（如：在你获取了实体对象之后，有人偷偷摸摸在数

据库中修改了该记录），你可以通过 refresh()方法刷新实体对象。容器会把数据库中的新值重写进实体对象。如果实体存在关联，并且设置了 CascadeType.REFRESH 或 CascadeType.ALL，刷新一个实体对象，那么与之关联的实体对象也会被刷新，刷新操作不会把实体对象的状态同步到数据库。

```
@PersistenceContext protected EntityManager em;
......
Person person = em.find(Person.class, 2);
//如果此时person对应的记录在数据库中已经发生了改变，可以通过refresh()方法得到最新数据。
em.refresh (person);
```

当 refresh()是在事务范围之外被调用。如果持久上下文是 Transaction-scoped persistence context 时，则会引发 TransactionRequiredException 例外，如果持久上下文是 extended persistence context 时，则是合法的。如果传进 refresh()的参数不是实体 Bean，将会引发 IllegalArgumentException 例外。如果刷新的实体在数据库中不存在，则会引发 EntityNotFoundException 例外。

对处于不同状态的实体调用 refresh ()方法具有以下行为：

- 如果是新建状态的实体，那么操作会导致 IllegalArgumentException。
- 如果是托管状态的实体，那么将会刷新它的持久状态。
- 如果是删除状态的实体，那么操作会导致 IllegalArgumentException。
- 如果是游离（detached）状态的实体，那么操作会导致 IllegalArgumentException。

3.7.10 检测实体是否处于托管状态 contains()

contains()方法使用一个实体作为参数，如果当前的持久化上下文（persistence context）包含指定的实体对象，返回为 true，否则为 false。实体存在于 persistence context 中表明实体处在托管状态。
如果传递的参数不是实体 Bean，将会引发一个 IllegalArgumentException 例外。

```
@PersistenceContext protected EntityManager em;
Person person = em.find(Person.class, 2);
if (em.contains(person)){
  //正在被持久化内容管理
}else{
  //已经不受持久化内容管理
}
```

3.7.11 分离所有正在托管的实体 clear()

在处理大量实体的时候，如果你不把已经处理过的实体从 persistence context 中分离出来，将会消耗你大量的内存。调用 EntityManager.clear()方法后，所有托管状态的实体将会从持久上下文(persistence context)中清除，此时的实体成为游离状态。

特别说明，在事务没有提交前（事务默认在调用堆栈的最后提交，如：方法的返回）调用 clear()方法，之前对实体所作的任何改变将会掉失，建议你在调用 clear()方法前先调用 flush()方法保存更改。

3.7.12 刷新 flush()与设置 flush 模式 setFlushMode()

当你调用 EntityManager.persist( )、EntityManager.merge( )、EntityManager.remove( ) 或托管对象 setter 这些方法时，实体的状态并不会立刻同步到数据库中，直到 EntityManager 决定 flush 时，才会把实体的状态同步到数据库。这样做的目的是为了减少跟数据库交互的次数，以提高应用的性能。容器会将所有数据库操作集中到一个批处理中，并在某个时刻提交到数据库。

默认的 flush 模式是 FlushModeType.AUTO，代表 flush 是在查询语句执行前或事务提交时才发生，“查询语句”指的是通过 javax.persistence.Query 对象执行的查询，而并非 find()或 getreference()进行的查询，这两个方法都不会引起 flush，这是因为通过主键来查询实体是不会受到任何更新操作的影响。flush 还有另外一种刷新模式 FlushModeType.COMMIT，它代表 flush 只有在事务提交时才发生，如果是容器管理事务，事务提交的时间发生在方法执行结束时。Flush 的次数也就是 JDBC 跟数据库交互的次数。在跟数据库交互次数方面，FlushModeType.AUTO 模式大于等于 1，大于 1 时是发生在一系列更新操作中参杂了查询语句执行，等于 1 时是发生在一系列更新操作中没有任何查询语句执行。而 FlushModeType.COMMIT 模式等于 1，仅仅在事务提交时才执行一次。由此看来 FlushModeType.AUTO 模式比较智能，可以决定什么时候 flush，但 flush 的次数会因查询语句存在而大于 1。FlushModeType.COMMIT 模式显的更有效率，不管是否存在查询语句，它都只会发生一次 flush。按理这么说，直接使用 FlushModeType.COMMIT 模式就挺好的了，为什么还要 FlushModeType.AUTO 模式。下面我们看看这种情况：

这部分内容在《EJB3.0入门经典》中

3.7.13 获取持久化实现者的引用 getDelegate( )

用过 getDelegate( )方法，你可以获取 JPA 实现产品的引用。如：获取 Jboss JPA 实现产品 Hibernate 的引用：

```
@PersistenceContext protected EntityManager em;
HibernateEntityManager manager = (HibernateEntityManager)em.getDelegate();
```

获取 Hibernate 的引用后，可以直接面对 Hibernate 进行编码，不过这种做法不建议使用。

3.8 关系/对象映射

3.8.1 双向一对多及多对一映射

现实应用中存在很多一对多的情况，如一项订单中存在一个或多个订购项。当 one 方存在与 many 方关系的定义，而 many 方同时也存在与 one 方关系的定义，这样的关系被称为双向关系。代码上体现为在 one 方有一个集合属性指向 many 方，而在 many 方也有一个属性指向 one 方，下面以订单为例介绍具有一对多及多对一双向关系的实体 bean 开发。

需要映射的数据库表（注意：你不需要创建数据表，本例子使用持久化驱动自动生成数据表)

orders

字段名称	字段类型属性	描述
orderid(主键)	Int not null	订单号
amount	Float null	订单金额
createdate	Datetime null	订单创建日期

orderitem

字段名称	字段类型属性	描述
id(主键)	Int not null	订单项 ID
productname	Varchar(255) not null	订购产品名称
price	Float null	产品价格
order_id	Int null	订单号

双向一对多关系，必须包含一个关系维护端。目前，持久化规范要求多的一方为关系维护端（owner side），一的一方为关系被维护端（inverse side）。我们可以在 one 方的@OneToMany 注释设置 mappedBy 属性，以指定它是这一关联中的被维护端，Many 方是关系维护端。

Order.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Date;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.OrderBy;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "Orders")
public class Order implements Serializable {
    private static final long serialVersionUID = 4970325922198249712L;
    private Integer orderid;
    private Float amount;
    private Set<OrderItem> orderItems = new HashSet<OrderItem>();
    private Date createdate;

    @Id
    @GeneratedValue
    public Integer getOrderid() {
        return orderid;
```

```
    }
    public void setOrderid(Integer orderid) {
        this.orderid = orderid;
    }

    public Float getAmount() {
        return amount;
    }
    public void setAmount(Float amount) {
        this.amount = amount;
    }

    @OneToMany(mappedBy="order",cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @OrderBy(value = "id ASC")
    public Set<OrderItem> getOrderItems() {
        return orderItems;
    }
    public void setOrderItems(Set<OrderItem> orderItems) {
        this.orderItems = orderItems;
    }

    @Temporal(value=TemporalType.TIMESTAMP)
    public Date getCreatedate() {
        return createdate;
    }
    public void setCreatedate(Date createdate) {
        this.createdate = createdate;
    }

    public void addOrderItem(OrderItem orderitem) {
       if (!this.orderItems.contains(orderitem)) {
            this.orderItems.add(orderitem);
            orderitem.setOrder(this);
       }
    }

    public void removeOrderItem(OrderItem orderitem) {
        if (this.orderItems.contains(orderitem)) {
            orderitem.setOrder(null);
            this.orderItems.remove(orderitem);
        }
    }

    /**
```

```
     * 返回对象的散列代码值。该实现根据此对象
     * 中 orderid 字段计算散列代码值。
     * @return 此对象的散列代码值。
     */
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.orderid != null ? this.orderid.hashCode() : 0);
        return hash;
    }

    /**
     * 确定其他对象是否等于此 Order。当且仅当
     * 参数不为 null 且该参数是具有与此对象相同 orderid 字段值的 Order 对象时，
     * 结果才为 <code>true</code>。
     * @param 对象，要比较的引用对象
     * 如果此对象与参数相同，则 @return <code>true</code>；
     * 否则为 <code>false</code>。
     */
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Order)) {
            return false;
        }
        Order other = (Order)object;
        if (this.orderid != other.orderid && (this.orderid == null
|| !this.orderid.equals(other.orderid))) return false;
        return true;
    }

    /**
     * 返回对象的字符串表示法。该实现根据 orderid 字段
     * 构造此表示法。
     * @return 对象的字符串表示法。
     */
    @Override
    public String toString() {
        return this.getClass().getName()+ "[orderid=" + orderid + "]";
    }
}
```

上面声明一个 Set 类型的成员属性 orderItems 用来存放 OrderItem 对象集合，注释 @OneToMany(mappedBy="order",cascade = CascadeType.ALL, fetch = FetchType.LAZY)指明 Order 与 OrderItem 关联关系为一对多关系，下面我们看看@OneToMany 注释的定义：

public @interface OneToMany

```
{
    Class targetEntity( ) default void.class;
    CascadeType[] cascade( ) default {};
    FetchType fetch( ) default LAZY;
    String mappedBy( ) default "";
}
```

targetEntity( ) 属性指定你要关联的实体类。通常情况下你不需要指定该属性，persistence provider 会根据成员属性的类型自动对它进行设置。

cascade( ) 该属性定义类和类之间的级联关系。级联关系定义对当前对象的操作将波及到关联类对象，而且这种关系是递归调用的。举个例子：Order 和 OrderItem 有级联删除关系，那么删除 Order 时将同时删除关联的 OrderItem 对象。如果 OrderItem 和其它对象之间还有级联删除关系，那么这样的操作会一直递归执行下去。cascade 的值只能从 CascadeType.PERSIST（级联新建）、CascadeType.REMOVE（级联删除）、CascadeType.REFRESH（级联刷新）、CascadeType.MERGE（级联更新）中选择一个或多个，如果你想包含这四项，可以选择 CascadeType.ALL。这四种级联操作分别在 EntityManager 的 persist()、remove()、refresh ()和 merge()方法调用时才会发生。

fetch( ) 属性指定成员属性在首次加载实体时，是延迟加载还是立即加载。可选择项包括：FetchType.EAGER 和 FetchType.LAZY。FetchType.EAGER表示关系类(OrderItem 类)在主类(Order类)加载的时候同时加载，FetchType.LAZY表示关系类在被访问时才加载。默认值是FetchType.LAZY。

mappedBy( ) 定义类之间的双向关系。如果类之间是单向关系，不需要指定该值。如果类和类之间形成双向关系，我们就需要使用这个属性进行定义，否则会引起数据一致性的问题。该属性告诉 persistence manager，用于将该关联关系映射到数据表所需的信息是在被关联类中定义的。该属性值为被关联类中指向自己的成员属性。如：OrderItem 含有一个 order 成员属性，在该成员属性的 getter 方法上定义了与 Order 的关联关系。

```
@ManyToOne(cascade=CascadeType.REFRESH,optional=false)
public Order getOrder() {
    return order;
}
```

那么在 Order 中，@OneToMany 注释的 mappedBy()就应设为 order。

```
@OneToMany(mappedBy="order",cascade = CascadeType.ALL, fetch = FetchType.LAZY)
public Set<OrderItem> getOrderItems() {
}
```

@OrderBy(value = "id ASC")注释指明加载 OrderItem 时按 id 的升序排序，如果对多个属性进行排序可以用逗号分隔，如：@OrderBy(value = "id ASC, price desc")

OrderItem.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity
```

```
public class OrderItem implements Serializable {
    private static final long serialVersionUID = -1166337687856636179L;
    private Integer id;
    private String productname;
    private Float price;
    private Order order;

    public OrderItem() {}

    public OrderItem(String productname, Float price) {
        this.productname = productname;
        this.price = price;
    }

    @Id
    @GeneratedValue
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }

    @Column(length=100, nullable=false)
    public String getProductname() {
        return productname;
    }
    public void setProductname(String productname) {
        this.productname = productname;
    }

    public Float getPrice() {
        return price;
    }
    public void setPrice(Float price) {
        this.price = price;
    }

    @ManyToOne(cascade=CascadeType.REFRESH,optional=false)
    @JoinColumn(name = "order_id")
    public Order getOrder() {
        return order;
    }
    public void setOrder(Order order) {
```

```
        this.order = order;
    }

    /**
     * 返回对象的散列代码值。该实现根据此对象
     * 中 id 字段计算散列代码值。
     * @return 此对象的散列代码值。
     */
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.id != null ? this.id.hashCode() : super.hashCode());
        return hash;
    }

    /**
     * 确定其他对象是否等于此 OrderItem。当且仅当
     * 参数不为 null 且该参数是具有与此对象相同 id 字段值的 OrderItem 对象时，
     * 结果才为 <code>true</code>。
     * @param 对象，要比较的引用对象
     * 如果此对象与参数相同，则 @return <code>true</code>；
     * 否则为 <code>false</code>。
     */
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof OrderItem)) {
            return false;
        }
        OrderItem other = (OrderItem)object;
        if (this.id != other.id && (this.id == null || !this.id.equals(other.id)))
return false;
        return true;
    }

    /**
     * 返回对象的字符串表示法。该实现根据 id 字段
     * 构造此表示法。
     * @return 对象的字符串表示法。
     */
    @Override
    public String toString() {
        return this.getClass().getName()+ "[id=" + id + "]";
    }
}
```

注释@ManyToOne 指定 OrderItem 和 Order 之间为多对一关系，多个 OrderItem 实例关联的都是同一个 Order 对象。下面是@ManyToOne 注释的定义：

```
public @interface ManyToOne
{
    Class targetEntity( ) default void.class;
    CascadeType[] cascade( ) default {};
    FetchType fetch( ) default EAGER;
    boolean optional( ) default true;
}
```

targetEntity()、cascade()和 fetch()的具体含义和@OneToMany 注释的同名属性相同，但@ManyToOne 注释的 fetch()属性默认值是 FetchType.EAGER。

optional() 指定关联方是否可以为空(null)，该属性值默认为 true。若将其设为 false，则要求双方必须存在。如果使用 persistence provider 生成数据库 schema，optional()属性值会影响到生成的外键字段是否允许为空。当属性值为 false 时，外键字段不允许为 null，当属性值为 true 时，外键字段允许为 null。因此，我们不再需要设置@JoinColumn 注释的 nullable()属性。在执行 EntityManager.find()、EntityManager.getReference()查询时，optional()属性值影响关联查询的形式。如：通过 EntityManager.find()、EntityManager.getReference()查询 OrderItem 实体，当 optional=false 时，OrderItem 与 Order 关联形式为 inner join，当 optional=true 时，OrderItem 与 Order 关联形式为 left join。

代码片断解释如下：

```
public class OrderItem implements Serializable {
    @ManyToOne(cascade=CascadeType.REFRESH,optional=false)
    @JoinColumn(name = "order_id")
    public Order getOrder() {
        return order;
    }
}
@PersistenceContext EntityManager em;
OrderItem item = em.find(OrderItem.class, orderItemid);
```

此时的 SQL 为：select * from OrderItem item inner join Orders o on o.order_id=item.id

OrderItem 表与 orders 表都必须有关联记录时，查询结果才有记录。如果 OrderItem 表有记录，但 orders 表没有记录，查询结果为空

```
public class OrderItem implements Serializable {
    @ManyToOne(cascade=CascadeType.REFRESH,optional=true)
    @JoinColumn(name = "order_id")
    public Order getOrder() {
        return order;
}
@PersistenceContext EntityManager em;
OrderItem item = em.find(OrderItem.class, orderItemid);
```

此时的 SQL 为：select * from OrderItem item left outer join Orders o on o.order_id=item.id

如果 orders 表没有记录，OrderItem 表有记录，查询结果仍有记录。

@JoinColumn(name = "order_id")注释指定 OrderItem 表指向 Orders 表主键的外键字段，外健字段的名称为 order_id。下面是@JoinColumn 注释的定义：

```
public @interface JoinColumn
```

```
{
    String name() default "";
    String referencedColumnName() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
}
```

name() 指定外键字段的名称。

referencedColumnName() 指定外键与被关联实体那个字段关联，在本例它默认跟 Orders 表的主键关联，如果需要跟 Orders 表主键以外的其他字段关联，可以在此指定。

unique()属性指定字段的值具有唯一性。

nullable() 指定外键字段是否允许为 null。

insertable() 指定该字段是否出现在 insert 语句中。注：保存实体的时候，persistence provider 使用实体的成员属性映射的字段名构造 insert SQL 语句。默认所有成员属性映射的字段都会出现在 insert 语句中，如果你不希望某个字段出现在 insert 语句中，可以设为 false。

updatable()指定该字段是否出现在 update 语句中。注：更新实体的时候，persistence provider 使用实体的成员属性映射的字段名构造 update SQL 语句。默认所有成员属性映射的字段都会出现在 update 语句中，如果你不希望某个字段出现在 update 语句中，可以设为 false。

columnDefinition() 会在 persistence provider 自动生成数据库 schema 时用到，我们可以用它来指定 referencedColumnName()对应字段的 DDL。

table() 指定字段在那个表中。如果映射字段不是分散在多张表中，我们不需要指定该值。persistence provider 能自动识别实体所对应的表。

如果你采用 persistence provider 生成数据库 schema，@JoinColumn 注释并不是必须指定的。persistence provider 会自动为你生成必要的外键映射。如把上面的代码改写成：

```
@Entity
public class OrderItem implements Serializable {
    ...
    @ManyToOne(cascade=CascadeType.REFRESH, optional=false)
    public Order getOrder() {
        return order;
    }
    ...
}
```

将自动生成如下数据库表:

```
CREATE TABLE `orderitem` (
  `id` int(11) NOT NULL auto_increment,
  `productname` varchar(100) NOT NULL,
  `price` float default NULL,
  `order_orderid` int(11) default NULL,
  PRIMARY KEY  (`id`),
```

```
  KEY `FK60163F611F3BCD80` (`order_orderid`),
  CONSTRAINT `FK60163F611F3BCD80` FOREIGN KEY (`order_orderid`) REFERENCES `orders` (`orderid`)
)
```

默认映射会创建一个外键字段，并且具有外键约束，该外键字段的名字由如下几个部分组成：参与映射的成员属性名称(order)，加上一个“_”字符，再加上被引用表的主键字段名称(orderid)。

为了使用上面的实体 Bean，我们定义了一个 Session Bean，下面是 Session Bean 的业务接口。

OrderDAO.java

```
package com.foshanshop.ejb3;
import java.util.List;
import com.foshanshop.ejb3.bean.Order;

public interface OrderDAO {
    /**
     * 添加一个订单
     *
     */
    public void insertOrder(Order order);
    /**
     * 获取指定订单
     * @param orderid 订单号
     * @return
     */
    public Order getOrderByID(Integer orderid);
    /**
     * 获取所有订单
     * @return
     */
    public List<Order> getAllOrder();
}
```

OrderDAOBean.java

```
package com.foshanshop.ejb3.impl;
import java.util.List;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import com.foshanshop.ejb3.OrderDAO;
import com.foshanshop.ejb3.bean.Order;

@Stateless
@Remote (OrderDAO.class)
```

```
public class OrderDAOBean implements OrderDAO {
    @PersistenceContext protected EntityManager em;

    public void insertOrder(Order order){
        em.persist(order);
    }

    public Order getOrderByID(Integer orderid) {
       Order order =  em.find(Order.class, orderid);
       order.getOrderItems().size();
       //因为是延迟加载，通过执行size()这种方式获取订单下的所有订单项
       return order;
    }

    @SuppressWarnings("unchecked")
    public List<Order> getAllOrder() {
        Query query = em.createQuery("select DISTINCT o from Order o inner join fetch
o.orderItems order by o.orderid");
        return (List<Order>) query.getResultList();
    }
}
```

上面 Order 对关联类(OrderItem)设置了延迟加载，所谓延迟加载，就是指在首次从数据库加载实体(Order)时，延迟属性(orderItems)并不会被初始化，只有在首次访问延迟属性时才会被加载。首次访问延迟属性应在实体对象还处于托管状态时进行。如果实体对象成为游离对象，你不能访问延迟属性，否则会引发加载例外。

在前面的章节中，我们了解到，随着 persistence context 关闭，实体对象就会成为游离对象。游离对象有个值的注意的特征是，它可以被序列化并通过网络发送给远程客户端。在客户端，如果你需要访问延迟属性，应确保实体对象返回客户端前加载延迟属性，否则会抛出加载例外。因为延迟加载会导致 JDBC 跟数据库多次交互，为了提高性能，你可以通过 join fetch 关联语句查询实体，这样只需要一次数据库交互，如：本例的业务方法 getAllOrder 。

下面是本例子的持久化配置文件（文件放在 jar 文件的 META-INF 目录）：

persistence.xml

```
<?xml version="1.0"?>
<persistence 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_1_0.xsd" version="1.0">
  <persistence-unit name="foshanshop">
    <jta-data-source>java:/DefaultMySqlDS</jta-data-source>
    <properties>
     <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
     <!-- 调整JDBC抓取数量的大小: Statement.setFetchSize() -->
     <property name="hibernate.jdbc.fetch_size" value="18"/>
```

```
    <!-- 调整JDBC批量更新数量 -->
    <property name="hibernate.jdbc.batch_size" value="10"/>
    <!-- 显示最终执行的SQL -->
    <property name="hibernate.show_sql" value="true"/>
    <!-- 格式化显示的SQL -->
    <property name="hibernate.format_sql" value="true"/>
   </properties>
  </persistence-unit>
</persistence>
```

下面是 JSP 客户端代码：OneToManyTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.OrderDAO,com.foshanshop.ejb3.bean.*,
             javax.naming.*,
             java.util.*"%>
<%
    try {
        InitialContext ctx = new InitialContext();
        OrderDAO orderdao = (OrderDAO) ctx.lookup("OrderDAOBean/remote");
       Order neworder = new Order();
       neworder.setCreatedate(new Date());
       neworder.addOrderItem(new OrderItem("笔记本电脑", new Float(13200.5)));
       neworder.addOrderItem(new OrderItem("U盘", new Float(620)));
       neworder.setAmount(new Float(13200.5+620));
        orderdao.insertOrder(neworder);
       List<Order> list = orderdao.getAllOrder();
       for(Order od : list){
           out.println("======订单号: "+ od.getOrderid() +"======<br>");
           for(OrderItem item : od.getOrderItems()){
               out.println("订购产品:"+ item.getProductname() +"<br>");
           }
       }
    } catch (Exception e) {
        out.println(e.getMessage());
    }
%>
```

本例子的源代码在配套光盘的 OneToMany 文件夹。例子使用的数据源配置文件是 mysql-ds.xml，你可以在配套光盘中找到。部署数据源到 Jboss 前，请先把数据库驱动拷贝到[jboss 安装目录]/server/default/lib 目录（请用你的配置名替换 default），拷贝完成后需重启 Jboss。要恢复 OneToMany 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务

本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/OneToManyTest.jsp 访问客户端。

关联表映射

除了可以在 many 方建立外键外，我们也可以使用中间表建立外键，中间表信息使用@JoinTable 注释定义。下面是使用了中间表（order_orderitem）的数据图：

这部分内容在《EJB3.0入门经典》中

3.8.2 单向一对多

虽然大部分应用使用的都是双向关系。但有些情况下，两个实体并不需要互有关系。在单向关联中，两个实体中的一方定义了一个指向对方的成员属性，然后通过这个成员属性获取或设置另一方。单向一对多关联是在一方定义了一个含有 Many 方关系的成员属性，而在多方并没有定义含有 One 方关系的成员属性。我们可以把前面双向一对多例子改成单向一对多，代码如下：

这部分内容在《EJB3.0入门经典》中

3.8.3 单向多对一

单向多对一关联是在 many 方定义了一个含有 One 方关系的成员属性，而在 one 方并没有定义含有 Many 方关系的成员属性。我们可以把前面双向多对一例子改成单向多对一，代码如下：

这部分内容在《EJB3.0入门经典》中

3.8.4 双向一对一映射

每个人（Person）只有唯一的身份证 (IDCard)，而每张身份证只有唯一的主人，Person 与 IDCard 形成了一对一关系。因为它们之间互有对方的关系，因此形成的是双向关系。下面以它们为例介绍具有一对一关系的实体 Bean 开发。

需要映射的数据库表

person

字段名称	字段类型属性	描述
personid (主键)	Int(11) not null	人员 ID
PersonName	Varchar(32) not null	姓名
sex	Tinyint(1) not null	性别
age	Smallint(6) not null	年龄
birthday	Datetime null	出生日期
idcard_id	Int(11) null	作为外键指向 idcard 表的主键

idcard

字段名称	字段类型属性	描述
id (主键)	Int(11) not null	流水号
cardno	Varchar(18) not null	身份证号

在双向一对一关联中，我们需要在关系被维护端（inverse side）的@OneToOne 注释中定义 mappedBy 属性，以指定它是这一关联中的被维护端。同时需要在关系维护端（owner side）建立外键列指向关系被维护端的主键列。下面是关系维护端 Person.java 的源代码：

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import java.util.Date;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Person  implements Serializable{
    private static final long serialVersionUID = 8305710085432450318L;
    private Integer personid;
    private String name;
    private boolean sex;
    private Short age;
    private Date birthday;
    private IDCard idcard;

    @Id
    @GeneratedValue
    public Integer getPersonid() {
        return personid;
    }
```

```
    public void setPersonid(Integer personid) {
        this.personid = personid;
    }

    @Column(name = "PersonName",nullable=false,length=32)
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @Column(nullable=false)
    public boolean getSex() {
        return sex;
    }
    public void setSex(boolean sex) {
        this.sex = sex;
    }

    @Column(nullable=false)
    public Short getAge() {
        return age;
    }
    public void setAge(Short age) {
        this.age = age;
    }

    @Temporal(value=TemporalType.DATE)
    public Date getBirthday() {
        return birthday;
    }
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    @OneToOne(optional = true, cascade = CascadeType.ALL)
    @JoinColumn(name = "idcard_id", unique = true)
    public IDCard getIdcard() {
        return idcard;
    }
    public void setIdcard(IDCard idcard) {
        this.idcard = idcard;
    }
```

```
    /**
     * 返回对象的散列代码值。该实现根据此对象
     * 中 personid 字段计算散列代码值。
     * @return 此对象的散列代码值。
     */
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.personid != null ? this.personid.hashCode() : 0);
        return hash;
    }

    /**
     * 确定其他对象是否等于此 Person。当且仅当
     * 参数不为 null 且该参数是具有与此对象相同 personid 字段值的 Person 对象时，
     * 结果才为 <code>true</code>。
     * @param 对象，要比较的引用对象
     * 如果此对象与参数相同，则 @return <code>true</code>；
     * 否则为 <code>false</code>。
     */
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Person)) {
            return false;
        }
        Person other = (Person)object;
        if (this.personid != other.personid && (this.personid == null
|| !this.personid.equals(other.personid))) return false;
        return true;
    }

    /**
     * 返回对象的字符串表示法。该实现根据 personid 字段
     * 构造此表示法。
     * @return 对象的字符串表示法。
     */
    @Override
    public String toString() {
        return this.getClass().getName()+ "[personid=" + personid+ "]";
    }
}
```

@OneToOne 注释指定 Person 与 IDCard 具有一对一关系，下面是@OneToOne 注释的定义：

public @interface OneToOne

{

```
    Class targetEntity( ) default void.class;
    CascadeType[] cascade( ) default {};
    FetchType fetch( ) default EAGER;
    boolean optional( ) default true;
    String mappedBy( ) default "";
}
```

@OneToOne 注释的五个属性：targetEntity()、cascade()、fetch()、mappedBy()和 optional()，前四个属性的含义与@OneToMany 注释的同名属性一一对应，fetch()属性的默认值是 FetchType.EAGER。

optional()属性的含义与@ManyToOne 注释的同名属性对应，optional = true 允许 idcard 属性为 null，也就是允许获取到没有身份证的人员，因为现实生活中未成年人是没有身份证的。

@JoinColumn(name = "idcard_id", unique = true) 注释指定 Person 表指向 IDCard 表主键的外键字段。外键字段的名称为 idcard_id。unique = true 指定字段的值具有唯一性，不可重复。如果你采用 persistence provider 生成数据库 schema，@JoinColumn 注释并不是必须指定的，persistence provider 会自动为你生成必要的外键映射。如把上面的代码写成：

```
@Entity
public class Person  implements Serializable{
    ...
    private IDCard idcard;
    @OneToOne(optional = true, cascade = CascadeType.ALL)
    public IDCard getIdcard() {
        return idcard;
    }
    ...
}
```

将自动生成如下数据库表:

```
CREATE TABLE `person` (
   `personid` int(11) NOT NULL auto_increment,
   `idcard_id` int(11) default NULL,
   ...
   PRIMARY KEY    (`personid`),
   KEY `FK8E48877543A6CBE0` (`idcard_id`),
   CONSTRAINT `FK8E48877543A6CBE0` FOREIGN KEY (`idcard_id`) REFERENCES `idcard` (`id`)
)
```

对于双向的 one-to-one 关联，默认映射会创建一个外键字段。该外键的名称由如下几个部分组合：

参与映射的成员属性名称(idcard)，加上一个“-”字符，再加上被引用表(idcard)的主键字段名称(id)。像本例就生成了 idcard_id 字段。

IDCard.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class IDCard  implements Serializable{
    private static final long serialVersionUID = -4610000117411667607L;
    private Integer id;
    private String cardno;
    private Person person;

    public IDCard() {}

    public IDCard(String cardno) {
        this.cardno = cardno;
    }

    @Id
    @GeneratedValue
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }

    @Column(nullable=false,length=18,unique = true)
    public String getCardno() {
        return cardno;
    }
    public void setCardno(String cardno) {
        this.cardno = cardno;
    }

    @OneToOne(optional = false, cascade = CascadeType.REFRESH, mappedBy = "idcard")
    public Person getPerson() {
        return person;
    }
    public void setPerson(Person person) {
        this.person = person;
    }
    /**
     * 返回对象的散列代码值。该实现根据此对象
     * 中 id 字段计算散列代码值。
     * @return 此对象的散列代码值。
```

```
     */
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.id != null ? this.id.hashCode() : 0);
        return hash;
    }

    /**
     * 确定其他对象是否等于此 IDCard。当且仅当
     * 参数不为 null 且该参数是具有与此对象相同 id 字段值的 IDCard 对象时，
     * 结果才为 <code>true</code>。
     * @param 对象，要比较的引用对象
     * 如果此对象与参数相同，则 @return <code>true</code>；
     * 否则为 <code>false</code>。
     */
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof IDCard)) {
            return false;
        }
        IDCard other = (IDCard)object;
        if (this.id != other.id && (this.id == null || !this.id.equals(other.id)))
return false;
        return true;
    }

    /**
     * 返回对象的字符串表示法。该实现根据 id 字段
     * 构造此表示法。
     * @return 对象的字符串表示法。
     */
    @Override
    public String toString() {
        return this.getClass().getName()+ "[id=" + id + "]";
    }
}
```

@OneToOne 注释指定 IDCard 与 Person 具有一对一关系，IDCard 是关系被维护端（注：以 mappedBy()属性标记 OneToOne 关联时，也就指明了它是这一关联中的被维护端）。optional = false 指定 person 属性不能为 null（因为身份证肯定会有对应的主人）。mappedBy = "idcard"设置双向关联，告诉 persistence manager，用于该关联关系映射到数据表所需的信息是在 Person 的 idcard 属性中定义的。

因为 IDCard 是关系被维护端，所以如果你想将 IDCard 实例与另一个 Person 关联起来，就必须调用维护端 Person 的 setIdcard()方法，并传入 null，如下：

```
@PersistenceContext protected EntityManager em;
```

```
IDCard card = em.find(IDCard.class, 1);
card.getPerson().setIdcard(null);
Person person = em.find(Person.class, 500);
person.setIdcard(card);
```

为了使用上面的实体 Bean，我们定义了一个 Session Bean，下面是 Session Bean 的业务接口：

OneToOneDAO.java

```
package com.foshanshop.ejb3;
import com.foshanshop.ejb3.bean.Person;

public interface OneToOneDAO {
    /**
     * 添加一个人
     * @param person 人员
     */
    public void insertPerson(Person person);
    /**
     * 获取指定人员
     * @param orderid 人员id
     * @return
     */
    public Person getPersonByID(Integer orderid);
    /**
     * 更新人员姓名,身份证号码
     * @param personid 人员id
     * @param newname 新姓名
     * @param newIDcard 新身份证号码
     */
    public void updatePersonInfo(Integer personid, String newname, String
newIDcard);
    /**
     * 删除指定人员,因为设了级联删除,会连同身份证一起删除
     * @param personid 人员id
     */
    public void deletePerson(Integer personid);
}
```

OneToOneDAOBean.java

```
package com.foshanshop.ejb3.impl;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import com.foshanshop.ejb3.OneToOneDAO;
```

```
import com.foshanshop.ejb3.bean.Person;

@Stateless
@Remote (OneToOneDAO.class)
public class OneToOneDAOBean implements OneToOneDAO {
    @PersistenceContext protected EntityManager em;

    public void insertPerson(Person person) {
        em.persist(person);
    }

    public Person getPersonByID(Integer personid) {
        Person person = em.find(Person.class, personid);
        return person;
    }

    public void updatePersonInfo(Integer personid, String newname, String
newIDcard) {
        Person person = em.find(Person.class, personid);
        if (person!=null) {
            person.setName(newname);
            if (person.getIdcard()!=null){
                person.getIdcard().setCardno(newIDcard);
            }
        }
    }

    public void deletePerson(Integer personid) {
        Person person = em.find(Person.class, personid);
        if (person!=null) em.remove(person);
    }
}
```

下面是 JSP 客户端代码：OneToOneTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.OneToOneDAO,com.foshanshop.ejb3.bean.*,
                javax.naming.*,
                java.text.SimpleDateFormat,
                java.util.*"%>
<%
    try {
        InitialContext ctx = new InitialContext();
        String outformate = "<font color=blue>CMD>>Out>></font> ";
        OneToOneDAO oneToonedao = (OneToOneDAO)
```

```
ctx.lookup("OneToOneDAOBean/remote");
        SimpleDateFormat formatter = new SimpleDateFormat("MMddhhmmss");
        String endno = formatter.format(new Date()).toString();
       Person newperson = new Person();
       newperson.setName("叶子由");
       newperson.setSex(true);
       newperson.setAge((short)26);
       newperson.setBirthday(new Date());
       IDCard idcard = new IDCard("44011"+endno);
       idcard.setPerson(newperson);
       newperson.setIdcard(idcard);
        oneToonedao.insertPerson(newperson);
        //添加时请注意，身份证号不要重复，因为数据库字段身份证号是唯一的
        Person person = oneToonedao.getPersonByID(new Integer(1));
        if (person!=null){
            out.println(outformate +"寻找编号为1的人员<br>");
            out.println("姓名:"+ person.getName() +" 身份证: "+
person.getIdcard().getCardno() +"<br>");
        }else{
            out.println("没有找到编号为1的人员<br>");
        }
        out.println(outformate +"更新编号为1的人员的姓名为李明,身份证号为33012" +endno
+"<br>");
        oneToonedao.updatePersonInfo(new Integer(1), "李明", "33012" +endno);


        out.println("================删除编号为3的人员==============<br>");
        oneToonedao.deletePerson(new Integer(3));

    } catch (Exception e) {
        out.println(e.getMessage());
    }
%>
```

本例子的源代码在配套光盘的 OneToOne 文件夹。例子使用的数据源配置文件是 mysql-ds.xml，你可以在配套光盘中找到。部署数据源到 Jboss 前，请先把数据库驱动拷贝到[jboss 安装目录]/server/default/lib 目录（请用你的配置名替换 default），拷贝完成后需重启 Jboss。要恢复 OneToOne 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。

本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/OneToOneTest.jsp 访问客户端。

注意：在发布本例子 EJB 时，请御载前面含有 Person 类的例子（如: EntityBean.jar），否则会引起类型冲突。

3.8.5 单向一对一

对于前面的例子，我们可以把它转化成单向一对一关系。在 Person 类定义一个指向 IDCard 关系的成员属性。而在 IDCard 类不需要定义任何指向 Person 关系的成员属性。修改后的代码如下：

3.8.6 双向多对多映射

实现生活中，一个学生有多个老师，一个老师有多个学生，他们具有明显的多对多关系。多对多映射采取中间表连接的映射策略，建立的中间表将分别引入两边的主键作为外键。EJB3 对于中间表的元数据提供了可配置的方式，用户可以自定义中间表的表名、列名。下面以学生和老师为例介绍具有多对多关系的实体 Bean 开发。
需要映射的数据表：
student

字段名称	字段类型属性	描述
studentid (主键)	Int(11) not null	学生 ID
studentName	varchar(32) not null	学生姓名

teacher

字段名称	字段类型属性	描述
teacherid (主键)	Int(11) not null	教师 ID
teacherName	varchar(32) not null	教师姓名

中间表 teacher_student

字段名称	字段类型属性	描述
Student_ID	Int(11) not null	外键字段，指向 student 表的 studentid
Teacher_ID	Int(11) not null	外键字段，指向 teacher 表的 teacherid

在双向 ManyToMany 关联中，必须包含有一个关系维护端。我们可以在 Student 的@ManyToMany 注释中指定 mappedBy()属性，将其标识为关系被维护端，自然 Teacher 就成了关系维护端。
Student.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
```

```
@Entity
public class Student implements Serializable{
    private static final long serialVersionUID = 4283862967633995348L;
    private Integer studentid;
    private String studentName;
    private Set<Teacher> teachers = new HashSet<Teacher>();

    public Student() {}

    public Student(String studentName) {
        this.studentName = studentName;
    }

    @Id
    @GeneratedValue
    public Integer getStudentid() {
        return studentid;
    }
    public void setStudentid(Integer studentid) {
        this.studentid = studentid;
    }

    @Column(nullable=false, length=32)
    public String getStudentName() {
        return studentName;
    }

    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }

    @ManyToMany(mappedBy = "students")
    public Set<Teacher> getTeachers() {
        return teachers;
    }
    public void setTeachers(Set<Teacher> teachers) {
        this.teachers = teachers;
    }

    /**
     * 返回对象的散列代码值。该实现根据此对象
     * 中 studentid 字段计算散列代码值。
     * @return 此对象的散列代码值。
```

```
     */
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.studentid != null ? this.studentid.hashCode() :
super.hashCode());
        return hash;
    }

    /**
     * 确定其他对象是否等于此 Student。当且仅当
     * 参数不为 null 且该参数是具有与此对象相同 studentid 字段值的 Student 对象时，
     * 结果才为 <code>true</code>。
     * @param 对象，要比较的引用对象
     * 如果此对象与参数相同，则 @return <code>true</code>；
     * 否则为 <code>false</code>。
     */
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Student)) {
            return false;
        }
        Student other = (Student)object;
        if (this.studentid != other.studentid && (this.studentid == null
|| !this.studentid.equals(other.studentid))) return false;
        return true;
    }

    /**
     * 返回对象的字符串表示法。该实现根据 studentid 字段
     * 构造此表示法。
     * @return 对象的字符串表示法。
     */
    @Override
    public String toString() {
        return this.getClass().getName()+ "[studentid=" + studentid + "]";
    }
}
```

@ManyToMany 注释指定 Student 是多对多关系的一端，mappedBy 属性指定 Student 为双向关系的被维护端(inverse side)，另外也指出该关系映射信息是在被关联实体 Teacher 的成员属性 students 上定义的。下面是@ManyToMany 注释的定义：

```
public @interface ManyToMany
{
    Class targetEntity( ) default void.class;
```

```
    CascadeType[] cascade( ) default {};
    FetchType fetch( ) default LAZY;
    String mappedBy( ) default "";
}
```

@ManyToMany 注释的属性的含义与@OneToMany 注释的同名属性一一对应。不了解的同学可以查看前面章节内容。

Teacher.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;

@Entity
public class Teacher implements Serializable{
    private static final long serialVersionUID = 3248995998128746336L;
    private Integer teacherid;
    private String teacherName;
    private Set<Student> students = new HashSet<Student>();

    public Teacher() {}

    public Teacher(String teacherName) {
        this.teacherName = teacherName;
    }

    @Id
    @GeneratedValue
    public Integer getTeacherid() {
        return teacherid;
    }
    public void setTeacherid(Integer teacherid) {
        this.teacherid = teacherid;
    }

    @Column(nullable=false, length=32)
```

```
    public String getTeacherName() {
        return teacherName;
    }

    public void setTeacherName(String teacherName) {
        this.teacherName = teacherName;
    }

    @ManyToMany(cascade = {CascadeType.PERSIST,CascadeType.REFRESH}, fetch =
FetchType.LAZY)
    @JoinTable(name = "Teacher_Student",
        joinColumns = {@JoinColumn(name = "Teacher_ID")},
        inverseJoinColumns = {@JoinColumn(name = "Student_ID")})
    public Set<Student> getStudents() {
        return students;
    }
    public void setStudents(Set<Student> students) {
        this.students = students;
    }

    public void addStudent(Student student) {
        if (!this.students.contains(student)) {
            this.students.add(student);
        }
    }

     public void removeStudent(Student student) {
         if (this.students.contains(student)) {
          this.students.remove(student);
        }
    }

     /**
      * 返回对象的散列代码值。该实现根据此对象
      * 中 teacherid 字段计算散列代码值。
      * @return 此对象的散列代码值。
      */
     @Override
     public int hashCode() {
         int hash = 0;
         hash += (this.teacherid != null ? this.teacherid.hashCode() : 0);
         return hash;
     }
```

```
    /**
     * 确定其他对象是否等于此 Teacher。当且仅当
     * 参数不为 null 且该参数是具有与此对象相同 teacherid 字段值的 Teacher 对象时，
     * 结果才为 <code>true</code>。
     * @param 对象，要比较的引用对象
     * 如果此对象与参数相同，则 @return <code>true</code>；
     * 否则为 <code>false</code>。
     */
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Teacher)) {
            return false;
        }
        Teacher other = (Teacher)object;
        if (this.teacherid != other.teacherid && (this.teacherid == null
|| !this.teacherid.equals(other.teacherid))) return false;
        return true;
    }
    /**
     * 返回对象的字符串表示法。该实现根据 teacherid 字段
     * 构造此表示法。
     * @return 对象的字符串表示法。
     */
    @Override
    public String toString() {
        return this.getClass().getName()+ "[teacherid=" + teacherid + "]";
    }
}
```

@ManyToMany 注释指定 Teacher 是多对多关系的一端。

@JoinTable 注释定义中间表信息。在双向 ManyToMany 关系中，该注释需要定义在关系维护端，下面是该注释的定义：

```
public @interface JoinTable
{
    String name( ) default "";
    String catalog( ) default "";
    String schema( ) default "";
    JoinColumn[] joinColumns( ) default {};
    JoinColumn[] inverseJoinColumns( ) default {};
    UniqueConstraint[] uniqueConstraints( ) default {};
}
```

3.8.7 单向多对多

我们可以把前面的例子改成单向多对多关系，在 Teacher 中定义一个指向 Student 关系的成员属性，而在 Student 中不定义任何指向 Teacher 关系的成员属性。修改后的代码如下：

3.9 JPQL 查询

Java Persistence API 定义了一种查询语言(JPQL)，它具有与 SQL 相类似的语法，我们可以通过它查询/更新/删除实体。JPQL 是完全面向对象的，具备继承、多态和关联等特性。下面是一条比较简单的 JPQL 语句：
Select p from Person as p

3.9.1 命名参数查询

命名参数的格式为冒号加上参数名：“:+参数名”

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    ...
    private String NameQuery(){
        //获取指定personid的人员
        Query query = em.createQuery("select p from Person p where p.personid=:Id");
        query.setParameter("Id",new Integer(1));
        List<Person> result = (List<Person>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** NameQuery 结果打印
***************<BR>");
        for(Person person : result){
```

```
            out.append(person.getName()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.2 位置参数查询

位置参数的格式为问号加上位置编号：“?+位置编号”

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    ...
    private String PositionQuery(){
        //获取指定personid的人员
        Query query = em.createQuery("select p from Person p where p.personid=?1");
        query.setParameter(1,new Integer(1));
        List<Person> result = (List<Person>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** PositionQuery 结果打
印 ***************<BR>");
        for(Person person : result){
            out.append(person.getName()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.3 Date 参数

如果你需要将 java.util.Date 或 java.util.Calendar 作为参数传进一个参数查询，你需要使用对应的 setParameter() 方法，如下：

```
public interface Query {
   //命名参数查询时使用，参数类型为java.util.Date
   Query setParameter(String name, java.util.Date value, TemporalType
temporalType);
   //命名参数查询时使用，参数类型为java.util.Calendar
   Query setParameter(String name, Calendar value, TemporalType temporalType);

   //位置参数查询时使用，参数类型为java.util.Date
   Query setParameter(int position, Date value, TemporalType temporalType);
   //位置参数查询时使用，参数类型为java.util.Calendar
   Query setParameter(int position, Calendar value, TemporalType temporalType);
}
```

上面方法用到的 TemporalType 参数定义如下：

```
package javax.persistence;
public enum TemporalType {
    DATE, //java.sql.Date
    TIME, //java.sql.Time
    TIMESTAMP //java.sql.Timestamp
}
```

因为一个 Date 或 Calendar 对象能够描述一个真实的日期、时间或时间戳，所以我们需要告诉 Query 对象怎么使用这些参数，我们把 javax.persistence.TemporalType 作为参数传递进 setParameter 方法，告诉查询接口在转换 java.util.Date 或 java.util.Calendar 参数到本地 SQL 时使用什么数据库类型。

3.9.4 一个 JPQL 查询例子

为方便大家学习 JPQL 语句，本书为大家搭建了一个测试环境，本例子的实体 Bean 有 Person、Order、 OrderItem。它们之间的关系是：一个 Person 有多个 Order，一个 Order 有多个 OrderItem。
下面是各实体 Bean 的代码：
Person.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.OrderBy;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Person  implements Serializable{
    private static final long serialVersionUID = -7090476760197665776L;
    private Integer personid;
    private String name;
    private boolean sex;
    private Short age;
    private Date birthday;
```

```
    private Set<Order> orders = new HashSet<Order>();

    public Person(){}
    public Person(String name, boolean sex, Short age, Date birthday) {
        this.name = name;
        this.sex = sex;
        this.age = age;
        this.birthday = birthday;
    }

    @Id
    @GeneratedValue
    public Integer getPersonid() {
        return personid;
    }
    public void setPersonid(Integer personid) {
        this.personid = personid;
    }

    @Column(name = "PersonName",nullable=false,length=32)
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @Column(nullable=false)
    public boolean getSex() {
        return sex;
    }
    public void setSex(boolean sex) {
        this.sex = sex;
    }

    @Column(nullable=false)
    public Short getAge() {
        return age;
    }
    public void setAge(Short age) {
        this.age = age;
    }

    @Temporal(value=TemporalType.DATE)
```

```
    public Date getBirthday() {
        return birthday;
    }
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    @OneToMany(mappedBy="ower",cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @OrderBy(value = "orderid ASC")
    public Set<Order> getOrders() {
        return orders;
    }
    public void setOrders(Set<Order> orders) {
        this.orders = orders;
    }

    /**
     * 返回对象的散列代码值。该实现根据此对象
     * 中 personid 字段计算散列代码值。
     * @return 此对象的散列代码值。
     */
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.personid != null ? this.personid.hashCode() :
super.hashCode());
        return hash;
    }

    /**
     * 确定其他对象是否等于此 Person。当且仅当
     * 参数不为 null 且该参数是具有与此对象相同 personid 字段值的 Person 对象时，
     * 结果才为 <code>true</code>。
     * @param 对象，要比较的引用对象
     * 如果此对象与参数相同，则 @return <code>true</code>；
     * 否则为 <code>false</code>。
     */
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Person)) {
            return false;
        }
        Person other = (Person)object;
        if (this.personid != other.personid && (this.personid == null
```

```
|| !this.personid.equals(other.personid))) return false;
        return true;
    }

    /**
     * 返回对象的字符串表示法。该实现根据 personid 字段
     * 构造此表示法。
     * @return 对象的字符串表示法。
     */
    @Override
    public String toString() {
        return this.getClass().getName()+ "[personid=" + personid+ "]";
    }
}
```

Order.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Date;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.OrderBy;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "Orders")
public class Order implements Serializable {
    private static final long serialVersionUID = 686643084215220968L;
    private Integer orderid;
    private Float amount;
    private Person ower;
    private Set<OrderItem> orderItems = new HashSet<OrderItem>();
    private Date createdate;
```

```
    public Order(){}
    public Order(Float amount, Person ower, Date createdate) {
        this.amount = amount;
        this.ower = ower;
        this.createdate = createdate;
    }

    @Id
    @GeneratedValue
    public Integer getOrderid() {
        return orderid;
    }
    public void setOrderid(Integer orderid) {
        this.orderid = orderid;
    }

    public Float getAmount() {
        return amount;
    }
    public void setAmount(Float amount) {
        this.amount = amount;
    }

    @ManyToOne(cascade=CascadeType.ALL,optional=false)
    @JoinColumn(name = "person_id")
    public Person getOwer() {
        return ower;
    }
    public void setOwer(Person ower) {
        this.ower = ower;
    }

    @OneToMany(mappedBy="order",cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @OrderBy(value = "id ASC")
    public Set<OrderItem> getOrderItems() {
        return orderItems;
    }
    public void setOrderItems(Set<OrderItem> orderItems) {
        this.orderItems = orderItems;
    }

    public void addOrderItem(OrderItem orderitem) {
        if (!this.orderItems.contains(orderitem)) {
            this.orderItems.add(orderitem);
```

```
            orderitem.setOrder(this);
        }
     }

    public void removeOrderItem(OrderItem orderitem) {
        if (this.orderItems.contains(orderitem)) {
            orderitem.setOrder(null);
            this.orderItems.remove(orderitem);
        }
    }

    @Temporal(value=TemporalType.TIMESTAMP)
    public Date getCreatedate() {
        return createdate;
    }
    public void setCreatedate(Date createdate) {
        this.createdate = createdate;
    }
    /**
     * 返回对象的散列代码值。该实现根据此对象
     * 中 orderid 字段计算散列代码值。
     * @return 此对象的散列代码值。
     */
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.orderid != null ? this.orderid.hashCode() : super.hashCode());
        return hash;
    }

    /**
     * 确定其他对象是否等于此 Order。当且仅当
     * 参数不为 null 且该参数是具有与此对象相同 orderid 字段值的 Order 对象时，
     * 结果才为 <code>true</code>。
     * @param 对象，要比较的引用对象
     * 如果此对象与参数相同，则 @return <code>true</code>；
     * 否则为 <code>false</code>。
     */
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Order)) {
            return false;
        }
        Order other = (Order)object;
```

```
        if (this.orderid != other.orderid && (this.orderid == null
|| !this.orderid.equals(other.orderid))) return false;
        return true;
    }

    /**
     * 返回对象的字符串表示法。该实现根据 orderid 字段
     * 构造此表示法。
     * @return 对象的字符串表示法。
     */
    @Override
    public String toString() {
        return this.getClass().getName()+ "[orderid=" + orderid + "]";
    }
}
```

OrderItem.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity
public class OrderItem implements Serializable {
    private static final long serialVersionUID = -1166337687856636179L;
    private Integer id;
    private String productname;
    private Float price;
    private Order order;

    public OrderItem() {}

    public OrderItem(String productname, Float price) {
        this.productname = productname;
        this.price = price;
    }

    @Id
    @GeneratedValue
    public Integer getId() {
```

```
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }

    public String getProductname() {
        return productname;
    }
    public void setProductname(String productname) {
        this.productname = productname;
    }

    public Float getPrice() {
        return price;
    }
    public void setPrice(Float price) {
        this.price = price;
    }

    @ManyToOne(cascade=CascadeType.REFRESH,optional=false)
    @JoinColumn(name = "order_id")
    public Order getOrder() {
        return order;
    }
    public void setOrder(Order order) {
        this.order = order;
    }

    /**
     * 返回对象的散列代码值。该实现根据此对象
     * 中 id 字段计算散列代码值。
     * @return 此对象的散列代码值。
     */
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.id != null ? this.id.hashCode() : super.hashCode());
        return hash;
    }

    /**
     * 确定其他对象是否等于此 OrderItem。当且仅当
     * 参数不为 null 且该参数是具有与此对象相同 id 字段值的 OrderItem 对象时，
```

```
     * 结果才为 <code>true</code>。
     * @param 对象，要比较的引用对象
     * 如果此对象与参数相同，则 @return <code>true</code>；
     * 否则为 <code>false</code>。
     */
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof OrderItem)) {
            return false;
        }
        OrderItem other = (OrderItem)object;
        if (this.id != other.id && (this.id == null || !this.id.equals(other.id)))
return false;
        return true;
    }

    /**
     * 返回对象的字符串表示法。该实现根据 id 字段
     * 构造此表示法。
     * @return 对象的字符串表示法。
     */
    @Override
    public String toString() {
        return this.getClass().getName()+ "[id=" + id + "]";
    }
}
```

下面是本例的 Session Bean

Session Bean 的接口

```
package com.foshanshop.ejb3;
public interface QueryDAO {
    public String ExecuteQuery(int index);
    public void initdate();
}
```

Session Bean 的程序片断

```
package com.foshanshop.ejb3.impl;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import javax.ejb.Remote;
import javax.ejb.Stateless;
```

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import com.foshanshop.ejb3.QueryDAO;
import com.foshanshop.ejb3.bean.Order;
import com.foshanshop.ejb3.bean.OrderItem;
import com.foshanshop.ejb3.bean.Person;
import com.foshanshop.ejb3.bean.SimplePerson;

@Stateless
@Remote (QueryDAO.class)
@SuppressWarnings("unchecked")
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;

    public void initdate() {
        try {
            Query query = em.createQuery("select count(p) from Person p");
            Object result = query.getSingleResult();
            if (result == null || Integer.parseInt(result.toString()) == 0) {
                // 没有数据时，插入几条数据用作测试
                // ===================================
                SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
                Person person = new Person("liujun", true, new Short("26"),
                        formatter.parse("1980-9-30"));
                Set<Order> orders = new HashSet<Order>();

                Order order1 = new Order(new Float("105.5"), person, new Date());
                order1.addOrderItem(new OrderItem("U盘", new Float("105.5")));

                Order order2 = new Order(new Float("780"), person, new Date());
                order2.addOrderItem(new OrderItem("MP4", new Float("778")));
                order2.addOrderItem(new OrderItem("矿泉水", new Float("2")));
                orders.add(order1);
                orders.add(order2);
                person.setOrders(orders);

                Person person1 = new Person("yunxiaoyi", false,
                        new Short("23"), formatter.parse("1983-10-20"));
                orders = new HashSet<Order>();
                order1 = new Order(new Float("360"), person1, new Date());
                order1.addOrderItem(new OrderItem("香水", new Float("360")));

                order2 = new Order(new Float("1806"), person1, new Date());
```

```
            order2.addOrderItem(new OrderItem("照相机", new Float("1800")));
            order2.addOrderItem(new OrderItem("5号电池", new Float("6")));
            orders.add(order1);
            orders.add(order2);
            person1.setOrders(orders);

            // ====================================
            Person person2 = new Person("zhangming", false,
                   new Short("21"), formatter.parse("1985-11-25"));
            orders = new HashSet<Order>();

            order1 = new Order(new Float("620"), person2, new Date());
            order1.addOrderItem(new OrderItem("棉被", new Float("620")));

            order2 = new Order(new Float("3"), person2, new Date());
            order2.addOrderItem(new OrderItem("可乐", new Float("3")));
            orders.add(order1);
            orders.add(order2);
            person2.setOrders(orders);

            em.persist(person2);
            em.persist(person1);
            em.persist(person);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public String ExecuteQuery(int index) {
    String result = "";
    switch(index){
        case 1:
            result = this.NameQuery();
            break;
        case 2:
            result = this.PositionQuery();
            break;
        case 3:
            result = this.QueryOrderBy();
            break;
        case 4:
            result = this.QueryPartAttribute();
            break;
```

```
            case 5:
                result = this.QueryConstructor();
                break;
            case 6:
                result = this.QueryAggregation();
                break;
            case 7:
                result = this.QueryGroupBy();
                break;
            case 8:
                result = this.QueryGroupByHaving();
                break;
            case 9:
                result = this.QueryLeftJoin();
                break;
            case 10:
                result = this.QueryInnerJoin();
                break;
            case 11:
                result = this.QueryInnerJoinLazyLoad();
                break;
            case 12:
                result = this.QueryJoinFetch();
                break;
            case 13:
                result = this.QueryEntityParameter();
                break;
            case 14:
                result = this.QueryBatchUpdate();
                break;
            case 15:
                result = this.QueryBatchRemove();
                break;
            case 16:
                result = this.QueryNOTOperate();
                break;
            case 17:
                result = this.QueryBETWEENOperate();
                break;
            case 18:
                result = this.QueryINOperate();
                break;
            case 19:
                result = this.QueryLIKEOperate();
```

```
                break;
            case 20:
                result = this.QueryISNULLOperate();
                break;
            case 21:
                result = this.QueryISEMPTYOperate();
                break;
            case 22:
                result = this.QueryEXISTSOperate();
                break;
            case 23:
                result = this.QueryStringOperate();
                break;
            case 24:
                result = this.QueryMathLOperate();
                break;
            case 25:
                result = this.QuerySubQueryOperate();
                break;
            case 26:
                result = this.QueryNoneReturnValueStoreProcedure();
                break;
            case 27:
                result = this.QuerySingleObjectStoreProcedure();
                break;
            case 28:
                result = this.QueryStoreProcedure();
                break;
            case 29:
                result = this.QueryPartColumnStoreProcedure();
                break;
            case 30:
                result = this.QueryAllAnySomeOperate();
                break;
            case 31:
                result =  QueryMemberOf();
                break;
        }
        return result;
    }
    …这里省略后面的代码
}
```

下面是本例的 JSP 客户端代码：QueryTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.QueryDAO,
                javax.naming.*,
                java.util.*"%>
<%
    try {
        String index = request.getParameter("index");
        if (index!=null && !"".equals(index.trim())){
            InitialContext ctx = new InitialContext();
            QueryDAO querydao = (QueryDAO) ctx.lookup("QueryDAOBean/remote");
            querydao.initdate();
            out.println(querydao.ExecuteQuery(Integer.parseInt(index)));
        }
    } catch (Exception e) {
        out.println(e.getMessage());
    }
%>
```

本例子的源代码在配套光盘的 Query 文件夹。例子使用的数据源配置文件是 mysql-ds.xml，你可以在配套光盘中找到。部署数据源到 Jboss 前，请先把数据库驱动拷贝到[jboss 安装目录]/server/default/lib 目录（请用你的配置名替换 default），拷贝完成后需重启 Jboss。要恢复 Query 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。

本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/QueryTest.jsp?index=1(参数值可以是 1 到 31)访问客户端。

注意：在发布本例子 EJB 时，请御载前面含有 Person，Order，OrderItem 类的例子 (如: EntityBean.jar，OneToMany.jar，OneToOne.jar)，否则会引起类型冲突。

3.9.5 命名查询

你可以在实体 bean 上定义一个或多个查询语句，这样可以减少每次因书写错误而引起的 BUG。实际应用中，一般把经常使用的查询语句定义成命名查询，如下：

```
@NamedQuery(name="getPerson", query= "select o from Person o WHERE o.personid=?1")
@Entity
public class Person  implements Serializable{
```

上面@NamedQuery.name()属性定义查询的名称，@NamedQuery.query()属性定义查询语句。如果你需要定义多个命名查询，可以使用@javax.persistence.NamedQueries 注释。在该注释里面，你可以放置多个@NamedQuery 注释，如：

```
@NamedQueries({
    @NamedQuery(name="getPerson", query= "select o from Person o WHERE o.personid=?1"),
    @NamedQuery(name="getPersonList", query= "select o from Person o WHERE o.age>?1")
})
```

```
@Entity
public class Person  implements Serializable{
```

当命名查询定义好了之后，我们可以通过其名称执行查询，代码如下：

```
Query query = em.createNamedQuery("getPerson");
query.setParameter(1, 1);
```

3.9.6 排序(order by)

下面是一个排序例子，"ASC"和"DESC"分别为升序和降序，如果不显式指定，JPQL 默认使用 ASC 升序。（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryOrderBy(){
        //先按年龄降序排序，然后按出生日期升序排序
        Query query = em.createQuery("select p from Person p order by p.age desc,
p.birthday asc");
        List<Person> result = (List<Person>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryOrderBy 结果打印
***************<BR>");
        for(Person person : result){
            out.append(person.getName()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.7 查询部分属性

在前面的例子中，都是针对实体的查询，返回的结果也是实体类型。JPQL 允许查询返回我们需要的成员属性，而不是实体对象。在一些实体成员属性比较多的情况，这样的查询可以提高性能。（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryPartAttribute(){
        //直接查询我们感兴趣的属性(列)
        Query query = em.createQuery("select p.personid, p.name from Person p order
by p.personid desc ");
        //集合中的元素不再是Person,而是一个Object[]对象数组
```

```
        List<Object[]> result = (List<Object[]>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryPartAttribute 结果打印 ***************<BR>");
        for(Object[] row : result){//取每一行
            //数组中的第一个值是personid
            int personid = Integer.parseInt(row[0].toString());
            String PersonName = row[1].toString();
            out.append("personid="+ personid+ "; PersonName="+PersonName+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.8 查询中使用构造器(Constructor)

JPQL 支持将查询的结果直接作为一个 java 类的构造器参数，并产生类对象作为结果返回。(例子的源代码在 Query 文件夹)

SimplePerson.java

```
package com.foshanshop.ejb3.bean;
public class SimplePerson {
    private String name;
    private boolean sex;

    public SimplePerson() {
    }
    public SimplePerson(String name, boolean sex) {
        this.name = name;
        this.sex = sex;
    }
    public String getDescription() {
        return sex ? name + "是男孩" : name + "是女孩";
    }
}
```

下面将查询的属性结果直接作为 SimplePerson 的构造器参数。

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    ...
    private String QueryConstructor(){
        //我们把需要的两个属性作为SimplePerson的构造器参数，并使用new函数。
        Query query = em.createQuery("select new com.foshanshop.ejb3.bean.SimplePerson(p.name,p.sex) from Person p order by p.personid desc");
```

```
        //集合中的元素是SimplePerson对象
        List<SimplePerson> result = (List<SimplePerson>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryConstructor 结果
打印 ***************<BR>");
        for(SimplePerson simpleperson : result){
            out.append("人员介绍："+ simpleperson.getDescription()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.9 聚合查询(Aggregation)

像大部分的 SQL 一样，JPQL 也支持查询中的聚合函数。目前 JPQL 支持的聚合函数包括：

1. AVG() 求平均数，返回值类型为 Double
2. SUM() 求和，返回值类型为被求值的成员属性所对应的类型。
3. COUNT() 统计，返回类型为 Long，注意 count(*)语法并不属于 JPA 规范，它在 hibernate 中可用，但在 toplink 其它产品中并不可用。
4. MAX() 求最大值，返回值类型为被求值的成员属性所对应的类型。可用于基本数据类型，字符串及可序列化对象
5. MIN() 求最小值，返回值类型为被求值的成员属性所对应的类型。可用于基本数据类型，字符串及可序列化对象

聚合函数在统计时，会忽略掉带 null 值的记录。如果查询的集合为空（即没有记录），count()在处理时会返回 0，而 AVG()、SUM()、MAX()、MIN()返回 null 值。

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryAggregation(){
        //获取最大年龄
        Query query = em.createQuery("select max(p.age) from Person p");
        Object result = query.getSingleResult();
        String maxAge = result.toString();
        //获取平均年龄
        query = em.createQuery("select avg(p.age) from Person p");
        result = query.getSingleResult();
        String avgAge = result.toString();
        //获取最小年龄
        query = em.createQuery("select min(p.age) from Person p");
        result = query.getSingleResult();
        String minAge = result.toString();
        //获取总人数
        query = em.createQuery("select count(p) from Person p");
        result = query.getSingleResult();
```

```
        String countperson = result.toString();
        //获取年龄总和
        query = em.createQuery("select sum(p.age) from Person p");
        result = query.getSingleResult();
        String sumage = result.toString();
        StringBuffer out = new StringBuffer("*************** QueryConstructor 结果
打印 ****************<BR>");
        out.append("最大年龄: "+ maxAge+ "<BR>");
        out.append("平均年龄: "+ avgAge+ "<BR>");
        out.append("最小年龄: "+ minAge+ "<BR>");
        out.append("总人数: "+ countperson+ "<BR>");
        out.append("年龄总和: "+ sumage+ "<BR>");
        return out.toString();
    }
}
```

和 SQL 一样，如果聚合函数不是 select...from 的唯一一个返回列，需要使用"GROUP BY"语句。"GROUP BY"应该包含 select 语句中除了聚合函数外的所有属性。（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    ...
    private String QueryGroupBy(){
        //返回男女生各自的总人数
        Query query = em.createQuery("select p.sex, count(p) from Person p group by
p.sex");
        //集合中的元素不再是Person,而是一个Object[]对象数组
        List<Object[]> result = (List<Object[]>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryGroupBy 结果打印
****************<BR>");
        for(Object[] row : result){//取每一行
            //数组中的第一个值是sex
            boolean sex = Boolean.parseBoolean(row[0].toString());
            //数组中的第二个值是聚合函数COUNT返回值
            String sextotal = row[1].toString();
            out.append((sex ? "男生":"女生")+ "总共有"+ sextotal+ "人<BR>");
        }
        return out.toString();
    }
}
```

如果还需要加上查询条件，需要使用"HAVING"条件语句而不是"WHERE"语句。（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    ...
```

```
    private String QueryGroupByHaving(){
        //返回人数超过1人的性别
        Query query = em.createQuery("select p.sex, count(p) from Person p group by
p.sex having count(p)>?1");
        //设置查询中的参数
        query.setParameter(1, new Long(1));
        //集合中的元素不再是Person,而是一个Object[]对象数组
        List<Object[]> result = (List<Object[]>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryGroupByHaving 结
果打印 ***************<BR>");
        for(Object[] row : result){//取每一行
            //数组中的第一个值是sex
            boolean sex = Boolean.parseBoolean(row[0].toString());
            //数组中的第二个值是聚合函数COUNT返回值
            String sextotal = row[1].toString();
            out.append((sex ? "男生":"女生")+ "总共有"+ sextotal+ "人<BR>");
        }
        return out.toString();
    }
}
```

3.9.10 关联(join)

在 JPQL 中，仍然支持和 SQL 中类似的关联语法：
left out join/left join
inner join
left join/inner join fetch

left out join/left join 等，都是允许右边表达式的实体为空。（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryLeftJoin(){
        //获取26岁人的订单,不管Order中是否有OrderItem
        Query query = em.createQuery("select o from Order o left join o.orderItems
where o.ower.age=26 order by o.orderid");
        List<Order> result = (List<Order>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryLeftJoin 结果打
印 ***************<BR>");
        Integer orderid = null;
        for(Order order : result){
            if (orderid==null || !orderid.equals(order.getOrderid())){
                orderid = order.getOrderid();
```

```
                out.append("订单号：" + orderid + "<BR>");
            }
        }
        return out.toString();
    }
}
```

inner join 要求右边表达式的实体必须存在。

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryInnerJoin(){
        //获取26岁人的订单,Order中必须要有OrderItem
        Query query = em.createQuery("select o from Order o inner join o.orderItems
where o.ower.age=26 order by o.orderid");
        List<Order> result = (List<Order>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryInnerJoin 结果打
印 ***************<BR>");
        Integer orderid = null;
        for(Order order : result){
            if (orderid==null || !orderid.equals(order.getOrderid())){
                orderid = order.getOrderid();
                out.append("订单号：" + orderid + "<BR>");
            }
        }
        return out.toString();
    }
}
```

left/left out/inner join fetch 提供了一种灵活的查询加载方式来提高查询的性能。在默认的查询中，实体的延迟属性不会被加载。如下：

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryInnerJoinLazyLoad(){
        Query query = em.createQuery("select o from Order o inner join o.orderItems
where o.ower.age=26 order by o.orderid");
        List<Order> result = (List<Order>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryInnerJoinLazyLoad
结果打印 ***************<BR>");
        if( result.size()>0){
            //这时获得的Order实体中orderItems为空
            Order order = result.get(0);
            //当应用需要时，EJB3 Runtime才会执行一条SQL语句来加载属于当前Order的OrderItems
```

```
            for(OrderItem orderItem : order.getOrderItems()){
                out.append("订购产品名："+ orderItem.getProductname()+ "<BR>");
            }
        }
        return out.toString();
    }
}
```

在第一次执行查询时，上面 orderItems 延迟属性不会被初始化，只有在执行 Set<OrderItem> list = order.getOrderItems();时才会执行一条 SQL 来加载 orderItems。这样的查询在性能上有不足的地方。为了查询 N 个 Order，我们需要一条 SQL 获取所有 Order 的基本属性， 然后再需要 N 条语句加载延迟属性。为了避免 1+N 的性能问题，我们可以利用 join fetch，使其预先加载好关联的 orderItems 集合。如下：

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryJoinFetch(){
        //获取26岁人的订单,Order中必须存在OrderItem
        Query query = em.createQuery("select o from Order o inner join fetch
o.orderItems where o.ower.age=26 order by o.orderid");
        List<Order> result = (List<Order>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryJoinFetch 结果打
印 ***************<BR>");
        Integer orderid = null;
        for(Order order : result){
            if (orderid==null || !orderid.equals(order.getOrderid())){
                orderid = order.getOrderid();
                out.append("订单号："+ orderid+ "<BR>");
            }
        }
        return out.toString();
    }
}
```

上面由于使用了 fetch，这个查询只会产生一条 SQL 语句，比原来需要 1+N 条 SQL 语句在性能上有了极大的提升。

3.9.11 排除相同的记录 DISTINCT

使用关联查询，我们很经常得到重复的对象，如下面语句：
select o from Order o inner join fetch o.orderItems order by o.orderid
如果一个 Order 有多个 orderItem，返回的结果就会有多个相同的 Order，这时我们需要使用 DISTINCT 关键字来排除相同的对象。

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    ......
    public List<Order> getAllOrder() {
```

```
        Query query = em.createQuery("select DISTINCT o from Order o inner join fetch
o.orderItems order by o.orderid");
        return (List<Order>)query.getResultList();
    }
}
```

DISTINCT 操作符还可以与任何聚合函数结合使用，它会首先去掉重复值，然后再统计。

3.9.12 比较 Entity

在使用参数查询时，参数类型除了 String、原始数据类型(int, double 等)和它们的对象类型(Integer, Double 等)，也可以是实体对象。（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryEntityParameter(){
        //查询某人的所有订单
        Query query = em.createQuery("select o from Order o where o.ower =?1 order
by o.orderid");
        Person person = new Person();
        person.setPersonid(new Integer(1));
        //设置查询中的参数
        query.setParameter(1,person);
        List<Order> result = (List<Order>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryEntityParameter
结果打印 ***************<BR>");
        for(Order order : result){
        out.append("订单号："+ order.getOrderid()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.13 批量更新(Batch Update)

JPQL 支持批量更新. (例子的源代码在 Query 文件夹)

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
…
    private String QueryBatchUpdate(){
        //把所有订单的金额加10
        Query query = em.createQuery("update Order as o set o.amount=o.amount+10");
        //update的记录数
```

```
        int result = query.executeUpdate();
        StringBuffer out = new StringBuffer("*************** QueryBatchUpdate 结果
打印 ***************<BR>");
        out.append("更新操作影响的记录数："+ result+ "条<BR>");
        return out.toString();
    }
}
```

3.9.14 批量删除(Batch Remove)

JPQL 支持批量删除。(例子的源代码在 Query 文件夹)

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
…
    private String QueryBatchRemove(){
        //把金额小于100的订单删除,先删除订单子项,再删除订单
        Query query = em.createQuery("delete from OrderItem item where item.order
in(select o from Order as o where o.amount<100)");
        query.executeUpdate();
        query = em.createQuery("delete from Order as o where o.amount<100");
        int result = query.executeUpdate();//delete的记录数
        StringBuffer out = new StringBuffer("*************** QueryBatchRemove 结果
打印 ***************<BR>");
        out.append("删除操作影响的记录数："+ result+ "条<BR>");
        return out.toString();
    }
}
```

3.9.15 逻辑非运算符 NOT

逻辑非运算符。（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
…
    private String QueryNOTOperate(){
        //查询除了指定人之外的所有订单
        Query query = em.createQuery("select o from Order o where not(o.ower =?1)
order by o.orderid");
        Person person = new Person();
        person.setPersonid(new Integer(2));
        //设置查询中的参数
        query.setParameter(1,person);
```

```
        List<Order> result = (List<Order>) query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryNOTOperate 结果
打印 ***************<BR>");
        for(Order order : result){
          out.append("订单号："+ order.getOrderid()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.16 使用操作符 BETWEEN

定位某个范围之间的操作符，（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
…
    private String QueryBETWEENOperate(){
       StringBuffer out = new StringBuffer("*************** QueryBETWEENOperate 结
果打印 ***************<BR>");
        //查询金额在300到1000之间的订单
        Query query = em.createQuery("select o from Order as o where o.amount between
300 and 1000");
        List<Order> result = (List<Order>) query.getResultList();
        for(Order order : result){
            out.append("订单号："+ order.getOrderid()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.17 使用操作符 IN

在给定值列表中查找，（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryINOperate(){
       StringBuffer out = new StringBuffer("*************** QueryINOperate 结果打
印 ***************<BR>");
        //查找年龄为26,21的Person
        Query query = em.createQuery("select p from Person as p where p.age
in(26,21)");
```

```
        List<Person> result = (List<Person>) query.getResultList();
        for(Person person : result){
            out.append(person.getName()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.18 使用操作符 LIKE

模糊查找，（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryLIKEOperate(){
       StringBuffer out = new StringBuffer("*************** QueryLIKEOperate 结果
打印 ***************<BR>");
        out.append("---------- 查找以字符串\"li\"开头的Person ----------<BR>");
        Query query = em.createQuery("select p from Person as p where p.name like
'li%'");
        List<Person> result = (List<Person>) query.getResultList();
        for(Person person : result){
            out.append(person.getName()+ "<BR>");
        }
        out.append("---------- 查询所有name不以字符串\"ming\"结尾的Person
----------<BR>");
        query = em.createQuery("select p from Person as p where p.name not like
'%ming'");
        result = (List<Person>) query.getResultList();
        for(Person person : result){
            out.append(person.getName()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.19 使用操作符 IS NULL

判断成员属性是否为 null，它与 IS EMPTY 是不同的，IS EMPTY 用于判断集合是否为空，也就是判断集合有没有元素。（例子的源代码在 Query 文件夹）

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
```

```
    …
    private String QueryISNULLOperate(){
    StringBuffer out = new StringBuffer("*************** QueryISNULLOperate 结果打印 ***************<BR>");
    out.append("--------------- 查询含有购买者的所有Order -------------<BR>");
       Query query = em.createQuery("select o from Order as o where o.ower is not null order by o.orderid");
       List<Order> result = (List<Order>) query.getResultList();
       for(Order order : result){
          out.append("订单号："+ order.getOrderid()+ "<BR>");
       }
       out.append("--------------- 查询没有购买者的所有Order -------------<BR>");
       query = em.createQuery("select o from Order as o where o.ower is null order by o.orderid");
       result = (List<Order>) query.getResultList();
       for(Order order : result){
          out.append("订单号："+ order.getOrderid()+ "<BR>");
       }
       return out.toString();
    }
}
```

3.9.20 使用操作符 IS EMPTY

IS EMPTY 是针对集合属性的操作符，用于判断集合是否为空，也就是判断集合有没有元素。可以和 NOT 一起使用。注：低版权的 Mysql 不支持 IS EMPTY

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryISEMPTYOperate(){
       StringBuffer out = new StringBuffer("*************** QueryISEMPTYOperate 结果打印 ***************<BR>");
      out.append("--------------- 查询含有订单项的所有Order -------------<BR>");
       Query query = em.createQuery("select o from Order as o where o.orderItems is not empty order by o.orderid");
       List<Order> result = (List<Order>) query.getResultList();
       for(Order order : result){
          out.append("订单号："+ order.getOrderid()+ "<BR>");
       }
       out.append("--------------- 查询没有订单项的所有Order -------------<BR>");
       query = em.createQuery("select o from Order as o where o.orderItems is empty order by o.orderid");
       result = (List<Order>) query.getResultList();
```

```
        for(Order order : result){
            out.append("订单号："+ order.getOrderid()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.21 字符串函数

JPQL 定义了大量内置函数，这些函数的使用方法和 SQL 中相应的函数方法类似。字符串函数包括：

1. TRIM([[LEADING | TRAILING | BOTH] [trim_char] FROM] String) 去除指定字符，如果不指定字符，默认去除空格。LEADING 指定只去除前置的字符，TRAILING 指定只去除后置的字符，BOTH 指定前后都去除。使用例子：从 ABC 中去除后面的 C ，trim(TRAILING 'C' FROM 'ABC')。

3.9.22 日期和时间函数

CURRENT_DATE 当前日期
CURRENT_TIME 当前时间
CURRENT_TIMESTAMP 当前的日期时间
例：select p from Person p where p.birthday< CURRENT_DATE

3.9.23 数学函数

JPQL 中定义的计算函数包括：
ABS(number) 返回 number 的绝对值
SQRT(double) 返回 double 的平方根
MOD(int, int) 求余数，例 MOD(9,5)=4
SIZE() 取集合的数量

3.9.24 Member of

Member of 操作符用于判断实体是否是集合中的一员。如下面的例子，查询含有某个订单项的订单：

3.9.25 子查询

子查询可以用于 WHERE 和 HAVING 条件语句中。注：低版本的 Mysql 不支持子查询

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QuerySubQueryOperate(){
        //查询年龄为26岁的购买者的所有Order
        Query query = em.createQuery("select o from Order as o where o.ower in(select
p from Person as p where p.age =26) order by o.orderid");
        List<Order> result = (List<Order>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QuerySubQueryOperate
结果打印 ***************<BR>");
        out.append("--------------- 查询年龄为26岁的购买者的所有Order
-------------<BR>");
        for(Order order : result){
            out.append("订单号："+ order.getOrderid()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.26 EXISTS

EXISTS 需要和子查询配合使用，它用来判断子查询是否存在记录。注：低版权的 Mysql 不支持 EXISTS

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryEXISTSOperate(){
        StringBuffer out = new StringBuffer("*************** QueryEXISTSOperate 结
果打印 ***************<BR>");
        out.append("--------------- 如果存在订单号1，就获取所有OrderItem
```

```
-------------<BR>");
        Query query = em.createQuery("select oi from OrderItem as oi where exists
(select o from Order o where o.orderid=1)");
        List<OrderItem> result = (List<OrderItem>) query.getResultList();
        for(OrderItem item : result){
            out.append("所有订购的产品名："+ item.getProductname()+ "<BR>");
        }
        out.append("--------------- 如果不存在订单号10，就获取id为1的OrderItem
-------------<BR>");
        query = em.createQuery("select oi from OrderItem as oi where oi.id=1 and not
exists (select o from Order o where o.orderid=10)");
        OrderItem item = (OrderItem) query.getSingleResult();
        if(item!=null){
            out.append("订单项ID为1的订购产品名："+ item.getProductname()+ "<BR>");
        }
        return out.toString();
    }
}
```

3.9.27 All,ANY,SOME

当子查询返回多条记录时，你可以使用表达式 ALL、ANY 和 SOME 对结果做进一步限定。

这部分内容在《EJB3.0入门经典》中

3.9.28 结果集分页

有些时候执行一个查询会返回成千上万条记录，事实上我们只需要显示其中一部分数据。这时我们需要对结果集进行分页，Query API 有两个接口方法可以解决这个问题：setMaxResults( ) 和 setFirstResult( )
setMaxResults 方法设置获取多少条记录
setFirstResult 方法设置从结果集中的那个索引开始获取（索引从 0 开始）
分页代码片断：

```
public List<Person> getPersonList(int max, int whichpage) {
    int index = (whichpage-1) * max;
    Query query = em.createQuery("select p from Person p order by p.personid asc");
    List<Person> result = (List<Person>) query.setMaxResults(max)
                .setFirstResult(index).getResultList();
    return result;
```

```
}
```

JSP 客户端调用代码片断:

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.PersonDAO, com.foshanshop.ejb3.bean.Person,
                javax.naming.*, java.util.List"%>
<%
    try {
        InitialContext ctx = new InitialContext();
        PersonDAO persondao = (PersonDAO) ctx.lookup("PersonDAOBean/remote");
        out.println("<br>============  分页显示,每页记录数为2 =========<BR>");
        String index = request.getParameter("index");
        if (index==null || "".equals(index.trim())) index = "1";

        int max = 2; //每页记录数为2
        int whichpage = Integer.parseInt(index); //第几页
        List<Person> persons = persondao.getPersonList(max, whichpage);
        for(Person p : persons){
            out.println("人员编号:"+ p.getPersonid() + " 姓名: "+ p.getName() +
"<Br>");
        }
    } catch (Exception e) {
        out.println(e.getMessage());
    }
%>
```

上面代码设置每页记录数为 2，如果数据库中存在 7 条记录，第一页显示的记录索引应为（0，1），第二页为（2，3），第三页为(3，4)...等。

在分页中，为了显示数据有序，建议在查询中进行排序。

3.10 调用存储过程

要调用存储过程，我们可以通过 EntityManage.createNativeQuery()方法执行 SQL 语句(注意：这里说的是 SQL 语句，不是 JPQL)，调用存储过程的 SQL 格式如下：

{call 存储过程名称(参数 1, 参数 2, …)}

在 EJB3 中你可以调用的存储过程有两种

1．无返回值的存储过程。

2．返回值为 ResultSet（以 select 形式返回的值）的存储过程，EJB3 不能调用以 OUT 参数返回值的存储过程。

下面我们看看几种具有代表性的存储过程调用方法.

3.10.1 调用无返回值的存储过程

我们首先创建一个名为 AddPerson 的存储过程，它的 DDL 如下（注：本例使用的是 MySql 数据库）：

```
CREATE PROCEDURE `AddPerson`()
    NOT DETERMINISTIC
     SQL SECURITY DEFINER
    COMMENT ''
BEGIN
      INSERT into person(`PersonName`,`sex`,`age`) values('存储过程',1,25);
END;
```

下面的代码片断显示了无返回值存储过程的调用方法：

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryNoneReturnValueStoreProcedure(){
        //调用无返回参数的存储过程
        Query query = em.createNativeQuery("{call AddPerson()}");
        query.executeUpdate();
        StringBuffer out = new StringBuffer("***************
QueryNoneReturnValueStoreProcedure 结果打印 ***************<BR>");
        return out.toString();
    }
}
```

例子的源代码在 Query 文件夹，要运行本例子，你首先需要创建 AddPerson 存储过程，然后调用 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/QueryTest.jsp?index=26 访问例子。

3.10.2 调用返回单值的存储过程

我们首先创建一个名为 GetPersonName 的存储过程，它有一个 INTEGER 类型的输入参数，存储过程的 DDL 如下（注：本例使用的是 MySql 数据库）：

```
CREATE PROCEDURE `GetPersonName`(IN Pid INTEGER(11))
     NOT DETERMINISTIC
    SQL SECURITY DEFINER
    COMMENT ''
BEGIN
     select personname from person where `personid`=Pid;
END;
```

上面的 select 语句不一定要从表中取数据，你也可以写这样：select ‘foshanren’

下面代码片断显示了返回单值的存储过程调用方法：

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QuerySingleObjectStoreProcedure(){
        //调用返回单个值的存储过程
```

```
        Query query = em.createNativeQuery("{call GetPersonName(?)}");
        query.setParameter(1, new Integer(1));
        String result = query.getSingleResult().toString();
        StringBuffer out = new StringBuffer("***************
QuerySingleObjectStoreProcedure 结果打印 ****************<BR>");
        out.append("返回值(人员姓名)为："+ result+ "<BR>");
        return out.toString();
    }
}
```

例子的源代码在 Query 文件夹，要运行本例子，你首先需要创建 GetPersonName 存储过程，然后调用 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/QueryTest.jsp?index=27 访问例子。

3.10.3 调用返回表全部列的存储过程

我们首先创建一个名为 GetPersonList 的存储过程，它的 DDL 如下（注：本例使用的是 MySql 数据库）：

```
CREATE PROCEDURE `GetPersonList`()
    NOT DETERMINISTIC
    SQL SECURITY DEFINER
    COMMENT ''
BEGIN
     select * from person;
END;
```

下面代码片断显示了返回表全部列的存储过程调用方法，我们可以让 EJB3 Persistence 运行环境将列值直接填充入实体对象（本例填充进 Person 对象），返回结果为实体类型的 List。

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryStoreProcedure(){
        //调用返回Person全部属性的存储过程
        Query query = em.createNativeQuery("{call GetPersonList()}", Person.class);
        List<Person> result = (List<Person>)query.getResultList();
        StringBuffer out = new StringBuffer("*************** QueryStoreProcedure 结
果打印 ****************<BR>");
        for(Person person : result){
            out.append(person.getName()+ "<BR>");
        }
        return out.toString();
    }
}
```

例子的源代码在 Query 文件夹，要运行本例子，你首先需要创建 GetPersonList 存储过程，然后调用 Ant 的 deploy

任务。通过 http://localhost:8080/EJBTest/QueryTest.jsp?index=28 访问例子。

3.10.4 调用返回部分列的存储过程

我们首先创建一个名为 GetPersonPartProperties 的存储过程，它的 DDL 如下（注：本例使用的是 MySql 数据库）：

```
CREATE PROCEDURE `GetPersonPartProperties`()
    NOT DETERMINISTIC
    SQL SECURITY DEFINER
    COMMENT ''
BEGIN
     SELECT personid, personname from person;
END;
```

上面的 select 语句不一定要从表中取数据，你也可以这样写：select 3000, ‘foshanren’
下面代码片断显示了返回部分列的存储过程调用方法.

```
public class QueryDAOBean implements QueryDAO {
    @PersistenceContext protected EntityManager em;
    …
    private String QueryPartColumnStoreProcedure(){
        //调用返回记录集部分列的存储过程
        Query query = em.createNativeQuery("{call GetPersonPartProperties()}");
        List<Object[]> result = (List<Object[]>) query.getResultList();
        StringBuffer out = new StringBuffer("***************
QueryPartColumnStoreProcedure 结果打印 ***************<BR>");
        for(Object[] row : result){//取每一行
            out.append("人员ID="+ row[0]+ "; 姓名="+ row[1]+ "<BR>");
        }
        return out.toString();
    }
}
```

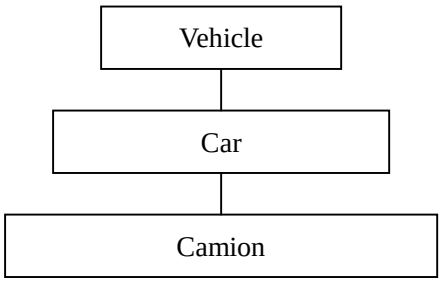
例子的源代码在 Query 文件夹，要运行本例子，你首先需要创建 GetPersonPartProperties 存储过程，然后调用 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/QueryTest.jsp?index=29 访问例子。

3.11 复合主键(Composite Primary Key)

这部分内容在《EJB3.0入门经典》中

3.12 实体继承

在本节我们定义了三个实体 Bean：Vehicle（交通工具），Car（汽车），Camion（卡车），它们的继承关系如下：



因为关系数据库的表之间不存在继承关系，为了将一个继承层次映射到关系数据库。Java Persistence 提供三种基本的继承映射策略：

- 每个类分层结构一张表(table per class hierarchy)
- 每个子类一张表(table per subclass)
- 每个具体类一张表(table per concrete class)

每种映射策略都有各自的优缺点，下面分别介绍这三种策略的使用。

3.12.1 每个类分层结构一张表(table per class hierarchy)

这种映射方式用一张表存放整个层次结构中每个类的全部成员属性，在表中还需要有一个字段用于区分子类的具体类型。

3.12.2 每个子类一张表(table per subclass)

这种映射方式为每个类创建一张表，每张表仅包含在当前类中定义的成员属性，不会包含父类或子类的成员属性。

3.12.3 每个具体类一张表(table per concrete class)

这种映射方式为每个类创建一张表，每张表都拥有当前类及其父类的所有成员属性映射的字段。

3.13 Entity 的生命周期和状态

当你调用 persist( )，merge( )，remove( )，find( ) 方法或执行 JPQL 查询的时候，将会触发实体 Bean 的生命周期事件。例如，persist( )方法触发数据插入事件，merge( )方法触发数据更新事件，remove( )方法触发数据删除事件，通过 JPQL 查询实体会触发数据载入事件。有时实体 Bean 得到这些事件发生的通知是非常有用的，例如，你想创建一个日志文件，用来跟踪对每条记录的操作（如：添，删，载入等）。

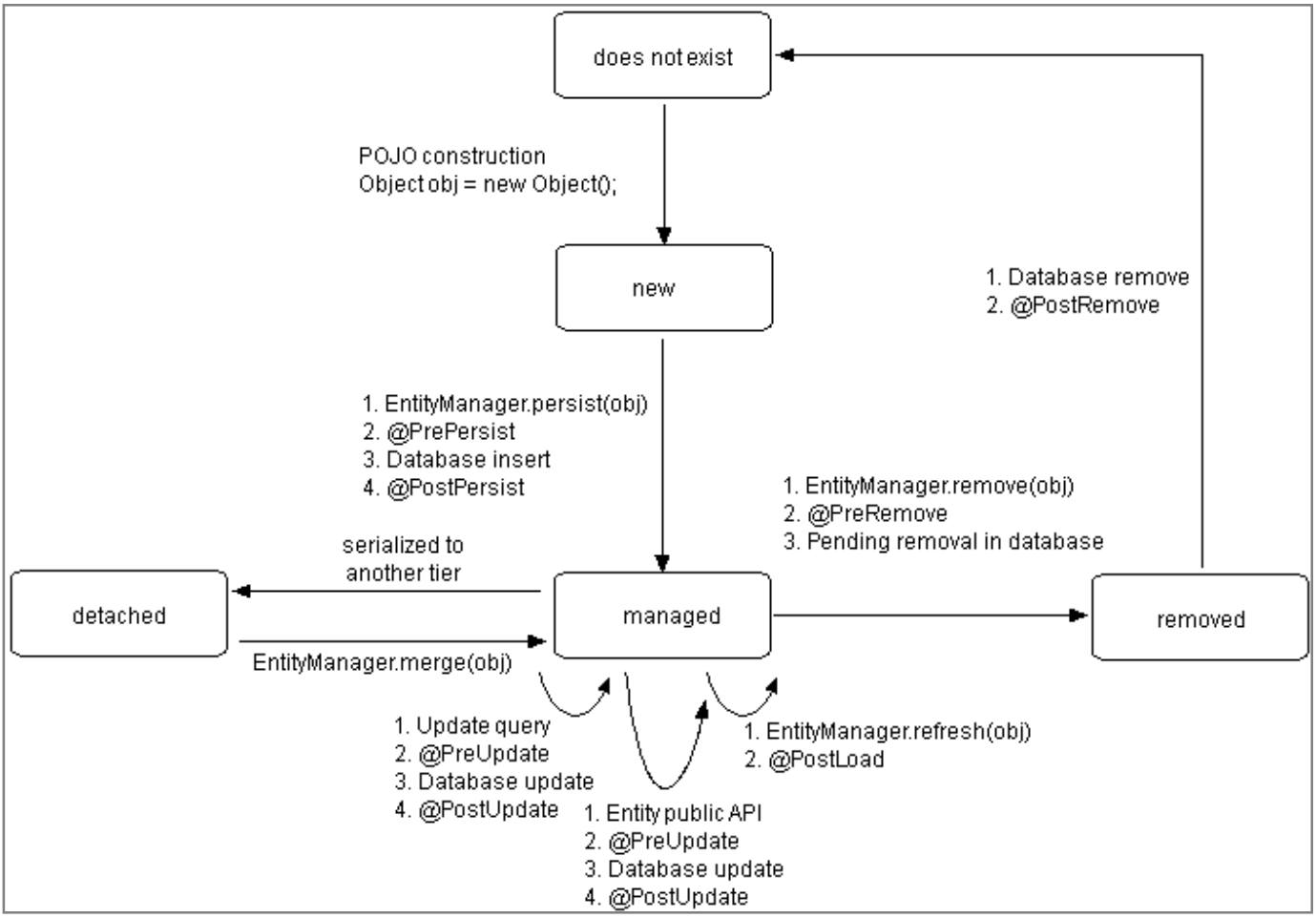
持久化规范允许你在实体类中实现回调方法，当这些事件发生时将会通知你的实体对象。当然你也可以使用一个外部类去拦截这些事件，这些类称作实体监听器。

这一节将教你如何在实体类中实现生命周期回调方法及怎样实现一个能拦截实体生命周期事件的实体监听器。

3.13.1 生命周期回调事件

如果需要在生命周期事件期间执行自定义逻辑，请使用以下注释关联生命周期事件与回调方法。EJB3.0 允许你将任何方法指定为回调方法，这些方法将会被容器在实体生命周期的不同阶段调用。
@javax.persistence.PostLoad
@javax.persistence.PrePersist
@javax.persistence.PostPersist
@javax.persistence.PreUpdate
@javax.persistence.PostUpdate
@javax.persistence.PreRemove
@javax.persistence.PostRemove

下图显示了实体生命周期事件发生的情况:



@PostLoad 事件在下列情况触发

1.执行 EntityManager.find( )或 getreference( )方法载入一个实体后

2.执行 JPQL 查询过后

3.EntityManager.refresh( )方法调用后

@PrePersist 和@PostPersist 事件在实体对象插入到数据库的过程中发生，@PrePersist 事件在调用 EntityManager.persist( )方法后立刻发生，级联保存的对象也会发生此事件，此时的数据还没有插进数据库。@PostPersist 事件在数据已经插进数据库后发生（注：这里说的“已经插进”是指事务未提交前的插进）。

@PreUpdate 和 @PostUpdate 事件的触发由更新实体引起，@PreUpdate 事件在实体的状态同步到数据库之前触发，此时的数据还没有真实更新到数据库。@PostUpdate 事件在实体的状态同步到数据库后触发，即容器进行 flush 时发生。

@PreRemove 和 @PostRemove 事件的触发由删除实体引起，@ PreRemove 事件在实体从数据库删除之前触发，即调用了 EntityManager.remove（）方法或者级联删除时发生，此时的数据还没有真实从数据库中删除。@PostRemove 事件在实体已经从数据库中删除后触发。

3.13.2 在外部类中实现回调

Entity listener（实体监听器）用作拦截实体生命周期事件，在 Entity listener 类里，你可以指定一个方法用于拦截

实体的某个生命周期事件，所指定的方法必须带有一个 Object 参数及返回值为 void，格式如下：
void <MethodName>(Object)
通过为方法加上事件注释，即完成特定事件与回调方法的关联，下面本例子代码：
EntityListenerLogger.java

```
package com.foshanshop.ejb3.bean.listener;
import javax.persistence.PostLoad;
import javax.persistence.PostPersist;
import javax.persistence.PostRemove;
import javax.persistence.PostUpdate;
import javax.persistence.PrePersist;
import javax.persistence.PreRemove;
import javax.persistence.PreUpdate;

public class EntityListenerLogger {

    @PostLoad
    public void postLoad(Object entity) {
        System.out.println("实体{" + entity.getClass().getName( ) + "}的@PostLoad
事件发生");
    }

    @PrePersist
    public void PreInsert(Object entity) {
        System.out.println("实体{" + entity.getClass().getName( )+ "}的@PrePersist
事件发生");
    }

    @PostPersist
    public void postInsert(Object entity) {
        System.out.println("实体{" + entity.getClass().getName( )+ "}的@PostPersist
事件发生");
    }

    @PreUpdate
    public void PreUpdate(Object entity) {
        System.out.println("实体{" + entity.getClass().getName( )+ "}的@PreUpdate
事件发生");
    }

    @PostUpdate
    public void PostUpdate(Object entity) {
        System.out.println("实体{" + entity.getClass().getName( )+ "}的@PostUpdate
事件发生");
    }
```

```
    @PreRemove
    public void PreRemove(Object entity) {
        System.out.println("实体{" + entity.getClass().getName( )+ "}的@PreRemove
事件发生");
    }

    @PostRemove
    public void PostRemove(Object entity) {
        System.out.println("实体{" + entity.getClass().getName( )+ "}的@PostRemove
事件发生");
    }
}
```

实体 bean 可以使用@javax.persistence.EntityListeners 注释为自己指定一个或多个监听器，多个监听器之间用逗号分隔，监听器按照定义的先后顺序执行。下面的实体使用上面定义好的监听器。

EntityLifecycle.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.EntityListeners;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import com.foshanshop.ejb3.bean.listener.EntityListenerLogger;

@Entity
@EntityListeners({EntityListenerLogger.class})
public class EntityLifecycle implements Serializable{
    private static final long serialVersionUID = 2619167645480125649L;
    private Integer id;
    private String name;

    public EntityLifecycle() {}

    public EntityLifecycle(String name) {
        this.name = name;
    }

    @Id
    @GeneratedValue
    public Integer getId() {
        return id;
    }
```

```
    public void setId(Integer id) {
        this.id = id;
    }

    @Column(nullable=false,length=32)
    public String getName() {
        return name;
    }

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

下面是本例 Session Bean 的业务接口及实现类

EntityLifecycleDAO.java

```
package com.foshanshop.ejb3;
import com.foshanshop.ejb3.bean.EntityLifecycle;

public interface EntityLifecycleDAO {
    public void Persist();
    public EntityLifecycle Load();
    public void Update();
    public void Remove();
}
```

EntityLifecycleDAOBean.java

```
package com.foshanshop.ejb3.impl;
import java.util.List;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import com.foshanshop.ejb3.EntityLifecycleDAO;
import com.foshanshop.ejb3.bean.EntityLifecycle;

@Stateless
@Remote (EntityLifecycleDAO.class)
@SuppressWarnings("unchecked")
public class EntityLifecycleDAOBean implements EntityLifecycleDAO {
    @PersistenceContext protected EntityManager em;
```

```
    public EntityLifecycle Load() {
        return em.find(EntityLifecycle.class, 1);//此处将会触发@PostLoad事件
    }

    public void Persist() {
        EntityLifecycle entitylifecycle = new EntityLifecycle("孙丽");
        em.persist(entitylifecycle);
    }

    public void Remove() {
        Query query = em.createQuery("select e from EntityLifecycle e");
        List<EntityLifecycle> result =
(List<EntityLifecycle>)query.getResultList();
        if (result.size()>0){
            EntityLifecycle entitylifecycle = result.get(0);
            em.remove(entitylifecycle);
        }
    }

    public void Update() {
        Query query = em.createQuery("select e from EntityLifecycle e");
        List<EntityLifecycle> result =
(List<EntityLifecycle>)query.getResultList();
        if (result.size()>0){
            EntityLifecycle entitylifecycle = result.get(0);
            entitylifecycle.setName("张权");
        }
    }
}
```

下面是 JSP 客户端：EntityListenerTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.*, com.foshanshop.ejb3.bean.*,
                javax.naming.*"%>
<%
    try {
        out.println("输出信息打印在Jboss控制台");
        InitialContext ctx = new InitialContext();
        EntityLifecycleDAO entitylifecycledao = (EntityLifecycleDAO)
ctx.lookup("EntityLifecycleDAOBean/remote");
        entitylifecycledao.Persist(); //添加测试数据
        entitylifecycledao.Load(); //载入数据
        entitylifecycledao.Update(); //更新数据
```

```
        entitylifecycledao.Remove(); //删除最后一条数据

    } catch (Exception e) {
        out.println(e.getMessage());
    }
%>
```

本例子的源代码在配套光盘的 EntityListeners 文件夹。例子使用的数据源配置文件是 mysql-ds.xml，你可以在配套光盘中找到。部署数据源到 Jboss 前，请先把数据库驱动拷贝到[jboss 安装目录]/server/default/lib 目录（请用你的配置名替换 default），拷贝完成后需重启 Jboss。要恢复 EntityListeners 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。
本例子的客户端代码在 EJBTest 文件夹，要发布客户端应用，你可以执行 Ant 的 deploy 任务。通过 http://localhost:8080/EJBTest/EntityListenerTest.jsp 访问客户端。

3.13.3 在 Entity 类中实现回调

除了可以在外部类实现生命周期事件的回调方法，你也可以把回调方法直接写在实体类中。要注意：直接写在实体类中的回调方法不需带任何参数，格式如下：
void <MethodName>()
EntityLifecycle.java

```
@Entity
public class EntityLifecycle implements Serializable{
    private static final long serialVersionUID = -579749403529370612L;
    private Integer id;
    private String name;

    public EntityLifecycle() {}

    public EntityLifecycle(String name) {
        this.name = name;
    }

    @Id
    @GeneratedValue
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
```

```
    @Column(nullable=false,length=32)
    public String getName() {
        return name;
    }

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

    @PostLoad
    public void postLoad() {
        System.out.println("实体{" + this.getClass().getName( ) + "}的@PostLoad事件
发生");
    }

    @PrePersist
    public void PreInsert() {
        System.out.println("实体{" + this.getClass().getName( )+ "}的@PrePersist事
件发生");
    }

    @PostPersist
    public void postInsert() {
        System.out.println("实体{" + this.getClass().getName( )+ "}的@PostPersist
事件发生");
    }

    @PreUpdate
    public void PreUpdate() {
        System.out.println("实体{" + this.getClass().getName( )+ "}的@PreUpdate事件
发生");
    }

    @PostUpdate
    public void PostUpdate() {
        System.out.println("实体{" + this.getClass().getName( )+ "}的@PostUpdate事
件发生");
    }

    @PreRemove
    public void PreRemove() {
        System.out.println("实体{" + this.getClass().getName( )+ "}的@PreRemove事件
发生");
    }
```

```
    @PostRemove
    public void PostRemove() {
        System.out.println("实体{" + this.getClass().getName( )+ "}的@PostRemove事
件发生");
    }
}
```

如果在实体类中既定义了外部监听器，内部又实现了回调方法。如：

```
@Entity
@EntityListeners({EntityListenerLogger.class})
public class EntityLifecycle implements Serializable{
    private Integer id;
    ...

    @PrePersist
    public void PreInsert() {
        System.out.println("实体{" + this.getClass().getName( )+ "}的@PrePersist事
件发生");
    }
}
```

这时会先执行外部监听器的回调方法，再执行实体内部的回调方法。

如果在一个实体继承层次中，实体监听器被用于基类，则所有子类都会继承这些监听器。如果子类也使用了监听器，则基类和子类的监听器都会与子类关联。如：

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@EntityListeners({EntityListenerLogger.class})
public class EntityLifecycle implements Serializable{
    private Integer id;
    ...
    @PrePersist
    public void PreInsert() {
        System.out.println("实体{" + this.getClass().getName( )+ "}的@PrePersist事
件发生");
    }
}

@Entity
@EntityListeners({SubListenerLogger.class})
public class SubLifecycle extends EntityLifecycle {
}
```

基类的监听器（本例为：EntityListenerLogger）会比子类的监听器（本例为：SubListenerLogger）先执行，在基类上定义的回调方法（如：本例的@PrePersist、PreInsert()方法）则在最后执行。如果你不希望执行从基类继承的监

听器，你可以使用@javax.persistence.ExcludeSuperclassListeners 注释来关闭。如：

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@EntityListeners({EntityListenerLogger.class})
public class EntityLifecycle implements Serializable{
    private Integer id;
    ...
    @PrePersist
    public void PreInsert() {
        System.out.println("实体{" + this.getClass().getName( )+ "}的@PrePersist事
件发生");
    }
}

@Entity
@EntityListeners({SubListenerLogger.class})
@ExcludeSuperclassListeners
public class SubLifecycle extends EntityLifecycle {
}
```

第四章　事务管理服务

事务可以确保一个工作单元中的多项任务要么一起成功，要么一起撤销（回滚）。对于事务，一般我们有两种事务管理的选择：本地事务和全局事务。对于本地事务，估计大家并不陌生，我们经常使用的 JDBC 事务就属于本地事务。下面是个典型的 JDBC 事务：

```
Connection conn = null;
try {
    DriverManager.registerDriver(new org.gjt.mm.mysql.Driver());
    Properties props = new Properties();
    props.put("user","root");
    props.put("password","123456");
    conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/foshanshop?useUnicode=true&
characterEncoding=GBK", props);
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("update person where name='叶子田'");
    stmt.executeUpdate("delete from person where personid=23");
    conn.commit();
    stmt.close();
} catch (Exception e) {
```

```
    e.printStackTrace();
}finally{
    try {
        if(null!=conn && !conn.isClosed()) conn.close();
    } catch (SQLException e) {
       e.printStackTrace();
    }
}
```

在上面的代码，不知大家有没有发现，JDBC 事务绑定在 Connnection 对象，你只有在同一个 Connnection 对象上执行多项任务，才能确保多项任务在同一个事务中。如果你需要访问多个不同的数据库，而且希望对不同数据库的操作能包含在同一个事务中。显然，JDBC 事务是无法满足这种需求的，因为每个数据库连接都有各自的 Connnection 对象，而 JDBC 事务的生命周期局限在了 Connnection 对象之内，所以它无法蹦出来将多个 Connnection 对象纳入到同一个事务中管理。全局事务正是冲着这种需求而来的，它可以管理多个事务性资源。这里说的事务性资源，不单单指关系数据库，像消息队列也属于事务性资源。纳入全局事务管理的多个事务性资源包含在同一个事务中，事务中的每一项任务必须全部成功，整个事务才能成功，如果有一项任务失败，则事务中由其它任务所做的修改都会回滚。

全局事务是通过 JTA 实现的，我们使用 JTA 的 UserTransaction 对事务进行操作。UserTransaction 由容器提供，要获取 UserTransaction，我们可以通过 JNDI 查找或注入注释。下面是使用 UserTransaction 操作全局事务的代码片断：

```
@Resource UserTransaction ut;
ut.begin( );
//执行一些任务
ut.commit( );
```

EJB 容器提供了两种使用全局事务的方式，一种是 Bean 管理事务（bean-managed transaction，BMT），它是编程式事务管理，为了更形象，作者在书中称它为手工管理事务。另一种是容器管理事务（Container-managed transaction，CMT），它是声明式事务管理。采用 CMT 的好处在于，你不需要在代码中操作事务，事务的开启/提交或回滚完全由容器负责处理。这样不但可以减少代码量，提高开发效率，而且使业务代码与事务代码之间实现了分离，对事务行为的修改不会影响业务逻辑，避免了像 JDBC 事务或 BMT 这样的入侵式的编程模型。没有了事务操作代码，代码也显的简单而优雅，CMT 是我们使用全局事务的首选。

4.1 容器管理事务(CMT)

这部分内容在《EJB3.0入门经典》中

4.2 Bean 管理事务(BMT)

4.3 事务并发的问题与处理

为了获得更好的运行性能，各个数据库都是允许多个事务同时运行。当这些事务访问或修改数据库中相同的数据时，如果没有采取必要的隔离机制，就会出现各种并发现象，这些并发现象可归纳为三种：

1. 脏读：一个事务读取到另一事务未提交的更新新据。
2. 不可重复读：在同一事务中，多次读取同一数据返回的结果有所不同。换句话说就是，后续读取可以读到另一事务已提交的更新数据。相反，“可重复读”在同一事务中多次读取数据时，能够保证所读数据一样，也就是，后续读取不能读到另一事务已提交的更新数据。
3. 幻读：一个事务读取到另一事务已提交的 insert 数据。

上面的并发现象在业务中倒底会给我们带来什么样的问题？下面就以作者交电费存工资为例。作者目前的银行存款为 1 万块（不小心把家底都露了^-^），每个月要交电费 200 元，每个月的工资收入为 12000 元。电费由银行代扣，每个月 1 号 12:00 点准时收取。工资由公司财务在每个月 1 号 12:00 点准时划入作者的银行账户。交电费与存工资刚好碰在一起，发生了事务并发。在现实生活中，交电费和存工资都完成后，作者的银行存款应该是 10000-200+12000=21800 元。下面我们假设当前的事务隔离级别为：Read Uncommitted，这种事务隔离级别导致上面三种并发现象都会出现。

4.4 因并发事务引起的更新丢失问题及处理

现实中，并发事务引起的更新丢失问题出现的概率很低，以至于我们的很多程序员没有及时发现及处理过此类问题。如果你的应用与金融有关，那你就不得不考虑此类问题。更新丢失问题是怎样导致的？如果作者用文字表达，估计你也很难理解，所以我们还是做个例子。前置条件是我们采用的是数据库默认的事务隔离级别，如 SQLSERVER 是 Read Commited，Mysql 是 Repeatable Read。下面还是以作者交电费存工资为例。

4.4.1 使用 SERIALIZABLE 隔离级别避免更新丢失

要避免更新丢失，必须保证第二个事务读取的数据是第一个事务提交后的数据，这就需要第二个事务在执行查询前必须等待第一个事务结束，要实现这个功能，我们需要为查询加上共享锁或排它锁，而且锁只能在事务结束时才解除。加锁工作可以是手工做，但一般是由数据库的自动管理锁机制负责，锁机制会依据事务的隔离级别自动为资源加上适合的锁。倒底会为资源加什么锁，各个数据库实现不太一样，你需要查看所用数据库的文档。Mysql 默认的事务隔离级别是不会为查询加共享锁的，SqlServer 的 REPEATABLE READ 能为查询加共享锁。如果希望在各种数据库下都能够让锁机制为 select 加共享锁，那么必须声明事务的隔离级别为 SERIALIZABLE。当事务的隔离级别设置为 SERIALIZABLE 后，我们再分析下面的时间片分配表：

4.4.2 修改代码逻辑来避免更新丢失

前面我们知道，造成更新丢失的源头是，多个并发事务读取同一资源，然后基于最初读到的值更新该资源。那我们能不能拿掉两个事务的读取操作，然后直接通过 update 语句进行累加或累减呢？

4.4.3 使用悲观锁避免更新丢失

悲观锁不是一种锁，更准确的说是一种悲观的加锁方案。每个人看待事物的态度都不一样，有些人比较悲观，有些人比较乐观。就拿对数据加锁的方案来说吧，有些人总认为别人会修改他读取的数据，所以在读取数据时就立刻加上排它锁，防止别人修改其读取的数据，这种行为就是悲观锁。

悲观锁避免更新丢失的原理和 SERIALIZABLE 隔离级别一样，也是通过保证第二个事务读取的是第一个事务提交后的数据。不过与 SERIALIZABLE 隔离级别不同，SERIALIZABLE 隔离级别是让数据库的锁机制负责为 select 加共享锁，影响面非常广。而悲观锁是通过手工为 select 加排它锁，对哪些 select 语句加锁由你决定。手工为 select 语句加排它锁的 SQL 语法如下：

```
Select * from users where id=1 for update
```

4.4.4 使用乐观锁避免更新丢失

和悲观锁一样，乐观锁也不是一种锁。它认为别人不会修改他读取的数据，所以在读取数据时并没有为记录加排它锁。难道它就任由其他并发事务覆盖其所作的更新？不是的，尽管它处事乐观，但也为自己留了一手，正所谓害人之心不可有，防人之心不可无呀!（看我不对数据加锁，你们就认为我好欺负，想覆盖我作的更新，没门!）。

那么乐观锁是如何防止其他并发事务覆盖前面事务所作的更新呢？原理很简单，它要求我们在数据库中增加一个版本字段（版本字段是什么？其实就是一个数字类型的字段，如 long），每次执行 update 语句时，这个字段都会进行累加，当其他事务以旧版本的值更新记录时，由于前面事务的 update 语句已经累加了版本字段，所以其他事务会找不到相应记录，这样避免了更新丢失。

这部分内容在《EJB3.0入门经典》中

第五章　消息服务（Java Message Service）

Java 消息服务（Java Message Service，简称 JMS）是用于访问企业消息系统的开发商中立的 API。企业消息系统可以协助应用软件通过网络进行消息交互。JMS 在其中扮演的角色与 JDBC 很相似，正如 JDBC 提供了一套用于访问各种不同关系数据库的公共 API，JMS 也提供了独立于特定厂商的企业消息系统访问方式。

使用 JMS 的应用程序被称为 JMS 客户端，处理消息路由与传递的消息系统被称为 JMS Provider，而 JMS 应用则是由多个 JMS 客户端和一个 JMS Provider 构成的业务系统。发送消息的 JMS 客户端被称为生产者（producer），而接收消息的 JMS 客户端则被称为消费者(consumer)。同一 JMS 客户端既可以是生产者也可以是消费者。

JMS 的编程过程很简单，概括为：应用程序 A 发送一条消息到消息服务器（也就是 JMS Provider）的某个目得地(Destination)，然后消息服务器把消息转发给应用程序 B。因为应用程序 A 和应用程序 B 没有直接的代码关连，所以两者实现了解偶。如下图：

消息传递系统的中心就是消息。一条 Message 由三个部分组成：

- 头（header）每条 JMS 消息都必须具有消息头。头字段包含用于路由和识别消息的值。可以通过多种方式来设置消息头的值：
  - 由 JMS 提供者在生成或传送消息的过程中自动设置
  - 由生产者客户机通过在创建消息生产者时指定的设置进行设置
  - 由生产者客户机逐一对各条消息进行设置

- 属性（property）消息可以包含称作属性的可选头字段。它们是以属性名和属性值对的形式指定的。可以将属性视为消息头的扩展，其中可以包括以下信息：创建数据的进程、数据的创建时间以及每条数据的结构。JMS 提供者也可以添加影响消息处理的属性，如是否应压缩消息或如何在消息生命周期结束时废弃消息。
- 主体（body）包含要发送给接收应用程序的内容。每个消息接口特定于它所支持的内容类型。

JMS 为不同类型的内容提供了它们各自的消息类型，但是所有消息都派生自 Message 接口。
StreamMessage：一种主体中包含 Java 基元值流的消息。其填充和读取均按顺序进行。
MapMessage：一种主体中包含一组名-值对的消息。没有定义条目顺序。
TextMessage：一种主体中包含 Java 字符串的消息（例如，XML 消息）。
ObjectMessage：一种主体中包含序列化 Java 对象的消息。
BytesMessage：一种主体中包含连续字节流的消息。

消息的传递模型：

JMS 支持两种消息传递模型：点对点（point-to-point，简称 PTP）和发布/订阅（publish/subscribe，简称 pub/sub）。这两种消息传递模型非常相似，但有以下区别：

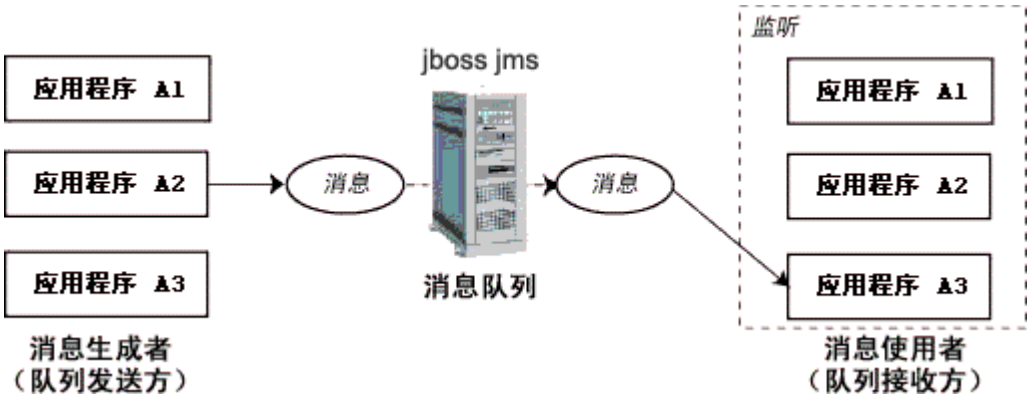
PTP 消息传递模型规定了一条消息只能传递给一个接收方。

Pub/sub 消息传递模型允许一条消息传递给多个接收方。

每种模型都通过扩展公用基类来实现。例如：javax.jms.Queue 和 javax.jms.Topic 都扩展自 javax.jms.Destination 类。

- 点对点消息传递

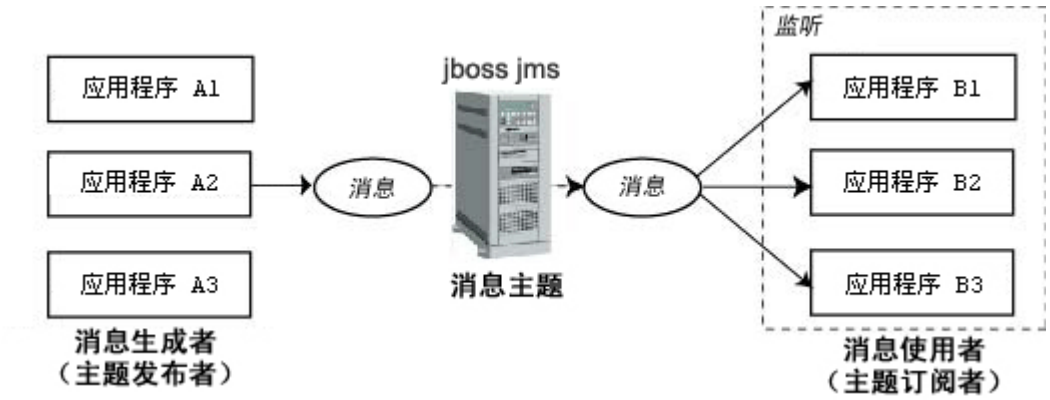
通过点对点 (PTP) 的消息传递模型，一个应用程序可以向另一个应用程序发送消息。在此传递模型中，目标类型是队列。消息首先被传送至队列目标，然后从该队列将消息传送至对此队列进行监听的某个消费者，如下图：



一个队列可以关联多个队列发送方和接收方，但一条消息仅传递给一个接收方。如果多个接收方正在监听队列上的消息，JMS Provider 将根据“先来者优先”的原则确定由哪个接收方接收下一条消息。如果没有接收方在监听队列，消息将保留在队列中，直至接收方连接到队列为止。这种消息传递模型是传统意义上的拉模型或轮询模型。在此类模型中，消息不是自动推送给客户端的，而是要由客户端从队列中请求获得。

- 发布/订阅消息传递

通过发布/订阅 (pub/sub) 消息传递模型，应用程序能够将一条消息发送到多个接收方。在此传送模型中，目标类型是主题。消息首先被传送至主题目标，然后传送至所有已订阅此主题的活动消费者。 如下图：



主题目标也支持长期订阅。长期订阅表示消费者已注册了主题目标，但在消息到达目标时该消费者可以处于非活动状态。当消费者再次处于活动状态时，将会接收该消息。如果消费者均没有注册某个主题目标，该主题只保留注册了长期订阅的非活动消费者的消息。

与 PTP 消息传递模型不同，pub/sub 消息传递模型允许多个主题订阅者接收同一条消息。JMS 一直保留消息，直至所有主题订阅者都收到消息为止。pub/sub 消息传递模型基本上是一个推模型。在该模型中，消息会自动广播，消费者无须通过主动请求或轮询主题的方式来获得新的消息。

上面两种消息传递模型里，我们都需要定义消息生产者和消费者，生产者把消息发送到 JMS Provider 的某个目标地址(Destination)，消息从该目标地址传送至消费者。消费者可以同步或异步接收消息，一般而言，异步消息消费者的执行和伸缩性都优于同步消息接收者，体现在：

1．异步消息接收者创建的网络流量比较小。单向推动消息，并使之通过管道进入消息监听器。管道操作支持将多条消息聚合为一个网络调用。

2．异步消息接收者使用的线程比较少。异步消息接收者在不活动期间不使用线程。同步消息接收者在接收调用期间内使用线程。结果，线程可能会长时间保持空闲，尤其是如果该调用中指定了阻塞超时。

3. 对于服务器上运行的应用程序代码，使用异步消息接收者几乎总是最佳选择，尤其是通过消息驱动 Bean。使用异步消息接收者可以防止应用程序代码在服务器上执行阻塞操作。而阻塞操作会使服务器端线程空闲，甚至会导致死锁。阻塞操作使用所有线程时则发生死锁。如果没有空余的线程可以处理阻塞操作自身解锁所需的操作，则该操作永远无法停止阻塞。

消息驱动 Bean 是异步消息消费者，它由 EJB 容器进行管理，具有一般的 JMS 消费者所不具有的优点。如：容器可创建多个消息驱动 Bean 实例来处理大量的并发消息，而一般的 JMS 消费者 (consumer)开发时则必须对此进行处理才能获得类似的功能。同时消息驱动 Bean 可取得 EJB 所能提供的标准服务，如容器管理事务等服务。

5.1 消息驱动 Bean (Message Driven Bean)

消息驱动 Bean(MDB)是设计用来专门处理基于消息请求的组件。MDB 负责处理消息，而 EJB 容器则负责处理服务（事务、安全、资源、并发、消息确认，等等），使 bean 开发者把精力集中在处理消息的业务逻辑上。如果你不使用 MDB，则必须编写一部分这些服务。MDB 像一个没有 local 和 remote 接口的无状态 Session Bean，它和无状态 Session Bean 一样也使用了实例池机制，容器可以为它创建大量的实例，用来并发处理成百上千个 JMS 消息。正因为 MDB 具有处理大量并发消息的能力，所以非常适合应用在一些消息网关产品。

一个 MDB 通常要实现 MessageListener 接口，该接口定义了 onMessage()方法。Bean 通过它来处理收到的 JMS 消息。

package javax.jms;

```
public interface MessageListener {
    public void onMessage(Message message);
}
```

当容器检测到 bean 守候的管道有消息到达时，容器调用 onMessage()方法，将消息作为参数传入 MDB。MDB 在 onMessage()中决定如何处理该消息。你可以使用注释指定 MDB 监听哪一个目标地址(Destination)。当 MDB 部署时，容器将读取其中的配置信息。

如果一个业务执行的时间很长，而执行结果无需实时向用户反馈时，也很适合使用 MDB。如订单成功后给用户发送一封电子邮件或发送一条短信等。

5.1.1 Queue 消息的发送与接收(PTP 消息传递模型)

开始 JMS 编程前，我们需要先配置消息到达的目标地址(Destination)，正如我们发送电子邮件一样，需要先知道对方的 email 地址。由于 JMS Provider 的实现产品众多（如：JbossMQ，IBM 的 MQSeries、BEA 的 WebLogic JMS service 等），因此每个应用服务器的配置方式都有所不同。下面以 jboss 为例。

Jboss 使用一个 XML 文件配置队列地址，其文件的取名格式应遵守*-service.xml，*号为自定义的一个或多个字符。下面是队列地址的配置文件：

foshanshop-service.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <mbean code="org.jboss.mq.server.jmx.Queue"
        name="jboss.mq.destination:service=Queue,name=foshanshop">
    <attribute name="JNDIName">queue/foshanshop</attribute>
    <depends
optional-attribute-name="DestinationManager">jboss.mq:service=DestinationManager
</depends>
  </mbean>
</server>
```

在这个配置文件，首先声明了 JMS 队列的 Mbean 类（org.jboss.mq.server.jmx.Queue），该类由 jboss 提供，然后是该 Mbean 的 JMX ObjectName (jboss.mq.destination:service=Queue,name=foshanshop)。ObjectName 的 name 属性定义了消息的目标地址，<attribute name="JNDIName">属性指定了该目标地址的全局 JNDI 名称。如果你不指定 JNDIName 属性，jboss 会为你生成一个默认的全局 JNDI，其名称由“queue”+“/”+目标地址名称组成。对本例而言，如果你不指定 JNDIName 属性，生成的全局 JNDI 为 queue/foshanshop。另外在任何队列或主题被部署之前，应用服务器必须先部署 Destination Manager Mbean，所以我们通过<depends>节点声明这一依赖。

配置文件编写完后，将它 copy 到[jboss 安装目录] server\default\deploy 目录下，这样会引发队列的热部署，你可以通过 http://localhost:8080/jmx-console 进入 Jboss 管理台，查看刚才部署的队列，如下：

其实我们在 JBOSS 中使用 MDB，可以不配置上述文件，Jboss 会根据 MDB 里的信息自动为我们创建队列地址。

当队列部署成功后，我们就可以针对该队列编写消息生产者，下面是一个普通 Java 应用，它发送 5 种类型的消息：

QueueSender.java

```
package com.foshanshop.ejb3.app;
import java.util.Properties;
import javax.jms.BytesMessage;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.StreamMessage;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.foshanshop.ejb3.bean.Man;

public class QueueSender {
    public static void main(String[] args) {
        QueueConnection conn = null;
        QueueSession session = null;
        try {
            Properties props = new Properties();
            props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
            props.setProperty(Context.PROVIDER_URL, "localhost:1099");
```

```
            props.setProperty(Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");
            InitialContext ctx = new InitialContext(props);

            QueueConnectionFactory factory = (QueueConnectionFactory)
ctx.lookup("QueueConnectionFactory");
            conn = factory.createQueueConnection();
            session = conn.createQueueSession(false,
QueueSession.AUTO_ACKNOWLEDGE);
            Destination destination = (Queue) ctx.lookup("queue/foshanshop");
            MessageProducer producer = session.createProducer(destination);

            //发送文本
            TextMessage msg = session.createTextMessage("佛山人您好，这是我的第一个消息
驱动Bean");
            producer.send(msg);

            //发送Ojbect(对象必须实现序列化,否则等着出错吧)
            producer.send(session.createObjectMessage(new Man("大美女", "北京朝阳区和
平里一号")));

            //发送MapMessage
            MapMessage mapmsg = session.createMapMessage();
            mapmsg.setObject("no1", "北京和平里一号");
            producer.send(mapmsg);

            //发送BytesMessage
            BytesMessage bmsg = session.createBytesMessage();
            bmsg.writeBytes("我是一个兵,来自老百姓".getBytes());
            producer.send(bmsg);

            //发送StreamMessage
            StreamMessage smsg = session.createStreamMessage();
            smsg.writeString("巴巴运动网,http://www.babasport.com");
            producer.send(smsg);

        } catch (Exception e) {
        e.printStackTrace();
        }finally{
            try {
                session.close ();
                conn.close();
            } catch (JMSException e) {
                e.printStackTrace();
```

```
            }
        }
    }
}
```

上面使用到的 Man.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;

public class Man implements Serializable{
    private static final long serialVersionUID = -1789733418716736359L;
    private String name;//姓名
    private String address;//地址

    public Man(String name, String address) {
        this.name = name;
        this.address = address;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
}
```

为了发送 JMS 消息，我们需要一个指向 JMS provider 的连接和一个消息目标地址。使用 JMS 连接工厂可以获得 JMS provider 连接，而消息目标地址则由 Queue/Topic 对象来表示。一旦拥有了 JMS provider 连接对象，就可以用它来创建 Session 对象，Session 对象允许你进行消息发送和接收的操作。

一般发送消息有以下步骤：

(1) 得到一个 JNDI 初始化上下文(Context)；

例子对应代码：

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
props.setProperty(Context.PROVIDER_URL, "localhost:1099");
props.setProperty(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
InitialContext ctx = new InitialContext(props);
```

(2) 根据上下文查找一个连接工厂 TopicConnectFactory/ QueueConnectionFactory （有两种连接工厂，根据 topic/queue 来使用相应的类型）。该连接工厂是由 JMS 提供的，不需我们自己创建，每个厂商都为它绑定了一个全局 JNDI，我们通过它的全局 JNDI 便可获取它；

例子对应代码：

QueueConnectionFactory factory = (QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");

(3) 从连接工厂得到一个连接(Connect 的类型有两种：TopicConnection/ QueueConnection);

例子对应代码：conn = factory.createQueueConnection();

(4) 通过连接来建立一个会话(Session);

例子对应代码：session = conn.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);

这句代码意思是：建立不需要事务的并且能自动确认消息已接收的会话。

(5) 查找目的地(目的地的类型有两种：Topic/ Queue);

例子对应代码：Destination destination = (Queue) ctx.lookup("queue/foshanshop");

(6) 根据会话以及目的地来建立消息生产者 MessageProducer （QueueSender 和 TopicPublisher 都扩展自 MessageProducer 接口）

例子对应代码：

MessageProducer producer = session.createProducer(destination);

TextMessage msg = session.createTextMessage("佛山人您好，这是我的第一个消息驱动 Bean");

producer.send(msg);

下面是 Queue 消息的接收方，它是一个 MDB

PrintBean.java

```
package com.foshanshop.ejb3.impl;
import java.io.ByteArrayOutputStream;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.BytesMessage;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;
import javax.jms.StreamMessage;
import javax.jms.TextMessage;
import com.foshanshop.ejb3.bean.Man;

@MessageDriven(activationConfig =
{
  @ActivationConfigProperty(propertyName="destinationType",
    propertyValue="javax.jms.Queue"),
  @ActivationConfigProperty(propertyName="destination",
    propertyValue="queue/foshanshop"),
  @ActivationConfigProperty(propertyName="acknowledgeMode",
    propertyValue="Auto-acknowledge")
})
public class PrintBean implements MessageListener {
```

```
    public void onMessage(Message msg) {
        try {
            if (msg instanceof TextMessage) {
                TextMessage tmsg = (TextMessage) msg;
                String content = tmsg.getText();
                System.out.println(content);
            }else if(msg instanceof ObjectMessage){
                ObjectMessage omsg = (ObjectMessage) msg;
                Man man = (Man) omsg.getObject();
                String content = man.getName()+ " 家住"+ man.getAddress();
                System.out.println(content);
            }else if(msg instanceof MapMessage){
                MapMessage map = (MapMessage) msg;
                String content = map.getString("no1");
                System.out.println(content);
            }else if(msg instanceof BytesMessage){
                BytesMessage bmsg = (BytesMessage) msg;
                ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
                byte[] buffer = new byte[256];
                int length = 0;
                while ((length = bmsg.readBytes(buffer)) != -1) {
                     byteStream.write(buffer, 0, length);
                }
                String content = new String(byteStream.toByteArray());
                byteStream.close();
                System.out.println(content);
            }else if(msg instanceof StreamMessage){
                StreamMessage smsg = (StreamMessage) msg;
                String content = smsg.readString();
                System.out.println(content);
            }

        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

上面使用@MessageDriven 注释指明这是一个消息驱动 Bean，它的定义如下：

```
@Target(value=TYPE) @Retention(value=RUNTIME)
public @interface MessageDriven{
    String mappedName() default "";
    String name() default "";
```

```
    Class messageListener Default Object.class
    ActivationConfigProperty[] activationConfig() Default {}
    String description() default "";
};
```

mappedName()属性：指定 MDB 正在监听的主题或者队列的 JNDI 名称。在 Weblogic/sun Application Server/Glassfish 应用服务器，你应使用该属性来指定主题或者队列的 JNDI 名称。

name()属性：指定 MDB 的名称，默认为 bean 类的非限定名。

messageListener()属性：指定 MDB 扩展的接口类。

description()属性：指定 bean 类的描述

activationConfig()属性接受一组@ActivationConfigProperty 注释作为参数，用一组简单的“名/值”对来描述 MDB 的配置。上面使用到的“名/值”对解析如下：

- destinationType 属性用于指定消息的类型，取值可以是 javax.jms.Queue 或 javax.jms.Topic。
- destination 属性指定消息的目标地址，取值是目标地址的 JDNI 名称。（该属性与厂商相关的）
- acknowledgeMode 属性指定消息的确认模式，取值可以是 Auto-acknowledge 或 Dups_ok_acknowledge。所谓消息确认（acknowledgment）是指 JMS 客户端通知 JMS provider 确认消息已经到达的一种机制。在 EJB 中，收到消息后发送确认是 EJB 容器的职责。确认一条消息，就是告诉 JMS provider，EJB 容器已经收到并处理了这条消息。如果没有经过确认。JMS provider 就不知道容器是否收到了消息，进而会重发消息。

Auto-acknowledge 模式告诉容器，在消息交给 MDB 实例处理后，容器应该立刻向 JMS provider 发送一个确认。Dups_ok_acknowledge 模式则表示，容器不必立即向 JMS provider 发送确认，容器在 MDB 实例收到消息后的任何时刻发送确认都是可接受的。使用 Dups_ok_acknowledge 模式时，MDB 容器可能延迟很长时间才会发送确认，于是 JMS provider 可能会认为容器没有收到消息，进而会重发消息。显然，在使用 Dups_ok_acknowledge 模式时，你的 MDB 必须能够正确处理重复消息。

Auto-acknowledge 模式可以避免重复消息，由于确认是立刻发送的，所以 JMS provider 不会发送重复消息。为了避免同一消息被两次处理，大多数 MDB 都使用 Auto-acknowledge 模式。Dups_ok_acknowledge 模式之所以存在，是因为它允许 JMS provider 优化对网络的使用，但实际上，确认的开销非常之小，而 MDB 容器和 JMS provider 通信的频率却非常之高，因此 Dups_ok_acknowledge 模式对性能提升作用不大。

实际上大多数情况下确认模式都会被忽略，除非 MDB 在 BMT 中执行，或者在具有 NotSupported 事务属性的 CMT 中执行。而在其他情况下，事务都是由容器来管理的，确认行为是在事务上下文中执行的，如果事务成功，消息就会被确认，如果事务失败，消息就不会被确认。当使用具有 Required 事务属性的容器管理事务时，我们通常不用指定确认模式。

@ActivationConfigProperty 注释定义的属性有部分跟 JMS provider 厂商有关，你应查看相关产品的文档。其中属于 EJB3.0 规范定义的属性有：acknowledgeMode, messageSelector, destinationType 和 subscriptionDurability。

运行本例子，当一条消息到达 queue/foshanshop 队列，EJB 容器会从队列中获取该消息，并把它交由 PrintBean 处理。此时会触发 onMessage 方法，消息作为一个参数传入。

本例子的源代码在配套光盘的 MessageDrivenBean 文件夹。要恢复 MessageDrivenBean 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。本例子的客户端代码在 com.foshanshop.ejb3.app 包，文件名为 QueueSender.java，因为是普通 Java 应用，所以你可以直接运行它。

如果你不想在普通 Java 应用中发送 Queue 消息，你也可以在 Session bean 中发送消息，相对普通 Java 应用，使用 Session bean 发送消息的代码量更少，如下：

5.1.2 Topic 消息的发送与接收(Pub/sub 消息传递模型)

从前面介绍的 pub/sub 消息传递模型中我们已经知道，Topic 消息允许多个主题订阅者接收同一条消息。所以本例子定义了两个消息接收者，当一条消息到达时，这两个接收者都可以收到。应用开发的第一步也是先配置消息到达的目标地址，和前面例子不同的是本例使用的是主题地址。配置如下：

foshanshop-service.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <mbean code="org.jboss.mq.server.jmx.Topic"
     name="jboss.mq.destination:service=Topic,name=chatTopic">
    <attribute name="JNDIName">topic/chatTopic</attribute>
    <depends
optional-attribute-name="DestinationManager">jboss.mq:service=DestinationManag
er</depends>
  </mbean>
</server>
```

如果你没有配置 JNDIName 属性，默认生成的全局 JNDI 名称为：“topic”+“/”+目标地址名称，对本例而言就是 topic/chatTopic。文件配置完后，我们将它 copy 到[jboss 安装目录] server\default\deploy 目录下。接下来，我们就可以针对该主题编写消息生产者了。下面是一个普通 Java 应用，它发送文本类型的消息：

TopicSender.java

```
package com.foshanshop.ejb3.app;
import java.util.Properties;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
public class TopicSender {

    public static void main(String[] args) {
        TopicConnection conn = null;
        TopicSession session = null;
        try {
            Properties props = new Properties();
            props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
            props.setProperty(Context.PROVIDER_URL, "localhost:1099");
            props.setProperty(Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");
            InitialContext ctx = new InitialContext(props);

            TopicConnectionFactory factory = (TopicConnectionFactory)
ctx.lookup("TopicConnectionFactory");
            conn = factory.createTopicConnection();
            session = conn.createTopicSession(false,
TopicSession.AUTO_ACKNOWLEDGE);
            Destination destination = (Topic) ctx.lookup("topic/chatTopic");
            MessageProducer producer = session.createProducer(destination);
            //发送文本
            TextMessage msg = session.createTextMessage("您好，这是我的第一个消息驱动
Bean");
            producer.send(msg);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }finally{
            try {
                session.close ();
                conn.close();
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Topic 消息发送的步骤与 queue 消息相同，但 Topic 消息是使用 TopicConnectionFactory 连接工厂创建到 Topic 目标地址的连接。

第一个 MDB：TopicPrintBeanOne.java

```
package com.foshanshop.ejb3.impl;
import javax.ejb.ActivationConfigProperty;
```

```
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(activationConfig =
{
  @ActivationConfigProperty(propertyName="destinationType",
    propertyValue="javax.jms.Topic"),
  @ActivationConfigProperty(propertyName="destination",
    propertyValue="topic/chatTopic")
})
public class TopicPrintBeanOne implements MessageListener{

    public void onMessage(Message msg) {
        try {
            TextMessage tmsg = (TextMessage) msg;
            String content = tmsg.getText();
            System.out.println(this.getClass().getName()+"=="+ content);
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

第二个 MDB：TopicPrintBeanTwo.java

```
package com.foshanshop.ejb3.impl;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(activationConfig =
{
  @ActivationConfigProperty(propertyName="destinationType",
    propertyValue="javax.jms.Topic"),
  @ActivationConfigProperty(propertyName="destination",
    propertyValue="topic/chatTopic")
})
public class TopicPrintBeanTwo implements MessageListener{

    public void onMessage(Message msg) {
        try {
```

```
            TextMessage tmsg = (TextMessage) msg;
            String content = tmsg.getText();
            System.out.println(this.getClass().getName()+"=="+ content);
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

本例子的源代码在配套光盘的 MessageDrivenBean 文件夹。要恢复 MessageDrivenBean 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。本例子的客户端代码在 com.foshanshop.ejb3.app 包，文件名为 TopicSender.java，因为是普通 Java 应用，所以你可以直接运行它。

如果你不想在普通 Java 应用中发送 Topic 消息，你也可以在 Session bean 中发送消息，如下：

5.1.3 消息选择器(Message selector)

消息选择器允许 MDB 选择性地接收来自队列或主题特定的消息。消息选择器是基于消息属性进行选择的。消息属性是一种可以被附加于消息之上的头信息，开发人员可以通过它为消息附加一些信息，而这些信息不属于消息正文。Message 接口提供了若干属性读写的方法。属性值可以是 String 类型或某种基本数据类型(boolean, byte, short, int, long, float, double)。属性的命名、取值以及类型转换规则，都由 JMS 给出了严格的定义。

当你的应用需要增加一种新业务，这种新业务需要在旧的消息格式上增加若干个参数。为了避免影响到其它业务模块，你决定增加新的 MDB 来处理新的业务消息，并且不希望新的业务消息被旧的业务模块所接收，这时你就需要使用到消息选择器。你可以为消息定义一个版本属性，规定旧消息使用 1.0 版本，新消息使用 2.0 版本。然后属于新业务的 MDB 只接收 2.0 版本的消息，旧业务的 MDB 只接收 1.0 版本的消息。具体操作如下：

第六章　Web 服务(Web Service)

Web 服务也是一种分布式技术，它与 EJB 最大的不同是，Web 服务属于行业规范，可以跨平台及语言。而 EJB

属于 java 平台的规范，尽管理论上可以跨平台，但实现起来比较复杂，所以其应用范围局限在了 java 平台。看上去两者好像是互相竞争的关系，其实不是。它们两者的偏重点不一样，Web 服务偏重的是这个系统对外提供什么功能，而 EJB 偏重的是如何用一个个组件组装这些功能。就好比一个硬盘，它对外提供的是存储服务，这是 web 服务的关注点，对于怎样组装这个硬盘，怎样构造这些小零件，web 服务并不关心，但这些却是 EJB 所关注的。

Web service 技术到今天仍在不断地发展，规范也在不断地升级，本书将以最新的 JAX-WS2.x 规范（Java API for XML-based Web Services）介绍 webservice 的开发。由于 web service 这个主题相当之大，关联的协议也比较多，如果深入地介绍它，完全可以再出一本书。尽管 web service 的内容比较多，但站在开发的角度来说，实际要我们掌握的知识并不多，因为很多实现细节都由平台或第三方组件实现了。本书浓缩了 web service 开发的精要，学完后完全可以胜任 Web service 方面的开发。

JavaEE 为 web service 提供了两种不同的编程模型：EJB 容器模型及 Web 容器模型，而 EJB 容器模型是作者推荐使用的。

6.1 EJB 容器模型的 Web Service 开发

这部分内容在《EJB3.0入门经典》中

6.2 Web 容器模型的 Web Service 开发

除了可以将一个 ejb 发布为 web service 外，我们还可以把一个 POJO 以 servlet 形式发布为 Web Service。开发步骤如下：

1. 建立一个 POJO 端点
2. 在 web.xml 中，将这个 POJO 端点配置成一个 servlet
3. 把 POJO 端点打包成一个 Web 应用(war 文件)

按照上面的开发步骤，下面我们开始建立一个 POJO 端点。

```
package com.foshanshop.web.ws;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(targetNamespace="http://ws.foshanshop.com",
        name = "HelloWorld",
        serviceName = "HelloWorldService")
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT)
public class HelloWorldService{
```

```
    @WebMethod(operationName="SayHello")
    public String SayHello(@WebParam(name="name") String name) {
        return name +"说：你好!世界,这是我的第一个web service哦.";
    }
}
```

接着把 POJO 端点定义成一个 servlet.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" 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_2_5.xsd">

    <servlet>
      <servlet-name>HelloWorldService</servlet-name>
      <servlet-class>com.foshanshop.web.ws.HelloWorldService</servlet-class>
    </servlet>

    <servlet-mapping>
      <servlet-name>HelloWorldService</servlet-name>
      <url-pattern>/HelloWorldService</url-pattern>
    </servlet-mapping>
</web-app>
```

最后把项目打包成一个 web 应用(*.war)，下面是 Ant 的配置文件 build.xml：

```
<?xml version="1.0"?>
<!-- ====================================================================== -->
<!--  JWS build file                                                        -->
<!-- ====================================================================== -->
<project name="JWS" default="war" basedir="..">
    <property environment="env" />
    <property name="app.dir" value="${basedir}\JWS" />
    <property name="src.dir" value="${app.dir}\src" />
    <property name="jboss.home" value="${env.JBOSS_HOME}" />
    <property name="jboss.server.config" value="default" />
    <property name="build.dir" value="${app.dir}\build" />
    <property name="build.classes.dir" value="${build.dir}\classes" />

    <!-- Build classpath -->
    <path id="build.classpath">
        <fileset dir="${jboss.home}\client">
            <include name="**/*.jar" />
```

```
        </fileset>
        <pathelement location="${build.classes.dir}" />
    </path>

    <!-- ================================================================== -->
    <!-- Prepares the build directory                                       -->
    <!-- ================================================================== -->
    <target name="prepare" depends="clean">
        <mkdir dir="${build.dir}" />
        <mkdir dir="${build.classes.dir}" />
    </target>

    <!-- ================================================================== -->
    <!-- Compiles the source code                                           -->
    <!-- ================================================================== -->
    <target name="compile" depends="prepare" description="编绎web服务">
        <javac srcdir="${src.dir}" destdir="${build.classes.dir}" debug="on"
deprecation="on" optimize="off" includes="**">
            <classpath refid="build.classpath" />
        </javac>
    </target>

    <target name="war" depends="compile" description="创建WS发布包">
        <war warfile="${app.dir}\Services.war"
webxml="${app.dir}\WEB-INF\web.xml">
            <classes dir="${build.classes.dir}">
                <include name="com/**" />
            </classes>
        </war>
    </target>

    <target name="deploy" depends="war">
        <copy file="${app.dir}\Services.war"
todir="${jboss.home}\server\${jboss.server.config}\deploy" />
    </target>

    <!-- ================================================================== -->
    <!-- Cleans up generated stuff                                          -->
    <!-- ================================================================== -->
    <target name="clean">
        <delete dir="${build.dir}" />
        <delete
file="${jboss.home}\server\${jboss.server.config}\deploy\Services.war" />
    </target>
```

```
</project>
```

本例子的源代码在配套光盘的 JWS 文件夹。要恢复 JWS 项目的开发环境请参考第一章“如何恢复本书配套例子的开发环境”。要发布本例子到 Jboss(确保配置了环境变量 JBOSS_HOME 及启动了 Jboss)，你可以执行 Ant 的 deploy 任务。

6.3 Web Service 的客户端调用

6.3.1 在 J2SE 或 Web 中调用 Web Service

在 J2SE 或独立 Tomcat 中调用 Web Service，开发步骤如下：

1. 在应用的类路径下放入 JAX-WS 的 jar 文件，对于 web 应用，应放入 WEB-INF/lib 目录。如果你使用的是 JDK6，这一步可以省略，因为 JDK6 已经绑定了 JAX-WS。目前 JDK6 绑定的 JAX-WS 版本是 2.0。这意味着，当某些应用使用的 JAX-WS 版本高于 2.0 时，就有可能发生版本问题，这时，你需要升级 JDK6 中的 JAX-WS 版本，方法如下：
   下载最高版本的 JAX-WS，在产品 lib 目录中找到 jaxws-api.jar 和 jaxb-api.jar ，把这两个文件 copy 到 JDK6_HOME/jre/lib/endorsed 目录下。
2. 利用 Web Service 客户端生成工具生成辅助类。
3. 第三步：借助辅助类调用 web service。

假设你使用的是 JDK5，你需要在应用的类路径下加入 JAX-WS 的 jar 文件。这些 jar 文件你可以从本书配套光盘的 lib/JAX-WS2.1.2 中得到（JAX-WS 版本为 2.1.2），或者从网上下载，下载路径为 https://jax-ws.dev.java.net。

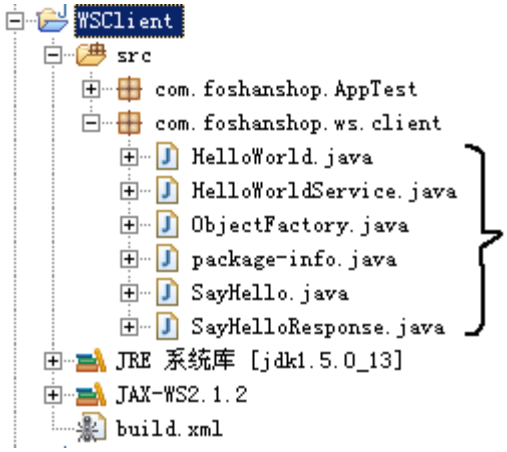
按照开发步骤，下面我们应该生成辅助类，这里我们使用 wsimport 工具的 Ant 任务类，如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="WSClient" default="wsclientgen" basedir="..">
    <property name="app.dir" value="${basedir}/WSClient" />
    <property name="src.dir" value="${app.dir}/src" />
    <!-- Build classpath -->
    <path id="build.classpath" description="设置类路径">
        <fileset dir="${basedir}/lib/JAX-WS2.1.2">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="wsclientgen" description="生成webservice客户端辅助代码，执行后请刷新项目">
    <taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport" classpathref="build.classpath"/>
    <wsimport wsdl="http://localhost:8080/WsHelloWorld/HelloWorldBean?wsdl"
        sourcedestdir="${src.dir}" package="com.foshanshop.ws.client" keep="true"
        verbose="true" destdir="${app.dir}/bin"/>
```

```
    </target>
</project>
```

上面通过<taskdef>节点声明 com.sun.tools.ws.ant.WsImport 任务标签，标签名为 wsimport，后面通过这个标签名使用这个任务。wsdl 属性为 web 服务的位置，package 属性为生成代码所使用的包名，sourcedestdir 属性为生成代码存放的目录，keep 属性指定是否生成文件，verbose 属性指定是否显示执行细节。

执行 wsclientgen 任务后，生成的辅助类如下：



HelloWorldService 为服务类，HelloWorld 为服务的端点接口，SayHello 为 SayHello()方法的请求消息封装类，SayHelloResponse 为 SayHello()方法的返回消息封装类。服务类用于跟 web service 进行通信，它必须扩展自 javax.xml.ws.Service，并提供一个方法用来获取服务的端点接口。另外它还应该使用 javax.xml.ws.WebServiceClient 注释来定义名称、名字空间，以及 WSDL 所在位置。由于这些文件都可以通过工具来生成，所以我们也就不太关心如何编写它们了。

注意：本书提供的 wsimport 工具生成的辅助类属于 JAX-WS 2.1.2 版本。

现在我们就可以借助辅助类调用 webservice 了，代码如下：

```
package com.foshanshop.AppTest;
import com.foshanshop.ws.client.HelloWorld;
import com.foshanshop.ws.client.HelloWorldService;

public class TestHelloWorld {
    public static void main(String[] args) {
       try {
        HelloWorldService service = new HelloWorldService();
        HelloWorld helloWorld = service.getHelloWorldPort();//从服务中取得入口端点
        String result = helloWorld.sayHello("张朗");
            System.out.println(result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在 JDK6 中，Sun 已经为我们提供了 wsimport 工具。该工具提供了两种使用方式：命令行和 Ant 任务。Ant 任务的使用方法和前面一样。命令行的语法如下：

wsimport [options] <wsdl>

例：

```
wsimport -d E:\JavaProject\WSClient\bin -keep -s E:\JavaProject\WSClient\src -verbose
http://localhost:8080/WsHelloWorld/HelloWorldBean?wsdl -p com.foshanshop.ws.client
```

上面各选项的含义如下：

-d <directory> 指定保存目录，存放生成类编绎后的 class 文件。

-s <directory>指定保存目录，存放生成的源文件。

-keep 指定是否生成文件。

-verbose 指定是否显示执行细节。

-p 指定生成文件所使用的包名。

注意：目前，该工具只生成支持 JAX-WS 2.0 的辅助类。

本例子的源代码在配套光盘的 WSClient 文件夹，项目使用到的类库在上级目录 lib/JAX-WS2.1.2 文件夹下。生成辅助代码时，应先发布 WsHelloWorld 项目的 web 服务，然后执行 ANT 的 wsclientgen 任务。要调用 webservice，你可以直接运行 TestHelloWorld。

6.3.1 在 EJB 中调用 Web Service

在 EJB 容器中我们一般通过注入注释来使用 Web Service。开发步骤如下：

1. 利用工具生成辅助类。
2. 使用注释注入服务类或服务端口。

这部分内容在《EJB3.0入门经典》中

第七章 在 Weblogic 中使用 EJB3.0

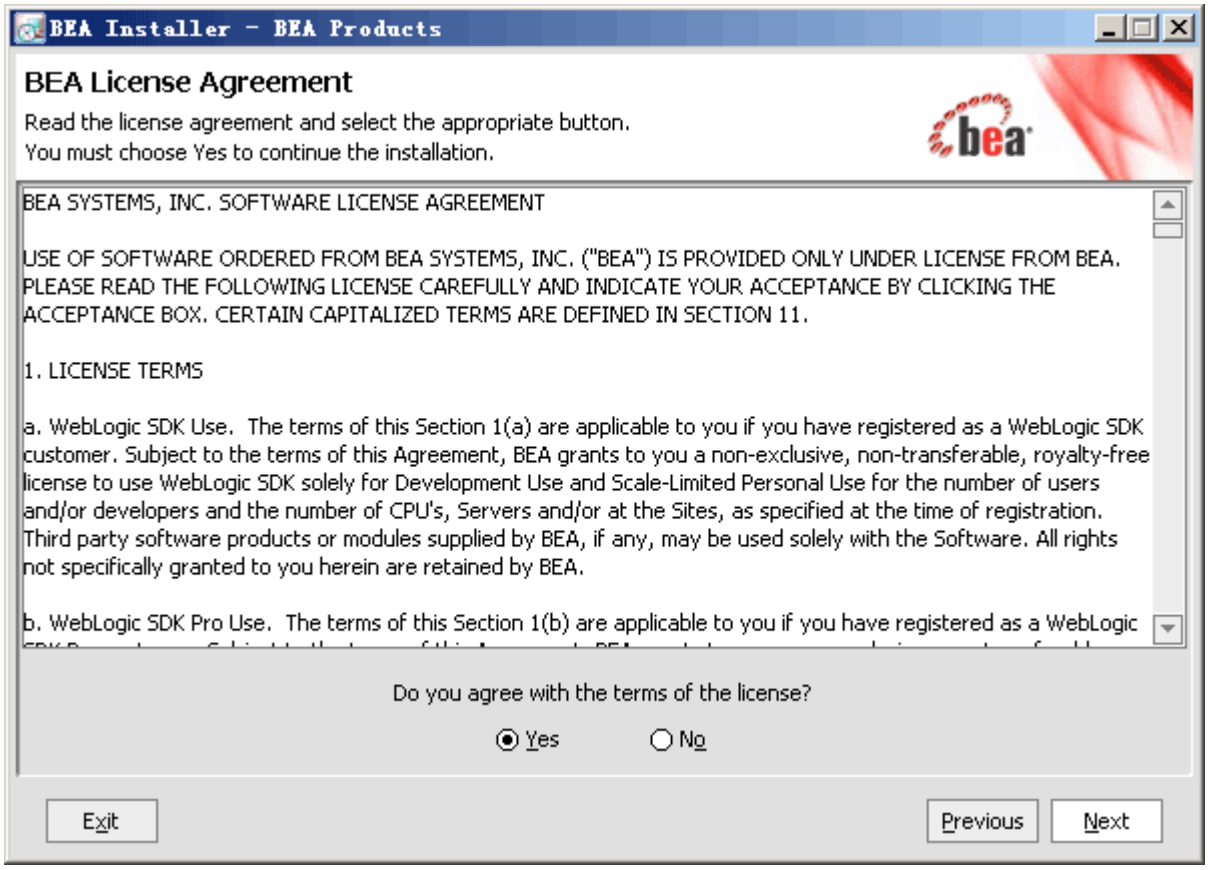
WebLogic Server 是一个功能丰富、基于标准的应用服务器，它为企业构建可靠、可伸缩和可管理的应用程序提供了一个坚实的基础。目前，在商业的 JavaEE 应用服务器市场中，weblogic 占据的市场份额是最大的。但在 ejb3.0 领域，作者在使用中发现，其稳定性不如 jboss，尤其是其 JPA 部分，因为采用的是 OpenJPA，而 OpenJPA 相对 Hibernate，TopLink 产品来说，还是有一定的差距，不过相信会越来越好的。

7.1 Weblogic 的安装

目前支持 EJB3.0 的版本是 Weblogic10 以上，本书使用的是 Weblogic10.3，大家可以从下面地址下载：

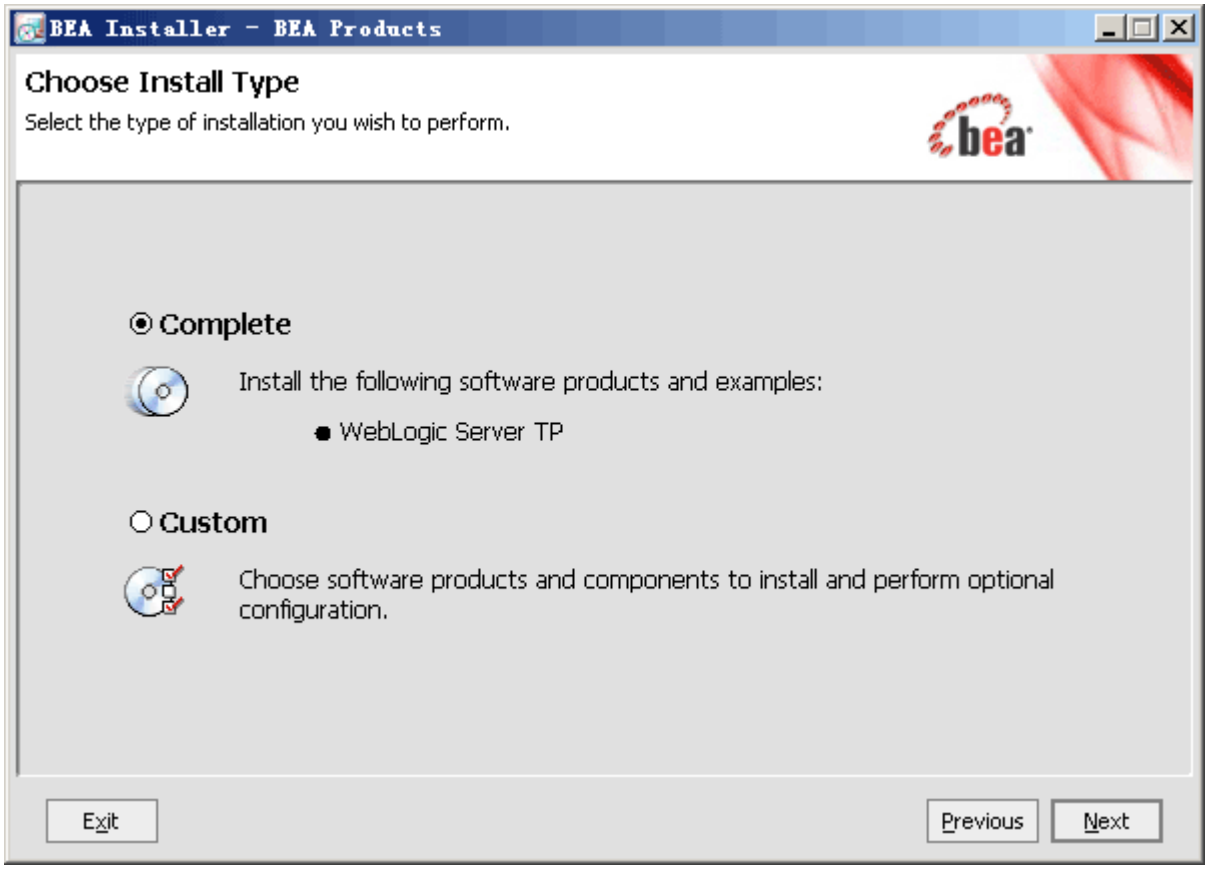
http://commerce.bea.com/products/weblogicplatform/weblogic_prod_fam.jsp?DL=www_WL-Fam_icon&WT.ac=DL_www_WL-Fam_icon

Weblogic 的安装非常简单，只要按照向导一步步安装即可。

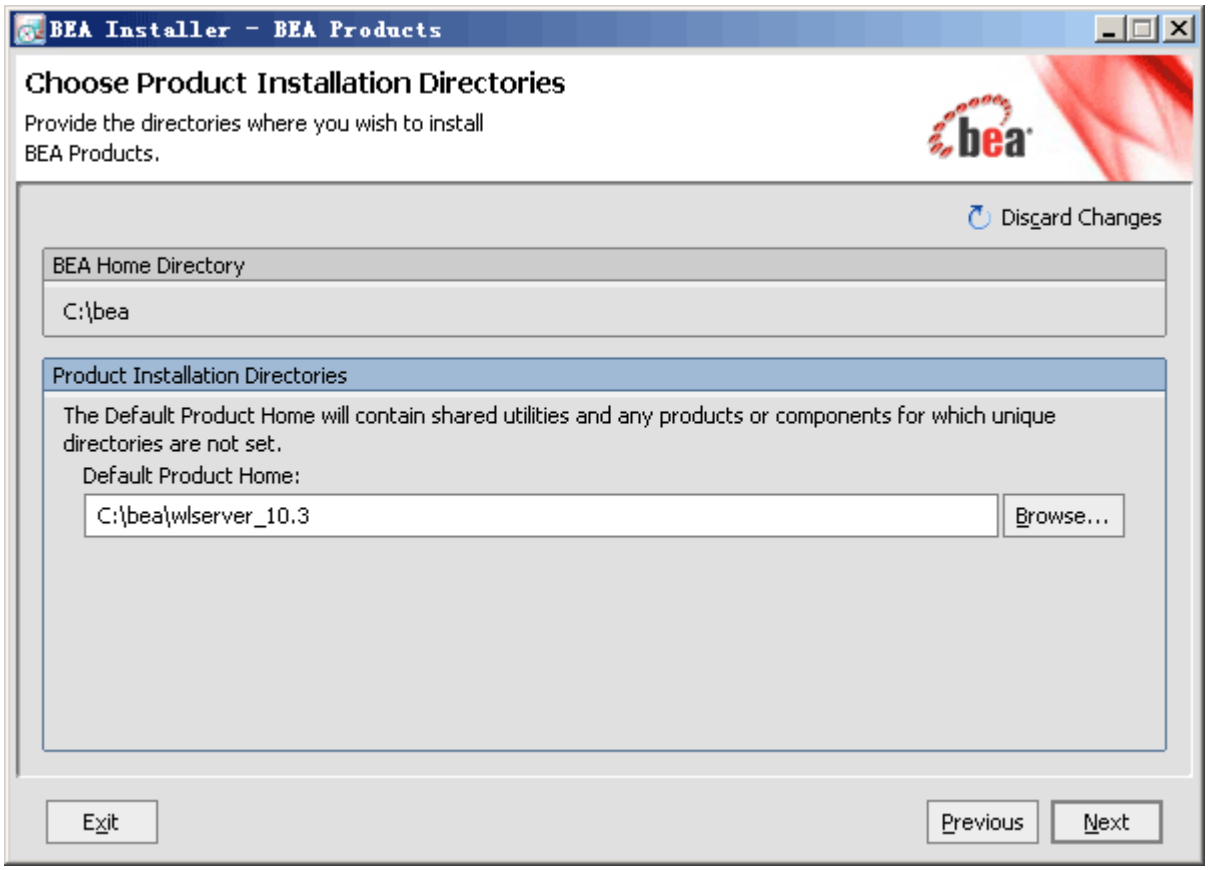


选择“YES”同意相关条约，点“Next”。

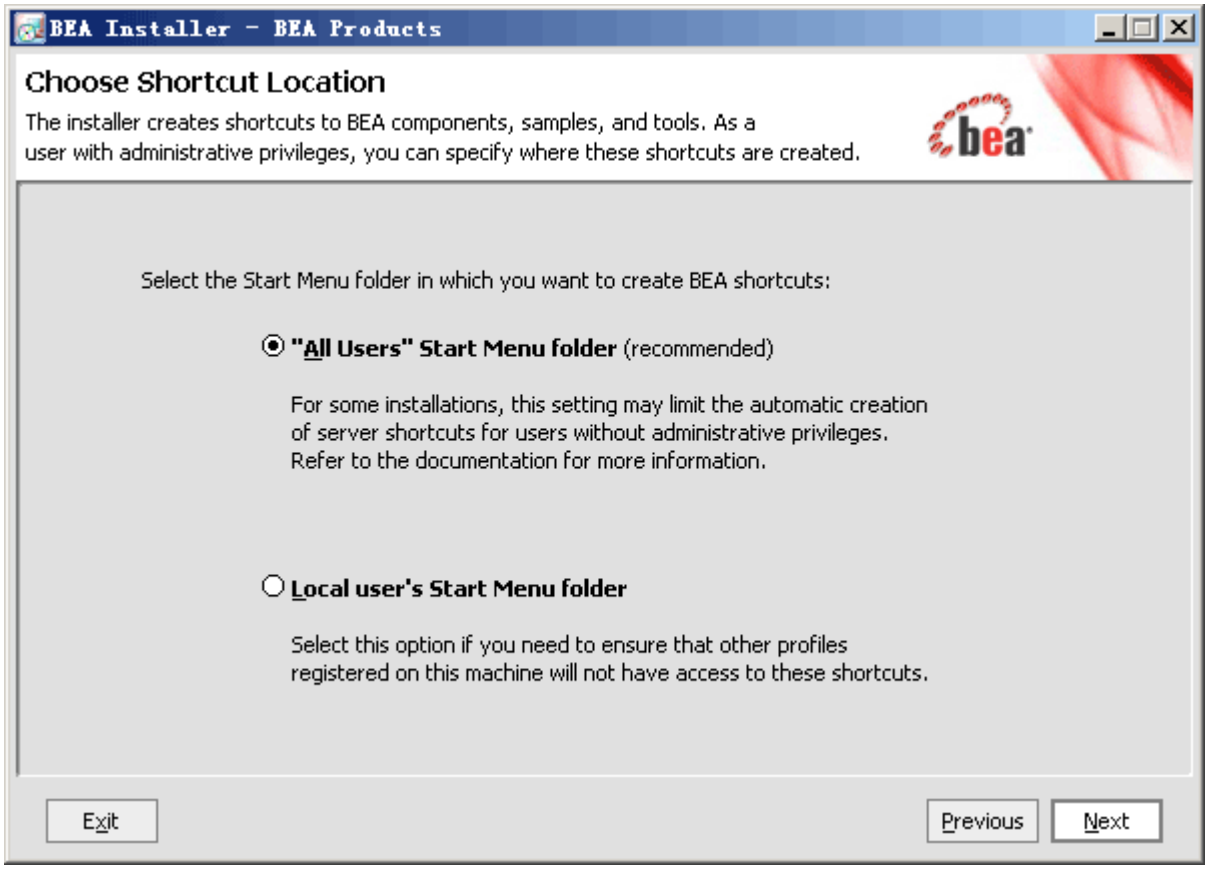
选择 bea 安装目录，默认安装在 c:\bea。书中例子 Ant 用到了该默认路径，如果你不想修改 build.xml，请安装在默认路径。



安装类型有“complete(全部)”和“Custom(自定义)”安装，这里我们选择“complete”安装。



接着选择产品的安装目录，这里我们安装在 bea 安装目录的 wlserver_10.3 下。(注：本书例子 Ant 使用了该路径)。



选择“All users”Start Menu folder，表明在 windows 开始菜单创建的文件夹供所有用户使用。

开始执行安装进程，安装完后的目录结构如下：

7.2 启动 weblogic examples 服务器

7.3 熟悉 weblogic 的管理控制台

7.4 关闭 weblogic examples 服务器

7.5 安装与删除企业应用

7.6 安装与删除 EJB 模块

7.7 安装与删除 Web 应用

这部分内容在《EJB3.0入门经典》中

7.8 安装和引用 JavaEE 共享库

这部分内容在《EJB3.0入门经典》中

7.9 使用 Ant 发布与卸载应用

在实际项目中，当应用处于开发阶段时，是需要不断地发布、测试及卸载的。如果你使用管理控制台向导来安装，启动，停止，卸载应用，那么等你的项目开发完，黄花菜都凉了。

这部分内容在《EJB3.0入门经典》中

7.10 创建 JDBC 数据源

首先确保数据库的 JDBC 驱动程序已经安装在 weblogic 上。目前，主流数据库的 JDBC 驱动程序已经随 WebLogic Server 附带，包括 BEA WebLogic Type 4 JDBC drivers for DB2、MySQL（支持版本：4.x/5. x）、Informix、MS SQL Server（支持版本：7.0/2000/2005）、Oracle（支持版本：9.0.1/9.2.0/10g）和 Sybase，这些驱动程序都不需要我们安装。

这部分内容在《EJB3.0入门经典》中

7.11 Weblogic 的 JNDI 名称

遵守 EJB3.0 规范编写的 EJB 是可以移植到任何实现了 JavaEE 规范的应用服务器中的。但在移植过程中，我们必须要面对这样一个现实情况：各个厂商使用的全局 JNDI 都有所不同，这意味着，你的远程客户端需要使用特定应用服务器的全局 JNDI 访问 EJB。

7.12 HelloWorld 例子

7.13 Entity Bean 应用例子

下面是一个完整的 Entity Bean 应用例子

7.14 Message-Driven Bean 应用例子

要进行 JMS 编程，我们应该首先创建消息到达的目标地址(Destination)，并为该 Destination 绑定一个 JNDI 名称。在程序中通过该 JNDI 名称获取 Destination，Destination 有两种类型：queue（队列）和 topic（主题）。

7.14.1 创建队列

7.14.2 创建主题

7.14.3 队列消息的发送与接收

在开始本例子前，你需要先创建队列，并设置队列绑定的 JNDI 名称为：queue/foshanshop。如果你不知道如何创建队列，请参考前面的内容。

7.14.4 主题消息的发送与接收

在开始本例子前，你需要先创建主题，并设置主题绑定的 JNDI 名称为：topic/chatTopic。如果你不知道如何创建主题，请参考前面的内容。

第八章　Struts+EJB3.0 和 JSF+EJB3.0 实战

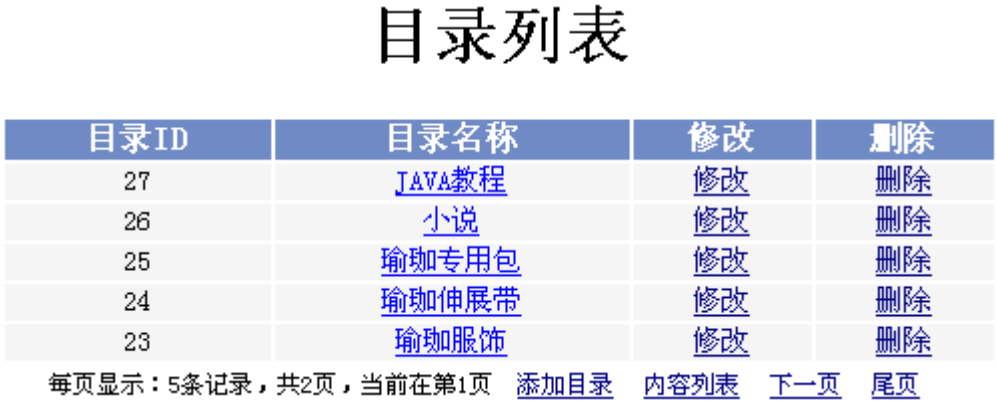
EJB 3.0 只专注于业务核心的开发，它并不关心客户端的类型及其使用的技术。你可以在 Web 中调用 EJB，也可以在 J2SE 或 J2ME 中调用 EJB。对于在 Web 中调用 EJB，你可以把代码直接写在 JSP 中，但这种做法已经被抛弃，原因是 java 代码与 html 代码相混合，不但可读性差，而且维护起来也很麻烦。目前，在 Web 开发上，我们可以引入第三方 web 框架，如 Struts，JSF，Spring MVC 等。其实使用 EJB 开发项目，对于选择 web 框架会显得不是那么重要，因为日后即使更换成另一种框架，对于我们的业务核心是不会构成任何影响的。即使如此，我们选择 web 框架仍需谨慎，因为谁也不想换来换去。目前，可推荐的 web 框架是 Struts 和 JSF，前者是这几年的主流框架，使用的开发人员比较多，后者属于 JavaEE 规范，有望成为 web 框架的主角。本章将为大家介绍 Struts+EJB 和 JSF+EJB。

本章引入的例子是一个简单的内容管理系统，主要完成新闻的编辑、修改和删除操作。例子使用了 JSF、Struts 和 JSTL，你要对这三种技术有初步的了解，如果不了解，请参考相关资料，本例子也可以作为你学习这三种技术的参照。例子的源代码在 JsfAndStrutsAndEJB3 文件夹，由于文件比较多，在学习本章时，你最好打开项目参照着学。

8.1 系统需求

内容管理系统应具有以下基本功能：

1. 系统必须可以运行在 Jboss 及 weblogic 两种平台下，并且提供 JSF 和 Struts 两种客户端实现，业务层代码不能因客户端实现不同而受影响。
2. 新闻含有 ID，标题，内容，来源和发布时间等属性，并且要实现分类。
3. 新闻的分类目录要含有类别 ID 和名称属性，对目录的管理要实现分页查看、添加、修改和删除。



4. 对新闻的管理要实现新闻浏览，新闻列表分页查看、添加、修改和删除。

内容列表

新闻标题	所在目录	创建日期	修改	删除
带你深入了解SQL Server 2008 特性	小说	2008-01-10	修改	删除
李彦宏揭秘：百度为什么推出自己的网站	JAVA教程	2008-01-10	修改	删除
2008十大应用热点猜想 SOA+BPM居首	JAVA教程	2008-01-10	修改	删除
Linux之父仍不采用GPLv3，并称不再更换许可证	小说	2008-01-10	修改	删除
《EJB3.0实例教程》改名为《EJB3.0入门经典》	JAVA教程	2008-01-10	修改	删除

每页显示：5条记录，共2页，当前在第1页 添加新闻 目录列表 下一页 尾页

新闻添加界面：

添加新闻：

标题 * ：

所属目录 * ： 小说

信息来源 ：

资讯内容 * ：

确 认

8.2 系统实现

这部分内容在《EJB3.0入门经典》中

8.2.1 建立实体模型

在以前，我们设计 java 系统时都比较热衷于数据库建模，也就是根据需求先建立数据库表和字段，然后通过 SQL

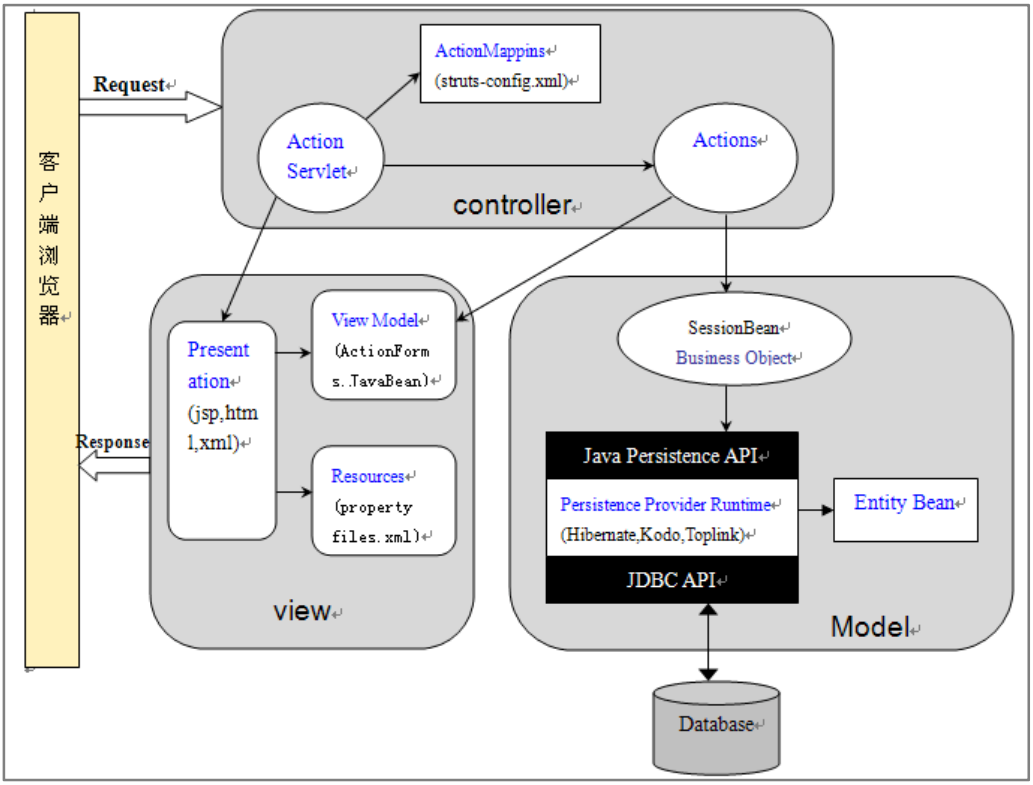
语句操作这些数据表。到目前，不少设计人员在设计实体 Bean 时，仍然摆脱不了这种影响，表现为先建立数据库表和字段，然后通过生成工具生成实体 Bean。尽管这种做法也实现了对数据库的面向对象操作，但在数据库设计的过程中很容易与面向对象设计思想产生冲突。所以建议大家应以实体 Bean 设计为出发点，至于数据库结构就由实体 Bean 自动生成吧。

8.2.2 建立持久化配置文件

8.2.3 建立会话 Bean

8.2.4 Struts 客户端

本节介绍的是 Struts 客户端应用，如果你对这种技术不感兴趣可以跳过阅读。使用 Struts 前，你需要往/WEB-INF/lib 目录下加入 Struts 依赖 jar，这些 jar 可以在本例子的 lib/struts-1.3.8 目录下获得。下面是 Struts 与 EJB3.0 的调用关系图：



8.2.5 JSF 客户端

本节介绍的是 JSF 客户端应用，如果你对这种技术不感兴趣可以跳过阅读。对于全面支持 JavaEE 规范的应用服务器，JSF 已经跟应用服务器整合在一起，因此，你不再需要往/WEB-INF/lib 目录下加入任何 JSF 依赖 jar。在 weblogic10.x 使用 JSF，你需要先安装 JSF1.2 共享库，并在 web 应用中通过 webloigc.xml 引用共享库，JSF1.2 共享库随 weblogic 附带，它在${wl.home}/common/deployable-libraries/jsf-1.2.war。关于如何安装和引用 JSF1.2 共享库，你可以参考前面章节“安装和引用 JavaEE 共享库”内容。注：如果你使用本例子 Ant 进行应用部署，deployToWeblogic 任务会自动帮你安装 JSF1.2 共享库。

8.2.6 创建 EAR 部署描述文件

8.2.7 使用 Ant 构建和部署程序

因为程序需要部署到 Jboss 及 weblogic 两种环境，所以使用 Ant 为这两种环境定义相关的打包及发布任务。

第九章　项目实用知识

9.1 使用了第三方类库的企业应用

当应用中的 web 模块和 ejb 模块使用了相同的第三方类库时，你可以把这些类库放进 EAR 文件内的 lib 目录中，这些类库将被 application 层次的 classloader 加载，可用于 ear 内的所有子模块（ejb，web 等），例如：

```
myapp.ear
|+ META-INF
    |+ applications.xml
|+ webapp.war
|+ ejbapp.jar
|+ lib
    |+ hibernate.jar（能被各个模块使用到的第三方类库）
```

EAR 中的 lib 目录是默认存放类库的目录，如果你不想使用这个目录，你也可以在 applications.xml 中自定义，如：
<library-directory>foshanshop</library-directory>
然后把类库 copy 到 EAR 中的 foshanshop 文件夹。

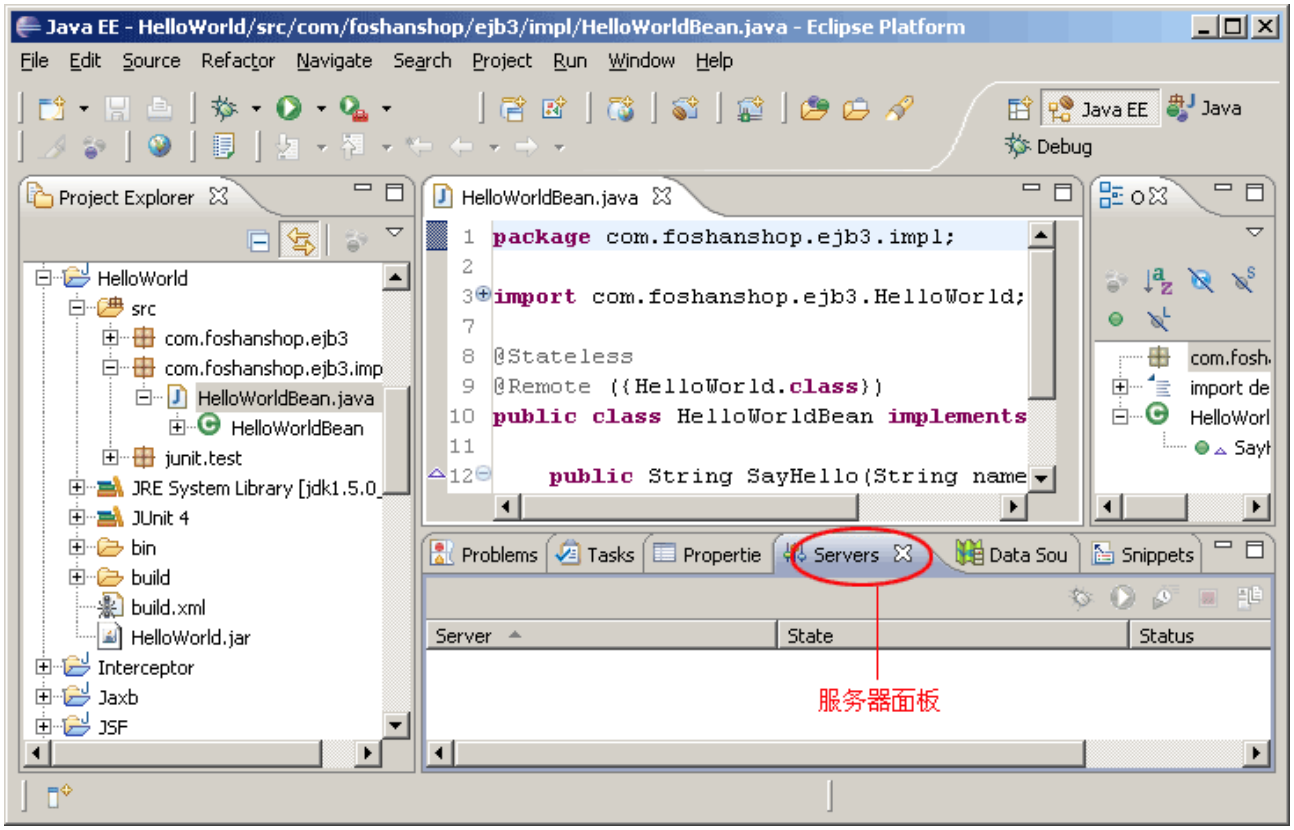
如果第三方类库需要被多个企业应用使用，那么你可以这样做：
在 jboss 中，你可以把第三方类库连同 EAR 文件一起 copy 到 Jboss 的 deploy 目录下。如果 jar 文件很多，你可以在 deploy 目录下建个文件夹，然后把 jar 文件放进去。
在 weblogic 中，你可以把第三方类库打包成 JavaEE 共享库（在 JAR 包的 META-INF/MANIFEST.MF 清单中需设置 Extension-Name，Specification-Version，Implementation-Version 等属性，详细请参考 weblogic 文档“创建共享 Java

EE 库和可选包”），然后在企业应用中使用 weblogic-application.xml 或 weblogic.xml 进行引用。

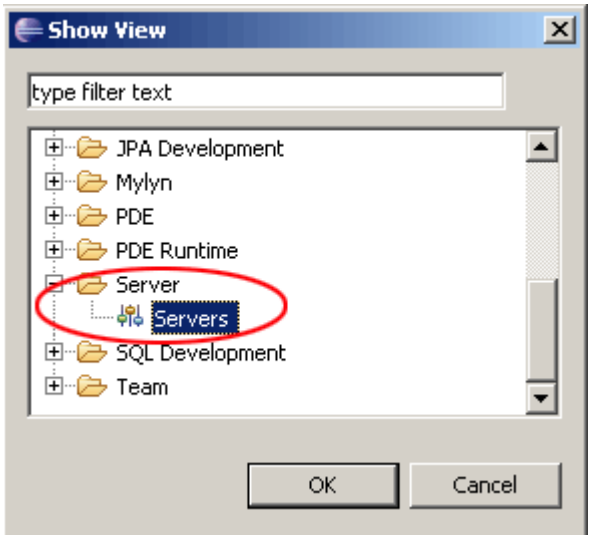
9.2 如何对 EJB3 进行调试

在对 EJB3 进行调试前，我们首先需要把 jboss 整合进 Eclipse 开发环境，操作步骤如下：

1> 在 Eclipse 开发界面中，检查是否存在“Servers”面板，如下图：



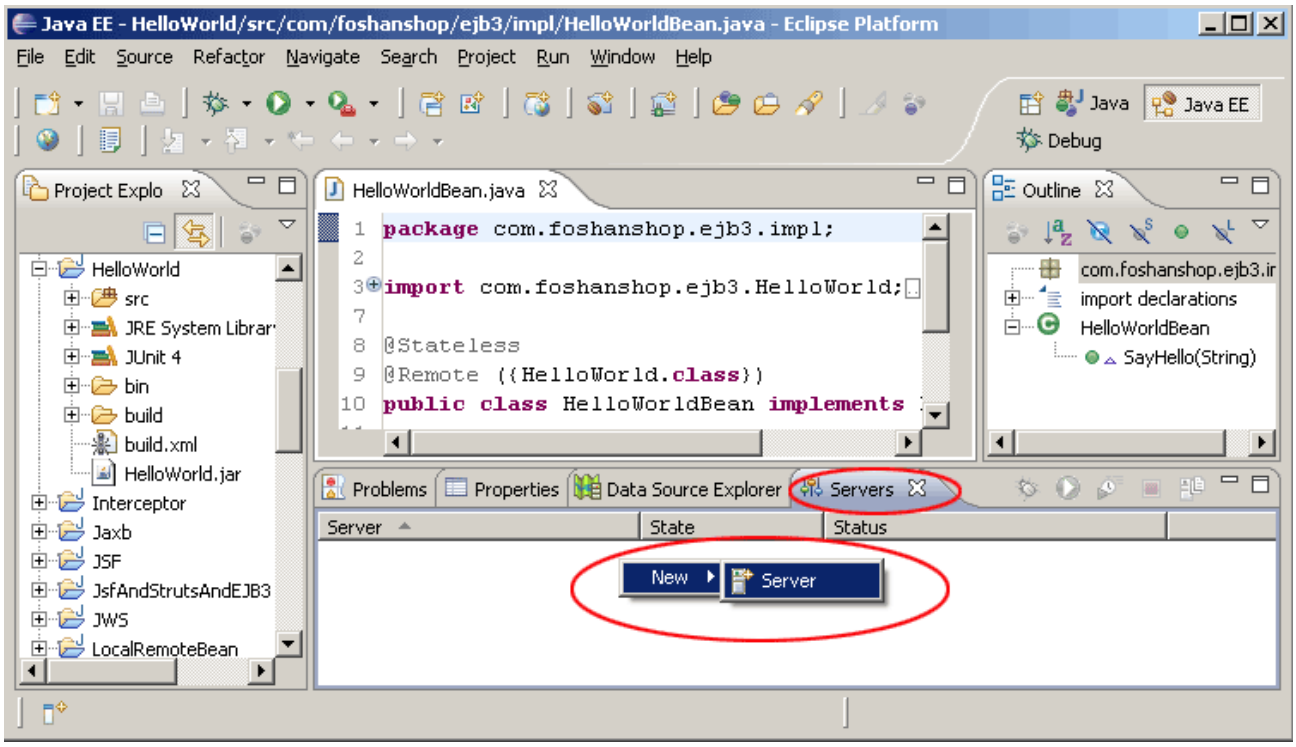
如果没有“Servers”面板，你可以在 Eclipse 菜单栏点击 “Window”-“Show View”-“Other”，在出现的“Show View”窗口中选择 Server 文件夹下的“Servers”，如下图：



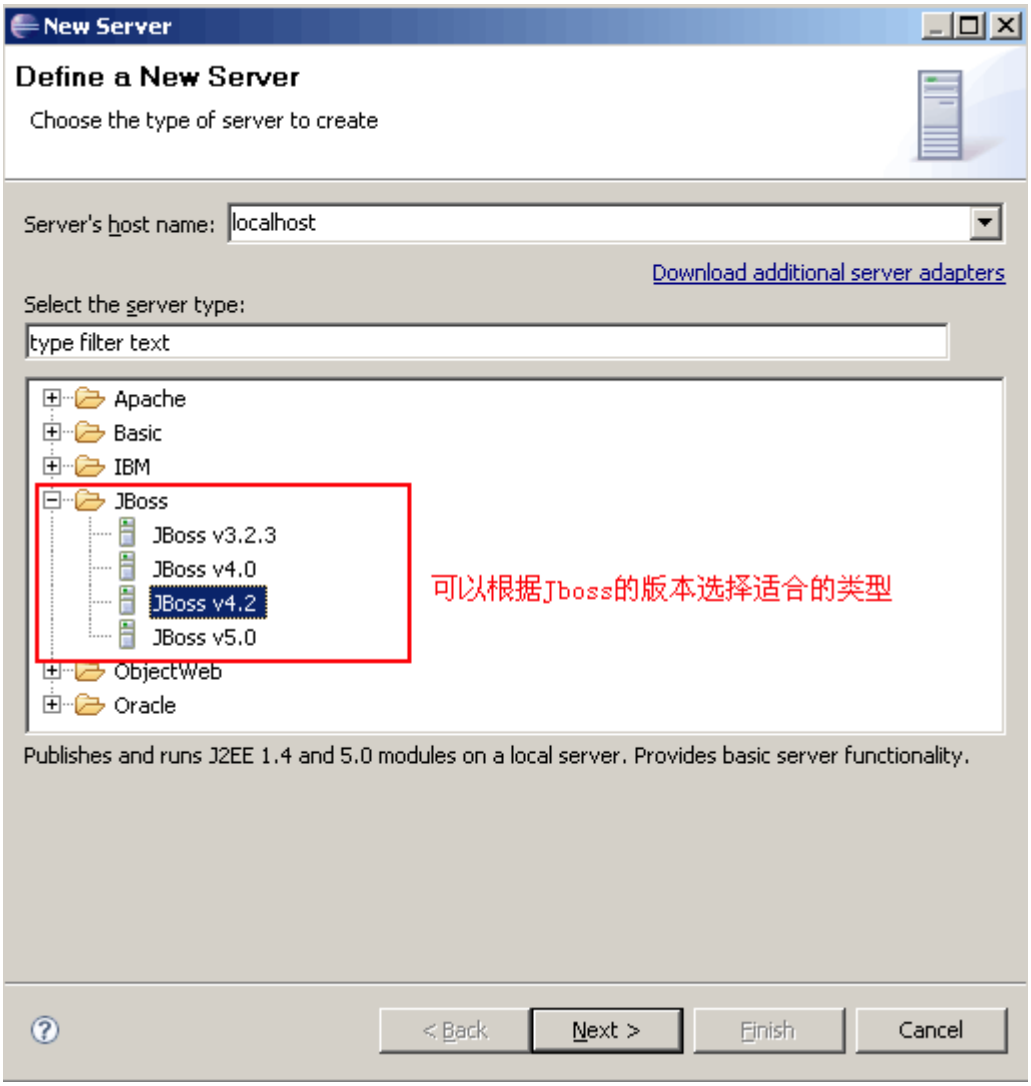
点击“OK”后，“Servers”面板将出现在 Eclipse 开发界面的控制区。

2> 接着我们需要添加一个 Jboss 服务器。在“Servers”面板上边白色区域点击鼠标右键，在出现的属性菜单中

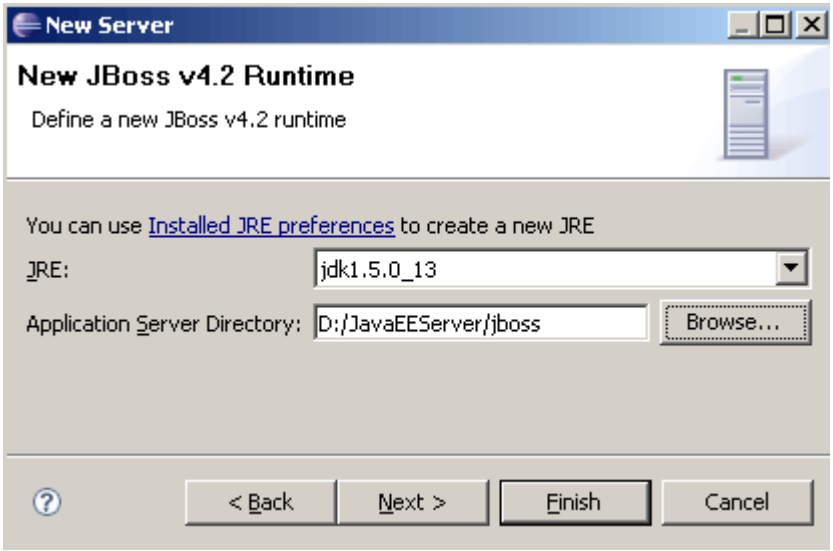
点击“New”-“Server”。如下图：



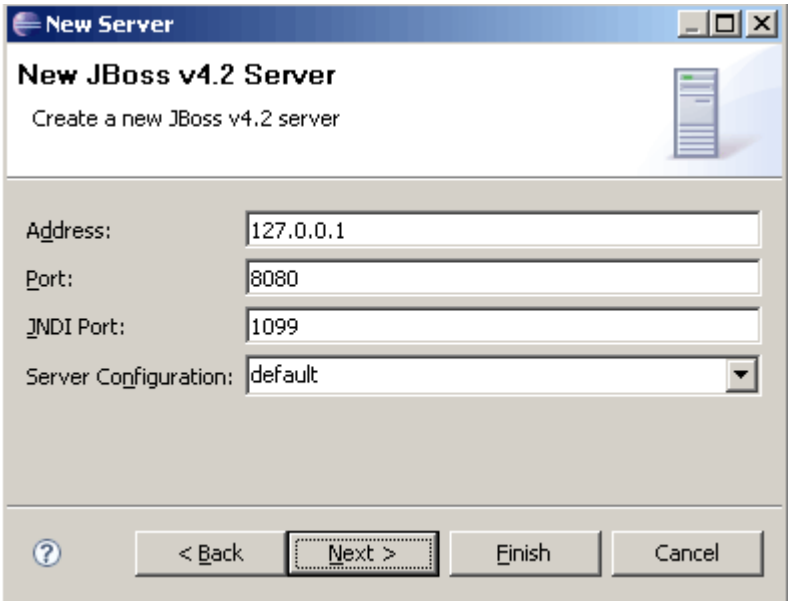
点击“Server”后出现下面窗口，在这个窗口中我们打开“Jboss”文件夹选择对应版本的 Jboss：



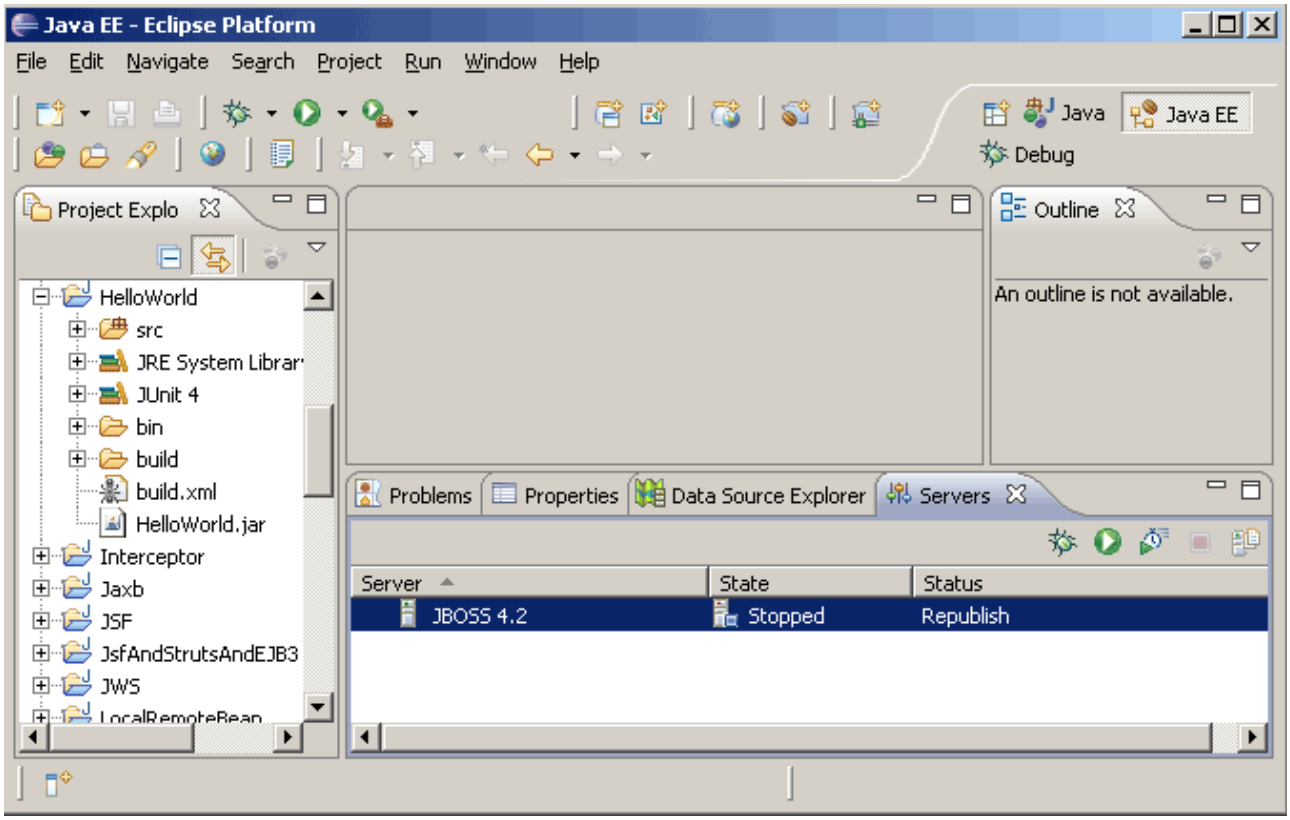
点击“Next”出现下面窗口，在这个窗口中我们选择使用的 JRE 和 Jboss 服务器所在目录（作者的 Jboss 安装在 D:\JavaEEServer\jboss）。



点“Next”出现下面窗口，在这个窗口中要求我们输入 Jboss 绑定的 IP、端口、JNDI 服务端口和配置名称。这里我们使用默认值，如果你的 EJB 要供远程客户端访问，IP 一栏需要输入公网或局域网 IP。

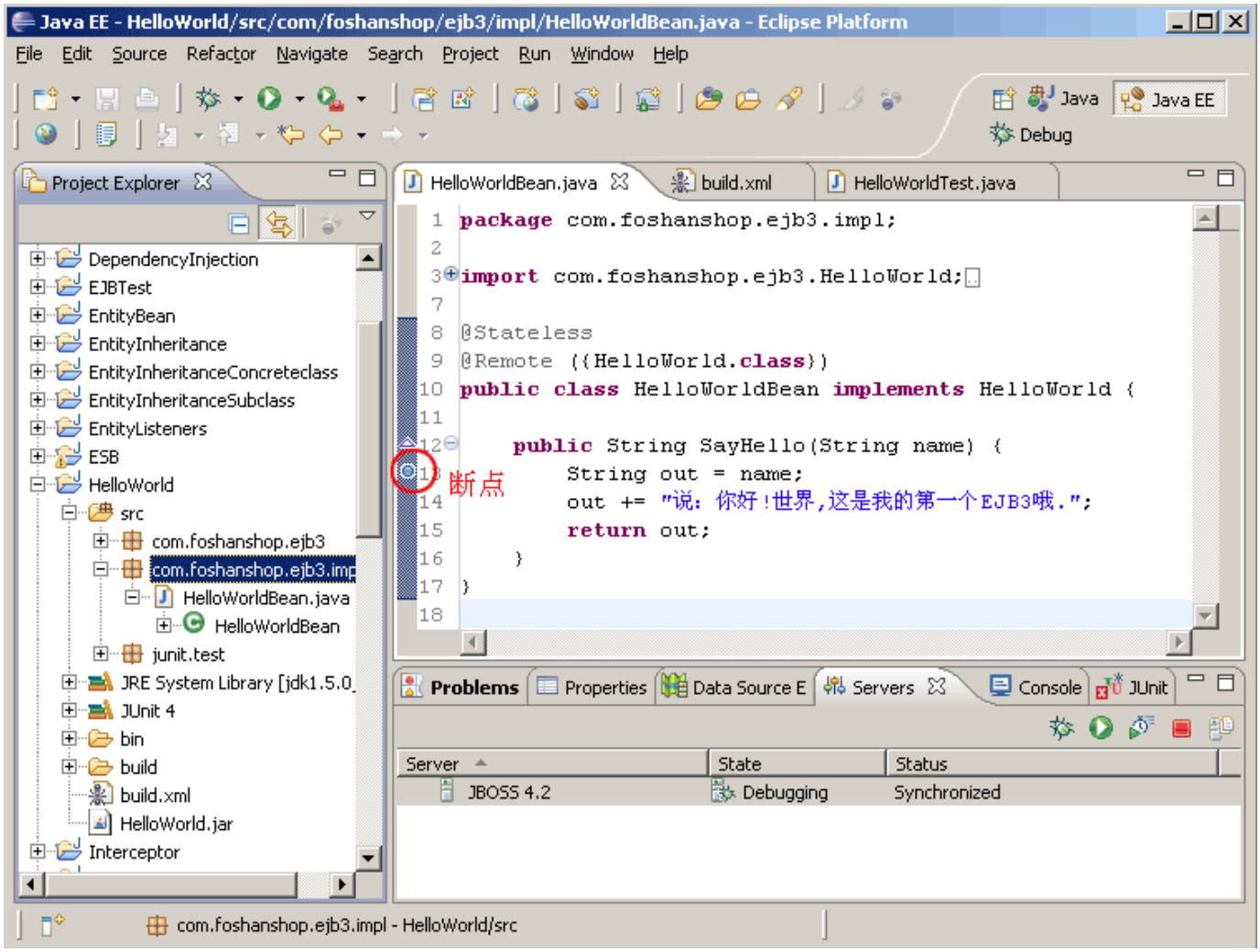


点击“Finish”结束添加服务器，新建的服务器将出现在“Servers”面板，如下图：

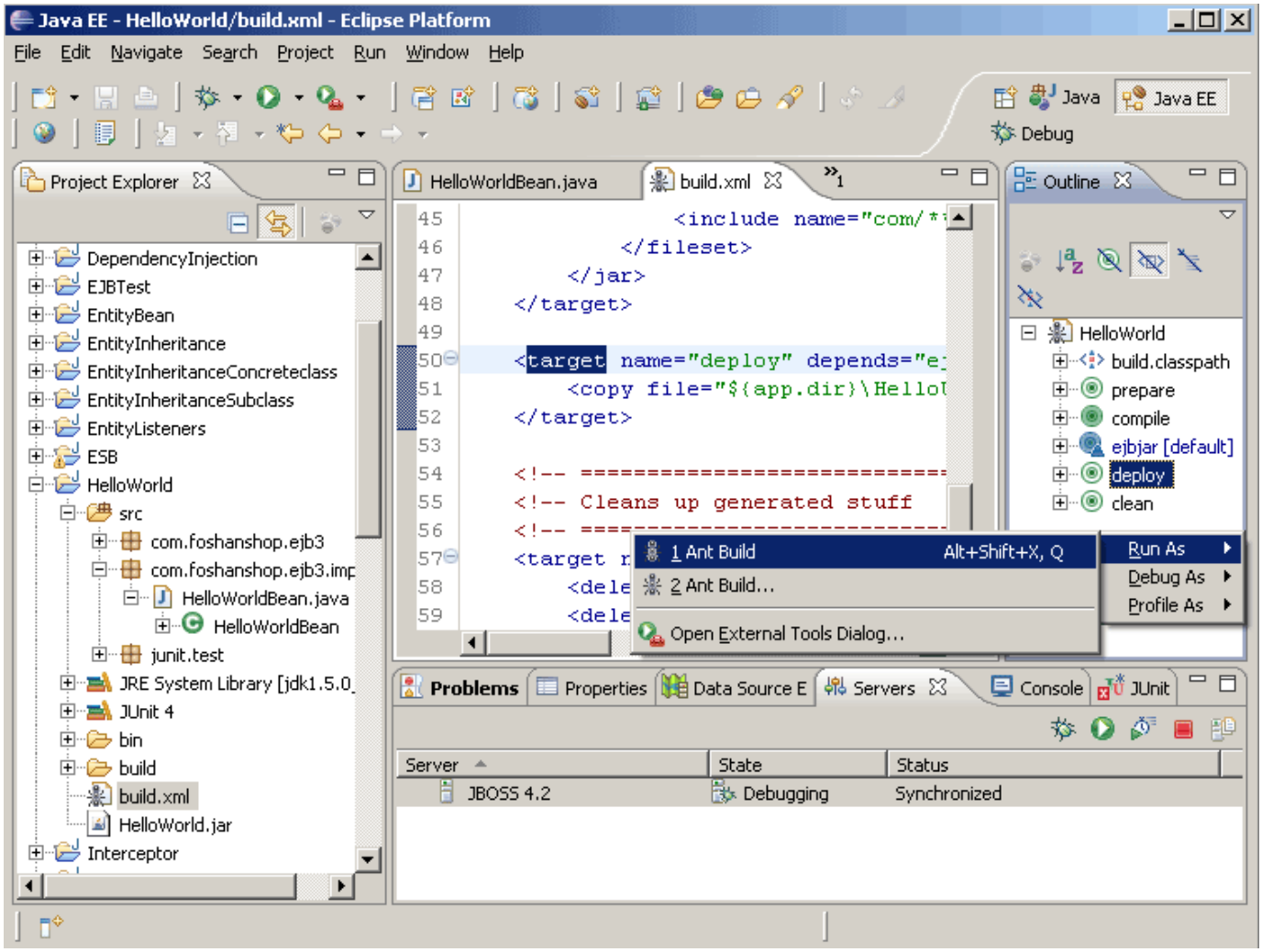


此时的服务器处于 Stopped 状态，要调试 EJB，你必须以 Debug 方式启动 Jboss，方法是：在选择的服务器上右击鼠标，在出现的属性菜单中点击“Debug”，或者你也可以点击“Servers”面板右下方的小甲虫图标。如果 Jboss 启动出错，首先检查命令行方式(run.bat)能否启动 Jboss，如果命令行下能正常启动 Jboss，那估计是你在建立 Jboss Server 的配置中出错，需要仔细检查。

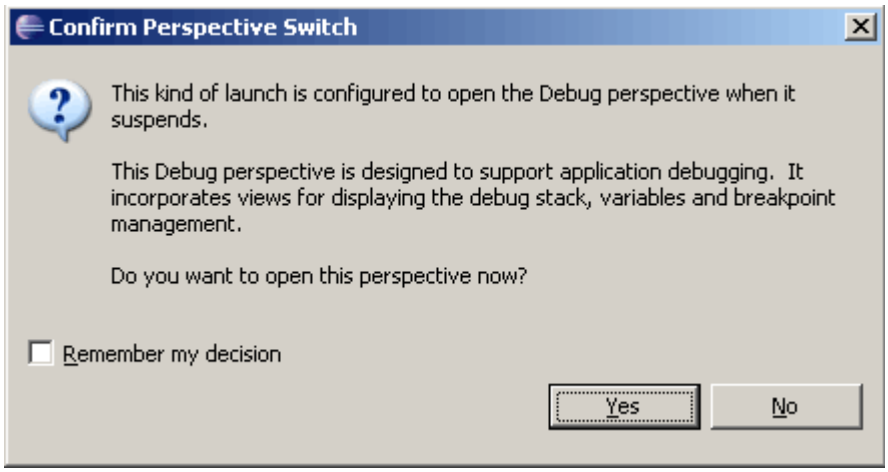
与 JBoss 的集成开发环境搭建好后，下面我们就以 HelloWorld 项目为例介绍 EJB3 的调试。因为 SayHello()方法代码不多，为了有代码可调试，我们在方法内部随便添加几行代码，并把断点设在代码的第一行，如下图：



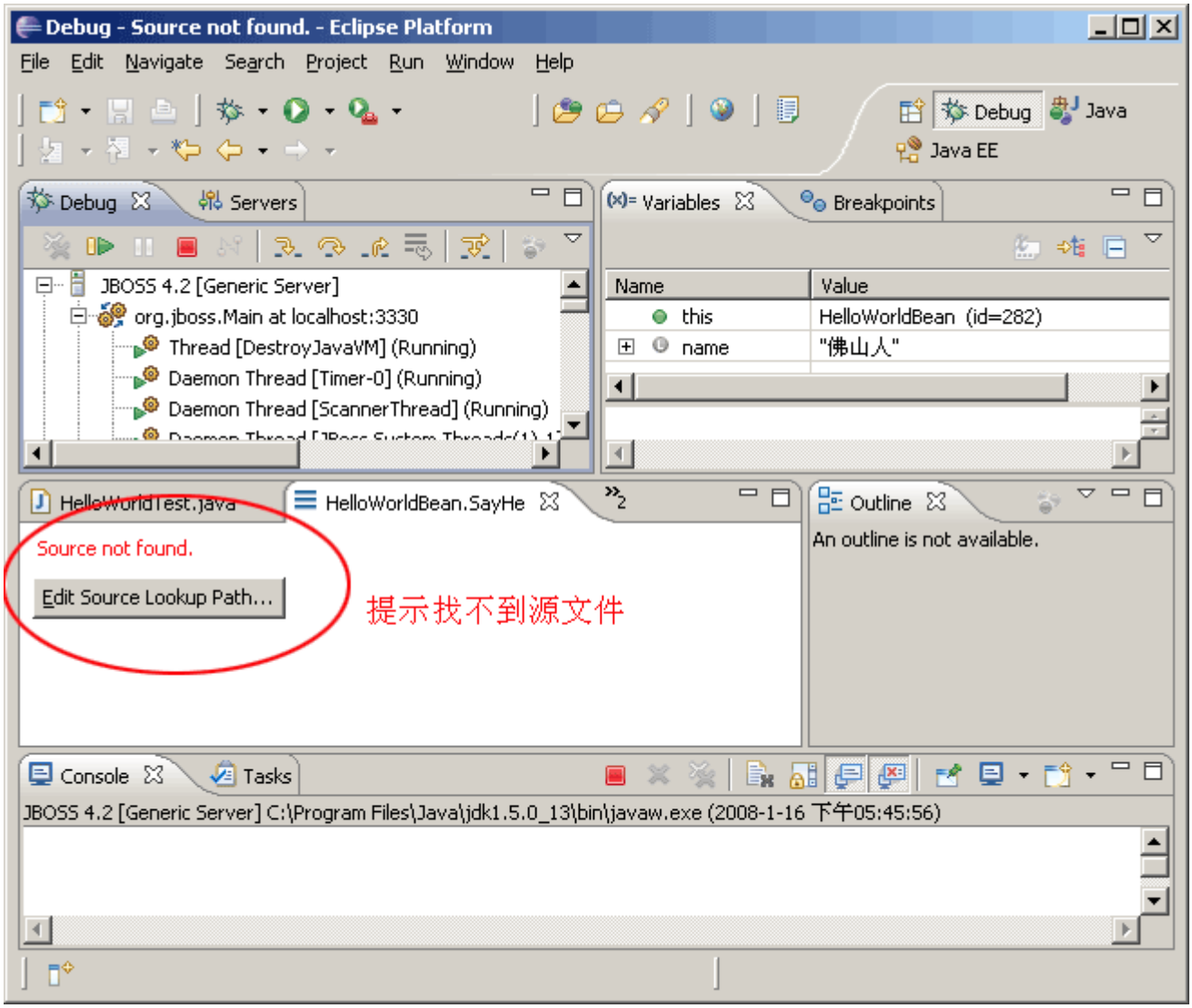
接着我们点击“Servers”面板右下方的小甲虫图标，以 Debug 方式启动 Jboss Server。启动完成后，我们打开项目的 build.xml 文件，在“Outline”视图中执行 Ant 的 deploy 任务，如下图：



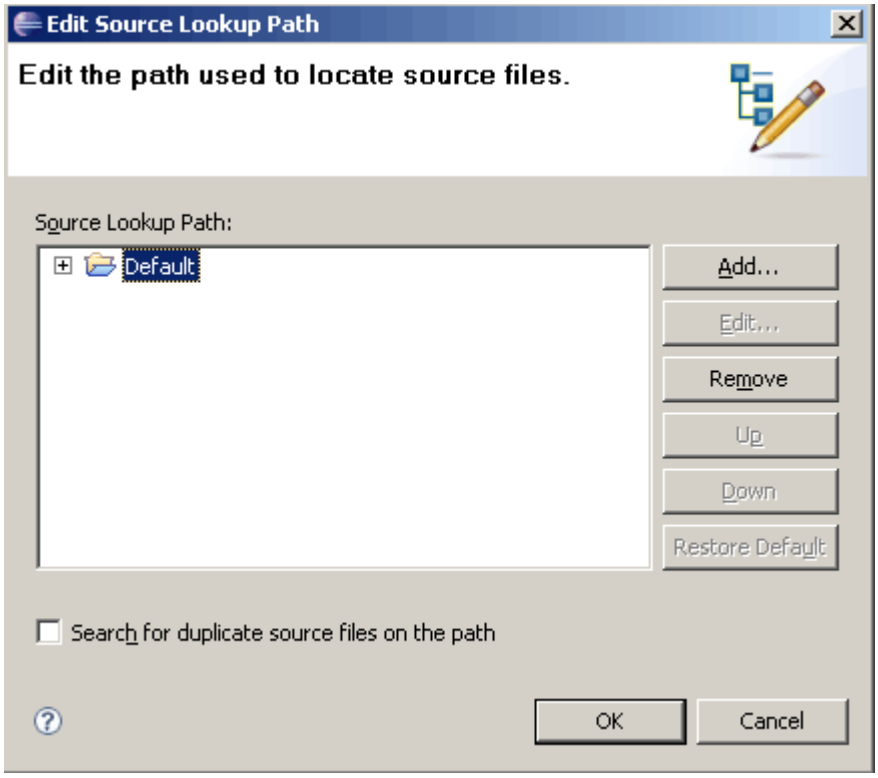
deploy 任务执行后，控制台将会输出 Ant 的任务信息及 Jboss 的发布信息，当 HelloWorld 发布完成后，我们执行 junit.test 包下的单元测试用例 HelloWorldTest.java，Eclipse 将会提示是否切换到调试透视图中，如下图，



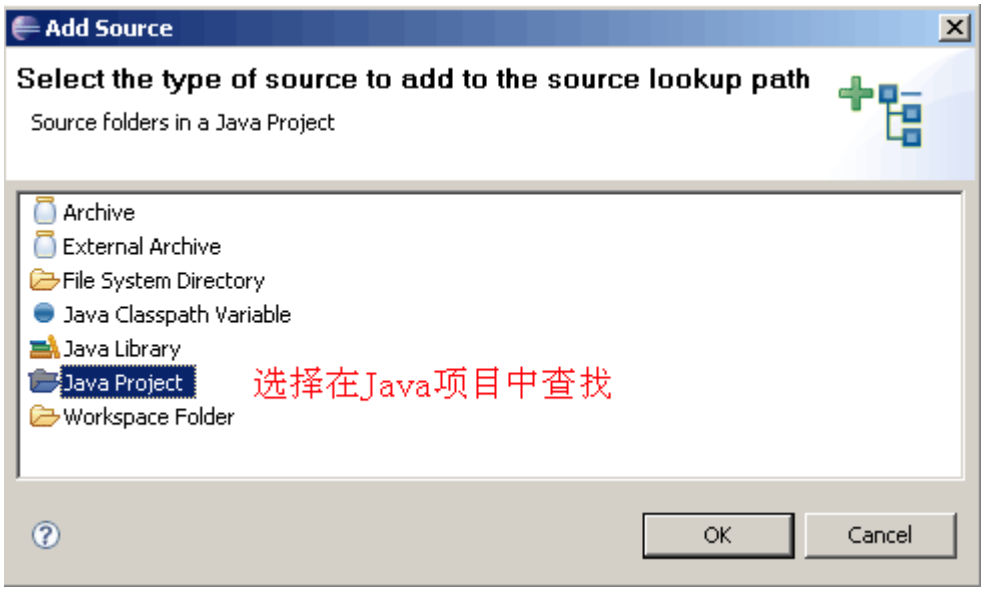
这里点“Yes”。如果你不想老出现这个对话框，可以勾选上“Remember my decision（记住我的决定）”。接着出现下面窗口，提示找不到 class 对应的 java 源文件。



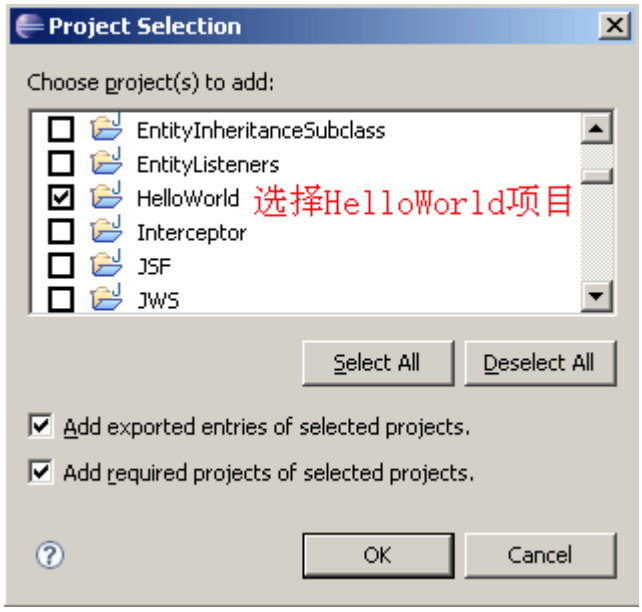
此时你可以在上图点击“Edit Source Lookup Path（编辑源查找路径）”，出现“编辑源查找路径”窗口，如下图：



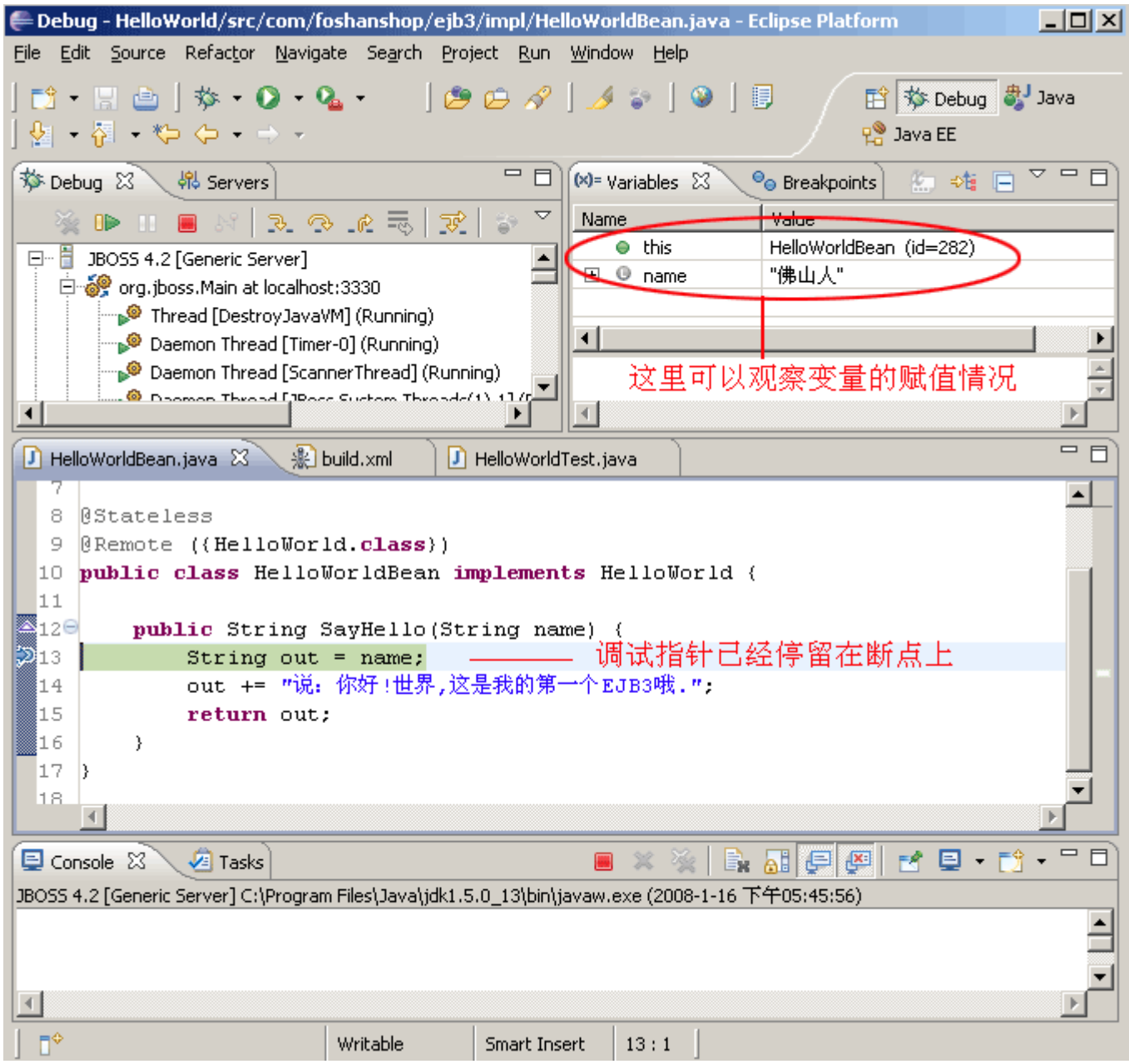
在上图中点击“Add”，出现“添加源代码”窗口，如下图：



上图我们选择在“Java Project”中寻找对应的源代码，点击“OK”出现下面窗口，如果你的源代码不在 eclipse 项目中，可以在“File System Directory（文件系统目录）”中寻找。



在上图，我们勾选 HelloWorld 项目，点击“OK”，此时将回到调试窗口，如下图：



在上图，调试指针已经落在了断点所在行，以后的调试方法就和普通的 java 应用程序一样，你可以使用单步跳入（F5），单步跳过（F6），单步返回（F7），运行至行（Ctrl+R）等调试方式，这里就不作详细介绍了，大家可以参考 Eclipse 关于调试方面的资料。

注意：调试 ejb3 时不需要反复重启 Jboss Server。

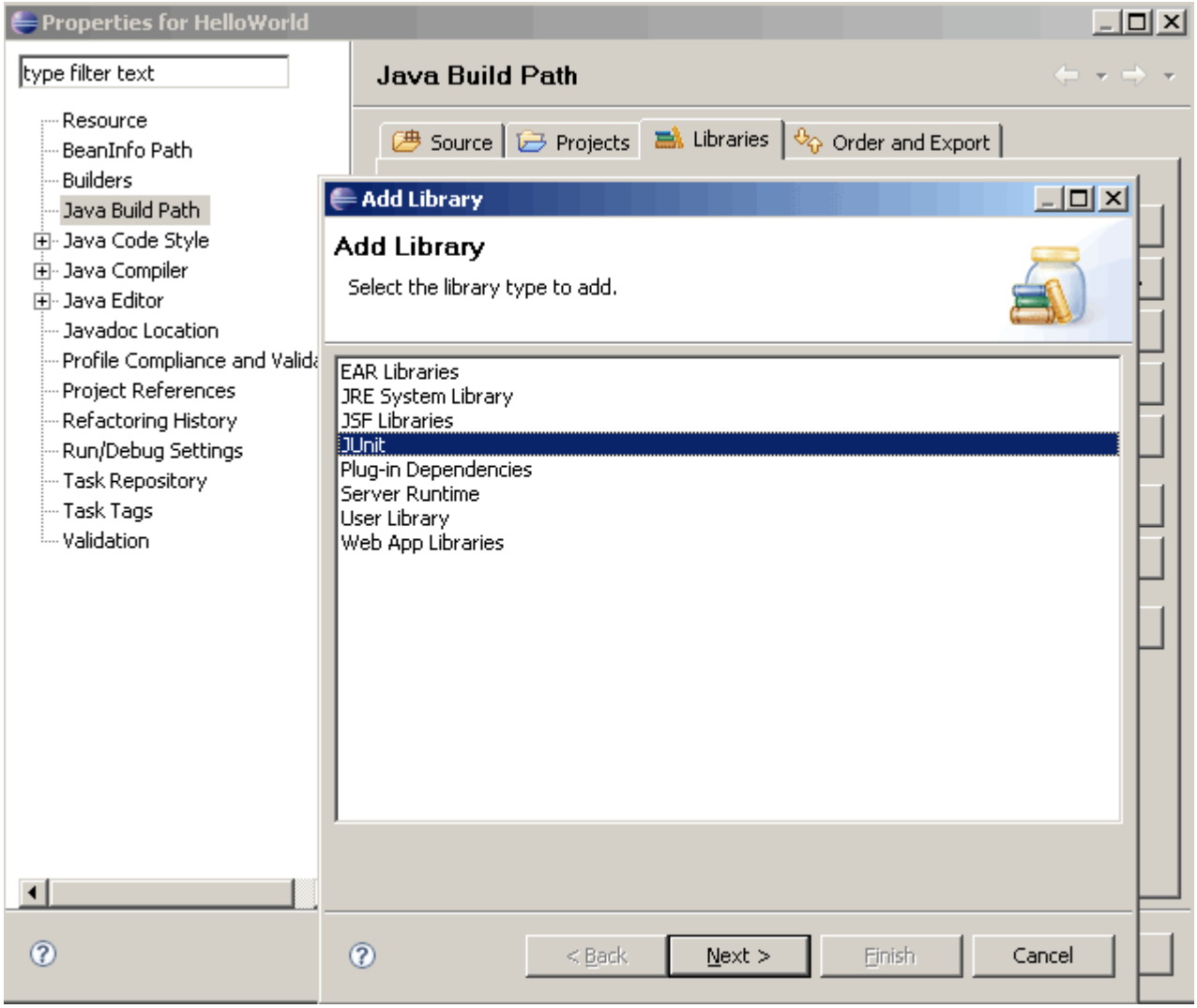
9.3 单元测试

在 EJB 开发的过程中，为保证软件的质量，我们需要不断地进行测试。测试的主要目的是检验程序能否正常运行，业务逻辑是否正确。测试业务逻辑时，我们可以给定输入参数，然后检验输出的结果是否符合期望值。

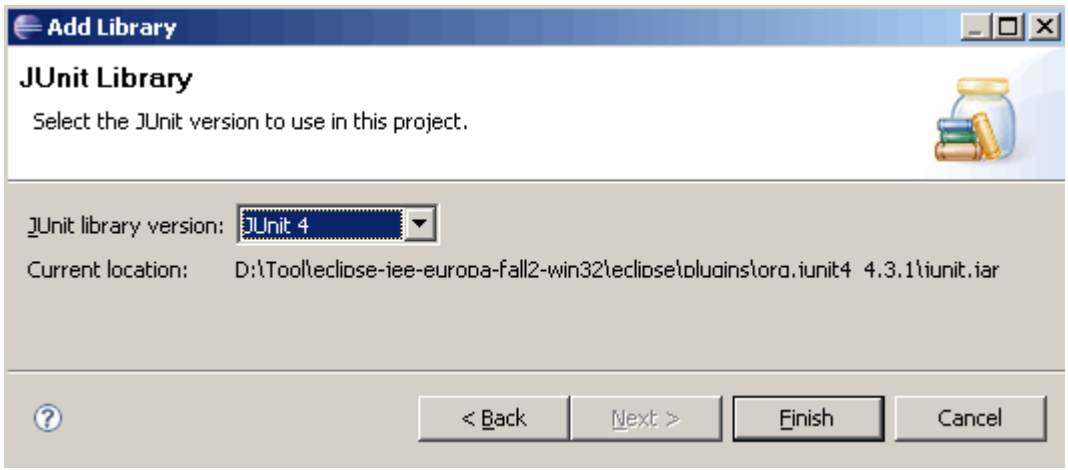
对 EJB 进行测试，我们一般采用单元测试工具 Junit。它是 EJB 开发过程中必不可少的工具，下面以 HelloWord 为例，介绍在 eclipse 中开发单元测试用例的步骤。

第一步：添加 Junit 依赖库，可以通过右击 HelloWord 项目，选择“Properties”，在属性窗口的左边选择“java Build

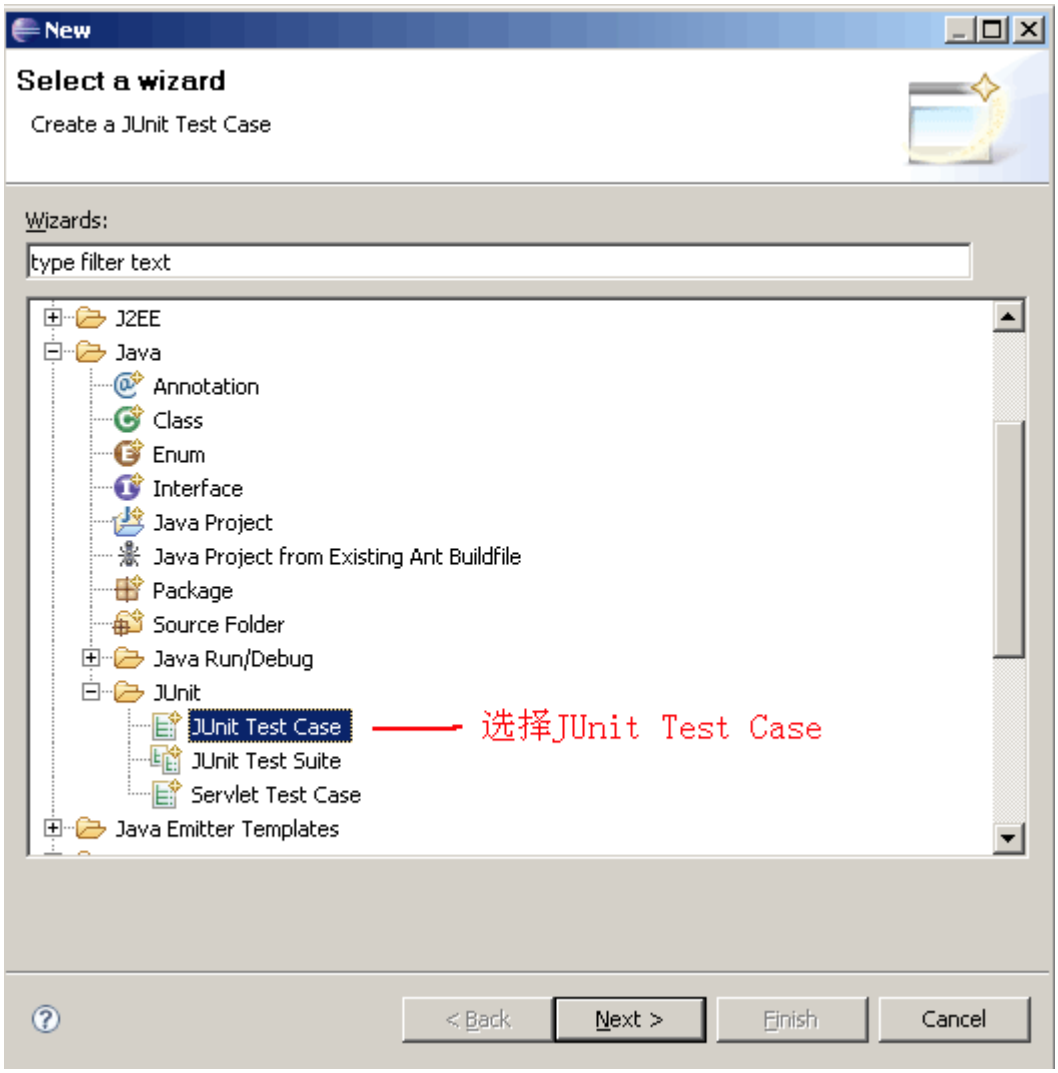
Path”，在右边框架中点击“Libraries”标签，接着点击“Add Library”，在出现的窗口中选择“JUnit”，如下图：



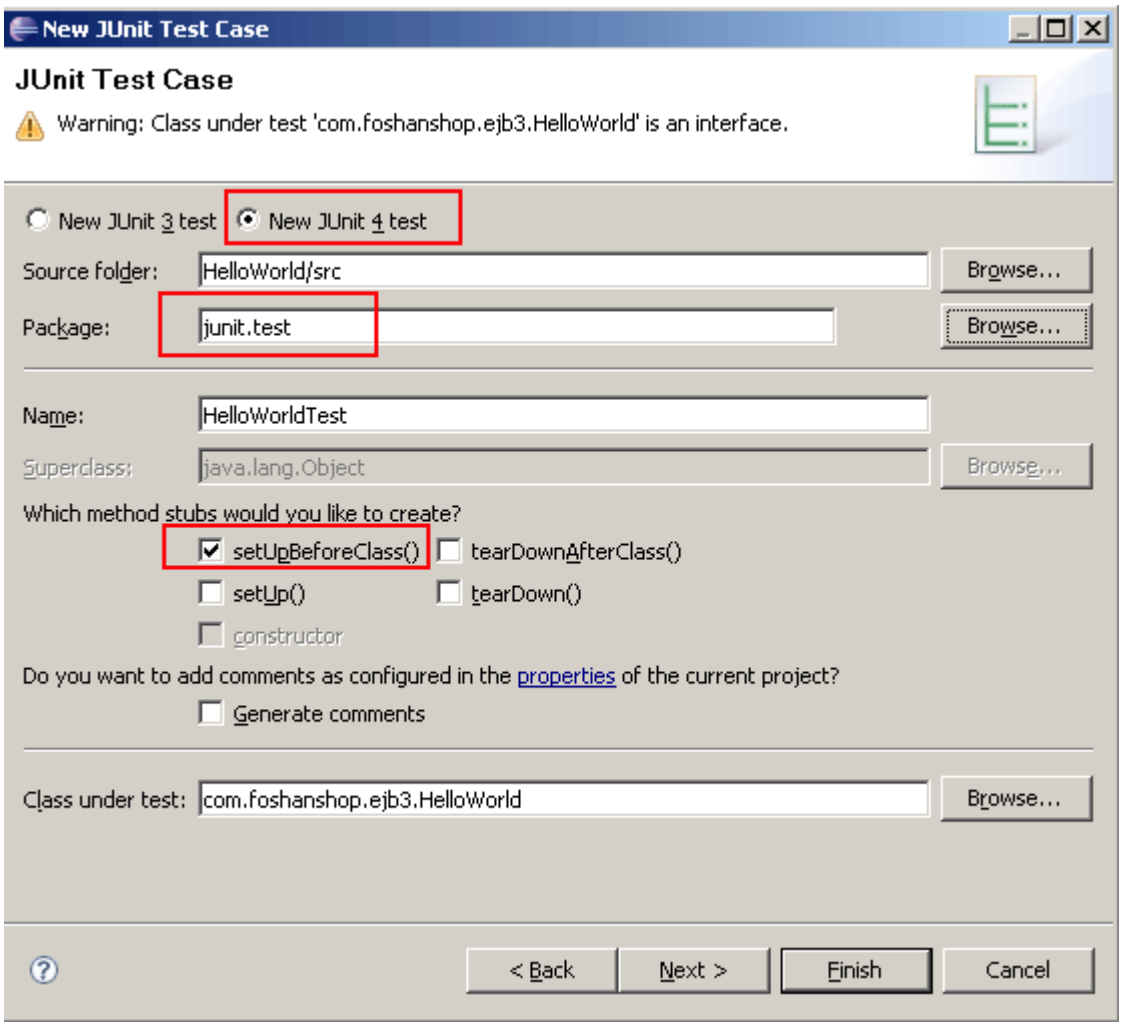
版本选择“JUnit 4”。如下图：



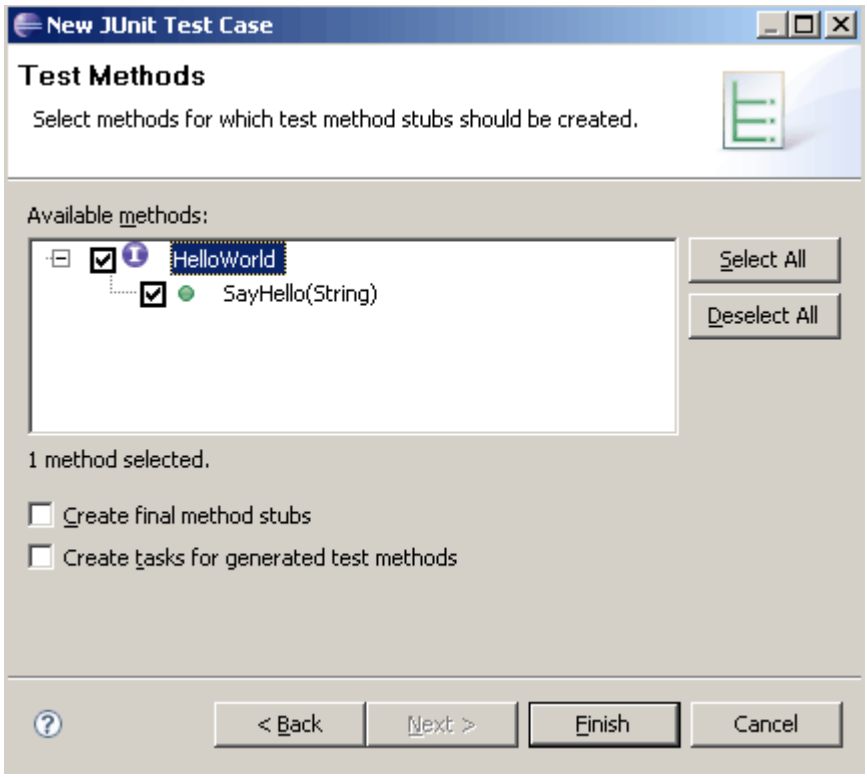
第二步：在需要测试的 EJB 接口（本例是 HelloWorld.java）上点击右键，在出现的属性菜单中点击“New”-“Other”，如下图：



依次展开“Java”文件夹、“JUnit”文件夹，选择“Junit Test Case”，点击“Next”，出现下面窗口：



点击“Next”，在出现的窗口中选择需要单元测试的方法：



点击“Finish”，生成的代码如下：

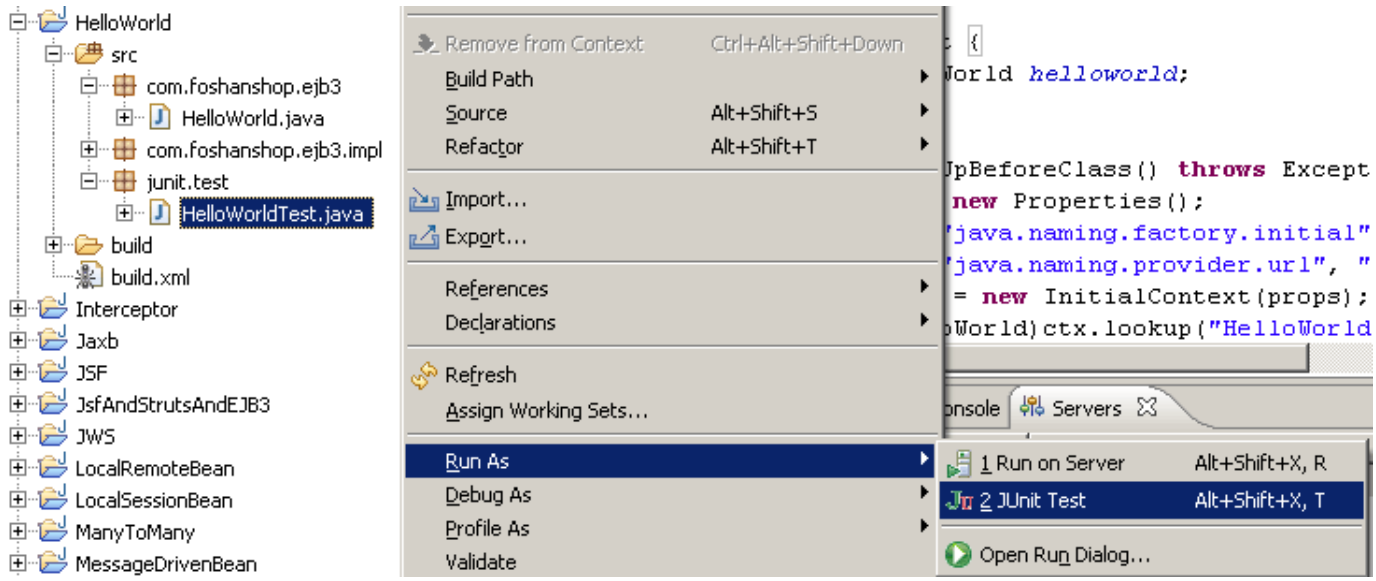
```
package junit.test;
import static org.junit.Assert.*;
import org.junit.BeforeClass;
import org.junit.Test;

public class HelloWorldTest {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @Test
    public void testSayHello() {
        fail("尚未实现");
    }
}
```

在生成的测试用例里面添加以下代码：

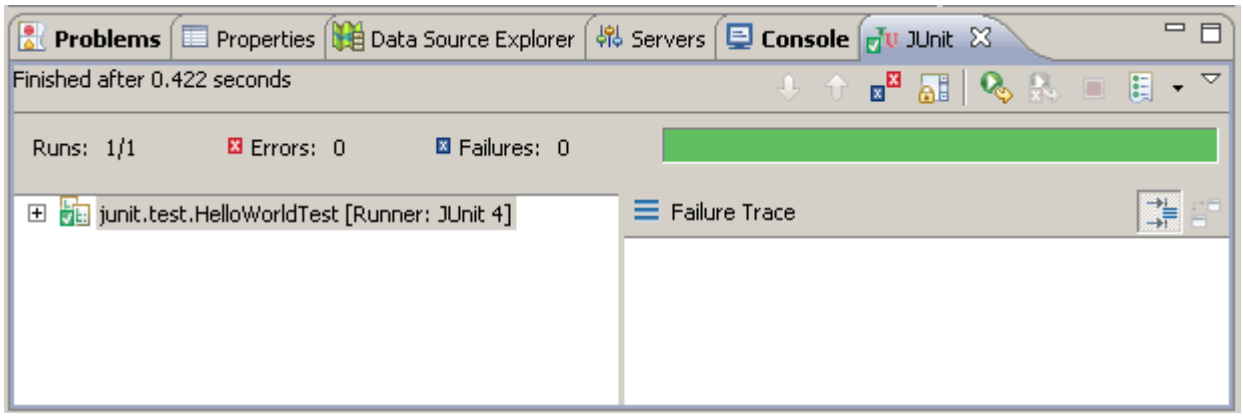
```
package junit.test;
import static org.junit.Assert.*;
import java.util.Properties;
import javax.naming.InitialContext;
import org.junit.BeforeClass;
import org.junit.Test;
import com.foshanshop.ejb3.HelloWorld;

public class HelloWorldTest {
    protected static HelloWorld helloworld;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        Properties props = new Properties();
        props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
        props.setProperty("java.naming.provider.url", "localhost:1099");
        InitialContext ctx = new InitialContext(props);
        helloworld = (HelloWorld)ctx.lookup("HelloWorldBean/remote");
    }

    @Test
    public void testSayHello() {
       String result = helloworld.SayHello("佛山人");
       assertEquals("佛山人说：你好!世界,这是我的第一个EJB3哦.", result);
    }
```

```
}
```

要运行 HelloWorldTest 测试用例，可以右键点击该文件，在属性菜单中点击“Run As”—“Junit Test”，如下图：



测试用例运行的结果如下：



绿色的进度条提示我们，测试通过。

9.4 在独立的 Web 服务器 或 J2SE 中调用 EJB

在正式的生产环境下，大部分调用 EJB 的客户端可能是单独的 Tomcat、Resin 或 J2SE。在这些客户端环境下调用远程服务器中的 EJB，应经过以下步骤：

1. 检查 JavaEE 应用服务器运行时是否已经绑定到某个可供外部访问的 IP 地址。Jboss 默认绑定到 127.0.0.1 地址，因此它不对外开放服务，只允许在本机访问 EJB。为了能让远程客户端访问 EJB，我们必须让 Jboss 启动时绑定到某个局域网 IP 或公网 IP。在 dos 命令行下进入[jboss 安装目录]\bin，输入：run –b 192.168.1.99，Jboss 启动时将绑定到 192.168.1.99 地址。
2. 远程客户端要跟应用服务器通信，需要使用应用服务器提供的客户端 jar。并且必须保证客户端使用的 jar 与服务器的版本一致，如：你不能使用 jboss4.2.1GA 的客户端 jar 去访问 jboss4.2.2GA。Jboss 提供的客户端 jar 在[Jboss 安装目录]/client 目录，Weblogic 提供的客户端 Jar 是[weblogic 安装目录]server\lib\weblogic.jar。你需要把这些 jar 设置在应用的类路径下，如果是 web 应用，你可以把这些 jar 拷贝到 WEB-INF/lib。

注意：[Jboss 安装目录]/client 目录下的 jar 比较多，其实我们只用到其中的一部分。你可以视应用的情况，把需要的 jar 挑出来。如果你觉得麻烦，可以把 jar 全部拷贝到 WEB-INF/lib。在学习时，为了避免出错，作者建议你把全部 jar 拷贝到 WEB-INF/lib。

3. 把 EJB 的接口类和实体 bean 类（如果使用了实体 bean 的话）放在应用的类路径下（最好是把类单独打成一个 jar）。对于 web 应用，你可以把这些类拷贝到/WEB-INF/classes，如果类打成了 jar 文件，可以放在/WEB-INF/lib。

   注意：EJB 的 Bean 类不需要放入客户端。
4. 客户端只能通过 EJB 的远程接口访问 EJB。
5. 客户端访问 EJB 时必须初始化为应用服务器的上下文，代码如下：

```
//Jboss
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
props.setProperty(Context.PROVIDER_URL, "192.168.1.99:1099");
InitialContext ctx = new InitialContext(props);
//weblogic
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
props.setProperty(Context.PROVIDER_URL, "t3://192.168.1.99:7001");
InitialContext ctx = new InitialContext(props);
```

在实际使用中，你应该对上面这段代码进行封装，而不应该出现在每次 JNDI 查找中。另外你也可以通过在类路径下放置 jndi.properties 来设置上下文。

对于 weblogic10.3 以上版本，由于 weblogic.jar 需要在 JDK1.6 下运行，所以你的客户端也必须运行在 JDK1.6。如果客户端使用了 JavaEE 的注释或类，你还需要在类路径下放置 JavaEE 的 jar，这些 jar 可以在本书源代码的 lib\javaee 下得到。

9.4.1 Struts+Spring+EJB3.0

当在独立的 Web 服务器或 J2SE 中调用 EJB 时，我们需要考虑两个问题：

1. 由于各个应用服务器的 JNDI 名称有所不同，因此客户端代码应该隔离对 JNDI 的访问。
2. 如果每次访问 EJB 都进行 lookup()查找，将会严重影响运行性能，因为应用服务器接到请求后需要在 JNDI 树中查找对象，然后把存根对象序列化后通过网络发送回客户端，频繁调用 lookup()方法将会有比较大的性能损耗。如果你没有采取任何方案，软件的运行性能将有一半耗在这上面，有很多开发人员叫喊 EJB 太慢，原因很大部分出在这里。实际上，如果 EJB 存根已经存在，我们就不需要调用 lookup()方法。因此我们在第一次 lookup()某个 EJB 时就应缓存 EJB 存根。

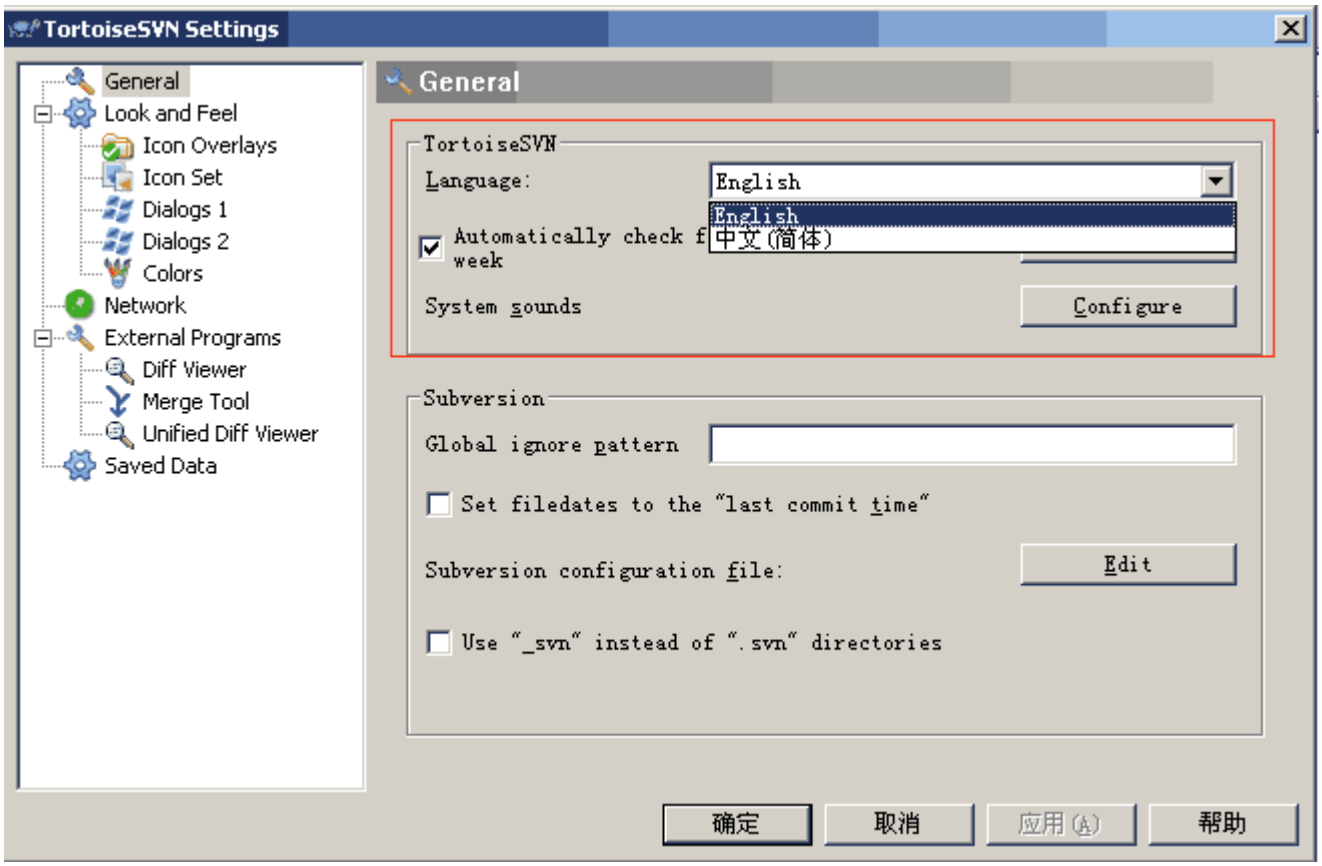
   注意：当你的 EJB 接口发生变化时，缓存的 EJB 存根将失效，此时应该调用 lookup()方法重新获取存根！

这部分内容在《EJB3.0入门经典》中

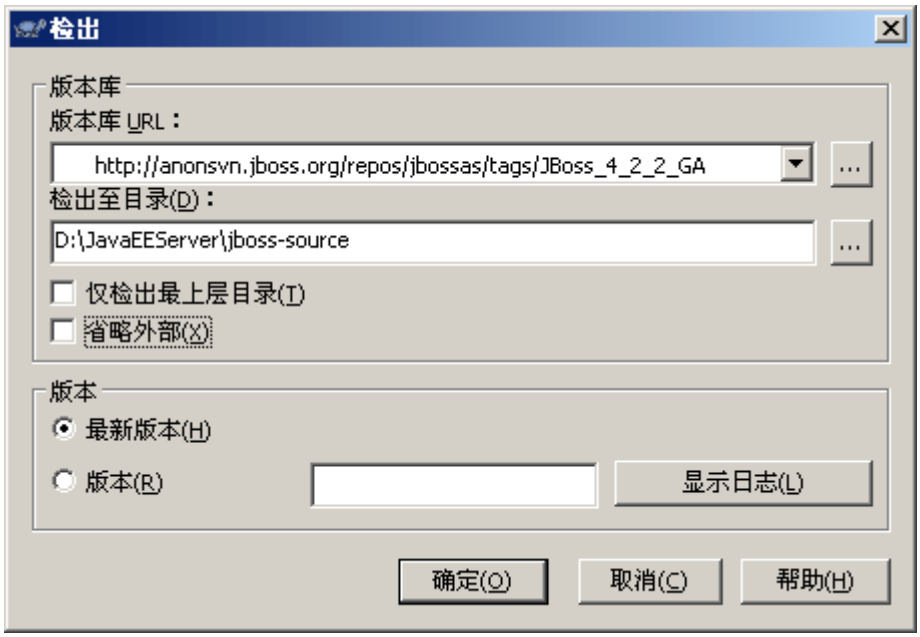
9.1 如何获取最新的 JBOSS 版本

有时候最新的 JBOSS 版本在官方网站上发布有些慢，如果你着急需要，可以从 Jboss 的版本库中获取。获取的方法很简单，首先我们安装一个版本控制软件 TortoiseSVN，下载路径：
http://superb-west.dl.sourceforge.net/sourceforge/tortoisesvn/TortoiseSVN-1.4.7.11792-win32-svn-1.4.6.msi
该软件也可以在配套光盘的“软件”目录中获取。双击文件进行安装，安装完后需要重启机器，软件的菜单通过右键属性菜单进入。

喜欢用中文的朋友还可以再下载中文语言包
http://superb-west.dl.sourceforge.net/sourceforge/tortoisesvn/LanguagePack-1.4.7.11792-win32-zh_CN.exe
该软件也可以在配套光盘的“软件”目录中获取。安装完后需要设置语言选项才可以显示中文，方法是：随便在桌面空白处右击鼠标，在出现的属性菜单中点击“TortoiseSVN”-“Settins”，在出现的属性窗口中选择“简体中文”，如下图所示：



下面我们就开始从 Jboss 版本库中获取最新版的 Jboss 源文件。
如我们要获取 JBoss_4_2_2_GA 版本的源文件。它的版本库路径为：http://anonsvn.jboss.org/repos/jbossas/tags/JBoss_4_2_2_GA
首先我们建一个用于存放 Jboss 源文件的文件夹，然后在文件夹上右击鼠标，在出现的属性菜单中选择“SVN Checkout”(中文：SVN 检出)，在出现的对话框中填入 http://anonsvn.jboss.org/repos/jbossas/tags/JBoss_4_2_2_GA，如下图：



点击“确定”后，程序就开始从版本库中获取源文件。文件大小有 73.37M 多，你就慢慢下载吧。

下载完后，进入 build 文件夹，执行 build.bat 批处理命令，程序就开始进行编绎，这过程需要 20 多分钟。成功执行完后，在 build/output 文件夹生成了 jboss-4.2.2.GA 发行版。把 jboss-4.2.2.GA 拷贝到某个安装目录中就可以直接使用了。