



最新
EJB3.0
实例教程
——企业应用开发核心技术

北京传智播客教育科技有限公司

——高级软件人才培训基地

<http://www.itcast.cn>

黎活明 著

| | |
|--|-----------|
| 第一章 前言 | 5 |
| 1.1 本教程适合人群 | 5 |
| 1.2 联系作者 | 5 |
| 1.3 《EJB3.0 实例教程》官方 MSN 群 | 5 |
| 1.4 版权声明 | 6 |
| 第二章 运行环境配置 | 6 |
| 2.1 下载与安装 | 6 |
| 2.2 运行一个 EJB3 例子 | 8 |
| 2.3 熟悉 JBoss 的目录结构 | 8 |
| 2.4 JBoss 中的部署 | 9 |
| 2.5 在单独的 TOMCAT 或 J2SE 中调用 EJB | 9 |
| 2.6 发布在 JBOSS 中的 WEB 应用调用 EJB | 10 |
| 2.7 如何获取最新的内嵌 EJB3 的 JBOSS 版本 | 10 |
| 第三章 基础知识学习 | 12 |
| 3.1 ENTERPRICE JAVA BEANS(EJB)的概念 | 12 |
| 3.2 一个基于 STRUTS+EJB3.0 应用的体系结构图 | 14 |
| 3.3 如何进行 EJB 打包 | 14 |
| 3.4 如何进行 WEB 应用打包 | 15 |
| 3.5 如何进行企业应用打包 | 16 |
| 3.6 使用了第三方类库的 EJB 如何打包 | 17 |
| 3.7 共用了第三方类库的 J2EE 应用如何打包 | 18 |
| 3.8 如何恢复本书配套例子的开发环境 | 19 |
| 3.9 如何对 EJB3 进行调试 | 24 |
| 3.10 单元测试 | 32 |
| 第四章 会话 BEAN(SESSION BEAN) | 36 |
| 4.1 STATELESS SESSION BEANS (无状态 BEAN) 开发 | 38 |
| 4.1.1 开发只存在 Remote 接口的无状态 Session Bean | 39 |
| 4.1.2 开发只存在 Local 接口的无状态 Session Bean | 42 |
| 4.1.3 开发存在 Remote 与 Local 接口的无状态 Session Bean | 43 |
| 4.2 STATEFUL SESSION BEANS (有状态 BEAN) 开发 | 46 |
| 4.3 STATELESS SESSION BEAN 与 STATEFUL SESSION BEAN 的区别 | 48 |
| 4.4 如何改变 SESSION BEAN 的 JNDI 名称 | 48 |
| 4.5 SESSION BEAN 的生命周期 | 50 |
| 4.6 拦截器(INTERCEPTOR) | 52 |
| 4.7 依赖注入(DEPENDENCY INJECTION) | 56 |
| 4.8 定时服务(TIMER SERVICE) | 60 |
| 4.9 安全服务(SEcurity SERVICE) | 62 |
| 4.9.1 自定义安全域 | 72 |
| 第五章 JMS (JAVA MESSAGE SERVICE) | 74 |
| 5.1 消息驱动 BEAN (MESSAGE DRIVEN BEAN) | 77 |
| 5.1.1 Queue 消息的发送与接收(PTP 消息传递模型) | 77 |

| | |
|---|-----------|
| 5.1.2 Topic 消息的发送与接收(Pub/sub 消息传递模型) | 82 |
| 第六章 实体 BEAN(ENTITY BEAN) | 87 |
| 6.1 持久化 PERSISTENCE.XML 配置文件 | 88 |
| 6.2 JBoss 数据源的配置 | 88 |
| 6.2.1 MySql 数据源的配置 | 89 |
| 6.2.2 Ms Sql Server2000 数据源的配置 | 89 |
| 6.2.3 Oracle9i 数据源的配置 | 90 |
| 6.3 实体 BEAN 发布前的准备工作 | 91 |
| 6.4 单表映射的实体 BEAN | 91 |
| 6.5 属性映射 | 100 |
| 6.6 持久化实体管理器 ENTITYMANAGER | 105 |
| 6.6.1 Entity 获取 find()或 getReference() | 105 |
| 6.6.2 添加 persist() | 105 |
| 6.6.3 更新实体 | 106 |
| 6.6.4 合并 Merge() | 106 |
| 6.6.5 删除 Remove() | 107 |
| 6.6.6 执行 JPQL 操作 createQuery() | 107 |
| 6.6.7 执行 SQL 操作 createNativeQuery() | 108 |
| 6.6.8 刷新实体 refresh() | 108 |
| 6.6.9 检测实体当前是否被管理中 contains() | 109 |
| 6.6.10 分离所有当前正在被管理的实体 clear() | 109 |
| 6.6.11 将实体的改变立刻刷新到数据库中 flush() | 109 |
| 6.6.12 改变实体管理器的 Flush 模式 setFlushMode() | 110 |
| 6.6.13 获取持久化实现者的引用 getDelegate() | 111 |
| 6.7 关系/对象映射 | 111 |
| 6.7.1 映射的表名或列名与数据库保留字同名时的处理 | 111 |
| 6.7.2 一对多及多对一映射 | 111 |
| 6.7.3 一对一映射 | 119 |
| 6.7.4 多对多映射 | 126 |
| 6.8 使用参数查询 | 132 |
| 6.8.1 命名参数查询 | 132 |
| 6.8.2 位置参数查询 | 132 |
| 6.8.3 Date 参数 | 133 |
| 6.9 JPQL 语言 | 133 |
| 6.9.1 大小写敏感性(Case Sensitivity) | 145 |
| 6.9.2 命名查询 | 145 |
| 6.9.3 排序(order by) | 145 |
| 6.9.4 查询部分属性 | 146 |
| 6.9.5 查询中使用构造器(Constructor) | 147 |
| 6.9.6 聚合查询(Aggregation) | 148 |
| 6.9.7 关联(join) | 150 |
| 6.9.8 排除相同的记录 DISTINCT | 153 |
| 6.9.9 比较 Entity | 154 |
| 6.9.10 批量更新(Batch Update) | 155 |

| | |
|---|------------|
| 6.9.11 批量删除(Batch Remove)..... | 155 |
| 6.9.12 使用操作符 NOT..... | 155 |
| 6.9.13 使用操作符 BETWEEN | 156 |
| 6.9.14 使用操作符 IN | 157 |
| 6.9.15 使用操作符 LIKE..... | 157 |
| 6.9.16 使用操作符 IS NULL..... | 158 |
| 6.9.17 使用操作符 IS EMPTY | 159 |
| 6.9.18 使用操作符 EXISTS..... | 160 |
| 6.9.19 字符串函数..... | 161 |
| 6.9.20 计算函数..... | 162 |
| 6.9.21 子查询..... | 163 |
| 6.9.22 结果集分页..... | 163 |
| 6.10 调用存储过程..... | 165 |
| 6.10.1 调用无返回值的存储过程..... | 165 |
| 6.10.2 调用返回单值的存储过程..... | 166 |
| 6.10.3 调用返回表全部列的存储过程..... | 167 |
| 6.10.4 调用返回部分列的存储过程..... | 167 |
| 6.11 事务管理服务 | 168 |
| 6.12 ENTITY 的生命周期和状态..... | 173 |
| 6.12.1 生命周期回调事件..... | 173 |
| 6.12.2 在外部类中实现回调..... | 175 |
| 6.12.3 在 Entity 类中实现回调..... | 180 |
| 6.13 复合主键(COMPOSITE PRIMARY KEY) | 181 |
| 6.14 实体继承..... | 189 |
| 6.14.1 每个类分层结构一张表(table per class hierarchy)..... | 190 |
| 6.14.2 每个子类一张表(table per subclass)..... | 196 |
| 6.14.3 每个具体类一张表(table per concrete class)..... | 199 |
| 第七章 WEB 服务(WEB SERVICE)..... | 204 |
| 7.1 WEB SERVICE 的创建..... | 204 |
| 7.2 WEB SERVICE 的客户端调用..... | 208 |
| 7.2.1 用 java 语言调用 Web Service | 208 |
| 7.2.2 用 asp 调用 Web Service..... | 211 |
| 第八章 使用 EJB3.0 构建轻量级应用框架 | 212 |
| 8.1 在 WEB 中使用 EJB3.0 框架..... | 212 |
| 8.1.1 如何使用 Session Bean | 213 |
| 8.1.2 如何使用 Message Driven Bean | 215 |
| 8.1.3 如何使用依赖注入(dependency injection) | 216 |
| 8.1.4 如何使用 Entity Bean..... | 217 |

第一章 前言

期待已久的 EJB3.0 最终规范已经发布了。虽然 EJB3.0 最终规范出来了一段时间,但对 EJB3.0 的应用还停留在介绍之中,应用实例更是少之又少,所以作者拟写本书,以简单的实例展现 EJB3.0 的开发过程,希望对大家有所帮助。

EJB3 最激动人心的是 POJO 编程模型,我想对开发人员的影响将是非常大的,因为他降低了开发人员编写 EJB 的要求。EJB3 的 bean 类将更像常规的 Java bean。不要求像过去那样实现特殊的回调界面或者扩展 EJB 类。所以它将使 EJB 的开发更像常规的 Java 开发。从作者几个 EJB3.0 项目的开发情况来看,除了第一个项目开发周期相对有些长之外(因为开发人员之前尚未掌握 EJB3.0,相当于边学边用),后面的项目从开发周期到以后的维护时间都明显优于 JDBC+java bean 或 sprint+hibernate 的项目,软件在模块划分上更清晰,业务模块重用方面也有所提高(多种客户端 J2ME, Web, Wap 重用业务对象),数据库移植性方面非常棒,只需修改一下数据源就很容易切换数据库。呵呵,怎么样,心动了吧?心动不如行动,赶快说声 HelloWorld 吧。

虽然使用 EJB3 做项目非常好,但如果你的团队里没有一个对实体 Bean 熟悉的人,最好先熟悉完了再开展项目,否则将会给项目带有巨大的性能问题。

作者对一些新的概念和知识理解也难免有误,有些概念和语义把握的不是很准,希望在这方面有经验和了解的朋友批评指正,欢迎多提意见。

因为 JBOSS EJB3.0 产品常未成熟,本教程随着新产品的推出将有所改动,请密切关注!

1.1 本教程适合人群

本教程适合具有 Java 语言基础的 EJB 初学者。不需要学习 EJB2.x 也可以直接学习 EJB3.0。虽然 EJB3 的知识点有很多,但实用的知识点有 7 天的学习时间就足够。作者周末为企业提供培训服务,个人需要培训的话就参加北京传智播客(www.itcast.cn)的 EJB3 培训班吧,作者定期到培训班讲课。

1.2 联系作者

黎活明,广东佛山人,毕业于中国农业大学,一直从事于 B/S 系统架构工作,目前就职于惠利至易科技技术/运营总监,北京传智播客“专家课堂”特约讲师。

电子邮件: lihuoming@sohu.com

手机: 13671323507 (谢绝保险推销)

1.3 《EJB3.0 实例教程》官方 MSN 群

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。如果您想参加相关面授培训,请与传智播客公司联系,网址: www.itcast.cn

1.4 版权声明

本电子书的内容全部为版权作品，仅供个人研究和学习之用，不得用于任何商业目的，未经作者书面许可，不得以其他任何方式进行出版、篡改、编辑。

未经作者书面许可，任何商业培训机构不得使用本电子书作为培训教程，否则将依法追究其法律责任。

第二章 运行环境配置

2.1 下载与安装

1>下载安装 **JDK5.0** <http://java.sun.com/j2se/1.5.0/download.jsp>

2>下载安装开发工具 **JBossIDE**(内含 Eclipse 3.2),直接解压缩即可完成安装。

<http://prdownloads.sourceforge.net/jboss/JBossIDE-2.0.0.Beta2-Bundle-win32.zip?download>

想使用中文的朋友可以下载中文语言包 [NLpack1-eclipse-SDK-3.2-win32.zip](#)

下载路径:

http://www.eclipse.org/downloads/download.php?file=/eclipse/downloads/drops/L-3.2_Language_Packs-200607121700/NLpack1-eclipse-SDK-3.2-win32.zip

解压语言包，把 features 及 plugins 文件夹拷贝复盖 JBossIDE 安装目录下的 features 及 plugins 文件夹。如果汉化失败，可能是你安装语言包之前运行过 eclipse，解决办法是：把 eclipse 安装目录下的 configuration 文件夹删除，从 JBossIDE 安装包中解压出 configuration 文件夹，把 configuration 文件夹拷贝到 JBossIDE 安装目录下。

3>下载和安装 **JBoss-4.2.1.GA 服务器**

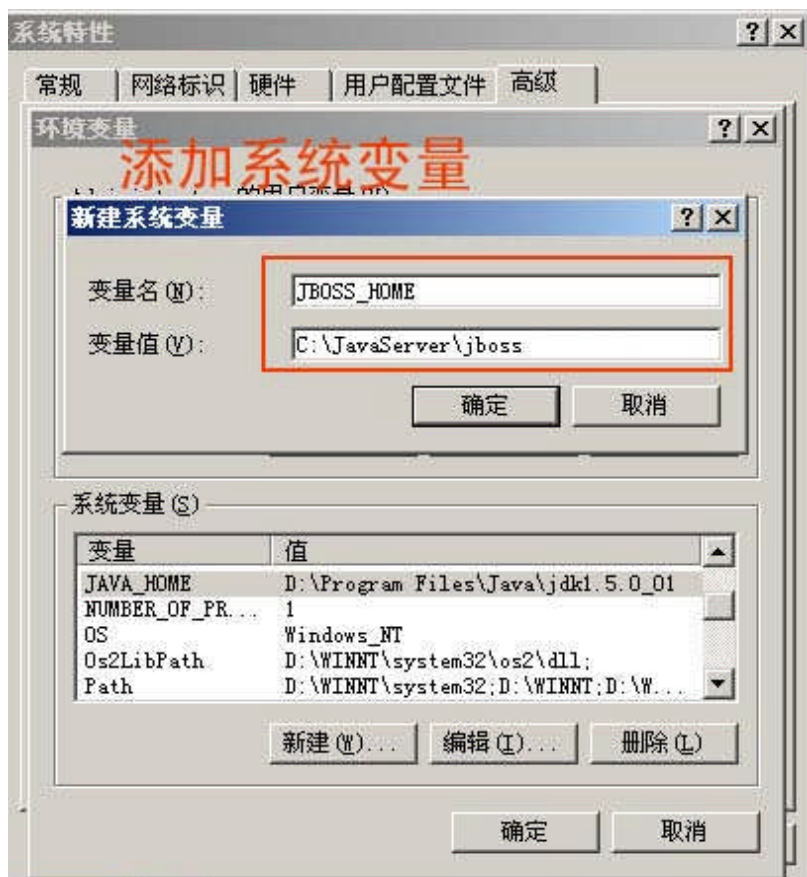
http://sourceforge.net/project/showfiles.php?group_id=22866&package_id=16942&release_id=523619

选择 jboss-4.2.1.GA.zip 文件下载 (大小为 90.2M)，**这里要注意：最好不要用下载工具下载,直接通过“目标另存为”下载。**

安装方法:

直接解压缩文件即可完成安装，为了避免日后产生莫名的错误，解压缩的路径不要带有空格，如“Program Files”。

安装完后请在“系统变量”里添加 **JBOSS_HOME** 变量，值为 Jboss 的安装路径。如下图



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

```

10:12:54,406 INFO [testQueue] Bound to JNDI name: queue/testQueue
10:12:54,484 INFO [UILServerILService] JBossMQ UIL service available at : /127.
0.0.1:8093
10:12:54,515 INFO [DLQ] Bound to JNDI name: queue/DLQ
10:12:54,734 INFO [ConnectionFactoryBindingService] Bound ConnectionManager 'jb
oss.jca:service=ConnectionFactoryBinding,name=JmsXA' to JNDI name 'java:JmsXA'
10:12:55,078 INFO [ConnectionFactoryBindingService] Bound ConnectionManager 'jb
oss.jca:service=DataSourceBinding,name=nxzxDS' to JNDI name 'java:nxzxDS'
10:12:55,468 INFO [JmxKernelAbstraction] creating wrapper delegate for: org.jbo
ss.ejb3.stateless.StatelessContainer
10:12:55,484 INFO [JmxKernelAbstraction] installing MBean: jboss.j2ee:jar=Hello
World.jar,name=HelloWorldBean,service=EJB3 with dependencies:
10:12:55,718 INFO [EJBContainer] STARTED EJB: com.foshanshop.ejb3.impl.HelloWor
ldBean ejbName: HelloWorldBean
10:12:55,781 INFO [EJB3Deployer] Deployed: file:/D:/JavaEEServer/jboss/server/a
ll/deploy/HelloWorld.jar
10:12:55,875 INFO [TomcatDeployer] deploy, ctxPath=/jmx-console, warUrl=.../dep
loy/jmx-console.war/
10:12:56,859 INFO [Http11Protocol] Starting Coyote HTTP/1.1 on http-127.0.0.1-8
080
10:12:56,875 INFO [AjpProtocol] Starting Coyote AJP/1.3 on ajp-127.0.0.1-8009
10:12:56,906 INFO [Server] JBoss <MX MicroKernel> [4.2.0.CR1 <build: SUNTag=JBo
ss_4_2_0_CR1 date=200703051212>] Started in 37s:453ms

```

你可以输入 <http://localhost:8080> 来到 Jboss 的欢迎主页。在 JBoss Management 栏中点击"JMX Console"进入 Jboss

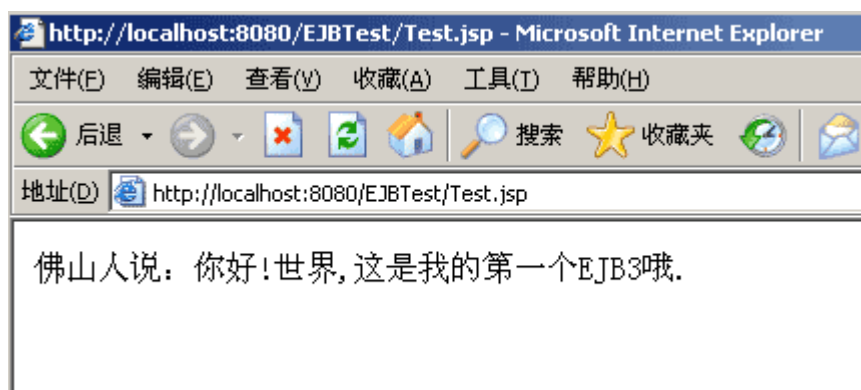
的管理界面，如果需要输入用户名及密码，默认的用户名及密码都是 `admin`。

如果启动 jboss 出现例外，先看看安装文件下载是否完整，jboss 所用端口有没有被占（如 1099,1098,8080,8083 等端口）。可以下载端口查看器（Active Ports）进行检查，如果端口被占用就关闭此进程。确定不是端口被占用，那很大可能是你的 JDK 安装不正确。怎么排错，你自己看着办吧。

2.2 运行一个 EJB3 例子

服务安装成功了，得来一个真家伙试试。在源代码的 HelloWorld 文件夹下（源代码下载：<http://www.foshanshop.net/>），把 HelloWorld.jar 拷贝到“jboss 安装目录/server/default/deploy/”目录下，jboss 支持热部署，HelloWorld 会被发现，并自动完成部署。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/”目录下。

在浏览器上输入：`http://localhost:8080/EJBTest/Test.jsp`。将会看见下图所示。



2.3 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（JARs） |
| docs | 配置的样本文件（数据库配置等） |
| docs/dtd | 在 JBoss 中使用的各种 XML 文件的 DTD。 |
| lib | 一些 JAR，JBoss 启动时加载，且被所有 JBoss 配置共享。（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：minimial，default 和 all，在你安装时可以进行选择。 |
| server/all | JBoss 的完全配置，启动所有服务，包括集群和 IIOP。 |
| server/default | JBoss 的默认配置。在没有在 JBoss 命令航中指定配置名称时使用。（本教程就采用此配置） |
| server/default/conf | JBoss 的配置文件。 |
| server/default /data | JBoss 的数据库文件。比如，嵌入的数据库，或者 JBossMQ。 |
| server/default /deploy | JBoss 的热部署目录。放到这里的任何文件或目录会被 JBoss 自动部署。EJB、WAR、EAR，甚至服务。 |
| server/default/lib | 一些 JAR，JBoss 在启动特定配置时加载他们。 |

| | |
|--------------------|--------------|
| server/default/log | JBoss 的日志文件 |
| server/default/tmp | JBoss 的临时文件。 |

2.4 JBoss 中的部署

JBoss 中的部署过程非常的简单、直接。在每一个配置中，JBoss 不断的扫描一个特殊目录的变化：

[jboss 安装目录]/server/config-name/deploy

此目录一般被称为“部署目录”。

你可以把下列文件拷贝到此目录下：

- * 任何 jar 库（其中的类将被自动添加到 JBoss 的 classpath 中）
- * EJB JAR
- * WAR (Web Application Archive)
- * EAR (Enterprise Application Archive)
- * 包含 JBoss MBean 定义的 XML 文件
- * 一个包含 EJB JAR、WAR 或者 EAR 的解压缩内容，并以.jar、.war 或者.ear 结尾的目录。

要重新部署任何上述文件（JAR、WAR、EAR、XML 等），用新版本的文件覆盖以前的就可以了。JBoss 会根据比较文件的时间发现改变，然后部署新的文件。要重新部署一个目录，更新他的修改时间即可。

2.5 在单独的 Tomcat 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 或 Resin。下面介绍如何在单独的 Tomcat 服务器中调用 EJB。在单独的 Tomcat 服务器中调用 EJB 需要有以下步骤：

根据应用的需要，把调用 EJB 所依赖的 Jar 包拷贝到 tomcat 下的/shared/lib 目录或 WEB 应用的 WEB-INF/lib 下，所依赖的 Jar 一般在 jboss 安装目录的 client，/server/default/deploy/jboss-aop-jdk50.deployer，/server/default/deploy/ejb3.deployer，/lib/endorsed 等文件夹下。

下面的 jar 文件是必需的：

[jboss 安装目录]\client\commons-logging.jar
[jboss 安装目录]\client\ concurrent.jar
[jboss 安装目录]\client\ ejb3-persistence.jar
[jboss 安装目录]\client\ hibernate-annotations.jar
[jboss 安装目录]\client\ hibernate-client.jar
[jboss 安装目录]\client\ javassist.jar
[jboss 安装目录]\client\ jboss-annotations-ejb3.jar
[jboss 安装目录]\client\ jboss-aop-jdk50-client.jar
[jboss 安装目录]\client\ jboss-aspect-jdk50-client.jar
[jboss 安装目录]\client\jboss-common-client.jar
[jboss 安装目录]\client\ jboss-ejb3-client.jar
[jboss 安装目录]\client\ jboss-ejb3x.jar
[jboss 安装目录]\client\jboss-j2ee.jar
[jboss 安装目录]\client\jboss-remoting.jar
[jboss 安装目录]\client\jbossx-client.jar
[jboss 安装目录]\client\jboss-transaction-client.jar

[jboss 安装目录]\client\jnp-client.jar
[jboss 安装目录]\client\trove.jar
[jboss 安装目录]\client\jbossws-client.jar
[jboss 安装目录]\client\jboss-jaxws.jar
[jboss 安装目录]\client\xmlsec.jar

1> 把 EJB 接口拷贝到应用的 /WEB-INF/classes/ 目录下

2> 客户端访问 EJB 时必须明确设置 InitialContext 环境属性, 代码如下:

```
Properties props = new Properties();
props.setProperty("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
props.setProperty("java.naming.provider.url", "localhost:1099");
props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");
InitialContext ctx = new InitialContext(props); //如果客户端和jboss运行在同一个jvm, 不需要传入props
HelloWorld helloworld = (HelloWorld) ctx.lookup("HelloWorldBean/remote");
out.println(helloworld.SayHello("佛山人"));
```

除了上面通过硬编码设置环境属性的方式外, 还可以在应用的 classpath 下放置一个 jndi.properties 文件。

注意: 在此环境下不能调用 EJB 的 Local 接口, 因为他与 JBOSS 不在同一个 VM 中。

J2se 中调用 EJB3 同样需要把上述 jar 及 EJB 接口放置在应用的类路径下。

2.6 发布在 JBOSS 中的 WEB 应用调用 EJB

有些调用 EJB 的 WEB 应用是直接发布在 Jboss 集成环境下, 本教程的客户端调用例子就是发布在 Jboss 中。在 Jboss 下发布 WEB 应用, 需要把 WEB 应用打包成 war 文件。另外在此环境下调用 EJB 不需要把 EJB 的接口类放入 /WEB-INF/classes/ 目录中, 否则在调用 Stateful Bean 就会发生类型冲突, 引发下面的例外。

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

备注: 在此环境下, EJB 的 Local 或 Remote 接口都可以被调用。

发布在 Jboss 下的客户端不需要明确设置 JNDI 访问的上下文环境, 可以直接通过 InitialContext ctx = new InitialContext() 获得上下文环境, 容器会自动赋给 InitialContext 正确的环境, 代码如下:

```
InitialContext ctx = new InitialContext(); //如果客户端和jboss运行在同一个jvm, 不需要传入props
HelloWorld helloworld = (HelloWorld) ctx.lookup("HelloWorldBean/remote");
out.println(helloworld.SayHello("佛山人"));
```

如果硬给 InitialContext 设置了访问属性, 反而会带来不可移植的问题, 因为你的应用有可能部署在 weblogic 等应用服务器。(本教程考虑到部分同学可能需要在独立的 J2se 中调用 EJB, 为了教学的方便, 把访问属性都设上了, 这样不管在 jboss、j2se 或独立 tomcat, 都能获得正确的 InitialContext)

2.7 如何获取最新的内嵌 EJB3 的 JBOSS 版本

有时候内嵌 EJB3 的 JBOSS 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, 首先我们得安装一个版本控制软件 TortoiseSVN, 下载路径(window 版本):

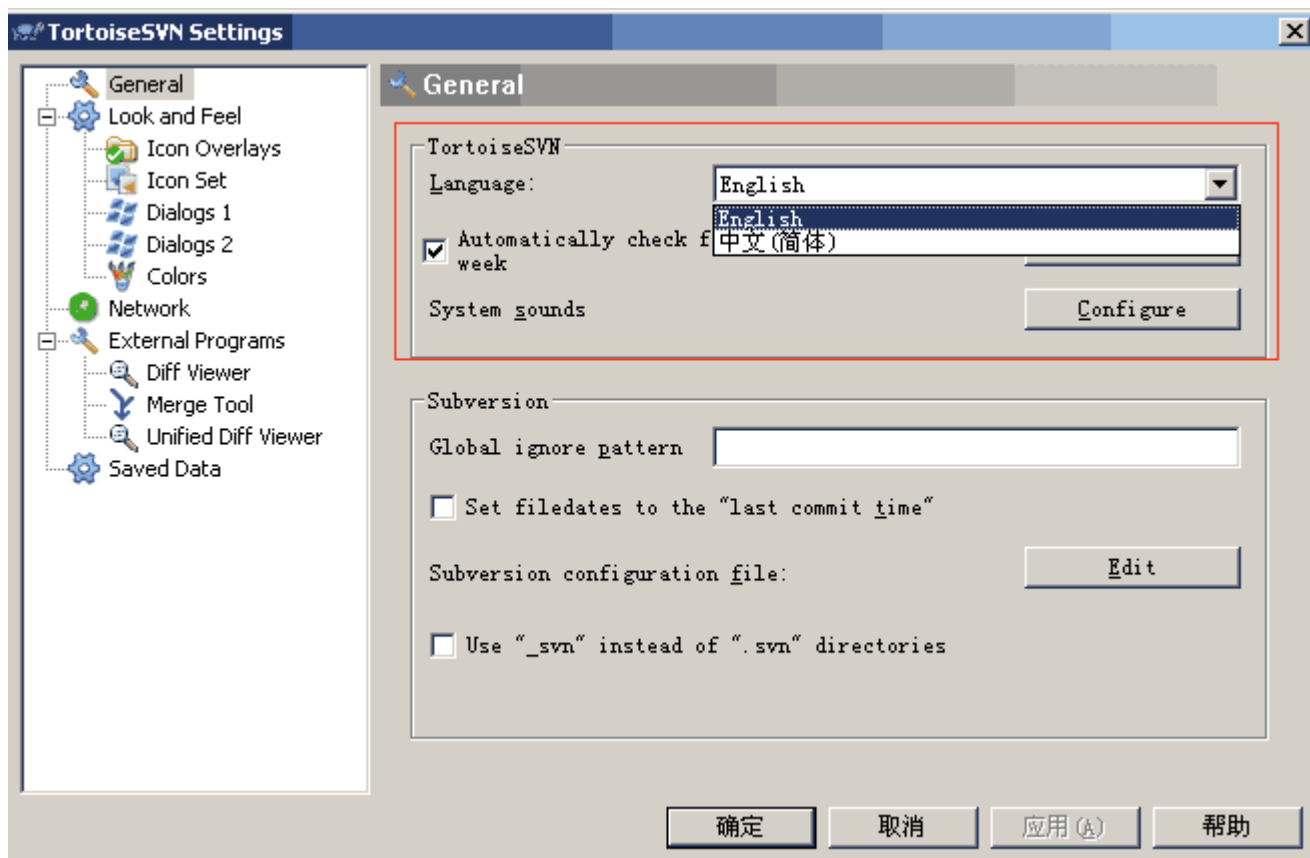
<http://prdownloads.sourceforge.net/tortoisesvn/TortoiseSVN-1.4.0.7501-win32-svn-1.4.0.msi?download>

双击进行安装，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://prdownloads.sourceforge.net/tortoisesvn/LanguagePack-1.4.0.7501-win32-zh_CN.exe?download

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：

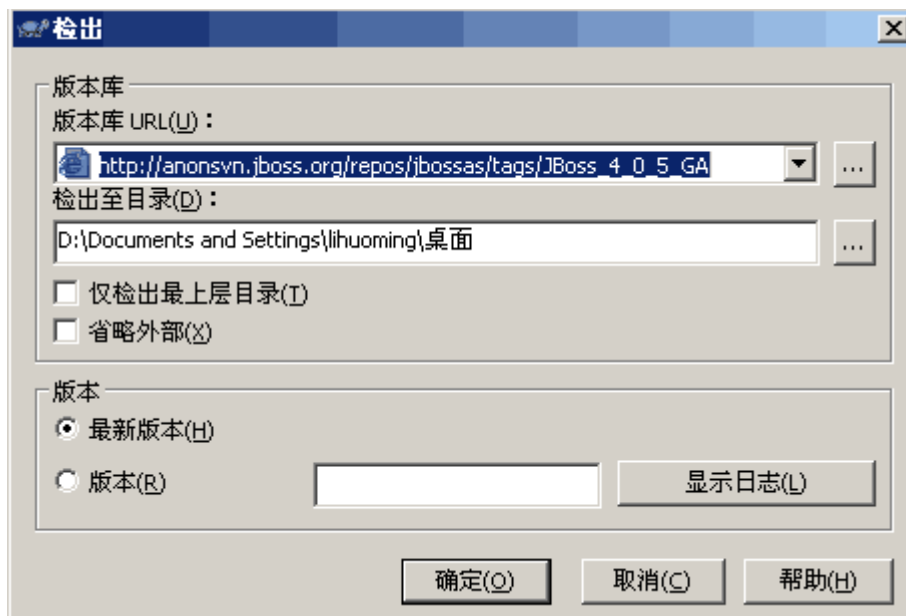


下面我们就开始从 Jboss 版本库中获取最新版的 Jboss 源文件。

如我们要获取 JBoss_4_0_5_GA 版的源文件。他的版本库路径为：

http://anonsvn.jboss.org/repos/jbossas/tags/JBoss_4_0_5_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，在跳出的属性菜单中选择“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_0_5_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 70M 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，程序就开始进行编译，这过程需要十来分钟。成功执行完后，在 build/output 文件夹生成了两个 Jboss 发行版(jboss-4.0.5.GA 和 jboss-4.0.5.GA-ejb3), jboss-4.0.5.GA-ejb3 内嵌 EJB3, jboss-4.0.5.GA 不带 EJB3。把 jboss-4.0.5.GA-ejb3 拷贝到某个目录中就可以直接使用了。

第三章 基础知识学习

3.1 Enterprise JavaBeans(EJB)的概念

Enterprise 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 Enterprise 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 Enterprise JavaBeans 规范的服务器平台，如 jboss、weblogic。

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)。

Session Bean: Session Bean 用于实现业务逻辑，它分为有状态 bean 和无状态 bean。每当客户端请求时，容器就会选择一个 Session Bean 来为客户端服务。Session Bean 可以直接访问数据库，但更多时候，它会通过 Entity Bean 实现数据访问。

下图展示了 Session Bean 通过 Entity Bean 往数据库插入一条记录。

Session Bean

```

package com.foshanshop.ejb3.impl;

import java.util.Date;

@Stateless(mappedName="PersonDAOBean")
@Remote ({PersonDAO.class})
public class PersonDAOBean implements PersonDAO {

    @PersistenceContext
    protected EntityManager em;

    public String getPersonNameByID(int personid) {}

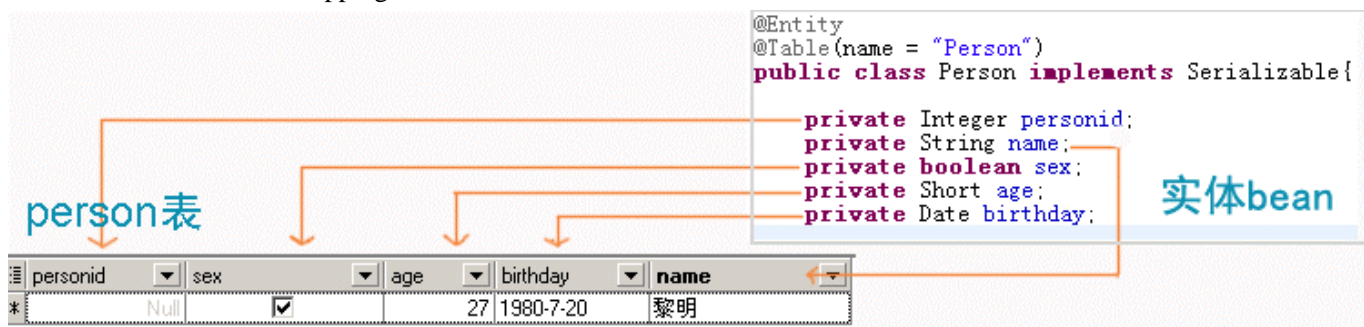
    public boolean insertPerson(String name, boolean sex, short age, Date birthday) {
        try {
            Person person = new Person();
            person.setName(name);
            person.setSex(sex);
            person.setAge(Short.valueOf(age));
            person.setBirthday(birthday);
            em.persist(person);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
        return true;
    }
}

```

实例bean

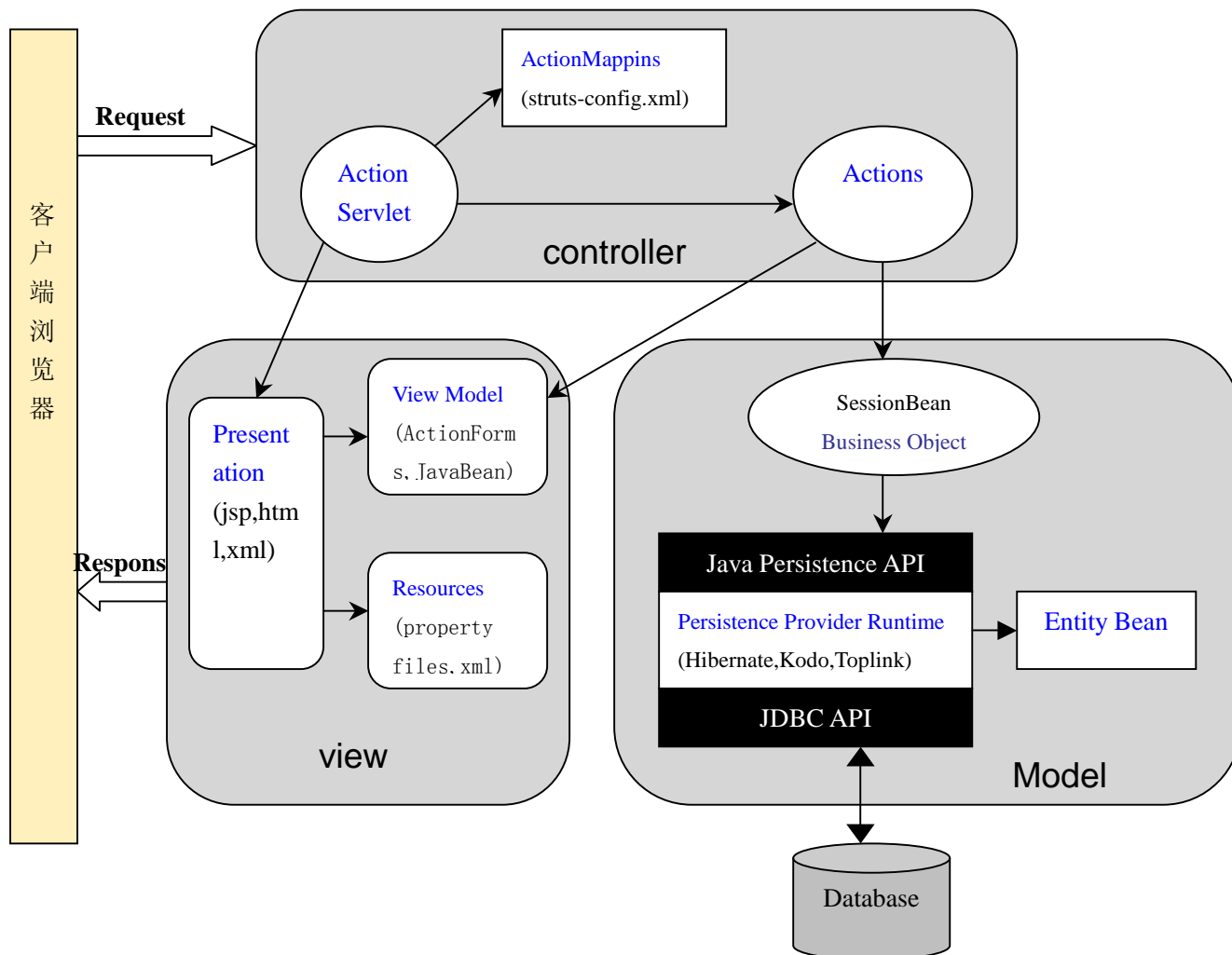
业务代码

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 JDBC+JavaBean 编程中, 通常把 JDBC 查询的结果信息存入 JavaBean, 然后供后面程序进行处理。在这里你可以把实体 Bean 看作是用来存放数据的 JavaBean。但比普通 JavaBean 多了一个功能, 实体 bean 除了担负起存放数据的角色, 还要负责跟数据库表进行对象与关系映射 (O/R Mapping), 下图就说明了这一映射:



消息驱动 Bean(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。

3.2 一个基于 Struts+EJB3.0 应用的体系结构图



3.3 如何进行 EJB 打包

要发布 EJB 时必须把她打成*.jar 文件，一个 EJB 打包后的目录结构如下：

```

EJB 应用根目录
|-- com (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)
  
```

打包的方式有很多，如：jar 命令行、集成开发环境的打包向导和 Ant 任务。下面介绍 Elispse 打包向导和 Ant 打包任务。

1. Elispse 打包向导

在 Elispse 开发环境下，可以通过向导进行打包。右击项目名称，在跳出的菜单中选择“导出”，在“导出”对话框选择“Jar 文件”，在“选择要导出的资源”时，选择源目录和用到的资源。然后选择一个存放目录及文件名。点“完成”就结束了打包。

2. Ant 打包任务

采用 Ant 进行打包是比较方便的，也是作者推荐的打包方式。下面我们看一个简单的打包任务。

```
<?xml version="1.0"?>
<project name="jartest" default="jar" basedir=".">
  <property name="build.dir" value="${basedir}/build" />
  <property name="build.classes.dir" value="${build.dir}/classes" />

  <target name="jar" description="打包成 Jar">
    <jar jarfile="${basedir}/ejbfile.jar">
      <fileset dir="${build.classes.dir}">
        <include name="**/*.class" />
      </fileset>
      <metainf dir="${basedir}/META-INF">
        <include name="*" />
      </metainf>
    </jar>
  </target>
</project>
```

上面建立了一个名为 jartest 的 Ant 项目，默认的任务为 default="jar"，项目的路径为 build.xml 文件所在目录 basedir="."。应用编译过后的 class 文件已经存在于应用的/build/classes/目录下。Ant 定义了一个属性“build.classes.dir”，他指向应用的/build/classes/目录。

<target name="jar" description="打包成 Jar">定义了一个名叫 jar 的任务，description 是描述信息。任务中使用 jar 命令把/build/classes/目录下的所有 class 文件打包进 jar 文件，同时也把应用下的 META-INF 目录下的所有文件打包进 jar 文件的 META-INF 目录。打包后的 jar 文件存放在应用目录下。文件名为：ejbfile.jar

3.4 如何进行 WEB 应用打包

一个 Web 应用发布到 Jboss 服务器时需要打成 war 包。Web 应用打包后的目录结构如下：

```
WEB 应用根目录
|-- **/*.jsp
|-- WEB-INF
|   |-- web.xml
|   |-- lib
|       |-- *.
|   |-- classes
|       |-- **/*.class
```

本节将介绍 jar 命令行及 Ant 任务两种 war 文件的打包方式。

1.在命令行下用 jar 命令进行 war 文件打包

在打包前把文件存成上面的目录结构：

在 Dos 窗口中进入到 WEB 应用根目录下，执行如下命令

```
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

例如 WEB 应用根目录在: D:\java\webapp\，命令输入如下：

D:\java\webapp> **jar cvf EJBTest.war ***

2. 在 Ant 任务中进行 war 文件打包

如果文件存放的结构如下面所示：

```
WEB 应用根目录
|-- build.xml
|-- **/*.jsp
|-- WEB-INF
    |-- web.xml
    |-- lib
        |-- *.jar
    |-- classes
        |-- **/*.class
```

那么 Ant 的 war 文件打包任务如下：

```
<?xml version="1.0"?>
<project name="wartest" default="war" basedir=".">
    <target name="war" description="创建 WEB 发布包">
        <war warfile="${basedir}/EJBTest.war" webxml="${basedir}/WEB-INF/web.xml">
            <classes dir="${basedir}/WEB-INF/classes"> <include name="**/*.class"/> </classes>
            <lib dir="${basedir}/WEB-INF/lib"> <include name="*.jar"/> </lib>
            <webinf dir="${basedir}/WEB-INF"> <include name="*.xml"/> </webinf>
        </war>
    </target>
</project>
```

`<target name="war" description="创建 WEB 发布包">` 定义一个名叫 war 的任务。任务中执行 war 打包操作，在 war 节点中，通过 webxml 指明 web.xml 的位置，通过 `<classes dir="${basedir}/WEB-INF/classes">` 指明 web 的 classes 目录位置，通过 `<lib dir="${basedir}/WEB-INF/lib">` 指明 web 的 lib 目录位置，通过 `<webinf dir="${basedir}/WEB-INF">` 指明 web 的 WEB-INF 目录位置。

3.5 如何进行企业应用打包

一个完整的企业应用包含 EJB 模块和 WEB 模块，在发布企业应用时，我们需要把它打成*.ear 文件，在打包前我们必须配置 application.xml 文件，该文件存放于打包后的 META-INF 目录。我们在 application.xml 文件中需要指定 EJB 模块和 WEB 模块的信息，一个 application.xml 配置例子如下：

```
<application xmlns="http://java.sun.com/xml/ns/j2ee" version="1.4"
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/application_1_4.xsd">
    <display-name>EJB3 Sample Application</display-name>
    <module>
        <web>
            <web-uri>web.war</web-uri>
            <context-root>web</context-root>
```

```

    </web>
  </module>
<module>
  <ejb>ejb3.jar</ejb>
</module>
</application>

```

上面<web>指定 Web 模块，<ejb>指定 EJB 模块，Web 模块或者 EJB 模块都可以存在多个。不管你使用何种方式打包，一个企业应用打包后的目录结构应该如下：

```

ear 应用根目录
|-- ejb3.jar  (你的 EJB 模块)
|-- web.war  (你的 WEB 模块)
|-- META-INF
    |-- MANIFEST.MF (如果使用工具打包，该文件由工具自动生成)
    |-- application.xml

```

3.6 使用了第三方类库的 EJB 如何打包

在实际项目中，我们经常需要使用第三方的类库。这些类库应该放在哪里？EJB 应用一般都有被“卸出”（这里指装入的反向过程）的能力，这种能力由部署时装入它们的类装载器支持。如果我们把第三方类库放入应用服务器的标准类路径（[jboss 安装目录]\server\default\lib），这些类很可能完全失去被卸出的能力。这样，如果 EJB 应用要更新某个第三方类库的版本，重新部署 EJB 应用时，第三方类库也要重新部署。在这种情形下，把第三方类库放入应用服务器标准类路径很不方便，因为每次部署 EJB 应用时，都要重新启动整个应用服务器，这显然不是理想的选择。适合放入应用服务器类路径的第三方类库通常是一些通用类库，如 JDBC 驱动。

对于针对特定应用的第三方类库，最理想的选择是把他们放入 EJB Jar 文件中。每一个 JAR 文件里都有一个 manifest 文件，这个文件由 jar 工具自动创建，默认名字是 MANIFEST.MF。我们可以在 manifest 文件中加入一个 Class-Path 属性，引用它所依赖的 JAR 文件。我们可以手工编辑 manifest.mf 文件，在原有内容的基础上，添加 Class-Path 属性。Class-Path 属性的值是用来搜索第三方类库的相对 URL。这个 URL 总是相对于包含 Class-Path 属性的组件。单个 Class-Path 属性内可以指定多个 URL，一个 manifest 文件可以包含多个 Class-Path 属性。

假设本例 EJB 使用了两个第三方类库，名为：Upload.jar，Socket.jar，修改后的 manifest.mf 文件内容如下：

```

Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: 1.5.0_01-b08 (Sun Microsystems Inc.)
Class-Path: Upload.jar Socket.jar

```

注意：Class-Path: 与 Upload.jar 之间有一空格分隔（缺少了空格就会发生找不到 jar 文件），多个 jar 文件之间需要用空格分隔。Class-Path 所在行还需要进行回车换行。

下面是打完包后的目录结构：

```

EJB 应用根目录
|-- com  (注：ejb 类包)
|-- Upload.jar (注：第三方类库)

```

```
| -- Socket.jar (注： 第三方类库)
| -- META-INF
| -- MANIFEST.MF (注： 加入了 Class-Path 属性)
```

3.7 共用了第三方类库的 J2EE 应用如何打包

一个 J2EE 项目通常由多个 EJB 和 Web 应用构成，如果多个 EJB 及 Web 应用共用了一个第三方类库，我们又如何打包呢？按照上节介绍的内容，我们会把第三方类库打进每个 EJB Jar 文件及放在 Web 应用的/WEB-INF/lib 目录下。虽然这种方案也能解决问题，但它明显地不够完善。封装 JAR 文件的目的是为了提高应用的模块化程度，把同一个类库放入多个 JAR 文件正好是背其道而行之。此外，多次放置同一个类库无谓地加大了应用的体积。最后，即使只改变一个类库的版权，每一个 EJB JAR 文件也都要重新构造，从而使构造过程复杂化。

下面的方案很好地解决了上面的问题

假设一个 J2EE 项目含有两个 EJB 及一个 Web 应用，他们的文件名分别为：HelloWorld.jar，HelloChina.jar，MyEJBTest.war。这三个模块都使用了一个第三方类库，名为：Tools.jar

现在我们要做的是编辑这三个模块的 manifest.mf 文件，在原有内容的基础上，添加 Class-Path 属性。

三个模块的 jar 文件修改后的 manifest.mf 文件内容如下：

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: 1.5.0_01-b08 (Sun Microsystems Inc.)
Class-Path: Tools.jar
```

注意：Class-Path: 与 Tools.jar 之间有一空格分隔（缺少了空格就会发生找不到 jar 文件），Class-Path 所在行还需要进行回车换行。

各个模块通过 manifest.mf 文件都能找到其所依赖的 Tools.jar 文件。

下面是打完包后的目录结构：

```
J2EE 应用根目录
| -- HelloWorld.jar
| -- META-INF
| -- MANIFEST.MF (注： 加入了 Class-Path 属性，引用它所依赖的 Tools.jar)
| -- HelloChina.jar
| -- META-INF
| -- MANIFEST.MF (注： 加入了 Class-Path 属性，引用它所依赖的 Tools.jar)
| -- MyEJBTest.war
| -- META-INF
| -- MANIFEST.MF (注： 加入了 Class-Path 属性，引用它所依赖的 Tools.jar)
| -- Tools.jar (注： 第三方类库)
| -- META-INF
| -- application.xml
| -- MANIFEST.MF (注： 由工具自动生成，没有加入 Class-Path 属性)
```

把第三方类库和 EJB 模块并排打进 jar 包，如果第三方类库很多的情况下，显的有些零乱而不雅观。在此作者建议大家建个文件夹专门用来存放第三方类库。如建个 lib 文件夹，把第三方类库放在此文件夹下。然后修改 J2EE

各模块的 manifest.mf 文件内容, 修改后的内容如下:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: 1.5.0_01-b08 (Sun Microsystems Inc.)
Class-Path: lib/Tools.jar
```

J2EE 应用的文件后缀为 ear, 应用使用到的各模块在 application.xml 文件中定义, 本例的 application.xml 内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/j2ee" version="1.4"
  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/application_1_4.xsd">
  <display-name>EJB3Trail</display-name>
  <description>J2EE Made Easy Trail Map</description>

  <module>
    <ejb>HelloWorld.jar</ejb>
  </module>
  <module>
    <ejb>HelloChina.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>MyEJBTest.war</web-uri>
      <context-root>MyEJBTest</context-root>
    </web>
  </module>
</application>
```

因为 EJB 打进了 EAR 文件, 在访问 EJB 时, JNDI 名应为以下格式:

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

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

例: 如果上面 J2EE 应用打成 MyJ2EE.ear 文件, 访问 HelloWorld EJB 远程接口的 JNDI 名是:
MyJ2EE/HelloWorldBean/remote

3.8 如何恢复本书配套例子的开发环境

登陆 <http://www.foshanshop.net>, 下载本书配套例子, 解压缩后可以看到以下文件夹 (本例解压缩到: E:\book):

| 名称 | 大小 | 类型 | 修改日期 |
|----------------------|-----------|------------|-----------------|
| CompositePK | | 文件夹 | 2006-7-23 18:18 |
| DependencyInjection | | 文件夹 | 2006-7-23 18:18 |
| EJBTest | | 文件夹 | 2006-7-23 18:18 |
| EmbeddedEJB3 | | 文件夹 | 2006-8-6 18:15 |
| EntityBean | | 文件夹 | 2006-7-23 18:18 |
| HelloWorld | | 文件夹 | 2006-7-23 18:18 |
| Interceptor | | 文件夹 | 2006-7-23 18:18 |
| JWS | | 文件夹 | 2006-7-23 18:18 |
| lib | | 文件夹 | 2006-7-23 18:18 |
| LocalRemoteBean | | 文件夹 | 2006-7-23 18:18 |
| LocalSessionBean | | 文件夹 | 2006-7-23 18:18 |
| ManyToMany | | 文件夹 | 2006-7-23 18:18 |
| MessageDrivenBean | | 文件夹 | 2006-7-23 18:18 |
| OneToMany | | 文件夹 | 2006-7-23 18:18 |
| OneToOne | | 文件夹 | 2006-7-23 18:18 |
| Query | | 文件夹 | 2006-7-23 18:18 |
| SessionBeanLifeCycle | | 文件夹 | 2006-7-23 18:18 |
| StatefulBean | | 文件夹 | 2006-7-23 18:18 |
| TransactionService | | 文件夹 | 2006-7-23 18:18 |
| WSClient | | 文件夹 | 2006-7-23 18:18 |
| Book-SourceCode.rar | 12,161 KB | WinRAR ... | 2006-8-6 18:16 |
| mysql-ds.xml | 1 KB | XML 文档 | 2006-7-7 22:29 |

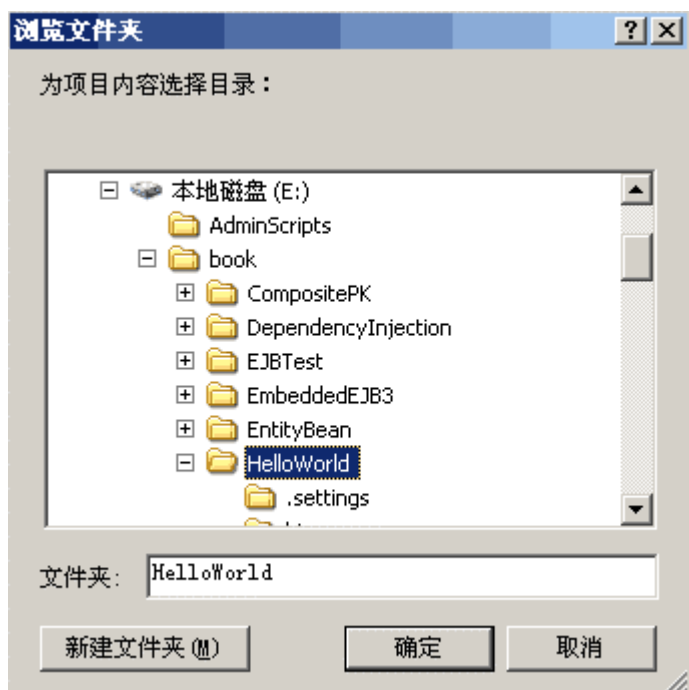
每个文件夹都是一个项目,lib 文件夹存放所有项目使用到的 jar 文件。下面我们以恢复 HelloWorld 项目为例,介绍开发环境的恢复。

Eclipse 必须是 eclipse3.1.x 以上版本,使用 JDK5.0 以上版本。

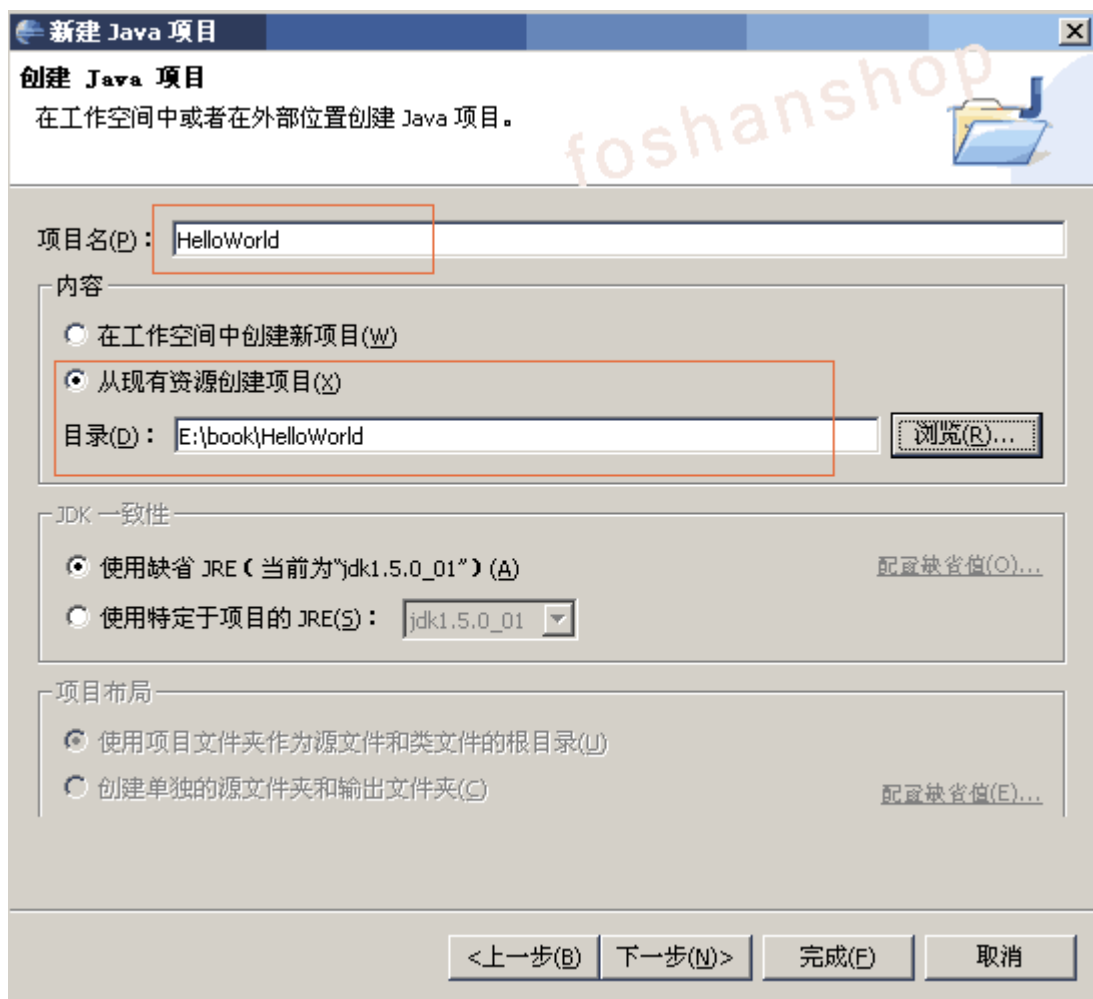
在 Eclipse 开发工具上点击“文件”->“新建”->“项目”,选择“Java 项目”,点击“下一步”,出现如下界面:



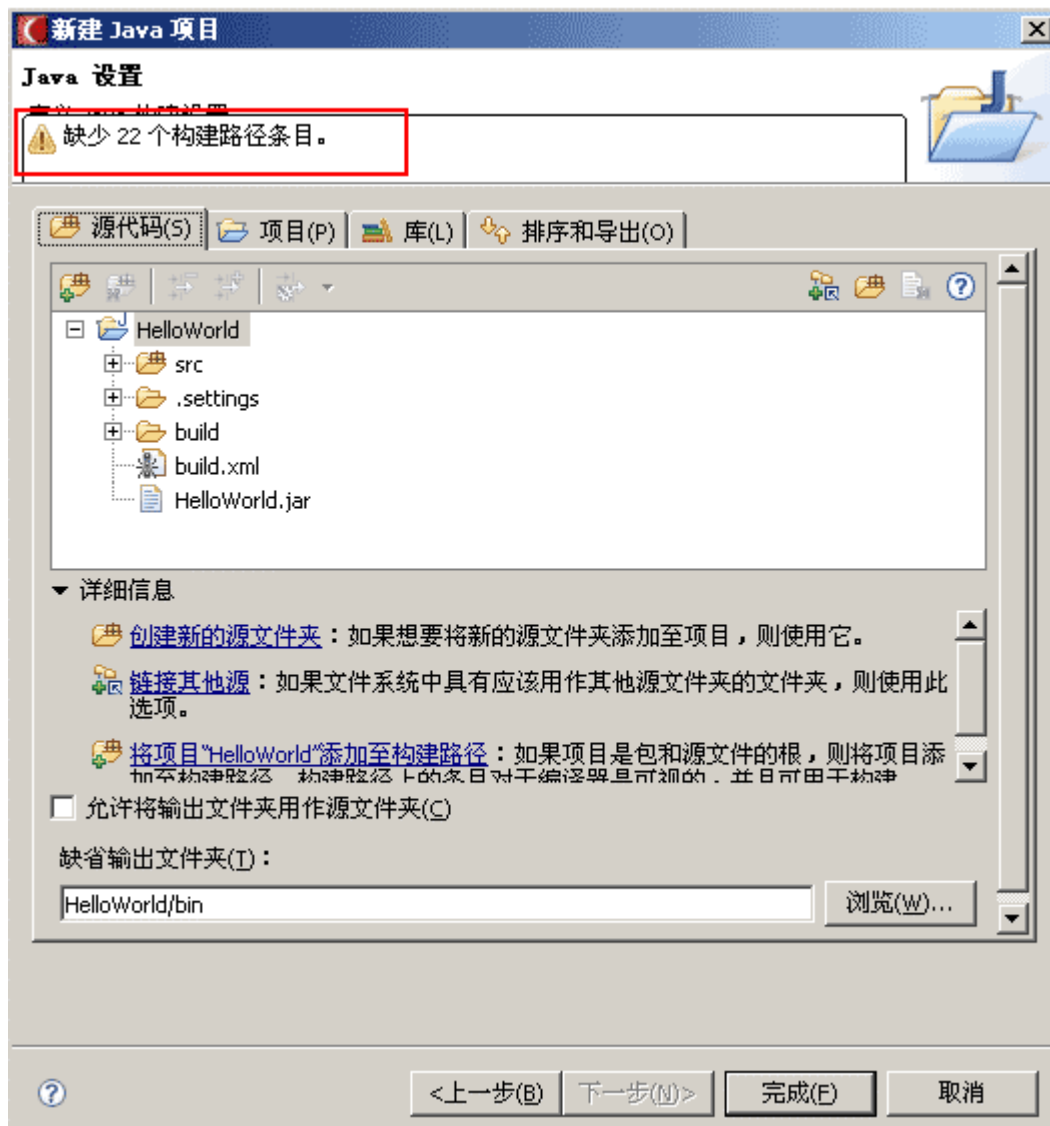
在上图中选择“从现有资源创建项目(X)”，点击“浏览(R)”，在出现的文件窗口中选择 E:\book\HelloWorld 文件夹，如下图：



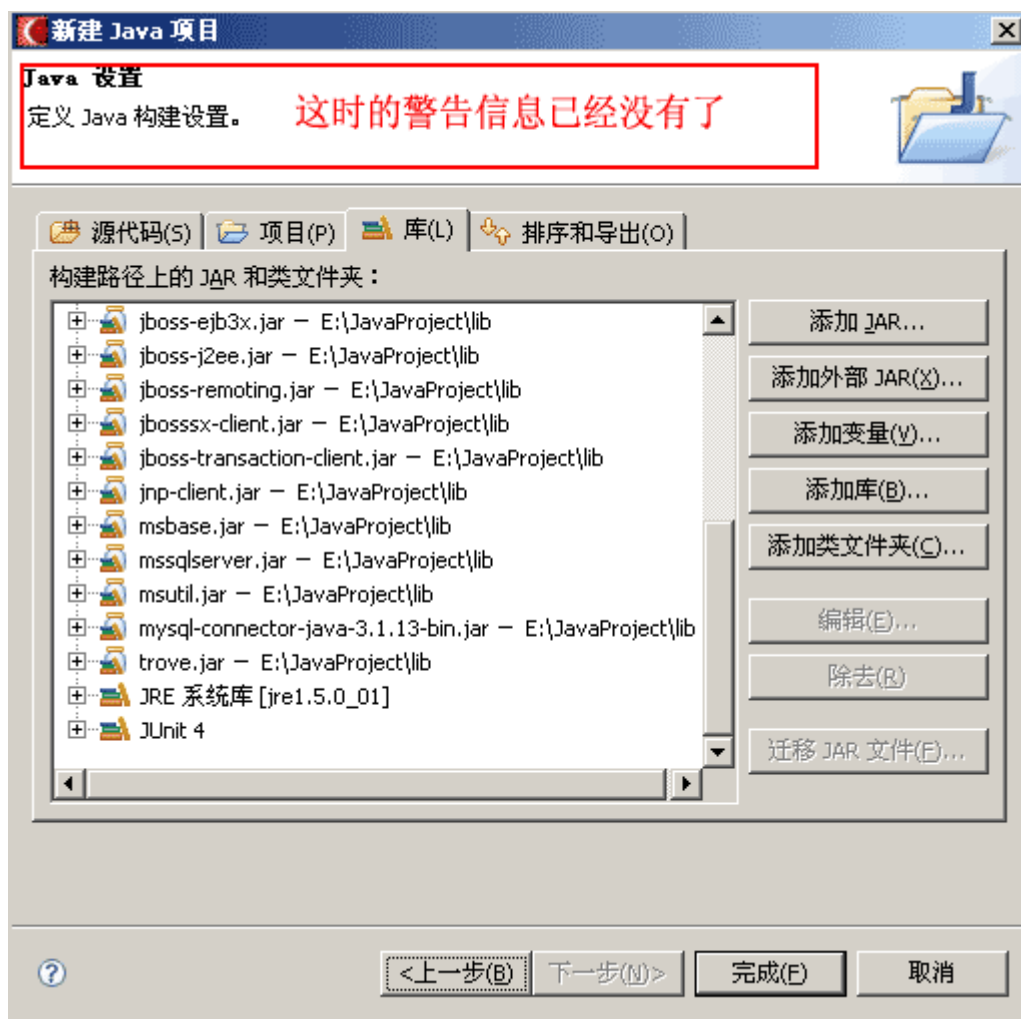
用文件夹名 HelloWorld 作为项目的名称，如下图：



在上图中点击“下一步”，Eclipse 将会检测已存在项目的配置信息，检测完后出现下图：



上图中“java 设置”出现感叹号,并提示缺少 N 个构建路径条目,实际上是 N 个 Jar 文件路径不正确。现在我们需要把 Jar 文件的路径调整正确。在上图中点击“库(L)”,在出现的界面中把全部 jar 文件,“JRE 系统库”及“Junit 4”删除,通过“添加库”按钮选择添加“JRE 系统库”和“Junit”,在添加“Junit”时请选择“Junit 4”版本,添加完“JRE 系统库”和“Junit”后,继续点击“添加外部 JAR(X)”按钮,在出现的文件窗口中把 E:\book\lib 文件夹下所有的 Jar 文件选上(Ctrl+A),回到了如下界面:



点击“完成”就结束了开发环境的建立。

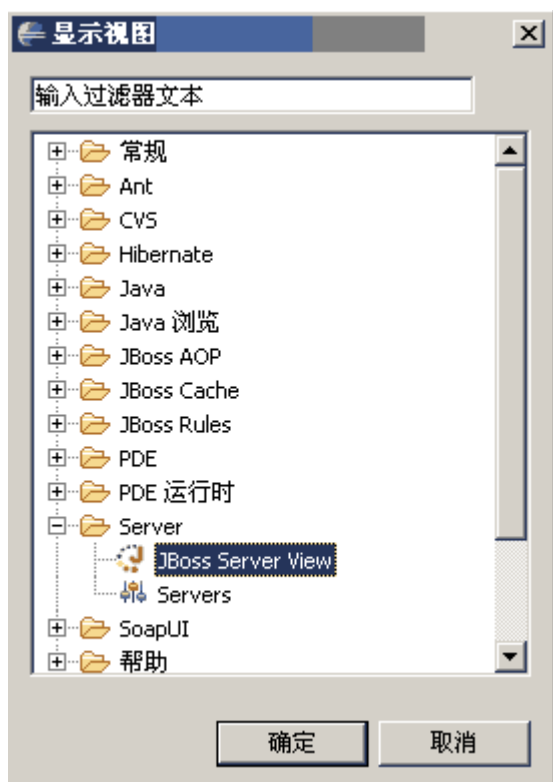
本书其他项目的开发环境恢复步骤如上。

3.9 如何对 EJB3 进行调试

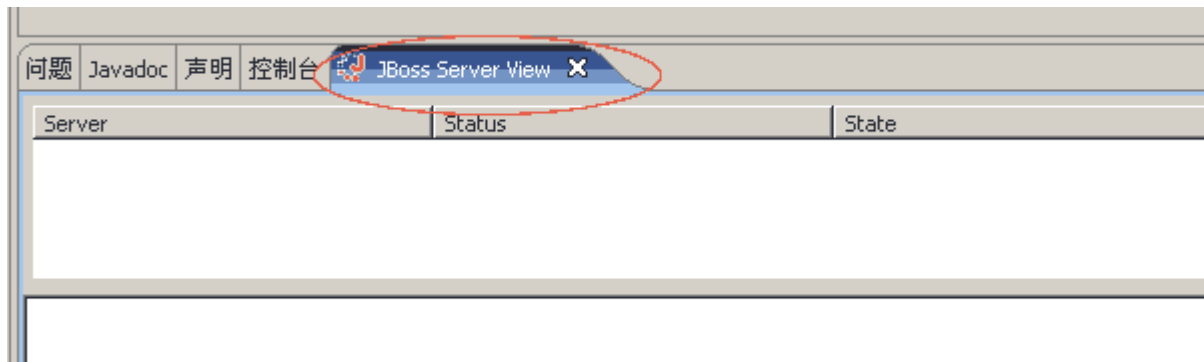
要对 EJB3 进行调试，我们需要用到一个工具：JBossIDE，这个工具就是第二章要你下载安装的开发工具，如果你没有安装，请参照第二章的内容进行安装。

在对 EJB3 进行调试前我们首先需要在 Eclipse 中配置 JBoss server 的调试环境。调试环境建立后，所有的项目都可以共用，配置步骤如下：

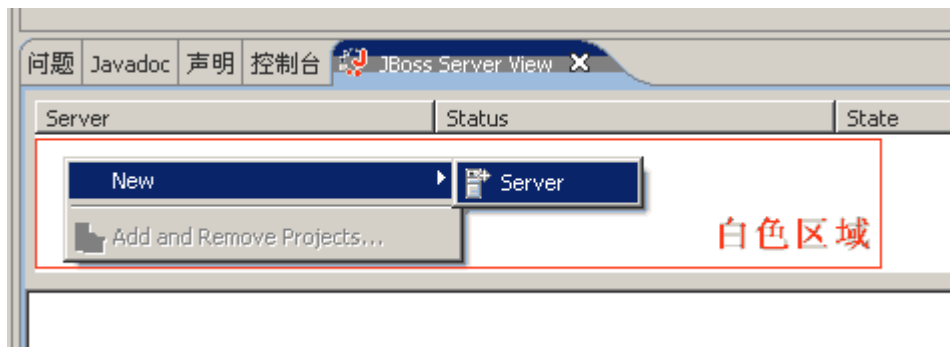
1> 首先需要在 Eclipse 开发界面中显示出“Jboss Server View”视图，你可以在 Eclipse 菜单栏点击“窗口(W)” - “显示视图(V)” - “其他(O)...”，在出现的“显示视图”窗口中选择 Server 文件夹下的“Jboss Server View”，如下图：



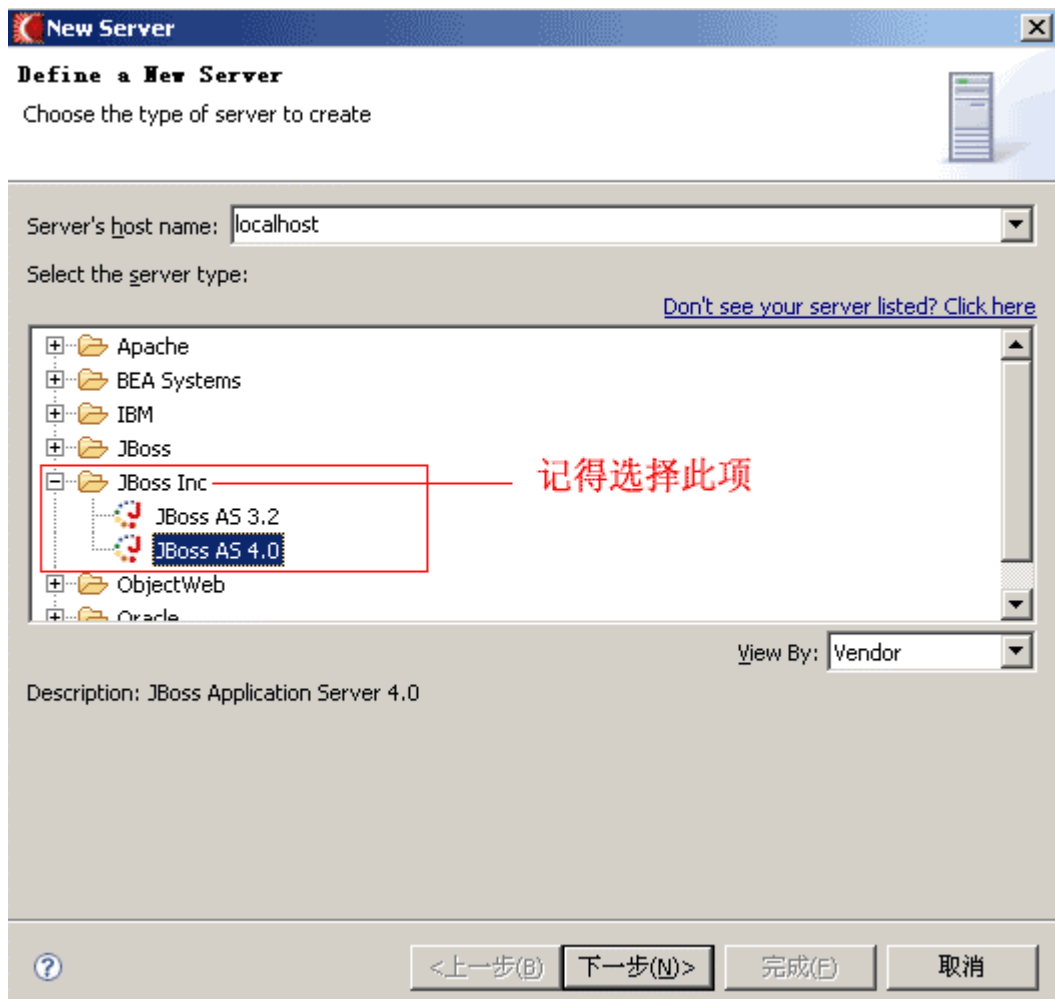
点击“确定”后，“JBoss Server View”视图将出现在 Eclipse 开发界面的控制区（如果你的开发界面已经很零乱，可以通过“窗口(W)” - “复位透视图”复位 Eclipse 原始界面），如下图：



2> 接着我们需要建立一个 Jboss Server。在“JBoss Server View”视图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：



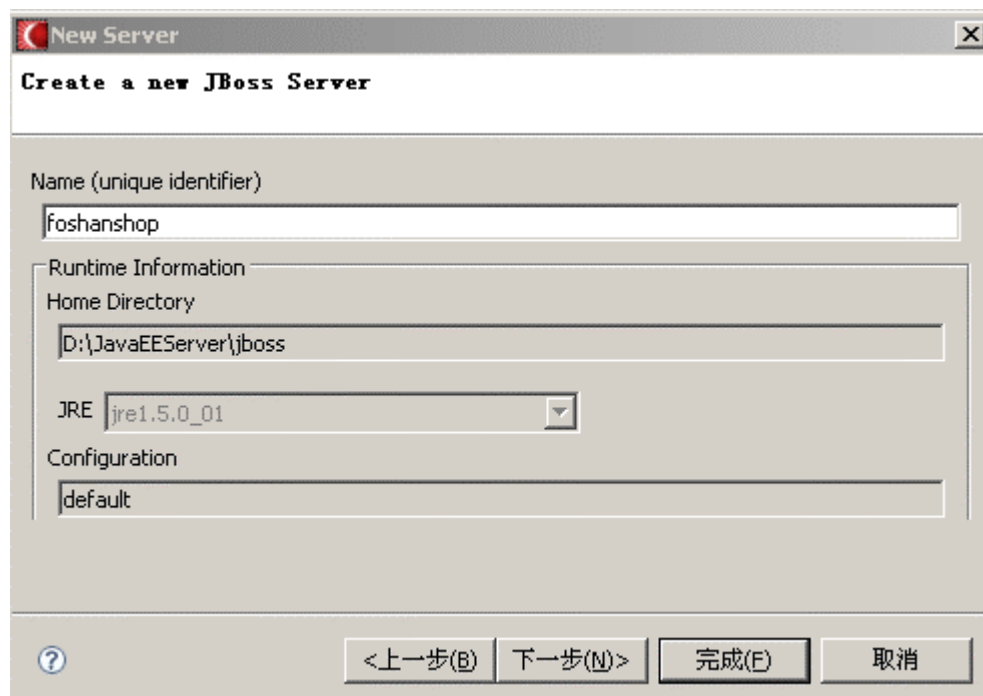
点击“Server”后跳出如下窗口，在这个窗口中我们选择“JBoss AS 4.0”，其他参数采用系统默认值。



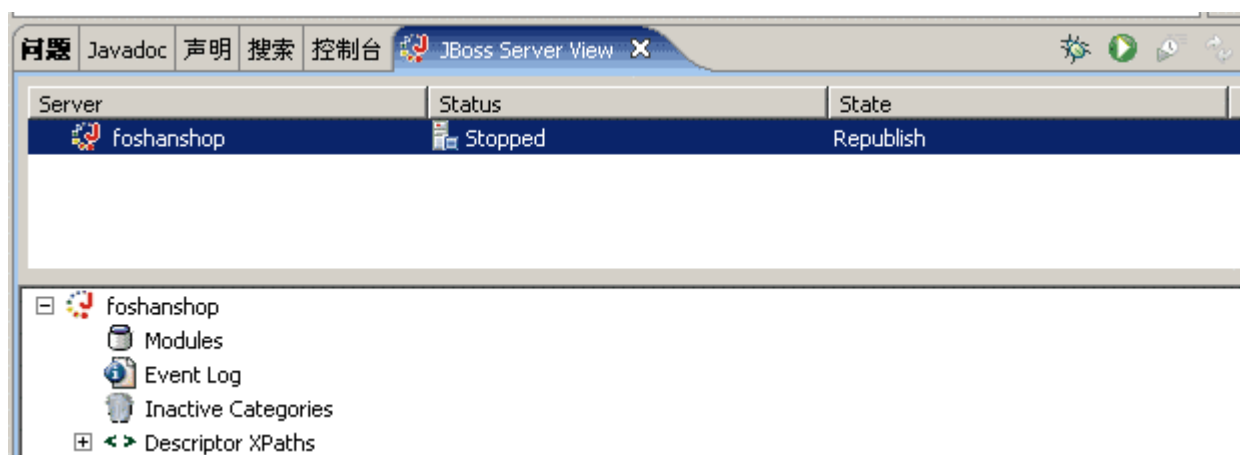
点击“下一步”后跳出如下窗口，在这个窗口中我们输入自定义名称，选择 Jboss 服务器所在目录（作者的 Jboss 安装在 C:\JavaServer\jboss）及使用的 jboss 配置项。



点击“下一步”，在出现的对话框中再次输入前面的自定义名称。

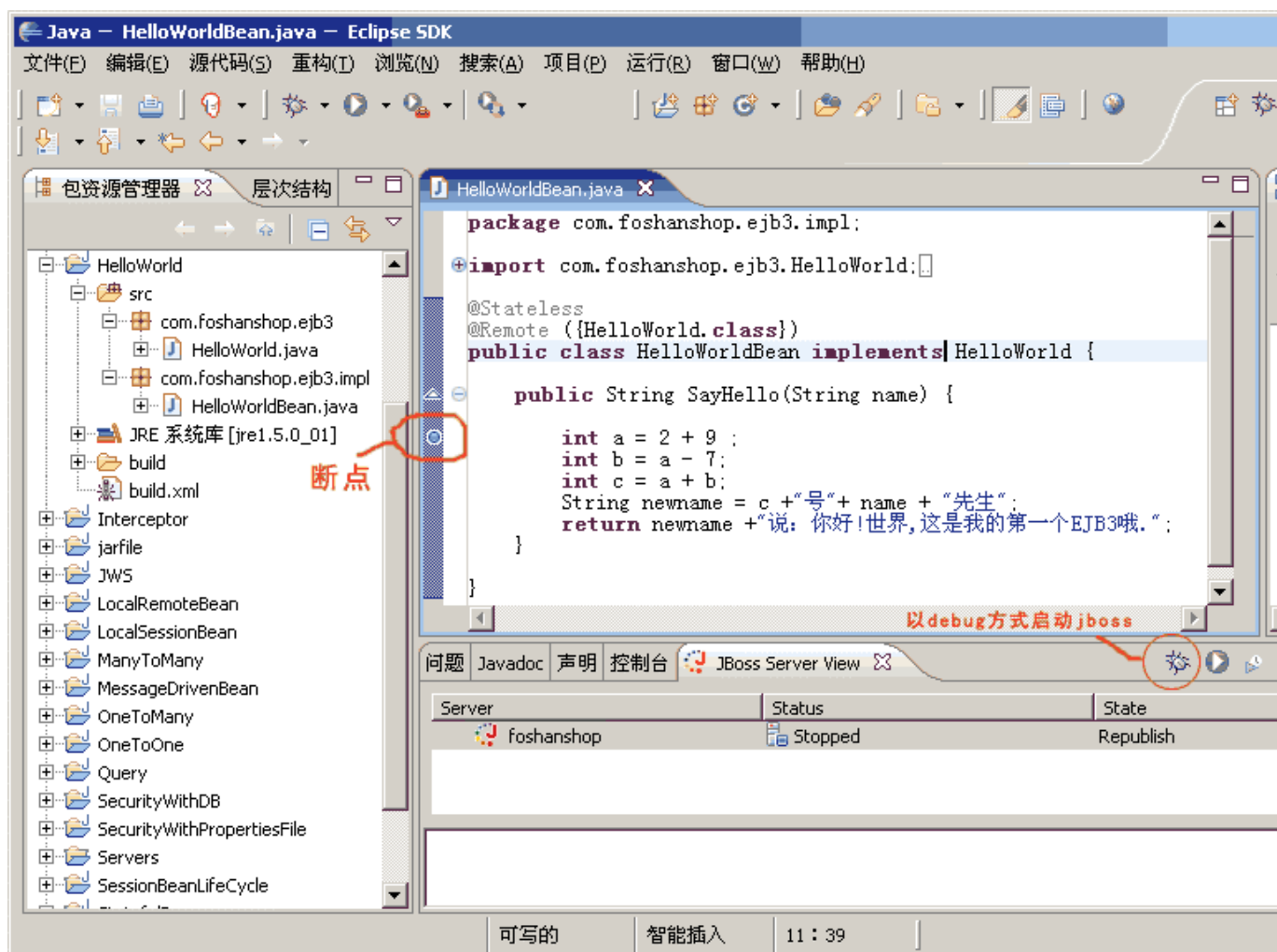


点击“完成”，“Jboss Server View”视图如下图所示：

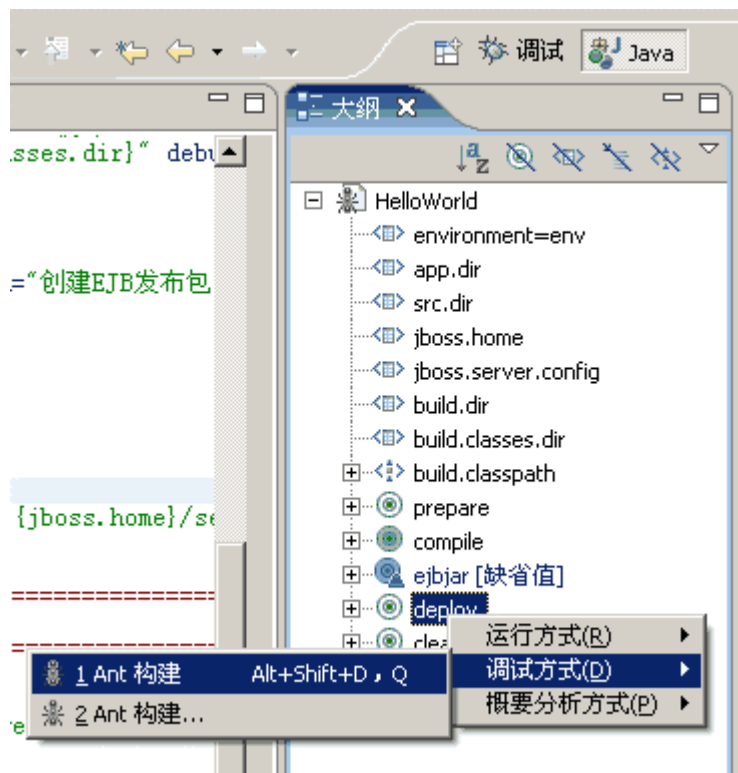


此时的服务器处于 **Stopped** 状态，你需要调试 EJB3 时可以以 **Debug** 方式启动 Jboss，方法是：在服务器 **foshanshop** 上右击鼠标，在跳出的属性菜单中点击“**Debug**”，或者点击“**Jboss Server View**”视图右边的小昆虫图标。控制台会输出 Jboss 的启动信息，如果启动出错，先看看在 DOS 窗口下以命令行方式(run.bat)能否启动 Jboss，如果 DOS 窗口下能正常启动 Jboss，那估计是你在建立 Jboss Server 的配置中出错，除了检查 Jboss 安装目录的选择是否正确之外，还要重点检查 Jboss 配置项选择是否正确。

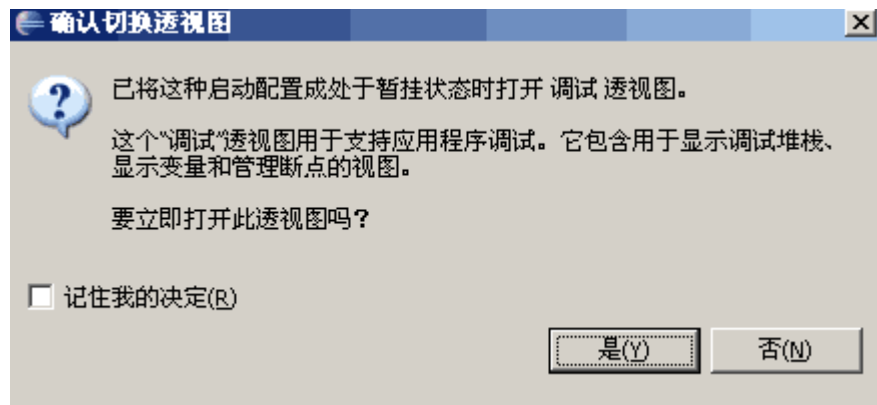
经过上面的步骤，JBoss server 的调试环境已经搭建好，下面我们就以 **HelloWorld** 项目为例介绍 EJB3 的调试。我们按照上面章节的步骤恢复 **HelloWorld** 项目的开发环境。因为 **SayHello(String name)** 方法内部代码不多，为了有代码可调试，我们在方法内部随便添加几行代码，并把断点设在代码的第一行，如下图：



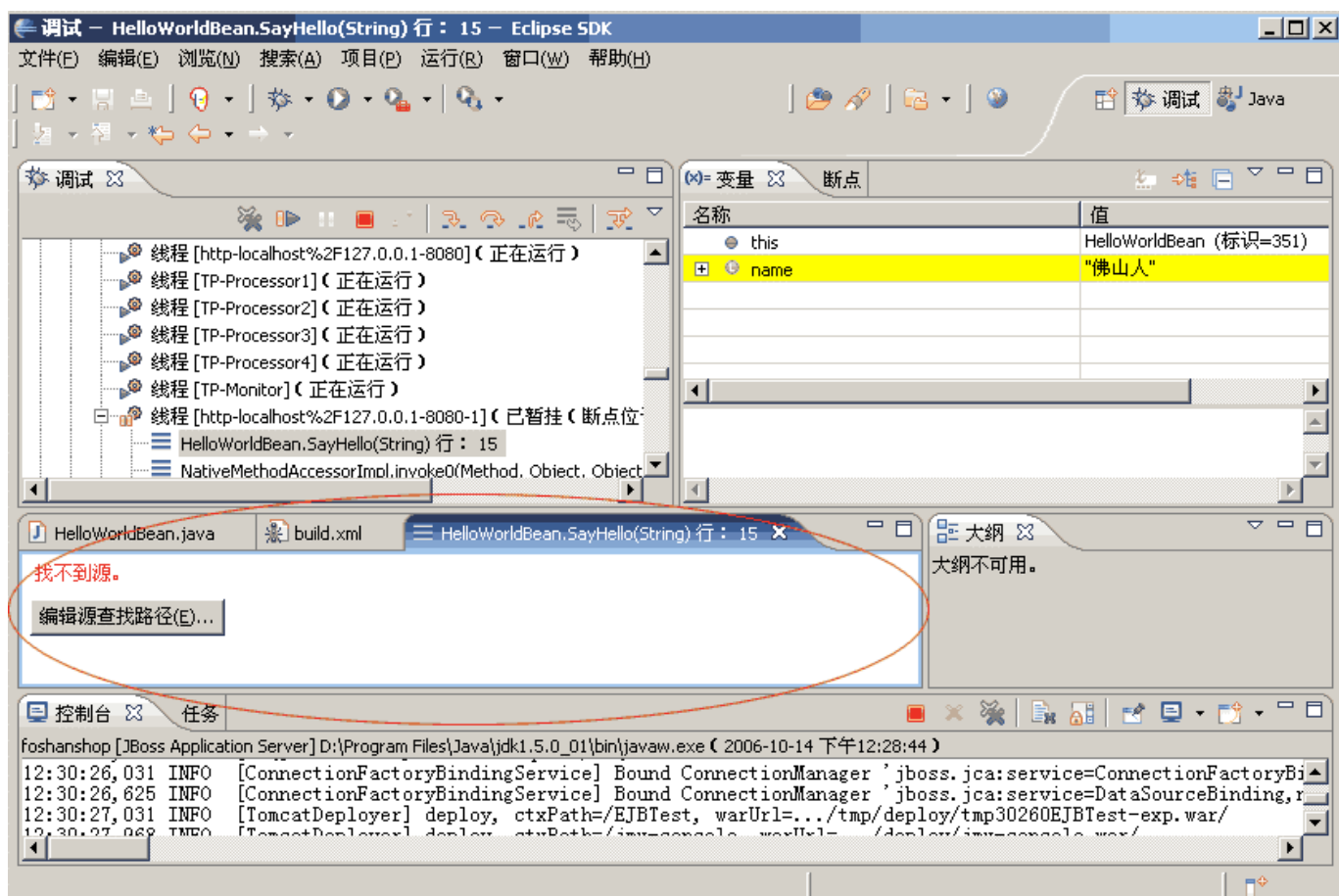
接着我们点击“Jboss Server View”视图右边的小昆虫图标，以 Debug 方式启动名为 foshanshop 的 Jboss Server。在“控制台”观察启动是否完成，如果启动完成，我们就打开 HelloWorld 项目下的 build.xml 文件，在“大纲”视图中以调试方式执行 Ant 的 deploy 任务，如下图：



执行“Ant 构建”后，在控制台将会输出 Ant 的任务信息及 HelloWorld 在 Jboss 中的发布信息，当 HelloWorld 发布完成后，我们在浏览器上输入 <http://localhost:8080/EJBTest/Test.jsp> (注意：确保已经发布了 Web 应用 EJBTest.war)，Eclipse 将会提示是否切换到调试透视图，如下图，



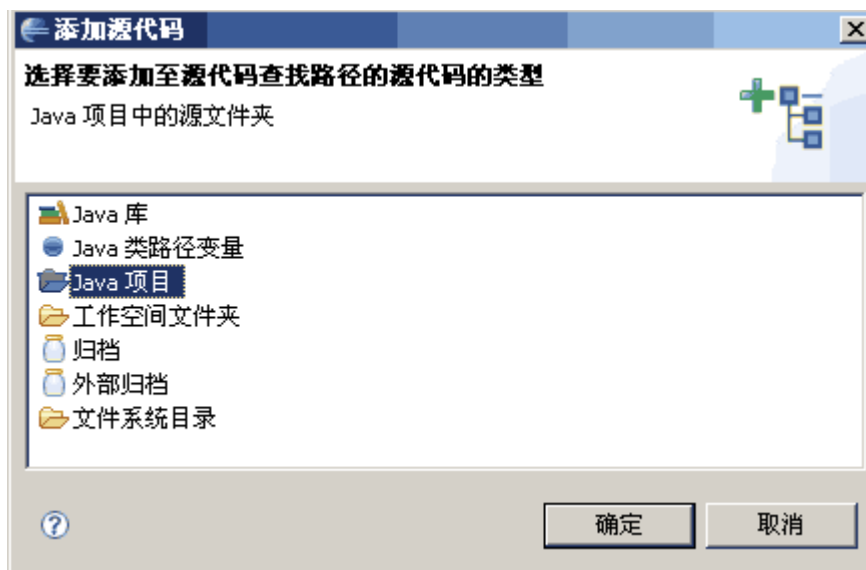
这里点“是”。如果你不想老出现这个对话框，可以勾选上“记住我的决定”。接着将出现如下窗口，提示找不 class 对应的 java 源文件。



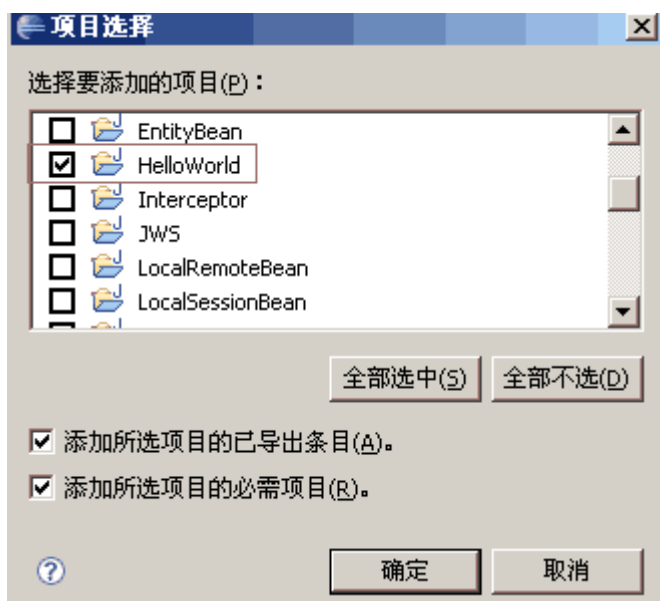
此时你可以在上图点击“编辑源查找路径(E)”，出现“编辑源查找路径”窗口，如下图：



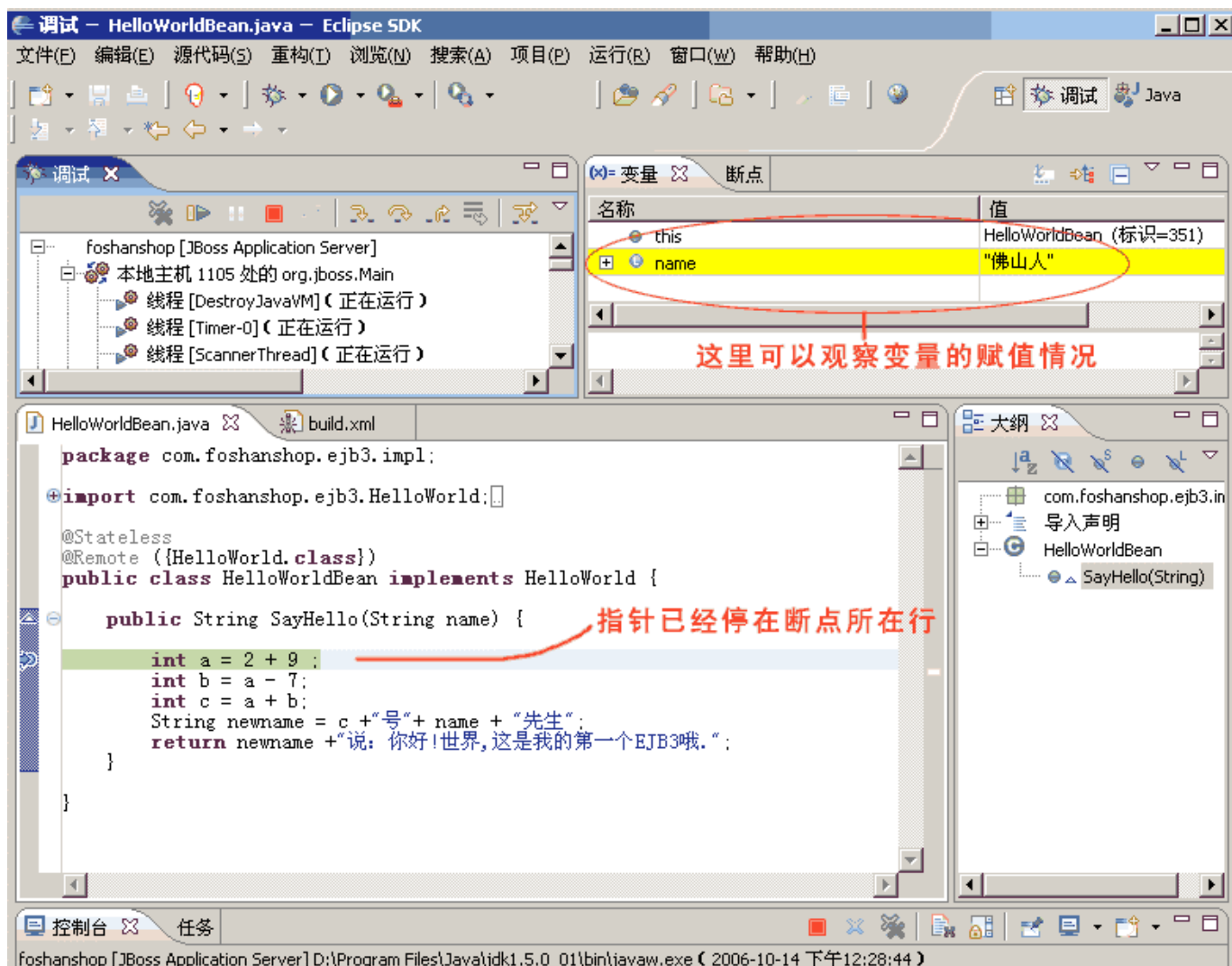
在上图中点击“添加(A)”，出现“添加源代码”窗口，如下图：



在上图我们选择从“java 项目”寻找对应的源代码，如果你的源代码不在 eclipse 的项目中，你可以通过“文件系统目录”进行寻找，把路径指向源代码所在的 src 目录。选择“java 项目”，点击“确定”后出现下面对话框：



在上图中我们勾选 HelloWorld 项目，点击“确定”。此时将回到调试窗口界面，如下图：



上图中可以看到调试指针已经落在了断点所在行,此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,单步跳过,单步返回,运行至行等调试方式,这里就不作详细解说了,想了解的朋友可以参考 Eclipse 关于调试方面的资料。

在这里需要说明下,调试指针(即绿色光亮行)的移动有时候比较慢,点击“单步跳入”等调试按钮十多秒后,调试指针的移动才有反应。

为了方便调试 EJB3,建议使用 Ant 执行编译、打包、发布等操作。调试 ejb3 时不需要反复重启 Jboss Server。

3.10 单元测试

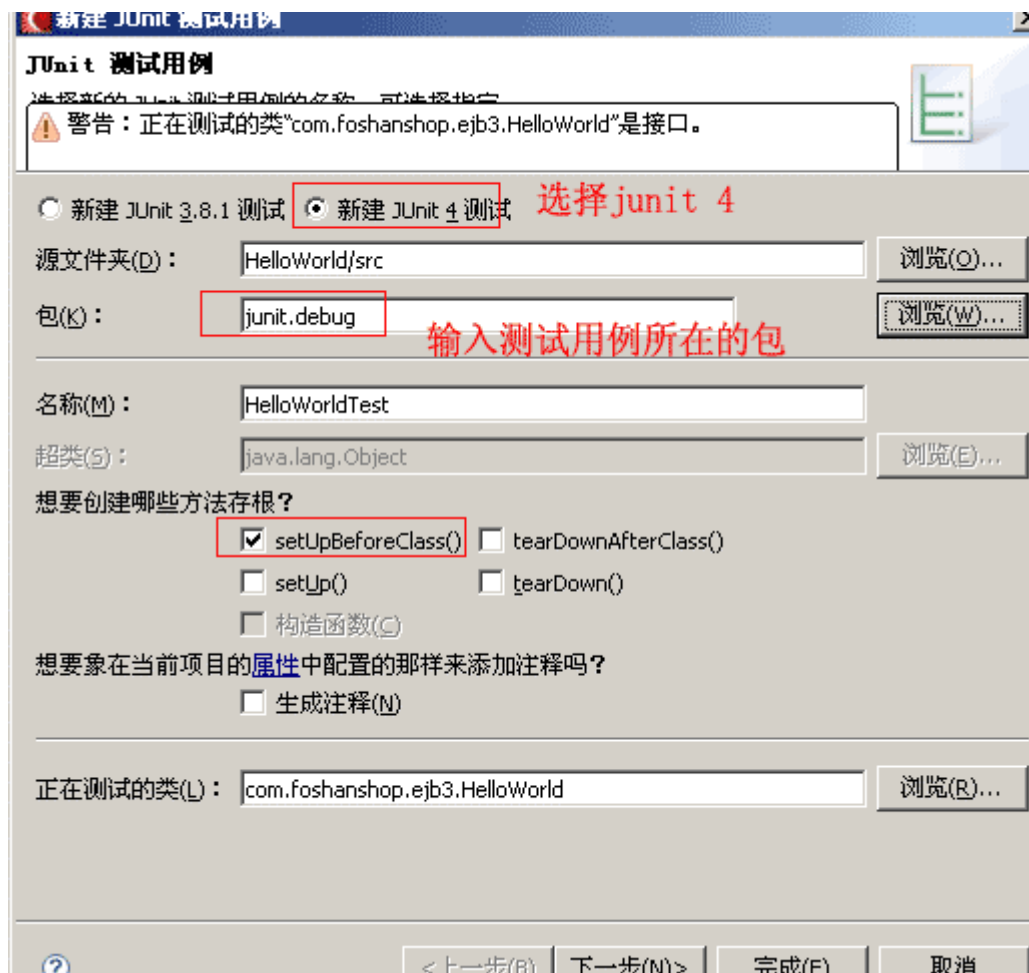
你可以对 ejb3 进行单元测试,下面以 HelloWorld 为例,介绍在 eclipse 中单元测试的步骤。

第一步:添加 Junit 依赖库,可以通过右击 HelloWorld 项目,选择“属性”,在出现的属性窗口左边选择“java 构建路径”,在右边点击“库”标签,点击“添加库(B)”,在出现的窗口中选择“junit”,版本选择“junit 4”。

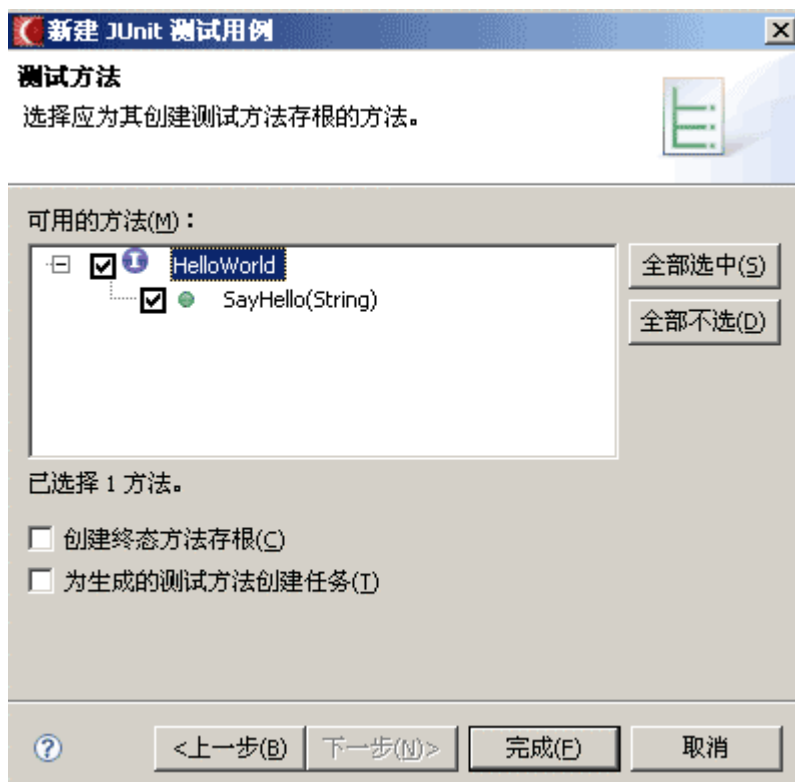
第二步:在需要测试的类(本例是 HelloWorld.java)上面点击右键,在出现的属性菜单中点击“新建”-“junit 测试用例”,如下图:



在出现的属性窗口中输入以下信息：



点击“下一步”，在出现的窗口中选择需要单元测试的方法：



点击“完成”，生成的测试用例如下：

```
package junit.debug;
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.debug;
import static org.junit.Assert.*;
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 {
    helloworld = (HelloWorld)EJBFactory.getEJB("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);
}
}

```

上面用到了 EJBFactory 类，他的实现代码如下：

```

package junit.debug;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.NamingException;

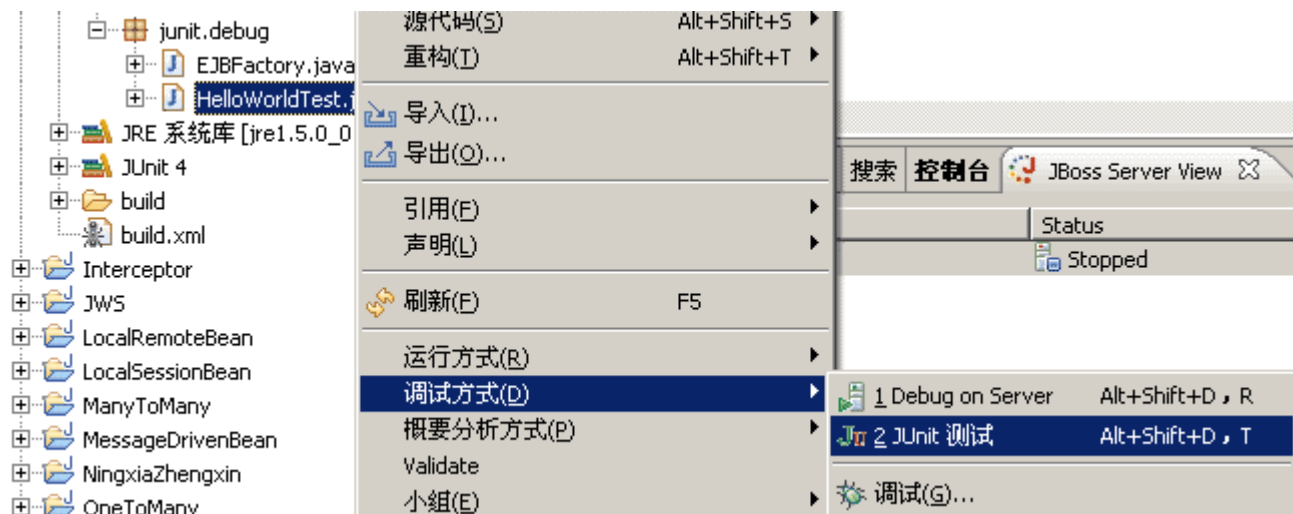
public class EJBFactory {
    public static Object getEJB(String jndipath) {
        try {
            Properties props = new Properties();

            props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
            props.setProperty("java.naming.provider.url", "localhost:1099");
            props.setProperty("java.naming.factory.url.pkgs",
"org.jboss.naming:org.jnp.interfaces");
            /*
            props.setProperty("java.naming.factory.initial",
"com.sun.enterprise.naming.SerialInitContextFactory");
            props.setProperty("java.naming.factory.url.pkgs", "com.sun.enterprise.naming");
            props.setProperty("java.naming.provider.url", "localhost:3700");
            props.setProperty("java.naming.factory.state",
"com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
            */
            InitialContext ctx = new InitialContext(props);
            return ctx.lookup(jndipath);
        } catch (NamingException e) {
            e.printStackTrace();
        }
        return null;
    }
}

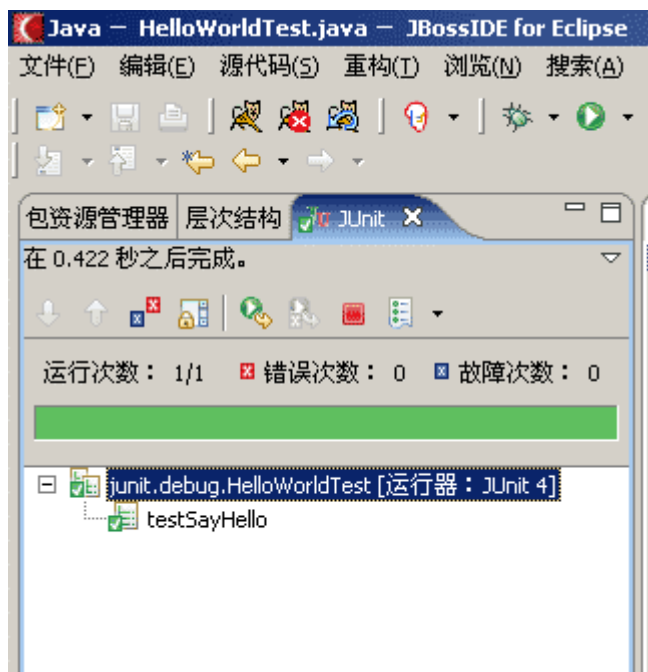
```

通过 **JUnit** 方式运行生成的测试用例，方法如下：

首先发布运行 **helloworld EJB**，然后在 **eclipse** 中右击 **HelloWorldTest** 文件，在属性菜单中点击“调试方式”——“JUnit 测试”，如下图：



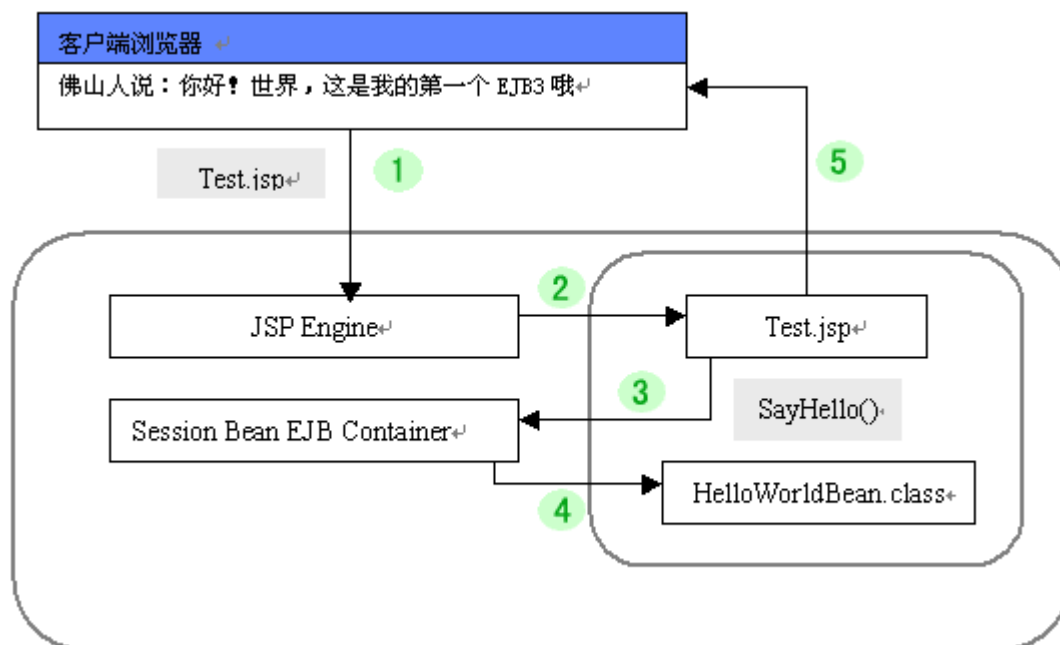
测试用例运行的结果如下：



绿色的进度条提示我们，测试运行通过了。关于 **JUnit** 更多知识请参考相关教程，这里不作详解。

第四章 会话 Bean(Session Bean)

Session Bean 用于实现业务逻辑，它分为有状态 bean 和无状态 bean。每当客户端请求时，容器就会选择一个 **Session Bean** 来为客户端服务。**Session Bean** 可以直接访问数据库，但更多时候，它会通过 **Entity Bean** 实现数据访问。**Session Bean** 作为业务处理对象出现在各种应用体系结构中，下面是本教程 **HelloWorld** 的应用结构：



- 1> 浏览器请求 Test.jsp 文件
- 2> 应用服务器的 JSP 引擎编译 Test.jsp
- 3> Test.jsp 通过 JNDI 查找 HelloWorld EJB, 获得 EJB 的 stub (代理存根), 调用存根的 SayHello() 方法, EJB 容器截获方法调用
- 4> EJB 容器调用 HelloWorld 实例的 SayHello() 方法.

因为客户端需要通过 JNDI 查找 EJB, 那么 JNDI 是什么?

JNDI(The Java Naming and Directory Interface, Java 命名和目录接口) 是一组在 Java 应用中访问命名和目录服务的 API。为开发人员提供了查找和访问各种命名和目录服务的通用、统一的方式。借助于 JNDI 提供的接口, 能够通过名字定位用户、机器、网络、对象服务等。

命名服务: 就像 DNS 一样, 通过命名服务器提供服务, 大部分的 J2EE 服务器都含有命名服务器。

目录服务: 一种简化的 RDBMS 系统, 通过目录具有的属性保存一些简单的信息。目录服务通过目录服务器实现, 比如微软 ACTIVE DIRECTORY 等。

JNDI 的好处:

- (1) 包含大量命名和目录服务, 可以使用相同 API 调用访问任何命名或目录服务。
- (2) 可以同时连接多个命名和目录服务。
- (3) 允许把名称同 JAVA 对象或资源关联起来, 不必知道对象或资源的物理 ID。
- (4) 使用通用接口访问不同种类的目录服务
- (5) 使得开发人员能够集中使用 and 实现一种类型的命名或目录服务客户 API 上。

什么是上下文: 由 0 或多个绑定构成。比如 java/MySQL, java 为上下文 (context), MySQL 为命名

什么是子上下文 (subContext), 上下文下的上下文。比如 MyJNDITree/ejb/helloBean, ejb 为子上下文。

JNDI 编程过程

因为 JNDI 是一组接口, 所以我们只需根据接口规范编程就可以。要通过 JNDI 进行资源访问, 我们必须设置初始

化上下文的参数，主要是设置 JNDI 驱动的类型名(`java.naming.factory.initial`)和提供命名服务的 URL (`java.naming.provider.url`)。因为 Jndi 的实现产品有很多。所以 `java.naming.factory.initial` 的值因提供 JNDI 服务器的不同而不同，`java.naming.provider.url` 的值包括提供命名服务的主机地址和端口号。下面是访问 Jboss 服务器的例子代码：

```
Properties props = new Properties();
props.setProperty("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
props.setProperty("java.naming.provider.url", "localhost:1099");
InitialContext = new InitialContext(props);
HelloWorld helloworld = (HelloWorld) ctx.lookup("HelloWorldBean/remote");
```

下面是访问 Sun 应用服务器的例子代码：

```
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");
```

下面是访问 Weblogic10 应用服务器的例子代码：

```
Properties props = new Properties();
props.setProperty("java.naming.factory.initial", "weblogic.jndi.WLInitialContextFactory");
props.setProperty("java.naming.provider.url", "t3://localhost:7001");
InitialContext = new InitialContext(props);
HelloWorld helloworld = (HelloWorld) ctx.lookup("HelloWorldBean
#com.foshanshop.ejb3.HelloWorld");
```

JBOSS 环境下 JNDI 树的命名约定：

(1) `java:copm`

这个上下文环境和其子上下文环境仅能被与之相关的特定应用组件访问和使用

(2) `java:`

子上下文环境和绑定的对象只能被 Jboss 服务器虚拟机内的应用访问

(3) 其他上下文环境

只要实现序列化就可以被远程用户调用。

4.1 Stateless Session Beans（无状态 bean）开发

无状态会话 Bean 主要用来实现单次使用的服务，该服务能被启用许多次，但是由于无状态会话 Bean 并不保留任何有关状态的信息，其效果是每次调用提供单独的使用。在很多情况下，无状态会话 Bean 提供可重用的单次使用服务。

尽管无状态会话 Bean 并不为特定的客户维持会话状态，但会有一个以其成员变量形式表示的过度状态。当一个客户调用无状态会话 Bean 的方法时，Bean 的成员变量的值只表示调用期间的一个过度状态。当该方法完成时，这个状态不再保留。

除了在方法调用期间，所有的无状态会话 Bean 的实例都是相同的，允许 EJB 容器给任何一个客户赋予一个实例。许多应用服务器利用这个特点，共享无状态会话 Bean 以获得更好的性能。

由于无状态会话 Bean 能够支持多个客户，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 更具优势的是其性能，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。

4.1.1 开发只存在 Remote 接口的无状态 Session Bean

步骤如下：

第一步：要定义一个会话 Bean，首先需要定义一个包含他所有业务方法的接口。这个接口不需要任何注释，就像普通的 java 接口那样定义。调用 EJB 的客户端通过使用这个接口引用从 EJB 容器得到的会话 Bean 对象 stub。接口的定义如下：

HelloWorld.java

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

第二步：实现上面的接口并加入两个注释 @Stateless, @Remote，第一个注释定义这是一个无状态会话 Bean，第二个注释指明这个无状态 Bean 的 remote 接口。在使用这两个注释时需要使用一些 EJB 的类包，这些类包都可以在 jboss 安装目录的 client, /server/default/deploy/jboss-aop-jdk50.deployer, /server/default/deploy/ejb3.deployer, /lib/endorsed 等文件夹下找到，或者在源代码的 Lib 文件夹下获得(下载地址：<http://www.foshanshop.net/>)。

经过上面的步骤一个只存在 Remote 接口的无状态会话 Bean 就开发完成。无状态会话 Bean 是一个简单的 POJO(纯粹的面向对象思想的 java 对象)，EJB3.0 容器自动地实例化及管理这个 Bean。下面是 HelloWorld 会话 Bean 的实现代码：

HelloWorldBean.java。实现类的命名规则是：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.";
    }
}
```

HelloWorld 会话 Bean 开发完了，现在我们把她发布到 Jboss 中。在发布前需要把她打成 Jar 包或 EAR 包 (如何打包请参照 [第三章:如何进行 EJB 打包](#))。

打完包后, 启动 Jboss, 把发布包拷贝到[jboss 安装目录]\server\default\deploy\目录下。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。

```
12:00:21.020 INFO [Server] JBoss <MX MicroKernel> [4.0.4.GA <build: CUSag=JBos
s_4_0_4_GA date=200605151000>] Started in 47s:250ms
12:00:52.645 INFO [Ejb3Deployment] EJB3 deployment time took: 625
12:00:52.863 INFO [JmxKernelAbstraction] installing MBean: jboss.j2ee:jar=Hello
World.jar,name=HelloWorldBean,service=EJB3 with dependencies:
12:00:53.566 INFO [EJBContainer] STARTED EJB: com.foshanshop.ejb3.inpl.HelloWor
ldBean ejbName: HelloWorldBean
12:00:53.738 INFO [EJB3Deployer] Deployed: file:/C:/JavaServer/jboss/server/all
/deploy/HelloWorld.jar
```

一旦发布成功, 你就可以在 jboss 的管理平台查看她们的 JNDI 名, 输入下面 URL

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

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

jboss

- [database=localDB,service=Hypersonic](#)
- [name=PropertyEditorManager,type=Service](#)
- [name=SystemProperties,type=Service](#)
- [partitionName=DefaultPartition,service=DistributedReplicant](#)
- [partitionName=DefaultPartition,service=DistributedState](#)
- [readonly=true,service=invoker,target=Naming,type=http](#)
- [service=AttributePersistenceService](#)
- [service=ClientUserTransaction](#)
- [service=DefaultPartition](#)
- [service=HAJNDI](#)
- [service=HASessionState](#)
- [service=JNDIView](#)
- [service=KeyGeneratorFactory,type=HiLo](#)

查看 JNDI 名称

在出现的页面中, 找到 “List of MBean operations:” 栏。点击 “Invoke” 按钮, 出现如下界面:

```
+ QueueConnectionFactory (class: org.jboss.naming.LinkRefPair)
+ UUIDKeyGeneratorFactory (class: org.jboss.ejb.plugins.keygenerator.uuid.UUIDKeyGeneratorFactory)
+ HelloWorldBean (class: org.jnp.interfaces.NamingContext)
| + remote (proxy: $Proxy66 implements interface com.foshanshop.ejb3.HelloWorld,interface org.jbc
```

这个就是EJB HelloWorld的JNDI名, 他的组成格式是: 上层名称/下层名称/... 本例中的JNDI名是: HelloWorldBean/remote

在上图中可以看见 HelloWorld 会话 Bean 的 JNDI(Java Naming and Directory Interface)路径, JNDI 路径名的组成规则是“上层名称/下层名称“, 每层之间以”/”分隔。HelloWorld 会话 Bean 的 JNDI 路径名是: HelloWorldBean/remote。

HelloWorld 会话 Bean 发布成功后, 接下来介绍客户端如何访问她。

当一个无状态会话 Bean 发布到 EJB 容器时, 容器就会为她创建一个对象 stub, 并把她注册进容器的 JNDI 目录, 客户端代码使用她的 JNDI 名从容器获得他的 stub。通过这个 stub, 客户端可以调用她的业务方法。例子代码如下:

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");
        props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

        InitialContext ctx;
        try {
            ctx = new InitialContext(props); // 如果客户端和 jboss 运行在同一个 jvm, 不需要传入 props
            HelloWorld helloworld = (HelloWorld) ctx.lookup("HelloWorldBean/remote");
            out.println(helloworld.SayHello("佛山人"));
        } catch (NamingException e) {
            out.println(e.getMessage());
        }
    }
}
%>

```

ctx = new InitialContext(props); 是设置 JNDI 访问的环境，本例使用 JBoss，所以把环境设为 JBoss 的上下文环境。如果客户端运行在 jboss，不需要传入 props。本例可以省略传入 props 参数。

在这里作者要重点说明一下 Jboss EJB JNDI 名称默认的命名规则，命名规则如下：

1> 如果 EJB 打包进后缀为 *.ear 的 J2EE 发布文件，默认的 JNDI 路径名称是

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

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

例：EJB HelloWorld 打包进名为 HelloWorld.ear 的 J2EE 应用，访问她远程接口的 JNDI 名是：
HelloWorld/HelloWorldBean/remote

2> 如果 EJB 应用打包成后缀为 *.jar 的发布文件，默认的 JNDI 路径名称是

访问本地接口：EJB-CLASS-NAME/local

访问远程接口：EJB-CLASS-NAME/remote

例：HelloWorld 应用打包成 HelloWorld.jar 文件，访问她远程接口的 JNDI 名称是：HelloWorldBean/remote

另外有一点要注意：EJB-CLASS-NAME 是不带包名的，如 com.foshanshop.ejb3.impl.HelloWorldBean 只需取 HelloWorldBean。

目前网上很多教材获取 JNDI 路径名的方式不适用在 jboss 下，如：

HelloWorld helloworld = (HelloWorld) ctx.lookup(HelloWorld.class.getName()); 这种方式适用于 Sun Application Server 及 glassfish

我们把上面的客户端应用打成 war 文件。然后把她拷贝到 “[jboss 安装目录]\server\default\deploy” 目录下。如果 war 文件的文件名为 EJBTest.war，我们可以通过 http://localhost:8080/EJBTest/Test.jsp 访问客户端。

本例子的 EJB 源代码在 HelloWorld 文件夹（源代码下载：<http://www.foshanshop.net/>），项目使用到的类库在上级目录 lib 文件夹下。要恢复 HelloWorld 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子（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/Test.jsp 访问客户端。

4.1.2 开发只存在 Local 接口的无状态 Session Bean

开发只存在 Local 接口的无状态会话 Bean 的步骤和上节开发只存在 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 注释指明实现的接口是远程接口，后者使用 @Local 注释指明实现的接口是本地接口。当 @Local 和 @Remote 注释都不存在时，会话 Bean 实现的接口默认为 Local 接口。如果在本机调用 EJB（确保客户端与 EJB 容器运行在同一个 JVM），采用 Local 接口访问 EJB 优于 Remote 接口，因为 Remote 接口访问 EJB 需要经过远程方法调用(RPCs)环节，而 Local 接口访问 EJB 直接从 JVM 中返回 EJB 的引用。下面是只存在 Local 接口的无状态会话 Bean 代码。

LocalHelloWorld.java

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

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";
    }
}
```

客户端调用代码:

LocalSessionBeanTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.LocalHelloWorld, 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");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx;
    try {
        ctx = new InitialContext(props);
        LocalHelloWorld helloworld = (LocalHelloWorld)
ctx.lookup("LocalHelloWorldBean/local");
```

```

        out.println(helloworld.SayHello("佛山人"));
    } catch (NamingException e) {
        out.println(e.getMessage());
    }
}

```

```
%>
```

上面的客户端代码打包成 war 文件发布到 jboss 中。如果你试图在独立的 Tomcat 服务器中执行客户端代码（如何在独立的 Tomcat 环境中调用 EJB 请参照 [第二章：在独立的 Tomcat 中调用 EJB](#)），你将获得如下例外：

`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 VM(虚拟内存堆)。相对于发布到 jboss deploy 目录下的客户端应用而言，他与 EJB 容器运行在同一个 VM。如果客户端与 EJB 容器在不同的 VM，只能通过其 Remote 接口进行访问。

本例子的 EJB 源代码在 LocalSessionBean 文件夹（源代码下载：<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 LocalSessionBean 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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 EJCTest 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/EJCTest/LocalSessionBeanTest.jsp> 访问客户端。

4.1.3 开发存在 Remote 与 Local 接口的无状态 Session Bean

在实际应用中，一个无状态 Session Bean 都应该实现 Remote 与 Local 接口。当会话 Bean 的某些方法只供 EJB 容器内部调用而不对外暴露时，可以把他定义在 Local 接口。本节介绍如何开发一个具有 Remote 与 Local 接口的无状态 Session Bean。开发前先介绍一下两个接口具有的方法，两者都含有方法 `Add(int a, int b)`，而 Local 接口含有一个自己的方法：`getResult()`。下面是例子的源代码。

定义远程接口：Operation.java

```

//author:lihuoming
package com.foshanshop.ejb3;
public interface Operation {
    public int Add(int a, int b);
}

```

定义本地接口：LocalOperation.java，本地接口具有远程接口的所有方法，另外有自己的方法 `getResult()`

```

package com.foshanshop.ejb3;
public interface LocalOperation extends Operation {
    public int getResult();
}

```

实现本地接口与远程接口的会话 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;
    private int addresult = 0;

    public int Add(int a, int b) {
        addresult = a + b;
        return addresult;
    }

    public int getResult() {
        total += addresult;
        return total;
    }
}

```

JSP 客户端: OperationBeanTest.jsp

```

<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.Operation, com.foshanshop.ejb3.LocalOperation,
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");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx = new InitialContext(props);
    try {
        Operation operation = (Operation) ctx.lookup("OperationBean/remote");
        out.println("通过远程接口调用EJB成功");
        out.println("<br>(通过远程接口调用EJB) 相加的结果是: "+ operation.Add(1,1));
    } 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 A = (LocalOperation) ctx.lookup("OperationBean/local");

```

```

        out.println("<br>(通过本地接口调用EJB)调用A.Add()的结果是：" + A.Add(1,1));
        out.println("<br>调用A.getResult()的结果是：" + A.getResult());

        LocalOperation B = (LocalOperation) ctx.lookup("OperationBean/local");
        out.println("<br>(通过本地接口调用EJB)调用B.Add()的结果是：" + B.Add(1,1));
        out.println("<br>调用B.getResult()的结果是：<font color=red>" + B.getResult() +
"</font>");
    } catch (Exception e) {
        out.println("<br>本地接口调用失败");
    }
}
%>

```

把会话 Bean 及客户端 Web 应用发布到 jboss,客户端的调用结果如下:

```

通过远程接口调用 EJB 成功
(通过远程接口调用 EJB)相加的结果是: 2

=====

(通过本地接口调用 EJB)调用 A.Add()的结果是: 2
调用 A.getResult()的结果是: 2
(通过本地接口调用 EJB)调用 B.Add()的结果是: 2
调用 B.getResult()的结果是: 4

```

细心的你可能会发现调用 Local 接口时,两次累加的结果都不一样,一个是 2,一个是 4。调用本地接口的客户端代码片段如下:

```

//通过本地接口调用 EJB
LocalOperation A = (LocalOperation) ctx.lookup("OperationBean/local");
out.println("<br>(通过本地接口调用 EJB)调用 A.Add()的结果是：" + A.Add(1,1));
out.println("<br>调用 A.getResult()的结果是：" + A.getResult());
LocalOperation B = (LocalOperation) ctx.lookup("OperationBean/local");
out.println("<br>(通过本地接口调用 EJB)调用 B.Add()的结果是：" + B.Add(1,1));
out.println("<br>调用 B.getResult()的结果是：<font color=red>" + B.getResult() +
"</font>");

```

你可能认为 A 和 B 得到的应该是两个不同的对象,根据 OperationBean.java 的代码判断,他们的结果都是 2 才对。为何一个为 2,另一个为 4 呢。

这是因为 Stateless Session Bean 不负责记录使用者状态,Stateless Session Bean 一旦实例化就被加进会话池中,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,它可能依然存在于会话池中,供其他用户调用。如果它有自己的属性(变量),那么这些变量就会受到所有调用它的用户的影响。

本例子的 EJB 源代码在 LocalRemoteBean 文件夹(源代码下载:<http://www.foshanshop.net/>),项目中使用到的类库在上级目录 lib 文件夹下。要恢复 LocalRemoteBean 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”,要发布本例子 EJB (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 EJCTest 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/EJCTest/OperationBeanTest.jsp> 访问客户端。

在 Jboss 网站看到,在 EJB3.0 RC9 以上版本中 Remote 及 Local 接口可以指向同一个业务接口,这样客户端就不会因调用接口的不同而来回切换业务接口类。当然这种使用场合是在 Remote 和 Local 的接口方法相同的情况下。如:

session bean 片断:

```
@Stateless
@Remote({Same.class})
@Local({Same.class})
public class SameBean implements Same {
    public String say(String who) {
        return who + "说: 我就爱使用同一个业务接口";
    }
}
```

JSP 客户端

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.Same, 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");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx;
    try {
        ctx = new InitialContext(props);
        Same same = (Same) ctx.lookup("SameBean/remote");
        out.println(same.say("佛山人"));
        out.println("<br>-----<br>");
        Same same2 = (Same) ctx.lookup("SameBean/local");
        out.println(same2.say("黎明"));
    } catch (NamingException e) {
        out.println(e.getMessage());
    }

%>
```

上面的 Jsp 客户端是运行在 Jboss 中的, 如果你想运行在独立的 tomcat 中, 你只能调用其 remote 接口。

需要独立运行在 tomcat 中, 你可以参考第二章“在单独的 Tomcat 中调用 EJB”。

4.2 Stateful Session Beans (有状态 bean) 开发

有状态 Bean 是一个可以维持自身状态的会话 Bean。每个用户都有自己的一个实例, 在用户的生存期内, Stateful Session Bean 保持了用户的信息, 即“有状态”; 一旦用户灭亡 (调用结束或实例结束), Stateful Session Bean 的生命期也告结束。即每个用户最初都会得到一个初始的 Stateful Session Bean。

Stateful Session Bean 的开发步骤与 Stateless Session Bean 的开发步骤相同。先定义业务接口, 例子代码如下:

MyAccount.java

```

package com.foshanshop.ejb3;
import java.io.Serializable;
public interface MyAccount extends Serializable {
    public int Add(int a, int b);
    public int getResult() ;
}

```

大家注意 stateful session bean 必须实现 Serializable 接口，这样 EJB 容器才能在她们不再使用时序列化存储她们的状态信息。

MyAccountBean.java

```

package com.foshanshop.ejb3.impl;
import javax.ejb.Remote;
import javax.ejb.Stateful;
import com.foshanshop.ejb3.MyAccount;

@SuppressWarnings("serial")
@Stateful
@Remote(MyAccount.class)
public class MyAccountBean implements MyAccount {
    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;
    }
}

```

上面 MyAccountBean 直接实现 MyAccount 接口，通过 @Stateful 注释定义这是一个有状态会话 Bean，@Remote 注释指明有状态 Bean 的 remote 接口。@SuppressWarnings("serial") 注释屏蔽缺少 serialVersionUID 定义的警告。

下面是 MyAccountBean 的 JSP 客户端代码：

StatefulBeanTest.jsp

```

<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.MyAccount, 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");
    props.setProperty("java.naming.factory.url.pkgs",
"org.jboss.naming:org.jnp.interfaces");

```



```

    InitialContext ctx = new InitialContext(props);
    try {
        MyAccount A = (MyAccount) ctx.lookup("MyAccountBean/remote");
        out.println("调用A.Add() 的结果是: " + A.Add(1, 1));
        out.println("<br>调用A.getResult() 的结果: " + A.getResult());
        out.println("<br>=====");
        MyAccount B = (MyAccount) ctx.lookup("MyAccountBean/remote");
        out.println("<br>调用B.Add() 的结果是: " + B.Add(1, 1));
        out.println("<br>调用B.getResult() 的结果: " + B.getResult());
    } catch (Exception e) {
        out.println(e.getMessage());
    }
%>

```

上面 JSP 客户端, 有 A 和 B 两个用户在使用 MyAccountBean, 他们的执行结果如下:

```

调用 A.Add() 的结果是: 2
调用 A.getResult() 的结果: 2
=====
调用 B.Add() 的结果是: 2
调用 B.getResult() 的结果: 2

```

因为 stateful session bean 的每个用户都有自己的一个实例, 所以两者对 stateful session bean 的操作不会影响对方。另外注意: 如果后面需要操作某个用户的实例, 你必须在客户端缓存 Bean 的 Stub 对象(JSP 通常的做法是用 Session 缓存), 这样在后面每次调用中, 容器才知道要提供相同的 bean 实例。

本例子的 EJB 源代码在 StatefulBean 文件夹 (源代码下载:<http://www.foshanshop.net/>), 项目中使用到的类库在上级目录 lib 文件夹下。要恢复 StatefulBean 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”, 要发布本例子 EJB (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 EJCTest 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/EJCTest/StatefulBeanTest.jsp> 访问客户端。

4.3 Stateless Session Bean 与 Stateful Session Bean 的区别

这两种 Session Bean 都可以将系统逻辑放在方法之中执行, 不同的是 Stateful Session Bean 可以记录呼叫者的状态, 因此一个使用者会有自己的一个实例。Stateless Session Bean 虽然也是逻辑组件, 但是他却不负责记录使用者状态, 也就是说当使用者呼叫 Stateless Session Bean 的时候, EJB 容器并不会寻找特定的 Stateless Session Bean 的实体来执行这个 method。换言之, 很可能数个使用者在执行某个 Stateless Session Bean 的 methods 时, 会是同一个 Bean 的实例在执行。从内存方面来看, Stateful Session Bean 与 Stateless Session Bean 比较, Stateful Session Bean 会消耗 J2EE Server 较多的内存, 然而 Stateful Session Bean 的优势却在于他可以维持使用者的状态。

4.4 如何改变 Session Bean 的 JNDI 名称

默认的 JNDI 命名规则前面已经介绍过, 但有些情况下需要自定义名称。在 Jboss 中要自定义 JNDI 名称, 可以使用 @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, 下面的代码展示了如何自定义 JNDI 名:


```
//author:lihuoming
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;
}
```

在 JSP 客户端调用上面 EJB 的代码片断如下:

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

在 weblogic10 中, 你可以通过 @Stateless.mappedName() 设置全局 JNDI 名称, 如:

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

客户端调用 EJB 的代码片断如下:

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

4.5 Session Bean 的生命周期

EJB 容器创建和管理 session bean 实例,有些时候,你可能需要定制 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 类中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:创建或销毁时)。通过使有下面所列的注释,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, EJB 2.1 中,所有的回调方法必须实现,即使是空的。EJB 3.0 中,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,使用了这个注释的方法会被立即调用。这个注释同时适用于有状态和无状态的会话 bean。

- **@PreDestroy:** 使用这个注释的方法会在容器从它的对象池中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的会话 bean。

- **@PrePassivate:** 当一个有状态的 session bean 实例空闲过长的时间,容器将会钝化(passivate)它,并把它的状态保存在缓存当中。使用这个注释的方法会在容器钝化 bean 实例之前调用。这个注释适用于有状态的会话 bean。当钝化后,又经过一段时间该 bean 仍然没有被操作,容器将会把它从存储介质中删除。以后,任何针对该 bean 方法的调用容器都会抛出例外。

- **@PostActivate:** 当客户端再次使用已经被钝化的有状态 session bean 时,新的实例被创建,状态被恢复。使用此注释的 session bean 会在 bean 的激活完成时调用。这个注释只适用于有状态的会话 bean。

- **@Init:** 这个注释指定了有状态 session 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。这取决于 bean 是如何创建的(细节请看 EJB 3.0 规范)。**@PostConstruct** 在**@Init**之后被调用。

另一个有用的生命周期方法注释是**@Remove**,特别是对于有状态 session bean。当应用通过存根对象调用使用了**@Remove**注释的方法时,容器就知道在该方法执行完毕后,要把 bean 实例从对象池中移走。

下面是这些生命周期方法注释在 LifecycleBean 中的一个示例:

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.Remove;
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("initialize() 方法被调用");
}

@PostConstruct
public void Construct () {
    System.out.println("Construct() 方法被调用");
}

@PreDestroy
public void exit () {
    System.out.println("exit() 方法被调用");
}

@PrePassivate
public void serialize () {
    System.out.println("serialize() 方法被调用");
}

@PostActivate
public void activate () {
    System.out.println("activate() 方法被调用");
}

@Remove
public void stopSession () {
    System.out.println("stopSession() 方法被调用");
    //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。
}
}

```

本例子的运行结果输出在 Jboss 控制台，请仔细观察。

下面是 LifeCycleBean 的 Remote 接口

LifeCycle.java

```

package com.foshanshop.ejb3;

public interface LifeCycle {
    public String Say();
    public void stopSession ();
}

```

下面是 Session Bean 的 JSP 客户端代码:

LifeCycleTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.LifeCycle, 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");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    try {
        LifeCycle lifecycle = (LifeCycle) session.getAttribute("lifecycle");
        if (lifecycle == null) {
            InitialContext ctx = new InitialContext(props);
            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>你可以执行会话Bean的stopSession方法, 把会话Bean实例从
对象池中移走. 在销毁这个会话Bean之前将会执行 @PreDestroy注释的方法<BR></font>");
        /*
        lifecycle.stopSession();
        */
    } catch (Exception e) {
        out.println(e.getMessage());
    }

%>
```

本例子的 EJB 源代码在 SessionBeanLifeCycle 文件夹 (源代码下载:<http://www.foshanshop.net/>), 项目中使用到的类库在上级目录 lib 文件夹下。要恢复 SessionBeanLifeCycle 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”, 要发布本例子 EJB (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 EJBTTest 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/EJBTTest/LifeCycleTest.jsp> 访问客户端。

4.6 拦截器(Interceptor)

拦截器可以监听程序的一个或所有方法。拦截器对方法调用流提供了细粒度控制。可以在无状态会话 bean、有

状态会话 bean 和消息驱动 bean 上使用它们。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。上面拦截器 HelloInterceptor 对 HelloChinaBean 中的所有方法进行监听。

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。用@AroundInvoke 注释指定的方法必须遵守以下格式:

public Object XXX(InvocationContext ctx) **throws** Exception
 XXX 代表方法名可以任意。

下面是 HelloChinaBean 的本地及远程业务接口

HelloChina.java

```

package com.foshanshop.ejb3;

public interface HelloChina extends HelloChinaRemote{
}

```

HelloChinaRemote.java

```

package com.foshanshop.ejb3;

public interface HelloChinaRemote {

    public String SayHello(String name);

    public String Myname();

}

```

下面是 Session Bean 的 JSP 客户端代码:

InterceptorTest.jsp

```

<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.HelloChinaRemote, 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");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx;
    try {
        ctx = new InitialContext(props);
        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());
    }
}

```

%>

除了可以在外部定义拦截器之外，还可以将 **Session Bean** 中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** 中如何定义拦截器。

HelloChinaBean.java

```

//author:lihuoming
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 EJB 源代码在 Interceptor 文件夹（源代码下载:<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 Interceptor 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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/InterceptorTest.jsp> 访问客户端。

4.7 依赖注入(dependency injection)

上面，你学到了如何开发耦合松散的服务组件。但是，为了存取那些服务对象，你需要通过服务器的 JNDI 来查找存根对象（session bean）或消息队列（MDB）。JNDI 查找是把客户端与实际的服务端实现解耦的关键步骤。但是，直接使用一个字符串来进行 JNDI 查找并不优雅。有这样几个原因：

- 客户端与服务端必须有一致的基于字符串的名字。它没有在编译时得到认证或在部署时得到检查。
- 从 JNDI 返回的服务对象的类型没有在编译时进行检查，有可能在运行时出现转换(casting)错误。
- 冗长的查找代码，有着自己的 try-catch 代码块，在应用之间是重复的和杂乱的

EJB 3.0，对任何 POJO,提供了一个简单的和优雅的方法来解耦服务对象和资源。使用@EJB 注释，你可以将 EJB 存根对象注入到任何 EJB 3.0 容器管理的 POJO 中。如果注释用在一个属性变量上，容器将会在它被第一次访问之前赋值给它。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。

Jboss4.0.5 版本中的@EJB 注释已从 javax.annotation 包移到 javax.ejb，如果本例子运行在 Jboss4.0.5 以下版本，将会出错。

下面的例子演示了怎样把 HelloWorldBean 无状态 session bean 的存根注入到 InjectionBean 类中。

InjectionBean.java

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

@Stateless
@Remote ({Injection.class})
public class InjectionBean implements Injection {
    @EJB (beanName="HelloWorldBean")
    HelloWorld helloworld;

    public String SayHello() {
        return helloworld.SayHello("注入者");
    }
}
```



```
}
```

@EJB 注释的 beanName 属性指定 EJB 的名称(如果没有设置过@Stateless 或@Stateful 的 name 属性, 默认为不带包名的类名), 他的另一个属性 mappedName 指定 EJB 的全局 JNDI 名。

下面的片断演示了如何使用 beanName 或 mappedName 属性查找 HelloWorldBean 会话 bean

```
public class InjectionBean implements Injection {
    @EJB (beanName="HelloWorldBean")
    //@EJB (mappedName="HelloWorldBean/remote")
    HelloWorld helloworld;
    ....
}
```

@EJB 注释如果被用在 JavaBean 风格的 setter 方法上时, 容器会在属性第一次使用之前, 自动地用正确的参数调用 bean 的 setter 方法。下面的片断演示了这是如何做的

```
public class InjectionBean implements Injection {
    HelloWorld helloworld;

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

关于@EJB 注释各个属性的使用, 有很多同学感到迷惑, 我们在使用 name、beanName、mappedName 属性时根据具体情况只需设置其中一个就行, 现把他与 ejb-jar.xml 对应起来, 估计大家心里就明白了

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"
)
```

从上面可以看到, 如果没有定义 ejb-jar.xml 或在 ejb-jar.xml 里没有配置 EJB 引用名称(通过<ejb-ref>,<ejb-local-ref>元素配置), 我们不需要设置@EJB.name()属性值

下面是 InjectionBean 的 Remote 业务接口

Injection.java

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

下面是 Session Bean 的 JSP 客户端代码:

InjectionTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.Injection, 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");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx;
    try {
        ctx = new InitialContext(props);
        Injection injection = (Injection) ctx.lookup("InjectionBean/remote");
        out.println(injection.SayHello());
    } catch (NamingException e) {
        out.println(e.getMessage());
    }

%>
```

@EJB 注释只能注入 EJB 存根对象, 除 @EJB 注释之外, EJB 3.0 也支持 @Resource 注释来注入来自 JNDI 的任何资源。下面的例子中演示了如何注入数据源。"java:/DefaultMySqlDS" 是数据源 DefaultMySqlDS 的全局 JNDI 名。有关数据源的配置请参考后面章节 [“JBoss 数据源的配置”](#)

```
//author:lihuoming
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.HelloWorld;
import com.foshanshop.ejb3.Injection;
import javax.annotation.Resource;
import javax.ejb.EJB;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.sql.DataSource;
```

```

@Stateless
@Remote( { Injection.class })
public class InjectionBean implements Injection {
    @EJB(beanName = "HelloWorldBean")
    HelloWorld helloworld;

    @Resource(mappedName = "java:/DefaultMySqlDS")
    DataSource myDb;

    public String SayHello() {
        String str = "";
        try {
            Connection 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();
        }
        return helloworld.SayHello(str);
    }
}

```

如果 JNDI 对象在本地(java:comp/env)JNDI 目录中, 你只需给定他的映射名称即可, 不需要带前缀,如下面例子注入一个消息 connection factory 和一个 messaging queue

```

@Resource(mappedName="ConnectionFactory")
QueueConnectionFactory factory;

@Resource(mappedName="queue/A")
Queue queue;

```

对于"well-known"对象, @Resource 注释可以不指定 JNDI 名就能注入他们, 他通过变量的类型就能获得他的 JNDI 名。下面是一些例子。

```

@Resource
TimerService tms;

@Resource
SessionContext ctx;

```

和@EJB 注释相同, @Resource 注释也可以被用在 JavaBean 风格的 setter 方法上。

本例子的 EJB 源代码在 DependencyInjection 文件夹 (源代码下载:<http://www.foshanshop.net/>), 项目中使用到的类

库在上级目录 lib 文件夹下。要恢复 DependencyInjection 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”,要发布本例子 EJB, 你可以执行 Ant 的 deploy 任务。发布前确保 HelloWorld.jar 已经发布到 Jboss 中。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/InjectionTest.jsp> 访问客户端。

注意: 在实际部署中, 如果 DependencyInjection.jar 和 HelloWorld.jar 文件同时存在于 jboss。启动 Jboss 时, Jboss 按默认的发布顺序先发布 DependencyInjection.jar, 后发布 HelloWorld.jar, 因为 DependencyInjection.jar 文件中使用到了 HelloWorld.jar 文件里的类, 所以就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, 作者给大家提供一个办法:

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

如给本例子 jar 文件编号, 修改后的文件名称如下:

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。

4.8 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, 估计各位在不同的场合中已经使用过。下面就直接介绍 EJB3.0 定时服务的开发过程。定时服务的开发过程与会话 Bean 的开发过程大致相同, 但比会话 Bean 多了几个操作, 那就是使用容器对象 SessionContext 创建定时器, 并使用 @Timeout 注释声明定时器方法。

下面定义一个每隔 3 秒触发一次事件的定时器, 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.TimerService;
import javax.annotation.Resource;
import javax.ejb.Remote;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;
```

```
@Stateless
```

```

@Remote ({TimerService.class})
public class TimerServiceBean implements TimerService {
    private int count = 1;
    private @Resource SessionContext ctx;

    public void scheduleTimer(long milliseconds){
        count = 1;
        ctx.getTimerService().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("-----");
        System.out.println("定时器事件发生, 传进的参数为: " + timer.getInfo());
        System.out.println("-----");

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

```

通过依赖注入@Resource SessionContext ctx, 我们获得 SessionContext 对象, 调用 ctx.getTimerService().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 你需要传给定时器的参数, 该参数必须实现 Serializable 接口。

当定时器创建完成后, 我们还需声明定时器方法。定时器方法的声明很简单, 只需在方法上面加入@Timeout 注释, 另外定时器方法必须遵守如下格式:

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

TimerService.java

```

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

```

下面是 TimerServiceBean 的 JSP 客户端代码:

TimerServiceTest.jsp

```

<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.TimerService, 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");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    try {
        InitialContext ctx = new InitialContext(props);
        TimerService timer = (TimerService) 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 EJB 源代码在 TimerService 文件夹（源代码下载:<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 TimerService 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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/TimerServiceTest.jsp> 访问客户端。

4.9 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），认证是完成用户名和密码的匹配校验；授权是决定用户可以访问哪些资源，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，你可以通过使用 @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 域表明文件是 classpath 中的 users.properties 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 |

本例使用了 Jboss 默认的安全域“other”，“other”安全域告诉容器到 classpath 中的 users.properties 和 roles.properties 找密码和用户角色列表。“other”安全域的定义在[jboss 安装目录]/server/default/conf/login-config.xml 文件。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 unauthenticated 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(不提供用户名及密码)访问的,如果你想使匿名用户也可访问通过 @PermitAll 注释定义的资源，可以修改“other”安全域配置，修改片断如下(蓝色部分)：

```
<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>
```

注意：本例子采用的是“other”域默认配置，没有修改过 login-config.xml 文件。匿名用户无法访问本例子 @PermitAll 注释的方法。要访问 @PermitAll 注释的方法，必须提供 users.properties 文件中的用户。

下面我们就开始安全服务的具体开发：

开发的第一步是定义安全域，安全域的定义有两种方法：

第一种方法：通过 Jboss 发布文件(jboss.xml)进行定义(本例采用的方法)，定义内容如下：

jboss.xml（本例使用 Jboss 默认的安全域“other”）


```
<?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>
```

jboss.xml 必须打进 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{
```

由于使用的是 Jboss 安全注释，程序采用了硬编码，不利于日后迁移到其他 J2EE 服务器（如：WebLogic），所以作者不建议使用这种方法定义安全域。

定义好安全域之后，因为我们使用 Jboss 默认的安全域“other”，所以必须使用 `users.properties` 和 `roles.properties` 存储用户名/密码及用户角色。

现在开发的第二步就是定义用户名，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。以下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
```

以上两个文件必须存放于类路径下。在进行用户验证时，Jboss 容器会自动寻找这两个文件。

开发的第三步就是为业务方法定义访问角色。本例定义了三个方法：`AdminUserMethod()`，`DepartmentUserMethod()`，`AnonymousUserMethod()`，第一个方法只允许具有 `AdminUser` 角色的用户访问，第二个方法只允许具有 `DepartmentUser` 角色的用户访问，第三个方法允许所有角色的用户访问。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，注：用户必须存在
users.properties文件哦";
    }
}

```

@RolesAllowed 注释定义允许访问方法的角色列表，如角色为多个，可以用逗号分隔。@PermitAll 注释定义所有的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();
}

```

到目前为止一个安全服务例子就开发完成。下面是打包后的目录结构：

```

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

```

我们看看客户端如何访问具有安全验证的 SecurityAccessBean。

SecurityAccessTest.jsp

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

<%

    Properties props = new Properties();
    props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
    props.setProperty("java.naming.provider.url", "localhost:1099");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");
    InitialContext ctx = new InitialContext(props);

    String user = request.getParameter("user");
    String pwd = request.getParameter("pwd");
    if (user!=null && !"".equals(user.trim())){
        SecurityAssociation.setPrincipal(new SimplePrincipal(user.trim()));
        SecurityAssociation.setCredential(pwd.trim().toCharArray());
    }

    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>");
    }
}
```

```

    }

    SecurityAssociation.clear();

%>

<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>&nbsp;&nbsp;&nbsp;密码: <STRONG>123456</STRONG>
</p>
<p>事业部  用户名: <STRONG>zhangfeng</STRONG>&nbsp;&nbsp;&nbsp;密码 <STRONG>111111</STRONG></p>
<p>合作伙伴  用户名: <STRONG>wuxiao</STRONG>&nbsp;&nbsp;&nbsp;密码 <STRONG>123</STRONG></p>
</body>
</html>

```

上面客户端代码只是为了教学的方便，并不适合在实际开发中使用。通常客户端应用要不在 JavaEE 容器，要不在 J2SE，下面介绍在 JavaEE 容器中的客户端安全应用。

因为使用的 web 文件较多，下面列出客户端 war 文件打包后的目录结构

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
| - jboss-web.xml
| - web.xml
```

为了使用容器的安全服务，我们需要在 jboss-web.xml 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>

```

需要配置的工作我们已经完成了，剩下来我们就要开始页面开发，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>&nbsp;&nbsp;&nbsp;密码: <STRONG>123456</STRONG>
</p>
  <p>事业部角色  用户名: <STRONG>zhangfeng</STRONG>&nbsp;&nbsp;&nbsp;密码
<STRONG>111111</STRONG></p>
  <p>合作伙伴角色  用户名: <STRONG>wuxiao</STRONG>&nbsp;&nbsp;&nbsp;密码 <STRONG>123</STRONG></p>
</body>

```

上面的粗体部分是 Jaas 规范定义的，登录 form 的 action 必须为 `j_security_check`，用户名字段名为 `j_username`，密码字段名为 `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>/loginFailed.html</form-error-page>` 指定的页面。

loginFailed.html 页面代码

```

<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>&nbsp;&nbsp;&nbsp;密码: <STRONG>123456</STRONG>
</p>
  <p>事业部角色  用户名: <STRONG>zhangfeng</STRONG>&nbsp;&nbsp;&nbsp;密码
<STRONG>111111</STRONG></p>
  <p>合作伙伴角色  用户名: <STRONG>wuxiao</STRONG>&nbsp;&nbsp;&nbsp;密码 <STRONG>123</STRONG></p>
</body>

```

notAuthenticated.html 页面代码

```
<body>
```



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

另外为了测试的需要,我们定义3种角色的访问页面(anonymousPage.jsp, departmentUser.jsp, adminPage.jsp)。下面是 adminPage.jsp 文件的代码,因为3种角色的页面只是调用的方法不一样,所以其他两个页面的代码不再列出

```
<%@ 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>
```

为了方便更换用户角色,我们开发一个登录退出页面。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>
```

本例子的 EJB 源代码在 SecurityWithPropertiesFile 文件夹（源代码下载:<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 SecurityWithPropertiesFile 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。

如果客户端是 J2SE,你只需要设置初始化上下文的参数就能直接访问带角色的 EJB，代码如下：

```
Properties props = new Properties();
props.setProperty("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
props.setProperty("java.naming.provider.url", "localhost:1099");
props.setProperty("java.naming.security.principal", "lihuoming");
props.setProperty("java.naming.security.credentials", "123456");
InitialContext = new InitialContext(props);
SecurityAccess securityaccess = (SecurityAccess) ctx.lookup("SecurityAccessBean/remote");
```

4.9.1 自定义安全域

把用户名/密码及角色存放在 users.properties 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。为此，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，他采用数据库存储用户名及角色。

安全域在[jboss 安装目录]/server/default/conf/login-config.xml 文件中定义，本例配置片断如下：

```
<!-- ..... 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), 他的 dsJndiName 属性(数据源 JNDI 名) 使用 DefaultMySqlDS 数据源(本教程自定义的数据源, 关于数据源的配置请参考后面章节: [JBoss 数据源的配置](#)), principalsQuery 属性定义 Jboss 通过给定的用户名如何获得密码, rolesQuery 属性定义 Jboss 通过给定的用户名如何获得角色列表, 注意:SQL 中的 'Roles' 常量字段不能去掉。unauthenticatedIdentity 属性允许匿名用户(不提供用户名及密码)访问。

“foshanshop”安全域使用的 sys_user 和 sys_userrole 表是自定义表, 实际项目开发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(实际项目开发中字段名可以自定义)

sys_user 表:

| 字段名 | 属性 |
|----------|----------------------|
| name(主键) | varchar(45) NOT NULL |
| password | varchar(45) NOT NULL |

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 |

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, 你还需往 sys_user, sys_userrole 表添加数据:

| name | password |
|-----------|----------|
| lihuoming | 123456 |
| zhangfeng | 111111 |
| wuxiao | 123 |

| username | rolename |
|-----------|----------------|
| lihuoming | AdminUser |
| lihuoming | DepartmentUser |
| lihuoming | CooperateUser |
| wuxiao | CooperateUser |

| | |
|-----------|----------------|
| zhangfeng | DepartmentUser |
|-----------|----------------|

在完成上面的配置后，我们就可以使用“foshanshop”安全域了，配置内容如下：

jboss.xml

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

本节的客户端代码与上节相同，代码这里不再列出。你只需要把 jboss-web.xml 使用的域修改为 foshanshop，修改后的配置如下：

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>
```

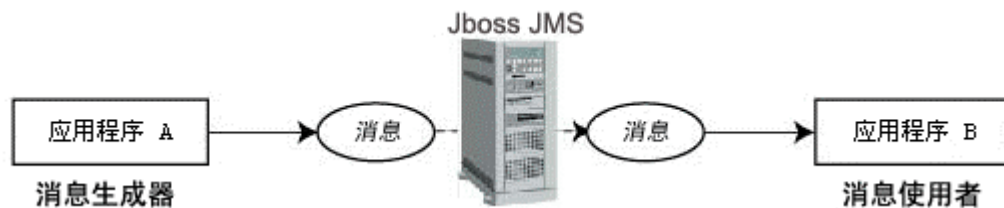
本例子的 EJB 源代码在 SecurityWithDB 文件夹（源代码下载：<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 SecurityWithDB 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（确保配置了环境变量 JBOSS_HOME 及启动了 Jboss），你可以执行 Ant 的 deploy 任务。本例子的客户端代码也在 SecurityWithDB 文件夹，客户端应用随 Ant 的 deploy 任务一起发布。通过 <http://localhost:8080/JaasTest> 访问客户端。

注意：别忘了配置数据源、“foshanshop”安全域、数据库表及例子所需数据。

第五章 JMS（Java Message Service）

Java 消息服务（Java Message Service，简称 JMS）是企业级消息传递系统，紧密集成于 Jboss Server 平台之中。企业消息传递系统使得应用程序能够通过消息的交换与其他系统之间进行通信。

下图说明 jboss JMS 消息传递。



消息组成

消息传递系统的中心就是消息。一条 **Message** 分为三个组成部分：

- 头 (header) 是个标准字段集，客户机和供应商都用它来标识和路由消息。
 - JMSMessageID**: 标识提供者发送的每一条消息，发送过程中由提供者设置
 - JMSDestination**: 消息发送的 Destination，由提供者设置
 - JMSDeliveryMode**: 包括 **DeliveryMode.PERSISTENT** (被且只被传输一次) 和 **DeliveryMode.NON_PERSISTENT** (最多被传输一次)
 - JMSTimestamp**: 提供者发送消息的时间，由提供者设置
 - JMSExpiration**: 消息失效的时间，是发送方法的生存时间和当前时间值的和，0 表明消息不会过期
 - JMSPriority**: 由提供者设置，0 最低，9 最高
 - JMSCorrelationID**: 用来链接响应消息和请求消息，由发送消息的 JMS 程序设置
 - JMSReplyTo**: 请求程序用它来指出回复消息应发送的地方
 - JMSType**: JMS 程序用来指出消息的类型
 - JMSRedelivered**: 消息被过早的发送给了 JMS 程序，程序不知道消息的接受者是谁
- 属性 (property) 支持把可选头字段添加到消息。如果您的应用程序需要不使用标准头字段对消息编目和分类，您就可以添加一个属性到消息以实现这个编目和分类。提供 **set<Type>Property(...)** 和 **get<Type>Property(...)** 方法以设置和获取各种 Java 类型的属性，包括 **Object**。JMS 定义了一个供应商选择提供的标准属性集。
 - JMSXUserID**: 发送消息的用户的身分
 - JMSXAppID**: 发送消息的应用程序的身分
 - JMSXDeliveryCount**: 尝试发送消息的次数
 - JMSXGroupID**: 该消息所属的消息组的身分
 - JMSXGroupSeq**: 该消息在消息组中的序号
 - JMSXProducerTxID**: 生成该消息的事物的身分
 - JMSXConsumerTxID**: 使用该消息的事物的身分
 - JMSXRcvTimestamp**: JMS 将消息发送给客户的时间
- 消息的主体 (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**: 包含 Java 基本数值流，用标准流操作来顺序的填充和读取。
 - **MapMessage**: 包含一组名/值对；名称为 **string** 类型，而值为 Java 的基本类型。
 - **TextMessage**: 包含一个 **String**。
 - **ObjectMessage**: 包含一个 **Serializable** Java 对象；能使用 JDK 的集合类。
 - **BytesMessage**: 包含未解释字节流：编码主体以匹配现存的消息格式。

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，只有以下区别：

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。

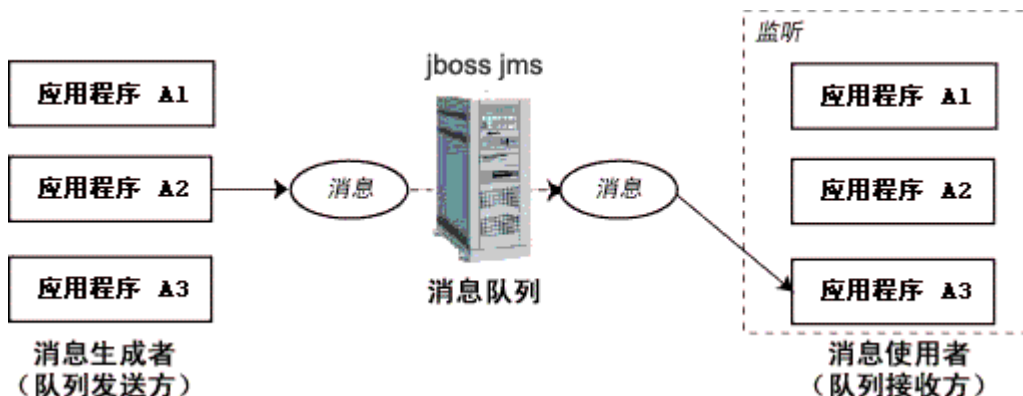
每种模型都通过扩展公用基类来实现。例如，PTP 类 `javax.jms.Queue` 和 pub/sub 类 `javax.jms.Topic` 都扩展 `javax.jms.Destination` 类。

以下部分将详细介绍每种消息传递模型。

1. 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。PTP 消息传递应用程序使用命名队列发送接收消息。队列发送方（生成者）向特定队列发送消息。队列接收方（使用者）从特定队列接收消息。

下图说明 PTP 消息传递。

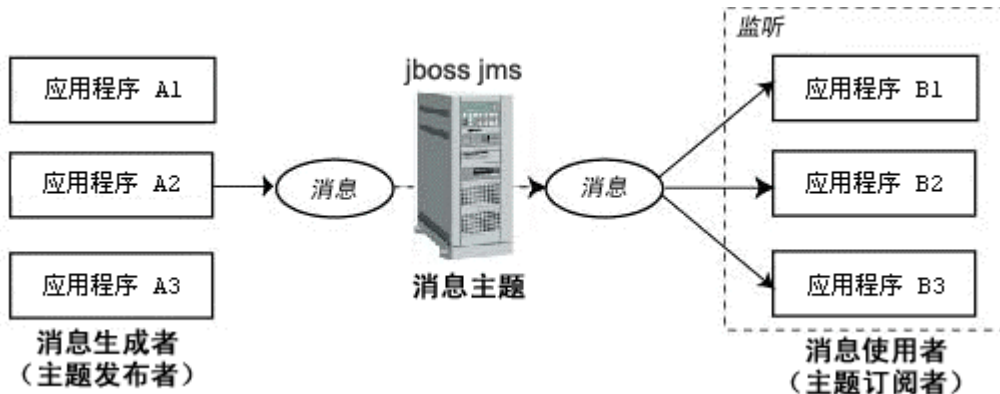


一个队列可以关联多个队列发送方和接收方，但一条消息仅传递给一个队列接收方。如果多个队列接收方正在监听队列上的消息，jboss JMS 将根据“先来者优先”的原则确定由哪个队列接收方接收下一条消息。如果没有队列接收方在监听队列，消息将保留在队列中，直至队列接收方连接队列为止。

2. 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应用程序。Pub/sub 消息传递应用程序可通过订阅主题来发送和接收消息。主题发布者（生成器）可向特定主题发送消息。主题订阅者（使用者）从特定主题获取消息。

下图说明 pub/sub 消息传递。



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。

上面两种消息传递模型里，我们都需要定义消息发送者和接收者，消息发送者把消息发送到 Jboss JMS 某个 Destination，而消息接收者从 Jboss JMS 的某个 Destination 里获取消息。消息接收者可以同步或异步接收消息，一般而言，异步消息接收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。

Message-Driven Bean 由 EJB 容器进行管理，具有一般的 JMS 接收者所不具有的优点，如对于一个 Message-driven Bean，容器可创建多个实例来处理大量的并发消息，而一般的 JMS 使用者 (consumer) 开发时则必须对此进行处理才能获得类似的功能。同时 Message-Driven Bean 可取得 EJB 所能获得的标准服务，如容器管理事务等服务。

正因为消息驱动 Bean 具有众多的优点，所以本教程使用消息驱动 Bean 作为消息接收者。

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。它是一个异步的无状态 Session Bean，客户端调用 MDB 后无需等待，立刻返回，MDB 将异步处理客户请求。一个 MDB 类必须实现 MessageListener 接口。当容器检测到 bean 守候的队列一条消息时，就调用 onMessage()方法，将消息作为参数传入。MDB 在 OnMessage()中决定如何处理该消息。你可以用注释来配置 MDB 监听哪一条队列。当 MDB 部署时，容器将会用到其中的注释信息。

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时，很适合使用消息驱动 Bean。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)

下面的例子展示了 queue 消息的发送与接收。代码如下：

QueueSender.java (Queue 消息的发送者)

```
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.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("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
            props.setProperty("java.naming.provider.url", "localhost:1099");
            props.setProperty("java.naming.factory.url.pkgs",
"org.jboss.naming:org.jnp.interfaces");
            InitialContext ctx = new InitialContext(props);

            QueueConnectionFactory factory = (QueueConnectionFactory)
ctx.lookup("ConnectionFactory");
            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Object (对象必须实现序列化, 否则等着出错吧)
            producer.send(session.createObjectMessage(new Man("美女", "北京和平里一号")));

            //发送MapMessage
            MapMessage mapmsg = session.createMapMessage();
            mapmsg.setObject("nol", "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);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

        //发送StreamMessage
        StreamMessage smsg = session.createStreamMessage();
        smsg.writeString("我就爱流读写");
        producer.send(smsg);

    } catch (Exception e) {
        System.out.println(e.getMessage());
    } finally {
        try {
            session.close();
            conn.close();
        } catch (JMSEException 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;
    }
}

```

上面的 J2SE 客户端用来向 queue/foshanshop 消息队列发送一条消息(在发送时, JNDI 为 queue/foshanshop 的 Destination 必须存在, 本例子的接收者 MDB 在发布时会自动创建该 Destination)。客户端发送消息一般有以下步骤:

(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("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
props.setProperty("java.naming.provider.url", "localhost:1099");
props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
InitialContext ctx = new InitialContext(props);
```

(2) 根据上下文来查找一个连接工厂 TopicConnectionFactory/ QueueConnectionFactory (有两种连接工厂, 根据是 topic/queue 来使用相应的类型);

例子对应代码:

```
QueueConnectionFactory factory = (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
```

(3) 从连接工厂得到一个连接(Connect 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接收消息收条的会话, 在非事务 Session 中, 消息传递的方式有三种:

Session.AUTO_ACKNOWLEDGE : 当客户机调用的 receive 方法成功返回, 或当 MessageListener 成功处理了消息, session 将会自动接收消息的收条。

Session.CLIENT_ACKNOWLEDGE : Session 对象依赖于应用程序对已收到的消息调用确认方法。一旦调用该方法, 会话将确认所有自上次确认后收到的消息。该方法允许应用程序通过一次调用接收、处理和确认一批消息。

Session.DUPS_OK_ACKNOWLEDGE : 一旦消息处理中返回了应用程序接收方法, Session 对象即确认消息接收, 允许重复确认。就资源利用情况而言, 此模式最高效。

(5) 查找目的地(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制造者 MessageProducer (扩展了 QueueSender 和 TopicPublisher 这两个基本接口)

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");//发送文本
producer.send(msg);
```

JMS 类说明

| JMS 类 | 描述 |
|---|--------------------------------------|
| ConnectionFactory (QueueConnectionFactory, TopicConnectionFactory) | 封装连接配置信息。将使用连接工厂创建连接。使用 JNDI 查找连接工厂。 |
| Connection (QueueConnection, TopicConnection) | 代表通往消息传递系统的开放式通信通道。将使用连接创建会话。 |
| Session (QueueSession, TopicSession) | 定义生成的消息和使用的消息的顺序。 |
| Destination | 标识队列或主题, 并封装特定提供程序的地址。队列和主题 |

| | |
|---|------------------------------------|
| | 目标分别管理通过 PTP 和 pub/sub 消息传递模型传递的消息 |
| MessageProducer | 提供发送消息的接口。消息生成器向队列或主题发送消息。 |
| MessageConsumer | 提供接收消息的接口。消息使用者从队列或主题接收消息。 |
| Message (类型有: StreamMessage, MapMessage, TextMessage, ObjectMessage, BytesMessage) | 封装要发送或要接收的信息。 |

下面是 Queue 消息的接收方，它是一个消息驱动 Bean

PrintBean.java (Queue 消息的接收者)

```
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")
})
public class PrintBean implements MessageListener {

    public void onMessage(Message msg) {
        try {
            if (msg instanceof TextMessage) {
                TextMessage tmsg = (TextMessage) msg;
                String content = tmsg.getText();
                this.print(content);
            } else if (msg instanceof ObjectMessage) {
                ObjectMessage omsg = (ObjectMessage) msg;
                Man man = (Man) omsg.getObject();
                String content = man.getName() + " 家住" + man.getAddress();
                this.print(content);
            } else if (msg instanceof MapMessage) {
```

```

        MapMessage map = (MapMessage) msg;
        String content = map.getString("nol");
        this.print(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();
        this.print(content);
    } else if (msg instanceof StreamMessage) {
        StreamMessage smsg = (StreamMessage) msg;
        String content = smsg.readString();
        this.print(content);
    }
}

} catch (Exception e) {
    e.printStackTrace();
}

}

private void print(String content) {
    System.out.println(content);
}
}

```

上面通过 `@MessageDriven` 注释指明这是一个消息驱动 Bean，并使用 `@ActivationConfigProperty` 注释配置消息的各种属性，其中 `destinationType` 属性指定消息的类型为 `queue`。`destination` 属性指定消息路径（Destination），消息驱动 Bean 在发布时，如果路径（Destination）不存在，容器会自动创建，当容器关闭时该路径将被删除。运行本例子，当一个消息到达 `queue/foshanshop` 队列，就会触发 `onMessage` 方法，消息作为一个参数传入，在 `onMessage` 方法里面得到消息体并调用 `print` 方法把消息内容打印到控制台上。

本例子的源代码在 `MessageDrivenBean` 文件夹（源代码下载：<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 `lib` 文件夹下。要恢复 `MessageDrivenBean` 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。

5.1.2 Topic 消息的发送与接收(Pub/sub 消息传递模型)

从之前介绍的 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。本例子特设定了两个消息接收者。代码如下：

TopicPrintBeanOne.java (Topic 消息接收者之一)

```
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.Topic"),
    @ActivationConfigProperty(propertyName="destination",
        propertyValue="topic/student")
})
public class TopicPrintBeanOne implements MessageListener{

    public void onMessage(Message msg) {
        try {
            if (msg instanceof TextMessage) {
                TextMessage tmsg = (TextMessage) msg;
                String content = tmsg.getText();
                this.print(content);
            } else if (msg instanceof ObjectMessage) {
                ObjectMessage omsg = (ObjectMessage) msg;
                Man man = (Man) omsg.getObject();
                String content = man.getName() + " 家住" + man.getAddress();
                this.print(content);
            } else if (msg instanceof MapMessage) {
                MapMessage map = (MapMessage) msg;
                String content = map.getString("no1");
                this.print(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();
    this.print(content);
} else if(msg instanceof StreamMessage) {
    StreamMessage smsg = (StreamMessage) msg;
    String content = smsg.readString();
    this.print(content);
}

} catch (Exception e) {
    e.printStackTrace();
}

}

private void print(String content) {
    System.out.println(this.getClass().getName()+"=="+ content);
}

}

```

TopicPrintBeanTwo.java (Topic 消息接收者之二)

```

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.Topic"),
    @ActivationConfigProperty(propertyName="destination",
        propertyValue="topic/student")
})

public class TopicPrintBeanTwo implements MessageListener {
    public void onMessage(Message msg) {

```



```

try {
    if (msg instanceof TextMessage) {
        TextMessage tmsg = (TextMessage) msg;
        String content = tmsg.getText();
        this.print(content);
    } else if (msg instanceof ObjectMessage) {
        ObjectMessage omsg = (ObjectMessage) msg;
        Man man = (Man) omsg.getObject();
        String content = man.getName() + " 家住" + man.getAddress();
        this.print(content);
    } else if (msg instanceof MapMessage) {
        MapMessage map = (MapMessage) msg;
        String content = map.getString("nol");
        this.print(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();
        this.print(content);
    } else if (msg instanceof StreamMessage) {
        StreamMessage smsg = (StreamMessage) msg;
        String content = smsg.readString();
        this.print(content);
    }
}

} catch (Exception e) {
    e.printStackTrace();
}

}

private void print(String content) {
    System.out.println(this.getClass().getName() + "==" + content);
}
}

```

TopicSender.java (Topic 消息发送者)

```

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.StreamMessage;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.naming.InitialContext;
import com.foshanshop.ejb3.bean.Man;

public class TopicSender {

    public static void main(String[] args) {
        TopicConnection conn = null;
        TopicSession session = null;
        try {
            Properties props = new Properties();
            props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
            props.setProperty("java.naming.provider.url", "localhost:1099");
            props.setProperty("java.naming.factory.url.pkgs",
"org.jboss.naming:org.jnp.interfaces");
            InitialContext ctx = new InitialContext(props);

            TopicConnectionFactory factory = (TopicConnectionFactory)
ctx.lookup("ConnectionFactory");
            conn = factory.createTopicConnection();
            session = conn.createTopicSession(false, TopicSession.AUTO_ACKNOWLEDGE);
            Destination destination = (Topic) ctx.lookup("topic/student");
            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);

            //发送Object (对象必须实现序列化，否则等着出错吧)
            producer.send(session.createObjectMessage(new Man("美女", "北京和平里一号")));

            //发送MapMessage
            MapMessage mapmsg = session.createMapMessage();
            mapmsg.setObject("nol", "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("我就爱流读写");
producer.send(smsg);

} catch (Exception e) {
    System.out.println(e.getMessage());
} finally {
    try {
        session.close();
        conn.close();
    } catch (JMSException e) {
        e.printStackTrace();
    }
}
}
}

```

上面 Topic 消息的发送步骤和 Queue 消息相同。

本例子的源代码在 MessageDrivenBean 文件夹（源代码下载:<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 MessageDrivenBean 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。

第六章 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 时，容器会自动将 Entity Bean 的状态和数据库同步。

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。

6.1 持久化 persistence.xml 配置文件

一个实体 Bean 应用由实体类和 persistence.xml 文件组成。persistence.xml 文件在 Jar 文件的 META-INF 目录。persistence.xml 文件指定实体 Bean 使用的数据源及 EntityManager 对象的默认行为。persistence.xml 文件的配置说明如下:

```
<persistence>
  <persistence-unit name="foshanshop">
    <jta-data-source>java:/DefaultMySqlDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </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 属性定义持久化名称。jta-data-source 节点指定实体 Bean 使用的数据源 JNDI 名称 (如何配置数据源请参考下节 [“Jboss 数据源的配置”](#)), 如果应用发布在 jboss 下数据源名称必须带有 java:/前缀, 数据源名称大小写敏感。properties 节点用作指定持久化产品的各项属性, 各个应用服务器使用的持久化产品都不一样如 Jboss 使用 Hibernate, weblogic10 使用 Kodo, glassfish/sun application server/Oracle 使用 Toplink。因为 jboss 采用 Hibernate, Hibernate 有一项属性 hibernate.hbm2ddl.auto, 该属性指定实体 Bean 发布时是否同步数据库结构, 如果 hibernate.hbm2ddl.auto 的值设为 create-drop, 在实体 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 文件中看见。

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 的改动能反应到数据表, 建议使用 update, 这样实体 Bean 添加一个属性时能同时在数据表增加相应字段。

6.2 JBoss 数据源的配置

Jboss 有一个默认的数据源 DefaultDS, 他使用 Jboss 内置的 HSQLDB 数据库。实际应用中你可能使用不同的数据库, 如 MySQL、MsSqlServer、Oracle 等。各种数据库的数据源配置模版你可以在[Jboss 安装目录]\docs\examples\jca 目录中找到, 默认名称为: 数据库名+ -ds.xml。

不管你使用那种数据库都需要把他的驱动类 Jar 包放置在[Jboss 安装目录]\server\default\lib 目录下, 放置后需要启动 Jboss 服务器。

本教程使用的数据库是 mysql-5.0.22 和 Ms Sql Server2000, 使用驱动 Jar 包如下:

MySQL: mysql-connector-java-3.1.13-bin.jar

Ms Sql Server2000: msbase.jar, mssqlserver.jar, msutil.jar

上面的 Jar 文件你可以在网上下载或在例子源码的 lib 文件夹下得到 (例子源码下载地址:

<http://www.foshanshop.net/>)。

下面介绍 Mysql 和 Ms Sql Server2000 的数据源配置，数据源配置文件的取名格式必须为 xxx-ds.xml，如：mysql-ds.xml，mssqlserver-ds.xml，oracle-ds.xml。

数据源文件配置好后需要放置在[jboss 安装目录]/server/config-name/deploy 目录下,本教程采用的配置名为：default,所以路径为[jboss 安装目录]/server/default/deploy 目录

6.2.1 MySQL 数据源的配置

下面定义一个名为 DefaultMySQLDS 的 Mysql 数据源，连接数据库为 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。大家只需修改数据库名及登录用户名密码就可以直接使用。

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&characterEncoding=GBK
    </connection-url>
    <driver-class>org.gjt.mm.mysql.Driver</driver-class>
    <user-name>root</user-name>
    <password>123456</password>
    <exception-sorter-class-name>org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
    </exception-sorter-class-name>
    <metadata>
      <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

6.2.2 Ms Sql Server2000 数据源的配置

下面定义一个名为 MSSQLDS 的 Ms Sql Server 数据源，连接数据库为 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。大家只需修改数据库名及登录用户名密码就可以直接使用。

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>
```

6.2.3 Oracle9i 数据源的配置

下面定义一个名为 OracleDS 的 Oracle9i 数据源，连接数据库为 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。大家只需修改数据库名及登录用户名密码就可以直接使用。

oracle-ds.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>OracleDS</jndi-name>
    <connection-url>jdbc:oracle:thin:@nd:1521:FS</connection-url>
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
    <user-name>root</user-name>
    <password>123456</password>
    <SetBigStringTryClob>true</SetBigStringTryClob>
  <exception-sorter-class-name>
    org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter</exception-sorter-class-name>
    <metadata>
      <type-mapping>Oracle9i</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

数据源发布后，你可以在 <http://localhost:8080/jmx-console/> 找到他，如下图：

jboss.jca

- [name='jboss-ha-local-jdbc.rar',service=RARDeployment](#)
- [name='jboss-ha-xa-jdbc.rar',service=RARDeployment](#)
- [name='jboss-local-jdbc.rar',service=RARDeployment](#)
- [name='jboss-xa-jdbc.rar',service=RARDeployment](#)
- [name='jms-ra.rar',service=RARDeployment](#)
- [name='quartz-ra.rar',service=RARDeployment](#)
- [name=DefaultDS,service=DataSourceBinding](#)
- [name=DefaultDS,service=LocalTxCM](#)
- [name=DefaultDS,service=ManagedConnectionFactory](#)
- [name=DefaultDS,service=ManagedConnectionPool](#)
- [name=DefaultMySqlDS,service=DataSourceBinding](#)
- [name=DefaultMySqlDS,service=LocalTxCM](#)
- [name=DefaultMySqlDS,service=ManagedConnectionFactory](#)
- [name=DefaultMySqlDS,service=ManagedConnectionPool](#)

数据源

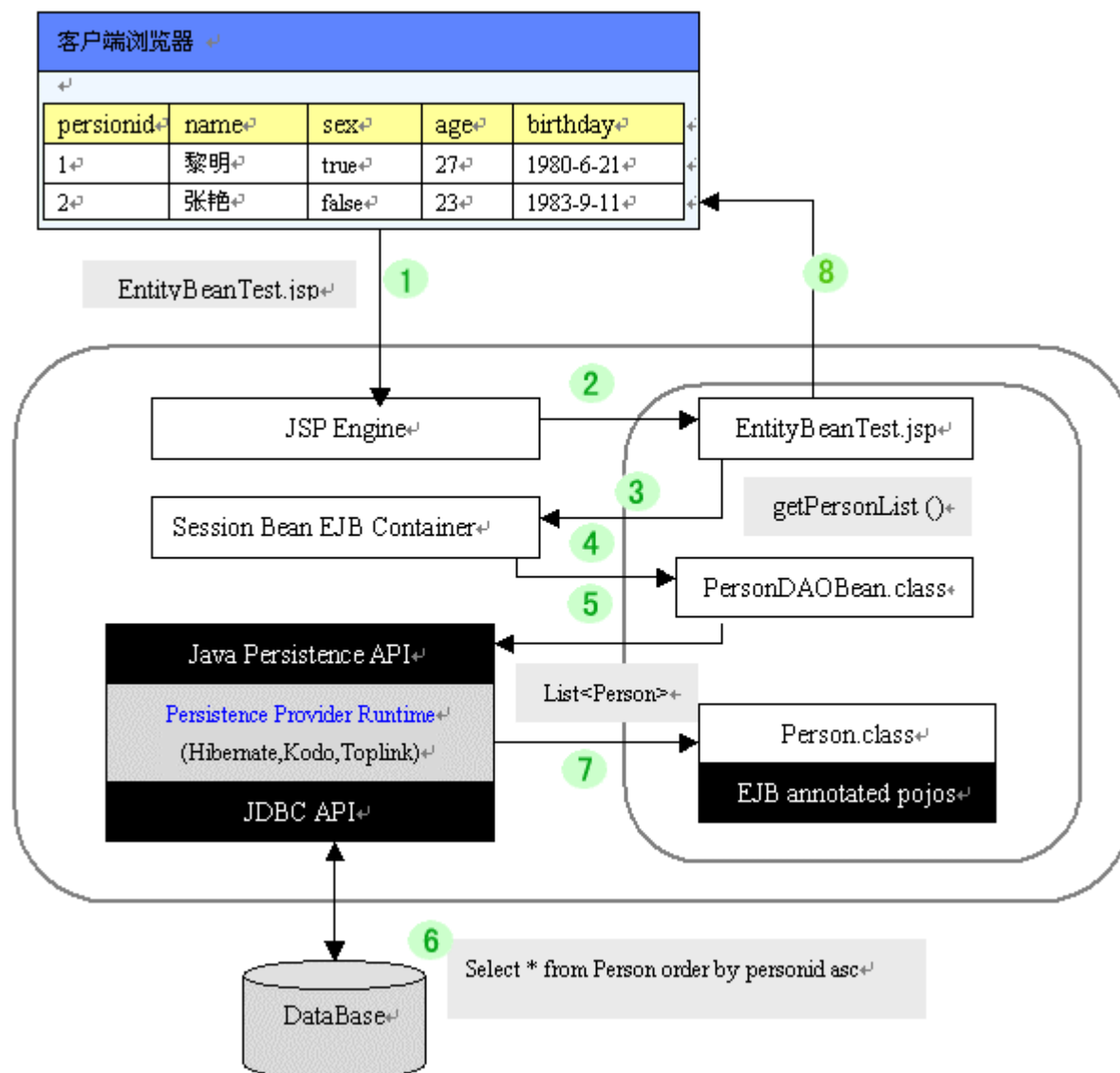
你可以点击 `name=DefaultMySqlDS,service=ManagedConnectionPool` 进入连接池属性修改界面。其中 `MaxSize` 属性指定了最大连接数, `InUseConnectionCount` 代表目前正在使用的连接数, 一旦 `InUseConnectionCount` 大于 `MaxSize`, 数据库连接将会报错, 这种情况一般都是因为手工操作 `jdbc`, 在使用完后没有立刻释放掉连接引起的。

6.3 实体 Bean 发布前的准备工作

- 2 配置数据源并放置在[jboss 安装目录]/server/default/deploy 目录, 把数据库驱动 Jar 包放置在[Jboss 安装目录]/server/default/lib 目录下, 放置后需要重启 Jboss 服务器。如果数据源已经存在就不需要配置。
- 3 配置 persistence.xml 文件, 在文件中指定使用的源据源及各项参数。
- 4 把实体类和 persistence.xml 文件打成 Jar, persistence.xml 放在 jar 文件的 META-INF 目录

6.4 单表映射的实体 Bean

这是本例子的应用体系结构图:



- 1> 浏览器请求 EntityBeanTest.jsp 文件
- 2> 应用服务器的 JSP 引擎编译 EntityBeanTest.jsp
- 3> EntityBeanTest.jsp 通过 JNDI 查找 PersonDAOBean EJB, 获得 EJB 的 stub(代理存根), 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("from Person order by personid asc") 进行持久化查询
- 6> Persistence provider runtime 通过 O/R Mapping annotation 把上面 JPQL 查询语句转译成 SQL 语句
- 7> Persistence provider runtime 把 SQL 查询结果处理成 Person 类型的 List
- 8> 遍历存放 Person 的 List, 打印在页面上。

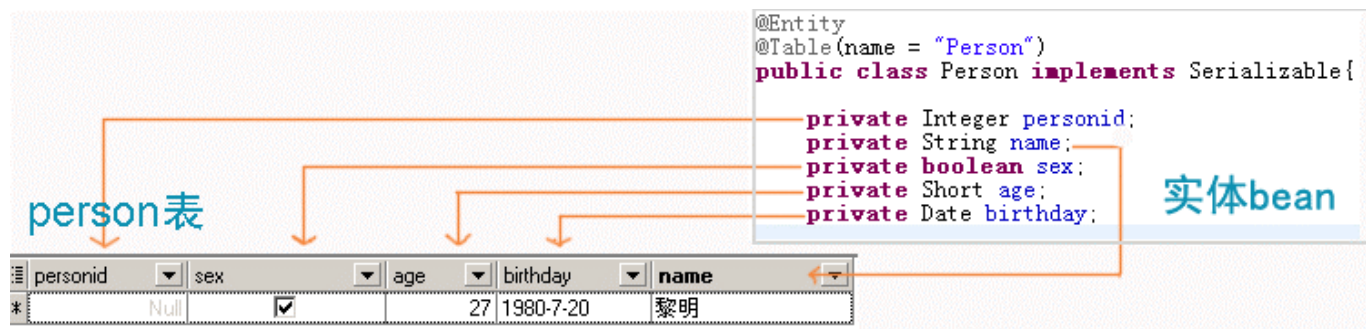
开发前先介绍需要映射的数据库表

person

| 字段名称 | 字段类型属性 | 描述 |
|---------------|----------------------|-------|
| personid (主键) | Int(11) not null | 人员 ID |
| 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 |

Person 表与实体的映射图如下:



现在按照上图建立与 Person 表进行映射的实体 Bean

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;
    }
}
```

从上面代码可以看到开发实体 Bean 非常简单，比起普通的 java bean 就是多了些注释。@Entity 注释指明这是一个实体 Bean，@Table 注释指定了 entity 所要映射的数据库表，其中@Table.name()用来指定映射表的表名。如果缺省@Table 注释，系统默认采用类名作为映射表的表名。实体 Bean 的每个实例代表数据表中的一行数据，行中的一列对应实例中的一个属性。

@javax.persistence.Column 注释定义了将成员属性映射到关系表中的哪一列和该列的一些结构信息（如列名是否唯一，是否允许为空，是否允许更新等），他的属性介绍如下：

- name: 映射的列名。如：映射 Person 表的 PersonName 列，可以在 name 属性的 getName 方法上面加入 @Column(name = "PersonName")，如果不指定映射列名，容器将属性名称作为默认的映射列名。

- **unique**: 是否唯一
- **nullable**: 是否允许为空
- **length**: 对于字符型列, **length** 属性指定列的最大字符长度
- **insertable**: 是否允许插入
- **updatable**: 是否允许更新
- **columnDefinition**: 定义建表时创建此列的 DDL
- **secondaryTable**: 从表名。如果此列不建在主表上 (默认建在主表), 该属性定义该列所在从表的名字。

@Id 注释指定 **personid** 属性为表的主键, 它可以有多种生成方式:

- **TABLE**: 容器指定用底层的数据表确保唯一。例子代码如下:

```
@TableGenerator(name="Person_GENERATOR", //为该生成方式取个名称
    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值的列的名称
    pkColumnValue="personid", //主键列的值(定位某条记录)
    allocationSize=1) //递增值

@Id
@GeneratedValue(strategy=GenerationType.TABLE, generator="Person_GENERATOR")
public Integer getPersonid() {
    return personid;
}
```

- **SEQUENCE**: 使用数据库的 **SEQUENCE** 列来保证唯一(Oracle 数据库通过序列来生成唯一 ID), 例子代码如下:

```
@SequenceGenerator(name="Person_SEQUENCE", //为该生成方式取个名称
    sequenceName="Person_SEQ") //sequence的名称(如果不存在, 会自动生成)
public class Person implements Serializable{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="Person_SEQ")
    public Integer getPersonid() {
        return personid;
    }
}
```

- **IDENTITY**: 使用数据库的 **IDENTITY** 列来保证唯一(像 mysql, sqlserver 数据库通过自增长来生成唯一 ID)
- **AUTO**: 由容器挑选一个合适的方式来保证唯一(由容器决定采用何种方式生成唯一主键, hibernate 会根据数据库类型选择适合的生成方式, 相反 **toplink** 就不是很近人情)
- **NONE**: 容器不负责主键的生成, 由调用程序来完成。

@GeneratedValue 注释定义了标识字段的生成方式, 本例 **personid** 的值由 MySQL 数据库自动生成。

注: 实体 bean 需要在网络上传送时必须实现 **Serializable** 接口, 否则将引发 **java.io.InvalidClassException** 例外。

@Temporal 注释用来指定 **java.util.Date** 或 **java.util.Calendar** 属性与数据库类型 **date, time** 或 **timestamp** 中的那一种类型进行映射。他的定义如下:

```
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
}
```

在 Jboss 中可以缺少 @Temporal 注释,但在使用了 TopLink 的服务器中,缺少该注释将会导致部署失败。@Temporal 注释的默认值为: TIMESTAMP

为了使用上面的实体 Bean,我们定义一个 Session Bean 作为他的使用者。下面是 Session Bean 的业务接口,他定义了五个业务方法 insertPerson,updatePerson,getPersonByID,getPersonList 和 getPersonNameByID,, insertPerson 用作添加一个 Person, updatePerson 更新 Person, getPersonByID 根据 personid 获取 Person, getPersonList 获取所有记录, getPersonNameByID 根据 personid 获取姓名。

PersonDAO.java

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

public interface PersonDAO {
    public boolean insertPerson(String name, boolean sex, short age, Date birthday);
    public String getPersonNameByID(int personid);
    public boolean updatePerson(Person person);
    public Person getPersonByID(int personid);
    public List getPersonList();
}
```

下面是 Session Bean 的实现

PersonDAOBean.java

```
package com.foshanshop.ejb3.impl;
import java.util.Date;
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
protected EntityManager em;

public String getPersonNameByID(int personid) {
    Person person = em.find(Person.class, Integer.valueOf(personid));
    return person.getName();
}

public boolean insertPerson(String name, boolean sex, short age, Date birthday) {
    try {
        Person person = new Person();
        person.setName(name);
        person.setSex(sex);
        person.setAge(Short.valueOf(age));
        person.setBirthday(birthday);
        em.persist(person);
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
    return true;
}

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

public boolean updatePerson(Person person) {
    try {
        em.merge(person);
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
    return true;
}

public List getPersonList() {
    Query query = em.createQuery("from Person order by personid asc");
    List list = query.getResultList();
    return list;
}
}
```

上面我们使用到了一个对象：EntityManager em，EntityManager 是由 EJB 容器自动地管理和配置的，不需要用户

自己创建，他用作操作实体 Bean。关于他的更多介绍请参考[持久化实体管理器 EntityManager](#)。

上面 em.find()方法用作查询主键 ID 为 personid 的记录。em.persist()方法用作向数据库插入一条记录。大家可能感觉奇怪，在类中并没有看到对 EntityManager em 进行赋值，后面却可以直接使用他。这是因为容器在实例化 SessionBean 后，就通过@PersistenceContext 注释动态注入 EntityManager 对象。

如果 persistence.xml 文件中配置了多个不同的持久化内容。在注入 EntityManager 对象时必须指定持久化名称，可以通过@PersistenceContext 注释的 unitName 属性进行指定，例：

```
@PersistenceContext(unitName="foshanshop")
```

```
EntityManager em;
```

如果只有一个持久化内容配置，不需要明确指定。

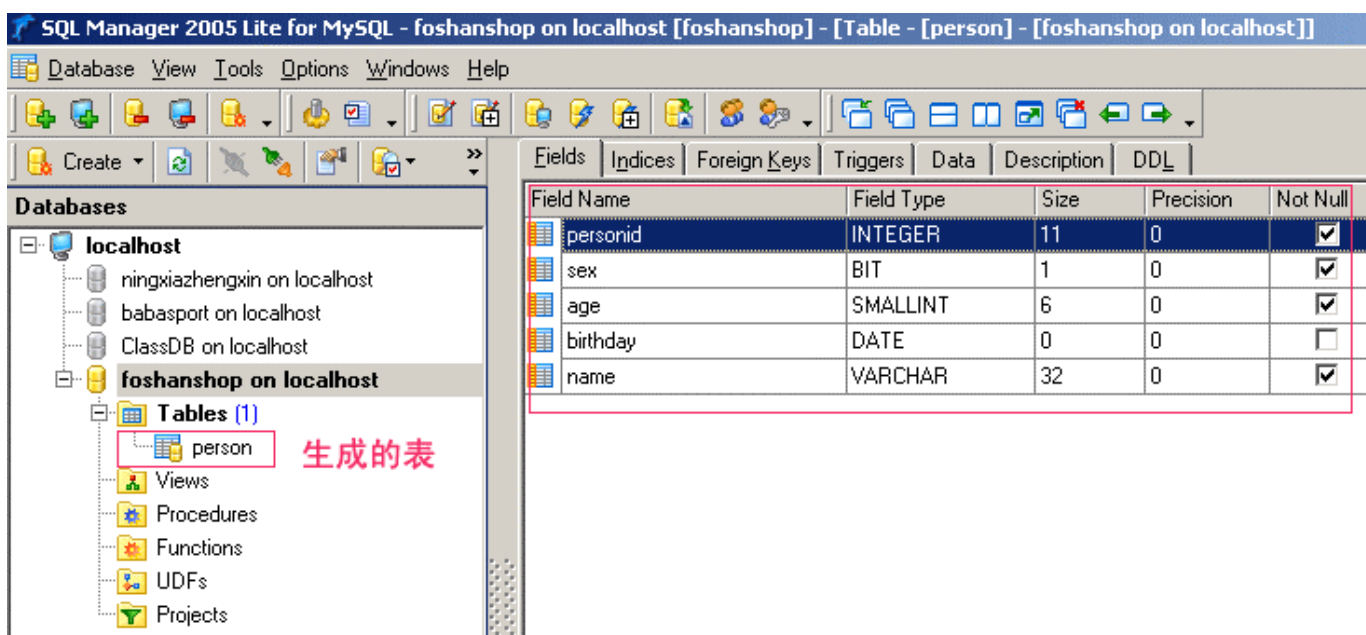
下面是 persistence.xml 文件的配置：

```
<persistence>
  <persistence-unit name="foshanshop">
    <jta-data-source>java:/DefaultMySqlDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

到目前为止，实体 bean 应用已经开发完成。我们按照上节“[实体 Bean 发布前的准备工作](#)”介绍的步骤把他打成 Jar 文件并发布到 Jboss 中。

在发布前请检查 persistence.xml 文件中使用的数据源是否配置（如何配置数据源请参考[Jboss 数据源的配置](#)），数据库驱动 Jar 文件是否放进了[Jboss 安装目录]\server\default\lib 目录下(放进后需要重启 Jboss)。

因为在 persistence.xml 文件中指定的 Hibernate 属性是<property name="hibernate.hbm2ddl.auto" value="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是否生成了 Person 表。下图是生成的表：



| Field Name | Field Type | Size | Precision | Not Null |
|------------|------------|------|-----------|-------------------------------------|
| personid | INTEGER | 11 | 0 | <input checked="" type="checkbox"/> |
| sex | BIT | 1 | 0 | <input checked="" type="checkbox"/> |
| age | SMALLINT | 6 | 0 | <input checked="" type="checkbox"/> |
| birthday | DATE | 0 | 0 | <input type="checkbox"/> |
| name | VARCHAR | 32 | 0 | <input checked="" type="checkbox"/> |

下面是 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,
    java.text.SimpleDateFormat"%>

<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 {
    Properties props = new Properties();
    props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
    props.setProperty("java.naming.provider.url", "localhost:1099");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");
    InitialContext ctx = new InitialContext(props);
    PersonDAO persondao = (PersonDAO) ctx.lookup("PersonDAOBean/remote");
    SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
    persondao.insertPerson("黎明", true, (short)26, formatter.parse("1980-9-30")); //添加一个
人

    List list = persondao.getPersonList();
    for(int i=0; i<list.size(); i++) {
        Person person = (Person)list.get(i);
        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人，然后获取全部记录打印出来。

本例子的 EJB 源代码在 EntityBean 文件夹（源代码下载：<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 EntityBean 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据源配置文件是 mysql-ds.xml，你可以在下载的文件中找到。

本例子使用的数据库名为 foshanshop，他的 DDL 为：

create database `foshanshop` DEFAULT CHARSET=gbk

在创建数据库时一定要得指定 Mysql 的字符集编码为 GBK，否则在插入中文字符时会报: Data too long for column, 如果你不想每次创建数据库时指定编码，可以修改 Mysql 默认的字符集编码，方法是：在 Mysql 安装目录下找到 my.ini 文件，打开文件找到“default-character-set”（共两处），把值设为 GBK，设置完后需重启 Mysql 服务，可以在服务管理器中重启或在 dos 窗口下输入：service mysql restart

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 Junit Test 例子 (PersonDAOTest.java)。

6.5 属性映射

如果不想让一些成员属性映射成数据库字段，我们可以使用 @Transient 注释进行标注，下面的 firstName 属性将不会被持久化成数据库字段

```
public class Person implements Serializable{

    private Integer personid;
    private String name;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    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;
    }
}
```

```

    @Transient
    public String getFristName() {
        return "li";
    }
}

```

如果你想映射枚举对象到数据库就需要使用 `@Enumerated` 注释进行标注，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
@Table
public class CommentContent implements Serializable{
    private Long id;//主键ID
    private CommentType type;//评论类型

    public CommentContent() {}

    public CommentContent(Long id) {
        this.id = id;
    }

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

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

```

```

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

/**
 * 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 CommentContent。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 CommentContent 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 CommentContent)) {
        return false;
    }
    CommentContent other = (CommentContent)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 + "]";
}
}

```

有些时候你可能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
在做新闻系统时，我们通常会使用到大文本数据，这时就有必要把字段加上 `@Lob` 注释

```
@Entity
@Table
public class News implements Serializable {
    private Long id; //主键 ID
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long getId() {
        return id;
    }

    public void setId(Long _id) {
        this.id = _id;
    }
    @Lob
    public String getContent() {
        return content;
    }

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

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能否为 `null`

```
@Entity
@Table
public class News implements Serializable {
    private Long id; //主键 ID
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long getId() {
        return id;
    }
}
```

```
}

public void setId(Long _id) {
    this.id = _id;
}

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

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

@Temporal 注释前面也讲到过，主要是用来指明 java.util.Date 或 java.util.Calendar 类型的属性具体与数据库 (date,time,timestamp) 三个类型中的那一个进行映射。注释的定义如下：

```
package javax.persistence;
```

```
public enum TemporalType
{
    DATE,
    TIME,
    TIMESTAMP
}
```

```
public class Person implements Serializable{

    private Integer personid;
    private Date birthday;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    public Integer getPersonid() {
        return personid;
    }

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

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

```

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

```

6.6 持久化实体管理器 EntityManager

EntityManager 是用来对实体 Bean 进行操作的辅助类。他可以用来产生/删除持久化的实体 Bean，通过主键查找实体 bean，也可以通过 EJB3 QL 语言查找满足条件的实体 Bean。实体 Bean 被 EntityManager 管理时，EntityManager 跟踪他的状态改变，在任何决定更新实体 Bean 的时候便会把发生改变的值同步到数据库中。当实体 Bean 从 EntityManager 分离后，他是不受管理的，EntityManager 无法跟踪他的任何状态改变。EntityManager 的获取前面已经介绍过，可以通过 @PersistenceContext 注释由 EJB 容器动态注入，例：

```
@PersistenceContext(unitName="foshanshop")
```

```
EntityManager em;
```

下面介绍 EntityManager 常用的 API

6.6.1 Entity 获取 find()或 getReference()

如果知道 Entity 的唯一标示符，我们可以用 find()或 getReference()方法来获得 Entity。

```

@PersistenceContext
protected EntityManager em;

...

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

```

当在数据库中没有找到记录时，getReference()和 find()是有区别的，find()方法会返回 null，而 getReference()方法会抛出 `javax.persistence.EntityNotFoundException` 例外，另外 getReference()方法不保证实体 Bean 已被初始化。如果传递进 getReference()或 find()方法的参数不是实体 Bean，都会引发 `IllegalArgumentException` 例外。

6.6.2 添加 persist()

保存 Entity 到数据库。

```

@PersistenceContext
protected EntityManager em;

...

```



```
Person person = new Person();
person.setName(name);
//把数据保存进数据库中
em.persist(person);
```

如果传递进 persist()方法的参数不是实体 Bean，会引发 IllegalArgumentException 例外。

6.6.3 更新实体

当实体正在被容器管理时，你可以调用实体的 set 方法对数据进行修改，在容器决定 flush 时，更新的数据才会同步到数据库。如果你希望修改后的数据实时同步到数据库，你可以执行 EntityManager.flush()方法。

```
@PersistenceContext
protected EntityManager em;
...
public void updatePerson() {
    try {
        Person person = em.find(Person.class, 1);
        person.setName("lihuoming"); //方法执行完后即可更新数据
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

6.6.4 合并 Merge()

merge ()方法是在实体 Bean 已经脱离了 EntityManager 的管理时使用，当容器决定 flush 时，数据将会同步到数据库中。

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

public boolean updatePerson(Person person) {
    try {
        em.merge(person);
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
    return true;
}
```

下面的代码把实体 Bean 返回到了客户端，这时的实体 Bean 已经脱离了容器的管理，在客户端对实体 Bean 进行修改，最后把他返回给 EJB 容器进行更新操作。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,此时的人已经脱离容器的管理
person.setName("张小艳");
persondao.updatePerson(person);
```

执行 em.merge(person)方法时，容器的工作规则：

1. 如果此时容器中已经存在一个受容器管理的具有相同 ID 的 person 实例，容器将会把参数 person 的内容拷贝进这个受管理的实例，merge()方法返回受管理的实例，但参数 person 仍然是分离的不受管理的。容器在决定 Flush 时把实例同步到数据库中。
2. 容器中不存在具有相同 ID 的 person 实例。容器根据传进的 person 参数 Copy 出一个受容器管理的 person 实例，同时 merge()方法会返回出这个受管理的实例，但参数 person 仍然是分离的不受管理的。容器在决定 Flush 时把实例同步到数据库中。

如果传递进 merge ()方法的参数不是实体 Bean，会引发一个 **IllegalArgumentException** 例外。

6.6.5 删除 Remove()

把 Entity 从数据库中删除。

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

如果传递进 remove ()方法的参数不是实体 Bean，会引发一个 **IllegalArgumentException** 例外。

6.6.6 执行 JPQL 操作 createQuery()

除了使用 find()或 getReference()方法来获得 Entity Bean 之外，你还可以通过 JPQL 得到实体 Bean。要执行 JPQL 语句，你必须通过 EntityManager 的 createQuery()或 createNamedQuery()方法创建一个 Query 对象。

```
@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

```

6.6.7 执行 SQL 操作 createNativeQuery()

注意这里操作的是 SQL 语句，并非 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一个 Entity 的实例，并将实例作为结果
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();

```

6.6.8 刷新实体 refresh()

如果你怀疑当前被管理的实体已经不是数据库中最新的数据，你可以通过 refresh()方法刷新实体，容器会把数据库中的新值重写进实体。这种情况一般发生在你获取了实体之后，有人更新了数据库中的记录，这时你需要得到最新的数据。当然你再次调用 find()或 getReference()方法也可以得到最新数据，但这种做法并不优雅。

```

@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);

```

如果传递进 refresh ()方法的参数不是实体 Bean，会引发一个 **IllegalArgumentException** 例外。

6.6.9 检测实体当前是否被管理中 contains()

contains()方法使用一个实体作为参数，如果这个实体对象当前正被持久化内容管理，返回值为 true，否则为 false。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
    }
}
```

6.6.10 分离所有当前正在被管理的实体 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 EntityManager 中分离出来，将会消耗你大量的内存。调用 EntityManager 的 clear()方法后，所有正在被管理的实体将会从持久化内容中分离出来。

有一点需要说明下，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），如果调用 clear()方法，之前对实体所作的任何改变将会掉失，所以建议你在调用 clear()方法之前先调用 flush()方法保存更改。

6.6.11 将实体的改变立刻刷新到数据库中 flush()

当实体管理器对象在一个 session bean 中使用时，它是和服务器的上下文绑定的。实体管理器在服务器的任务提交时提交并且同步它的内容。在一个 session bean 中，服务器的事务默认地会在调用堆栈的最后提交（如：方法的返回）。

```
@PersistenceContext
protected EntityManager em;

...

public void updatePerson(Person person) {
    try {
        Person person = em.find(Person.class, 2);
        person.setName("lihuoming");
        em.merge(person);
        //后面还有众多修改操作
    } catch (Exception e) {
        e.printStackTrace();
    }
}

//更新将会在这个方法的末尾被提交和刷新到数据库中
```

```
}
```

为了只在当事务提交时才将改变更新到数据库中，容器将所有数据库操作集中到一个批处理中，这样就减少了代价昂贵的与数据库的交互。

当你调用 `persist()`、`merge()` 或 `remove()` 这些方法时，更新并不会立刻同步到数据库中，直到容器决定刷新到数据库中时才会执行，默认情况下，容器决定刷新是在“相关查询”执行前或事务提交时发生，当然“相关查询”除 `find()` 和 `getReference()` 之外，这两个方法是不会引起容器触发刷新动作的，默认的刷新模式是可以改变的，具体请参考下节。如果你需要在事务提交之前将更新刷新到数据库中，你可以直接地调用 `EntityManager.flush()` 方法。这种情况下，你可以手工地来刷新数据库以获得对数据库操作的最大控制。

```
@PersistenceContext
protected EntityManager em;

...

public void updatePerson(Person person) {
    try {
        Person person = em.find(Person.class, 2);
        person.setName("lihuoming");
        em.merge(person);
        em.flush();//手动将更新立刻刷新进数据库
        //后面还有众多修改操作
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

6.6.12 改变实体管理器的 Flush 模式 setFlushMode()

我们对 Flush 模式进行修改需要使用到 `javax.persistence.FlushModeType`，他的定义如下：

```
public enum FlushModeType {
    AUTO,
    COMMIT
}
```

默认情况下，实体管理器的 Flush 模式为 `AUTO`，你可以改变他的值，如下：

```
entityManager.setFlushMode(FlushModeType.COMMIT);
```

下面说说这两者的区别及使用场合：

FlushModeType.AUTO: 刷新在查询语句执行前(除了 `find()` 和 `getReference()` 查询)或事务提交时才发生，使用场合：在大量更新数据的过程中没有任何查询语句(除了 `find()` 和 `getReference()` 查询)的执行。

FlushModeType.COMMIT: 刷新只有在事务提交时才发生，使用场合：在大量更新数据的过程中存在查询语句(除了 `find()` 和 `getReference()` 查询)的执行。

其实上面两种模式最终反映的结果是：**JDBC 驱动跟数据库交互的次数**。**JDBC 性能最大的增进是减少 JDBC 驱动与数据库之间的网络通讯**。**FlushModeType.COMMIT 模式使更新只在一次的网络交互中完成**，而 **FlushModeType.AUTO 模式可能需要多次交互（触发了多少次 Flush 就产生了多少次网络交互）**。

6.6.13 获取持久化实现者的引用 getDelegate()

用过 getDelegate() 方法, 你可以获取 EntityManager 持久化实现者的引用, 如 Jboss EJB3 的持久化产品采用 Hibernate, 可以通过 getDelegate() 方法获取对他的访问, 如:

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

获得对 Hibernate 的引用后, 可以直接面对 Hibernate 进行编码, 不过这种方法并不可取, 强烈建议不要使用。在 Weblogic 中, 你也可以通过此方法获取对 Kodo 的访问。

6.7 关系/对象映射

6.7.1 映射的表名或列名与数据库保留字同名时的处理

如果应用采用的数据库是 Mysql, 当映射的表名或列名与数据库保留字同名时, 持久化引擎转译后的 SQL 在执行时将会出错。

如:

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

表名 Order 与排序保留字 “Order”相同, 导致 SQL 语法出错。

针对上面的情况, 作者在 Jboss 持久化产品文档中没有找到相关的解决方案。在此作者采用了一种变通的方法来解决此问题。该方法针对具体数据库, 不利于数据库移植。建议大家在不得已的情况下使用。

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。Sqlserver 不加[]也能执行成功, 建议在出错的情况下使用。

6.7.2 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。下面就以订单为例介绍存在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

orders

版权所有: 黎活明

| 字段名称 | 字段类型属性 | 描述 |
|------------|----------|--------|
| orderid | Int | 订单号 |
| amount | float | 订单金额 |
| createdate | datetime | 订单创建日期 |

orderitems

| 字段名称 | 字段类型属性 | 描述 |
|-------------|--------------|--------|
| id | Int | 订单项 ID |
| productname | Varchar(255) | 订购产品名称 |
| price | float | 产品价格 |
| order_id | Int | 订单号 |

双向一对多关系，一是关系维护端（owner side），多是关系被维护端（inverse side）。在关系被维护端建立外键列指向关系维护端的主键列。

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;

@SuppressWarnings("serial")
@Entity
@Table(name = "Orders")
public class Order implements Serializable {
    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) {
        orderitem.setOrder(null);
        this.orderItems.remove(orderitem);
    }
}

```

上面声明一个 Set 变量 orderItems 用来存放多个 OrderItem 对象，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，下面是 @OneToMany 注释的属性介绍：

1>targetEntity

Class 类型的属性。

定义关系类的类型，默认是该成员属性对应的类类型，所以通常不需要提供定义。

2>mappedBy

String 类型的属性。

定义类之间的双向关系。如果类之间是单向关系，不需要提供定义，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，否则可能引起数据一致性的问题。

3>cascade

CascadeType[]类型。

该属性定义类和类之间的级联关系。定义的级联关系将被容器视为对当前类对象及其关联类对象采取相同的操作，而且这种关系是递归调用的。举个例子：Order 和 OrderItem 有级联关系，那么删除 Order 时将同时删除它对应的 OrderItem 对象。而如果 OrderItem 还和其他的对象之间有级联关系，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。还有一个选择是使用 CascadeType.ALL，表示选择全部四项。

4>fetch

FetchType 类型的属性。

可选择项包括：FetchType.EAGER 和 FetchType.LAZY。前者表示关系类(本例是 OrderItem 类)在主类(本例是 Order 类)加载的时候同时加载，后者表示关系类在被访问时才加载。默认值是 FetchType.LAZY。

@OrderBy(value = "id ASC")注释指明加载 OrderItem 时按 id 的升序排序

addOrderItem 和 removeOrderItem 方法用来添加/删除订单项。

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;
import javax.persistence.Table;

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

    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;
}

}
```

注释@ManyToOne指明 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。@ManyToOne 注释有四个属性: targetEntity、cascade、fetch 和 optional,前三个属性的具体含义和@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 属性是定义该关联类是否必须存在,值为 false 时,关联类双方都必须存在,如果关系被维护端不存在,

查询的结果为 null。值为 true 时，关系被维护端可以不存在，查询的结果仍然会返回关系维护端，在关系维护端中指向关系被维护端的属性为 null。optional 属性的默认值是 true。optional 属性实际上指定关联类与被关联类的 join 查询关系，如 optional=false 时 join 查询关系为 inner join，optional=true 时 join 查询关系为 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;
    }
}
//获取OrderItem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。
@ManyToOne(cascade=CascadeType. REFRESH, optional=true)
@JoinColumn(name = "order_id")
public Order getOrder() {
    return order;
}
//获取OrderItem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 映射表的 order_id 列作为外键与 Order 映射表的主键列关联。

为了使用上面的实体 Bean,我们定义一个 Session Bean 作为他的使用者。下面是 Session Bean 的业务接口，他定义了三个业务方法 insertOrder, getOrderById 和 getAllOrder,三个方法的业务功能是：

insertOrder 添加一个订单（带两个订单项）进数据库

getOrderById 获取指定订单号的订单

getAllOrder 获取所有订单

下面是 Session Bean 的业务接口及实现类

OrderDAO.java

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

public interface OrderDAO {

    public void insertOrder();
    public Order getOrderById(Integer orderid);
    public List getAllOrder();
}
```

OrderDAOBean.java

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

```

import java.util.Date;
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;
import com.foshanshop.ejb3.bean.OrderItem;

@Stateless
@Remote ({OrderDAO.class})
public class OrderDAOBean implements OrderDAO {
    @PersistenceContext
    protected EntityManager em;

    public void insertOrder() {
        Order order = new Order();
        order.setCreatedate(new Date());
        order.addOrderItem(new OrderItem("笔记本电脑", new Float(13200.5)));
        order.addOrderItem(new OrderItem("U盘", new Float(620)));
        order.setAmount(new Float(13200.5+620));
        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;
    }

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

```

上面有一点需要强调：当业务方法需要把一个实体 Bean 作为参数返回给客户端时，除了实体 Bean 本身需要实现

Serializable 接口之外, 如果关联类(**OrderItem**)是延迟加载, 还需在返回实体 **Bean** 之前通过访问关联类的方式加载关联类。否则在客户端访问关联类时将会抛出加载例外。另外不管是否延迟加载, 通过 **join fetch** 关联语句都可显式加载关联类, 如业务方法 **getAllOrder** 。

下面是 **Session Bean** 的 **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.*"%>

<%

    Properties props = new Properties();
    props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
    props.setProperty("java.naming.provider.url", "localhost:1099");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx = new InitialContext(props);
    try {
        OrderDAO orderdao = (OrderDAO) ctx.lookup("OrderDAOBean/remote");
        orderdao.insertOrder();
        /*
        Order order = orderdao.getOrderById(new Integer(1));
        out.println("订单总费用: "+ order.getAmount() + "<br>=====订单项
=====<br>");
        if (order!=null) {
            Iterator iterator = order.getOrderItems().iterator();
            while (iterator.hasNext()) {
                OrderItem SubOrder = (OrderItem) iterator.next();
                out.println("订购产品: "+ SubOrder.getProductname() + "<br>");
            }
        } else {
            out.println("没有找到相关订单");
        }
        */
        List list = orderdao.getAllOrder();
        if (list!=null) {
            for(int i=0; i<list.size();i++) {
                Order od = (Order) list.get(i);
                if (od!=null) {
                    out.println("=====订单号: "+ od.getOrderid()
+"=====<br>");
                    Iterator iterator = od.getOrderItems().iterator();
                    while (iterator.hasNext()) {
```

```

        OrderItem SubOrder = (OrderItem) iterator.next();
        out.println("订购产品:" + SubOrder.getProductname() + "<br>");
    }
}
}
} else {
    out.println("获取不到订单列表");
}
} catch (Exception e) {
    out.println(e.getMessage());
}
}
%>

```

本例子的 EJB 源代码在 OneToMany 文件夹（源代码下载：<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 OneToMany 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据源配置文件是 mysql-ds.xml，你可以在下载的文件中找到。数据库名为 foshanshop

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。

6.7.3 一对一映射

一个人（Person）只有唯一的身份证号（IDCard），Person 与 IDCard 是一对一关系。下面就以他们为例介绍存在一对一关系的实体 Bean 开发过程

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

| 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 |
| Person_ID | Int(11) not null | 作为外键指向 person 表的 personid |

一对一关系需要在关系维护端（owner side）的 @OneToOne 注释中定义 mappedBy 属性。在关系被维护端（inverse side）建立外键列指向关系维护端的主键列。

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.OneToOne;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@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;
    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, mappedBy = "person")
    public IDCard getIdcard() {
        return idcard;
    }

    public void setIdcard(IDCard idcard) {
        this.idcard = idcard;
    }
}

```

@OneToOne 注释指明 Person 与 IDCard 为一对一关系，@OneToOne 注释五个属性：targetEntity、cascade、fetch、optional 和 mappedBy，前四个属性的具体含义与 @ManyToOne 注释的同名属性一一对应，请大家参考前面章节中的内容，fetch 属性默认值是 FetchType.EAGER。mappedBy 属性的具体含义与 @OneToMany 注释的同名属性相同。上面的 optional = true 设置 idcard 属性可以为 null，也就是允许没有身份证，未成年人就是没有身份证的。

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.JoinColumn;

```

```
import javax.persistence.OneToOne;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Table(name = "IDCard")
public class IDCard implements Serializable{
    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)
    @JoinColumn(name = "Person_ID", referencedColumnName = "personid", unique = true)
    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }
}
```

```
}

```

@OneToOne 注释指明 IDCard 与 Person 为一对一关系, IDCard 是关系被维护端, optional = false 设置 person 属性值不能为 null, 也就是身份证必须有对应的主人。@JoinColumn(name = "Person_ID", referencedColumnName = "personid", unique = true)指明 IDCard 对应表的 Person_ID 列作为外键与 Person 对应表的 personid 列进行关联, unique = true 指明 Person_ID 列的值不可重复。

为了使用上面的实体 Bean,我们定义一个 Session Bean 作为他的使用者。下面是 Session Bean 的业务接口, 他定义了四个业务方法 insertPerson, getPersonByID, updatePersonInfo 和 deletePerson, 四个方法的业务功能是:

insertPerson 添加一个人员 (带一个身份证) 进数据库

getPersonByID 获取指定编号的人员

updatePersonInfo 更新人名及身份证号

deletePerson 删除人员, 连同其身份证一起删除

下面是 Session Bean 的业务接口及实现类

OneToOneDAO.java

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

public interface OneToOneDAO {

    public void insertPerson(String name, boolean sex, short age, Date birthday, String cardID);
    public Person getPersonByID(Integer orderid);
    public void updatePersonInfo(Integer personid, String newname, String newIDcard);
    public void deletePerson(Integer personid);

}
```

OneToOneDAOBean.java

```
package com.foshanshop.ejb3.impl;
import java.util.Date;
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.IDCard;
import com.foshanshop.ejb3.bean.Person;

@Stateless
@Remote ({OneToOneDAO.class})
public class OneToOneDAOBean implements OneToOneDAO {

    @PersistenceContext
    protected EntityManager em;
```

```

public void insertPerson(String name, boolean sex, short age, Date birthday, String cardID) {
    Person person = new Person();
    person.setName(name);
    person.setSex(sex);
    person.setAge(Short.valueOf(age));
    person.setBirthday(birthday);
    IDCard idcard = new IDCard(cardID);
    idcard.setPerson(person);
    person.setIdcard(idcard);
    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);
        }
        em.merge(person);
    }
}

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

```

下面是 Session Bean 的 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.util.Date,
                java.text.SimpleDateFormat,
                java.util.*"%>
<%

```

```

Properties props = new Properties();
props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
props.setProperty("java.naming.provider.url", "localhost:1099");
props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

InitialContext ctx = new InitialContext(props);
try {
    String outformat = "<font color=blue>CMD>>Out>></font> ";
    OneToOneDAO oneToonedao = (OneToOneDAO) ctx.lookup("OneToOneDAOBean/remote");
    SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
    SimpleDateFormat formatter1 = new SimpleDateFormat("MMddhhmmss");
    String endno = formatter1.format(new Date()).toString();
    oneToonedao.insertPerson("黎活明", true, (short)26, formatter.parse("1980-9-30"),
"44011"+endno);
    //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(outformat + "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(outformat + "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 EJB 源代码在 OneToOne 文件夹（源代码下载:<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 OneToOne 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据源配置文件是 mysql-ds.xml，你可以在下载的文件中找到。数据库名为 foshanshop

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

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。

6.7.4 多对多映射

学生和老师就是多对多的关系。一个学生有多个老师，一个老师教多个学生。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。

下面就以学生和老师为例介绍多对多关系的实体 Bean 开发
需要映射的数据库表

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 | 是 student 表 studentid 列的外键 |
| Teacher_ID | Int(11) not null | 是 teacher 表 teacherid 列的外键 |

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;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Table(name = "Student")
public class Student implements Serializable{
    private Integer studentid;
    private String StudentName;
    private Set<Teacher> teachers = new HashSet<Teacher>();

    public Student() {}
```

```
public Student(String studentName) {
    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) {
    StudentName = studentName;
}

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

public void setTeachers(Set<Teacher> teachers) {
    this.teachers = teachers;
}
}
```

@ManyToMany 注释表示 Student 是多对多关系的一边，mappedBy 属性定义了 Student 为双向关系的维护端 (owning side)。

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;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Table(name = "Teacher")
public class Teacher implements Serializable{
    private Integer teacherid;
    private String TeacherName;
    private Set<Student> students = new HashSet<Student>();

    public Teacher() {}

    public Teacher(String teacherName) {
        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) {
        TeacherName = teacherName;
    }

    @ManyToMany(cascade = CascadeType.PERSIST, fetch = FetchType.LAZY)
    @JoinTable(name = "Teacher_Student",
        joinColumns = {@JoinColumn(name = "Teacher_ID", referencedColumnName = "teacherid")},
        inverseJoinColumns = {@JoinColumn(name = "Student_ID", referencedColumnName =
"studentid")})
    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) {
    this.students.remove(student);
}
}

```

@ManyToMany 注释表示 Teacher 是多对多关系的一端。@JoinTable 描述了多对多关系的数据表关系。name 属性指定中间表名称, joinColumns 定义中间表与 Teacher 表的外键关系。上面的代码中, 中间表 Teacher_Student 的 Teacher_ID 列是 Teacher 表的主键列对应的外键列, inverseJoinColumns 属性定义了中间表与另外一端(Student) 的外键关系。

为了使用上面的实体 Bean, 我们定义一个 Session Bean 作为他的使用者。下面是 Session Bean 的业务接口, 他定义了三个业务方法 insertTeacher, getTeacherByID, 和 getStudentByID, 三个方法的业务功能是:

insertTeacher 添加一个教师(带学生)进数据库

getTeacherByID 获取指定编号的教师

getStudentByID 获取指定编号的学生

下面是 Session Bean 的业务接口及实现类

TeacherDAO.java

```

package com.foshanshop.ejb3;
import com.foshanshop.ejb3.bean.Student;
import com.foshanshop.ejb3.bean.Teacher;

public interface TeacherDAO {
    public void insertTeacher(String name, String[] studentnames);
    public Teacher getTeacherByID(Integer teacherid);
    public Student getStudentByID(Integer studentid);
}

```

TeacherDAOBean.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.TeacherDAO;
import com.foshanshop.ejb3.bean.Student;

```

```

import com.foshanshop.ejb3.bean.Teacher;

@Stateless
@Remote ({TeacherDAO.class})
public class TeacherDAOBean implements TeacherDAO {
    @PersistenceContext
    protected EntityManager em;

    public void insertTeacher(String name, String[] studentnames) {
        Teacher teacher = new Teacher(name);
        if (studentnames!=null) {
            for(int i=0;i<studentnames.length; i++){
                teacher.addStudent(new Student(studentnames[i]));
            }
        }
        em.persist(teacher);
    }

    public Teacher getTeacherByID(Integer teacherid) {
        Teacher teacher= em.find(Teacher.class, teacherid);
        if (teacher!=null) teacher.getStudents().size();
        return teacher;
    }

    public Student getStudentByID(Integer studentid) {
        Student student= em.find(Student.class, studentid);
        if (student!=null) student.getTeachers().size();
        return student;
    }
}

```

下面是 Session Bean 的 JSP 客户端代码:

ManyToManyTest.jsp

```

<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.TeacherDAO,
                com.foshanshop.ejb3.bean.*,
                javax.naming.*,
                java.util.*"%>

<%
    Properties props = new Properties();
    props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
    props.setProperty("java.naming.provider.url", "localhost:1099");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

```


6.8 使用参数查询

参数查询也和 SQL 中的参数查询类似。EJB3 QL 支持两种方式的参数定义方式：命名参数和位置参数。在同一个查询中只允许使用一种参数定义方式。

6.8.1 命名参数查询

命名参数的格式为：“: + 参数名”

```
@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** NameQuery 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            Person person= (Person)iterator.next();
            out.append(person.getName()+ "<BR>");
        }
    }
    return out.toString();
}
```

6.8.2 位置参数查询

位置参数的格式为“?+位置编号”

```
@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** PositionQuery 结果打印
*****<BR>");
    if (result!=null){
```

```

        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            Person person= (Person)iterator.next();
            out.append(person.getName()+ "<BR>");
        }
    }
    return out.toString();
}

```

6.8.3 Date 参数

如果你需要传递 `java.util.Date` 或 `java.util.Calendar` 参数进一个参数查询，你需要使用一个特殊的 `setParameter()` 方法，相关的 `setParameter` 方法定义如下：

```

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。

6.9 JPQL 语言

Java Persistence API 定义了一种查询语言，具有与 SQL 相类似的特征，JPQL 是完全面向对象的，具备继承、多态和关联等特性。本教程只介绍常用的语法，要了解更全面的知识请参考持久化产品使用手册（如

Hibernate,Kodo,Toplink)。

本小节把程序的测试框架搭建起来,方便各位后面学习 JPQL 语句。本例子的实体 Bean 有 Person,Order,OrderItem,他们之间的关系是: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.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@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;
    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;
    }
}

```

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;

@SuppressWarnings("serial")
@Entity
@Table(name = "Orders")
public class Order implements Serializable {
    private int hashCode = Integer.MIN_VALUE;
    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) {
    orderitem.setOrder(null);
    this.orderItems.remove(orderitem);
}
```

```

@Temporal(value=TemporalType.TIMESTAMP)
public Date getCreateddate() {
    return createddate;
}

public void setCreateddate(Date createddate) {
    this.createddate = createddate;
}

public boolean equals (Object obj) {
    if (null == obj) return false;
    if (!(obj instanceof Order)) return false;
    else {
        Order mObj = (Order) obj;
        if (null == this.getOrderid() || null == mObj.getOrderid()) return false;
        else return (this.getOrderid().equals(mObj.getOrderid()));
    }
}

public int hashCode () {
    if (Integer.MIN_VALUE == this.hashCode) {
        if (null == this.getOrderid()) return super.hashCode();
        else {
            String hashStr = this.getClass().getName() + ":" + this.getOrderid().hashCode();
            this.hashCode = hashStr.hashCode();
        }
    }
    return this.hashCode;
}
}

```

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;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity

```

```
@Table(name = "OrderItems")
public class OrderItem implements Serializable {

    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;
    }
}

```

下面是实体 Bean 的使用者 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.Iterator;
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})
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;
    }
    return result;
}
...这里省略后面的代码
}

```

下面是 Session Bean 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.Date,
    java.text.SimpleDateFormat,
    java.util.*"%>

<%

    Properties props = new Properties();
    props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
    props.setProperty("java.naming.provider.url", "localhost:1099");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    try {
        String index = request.getParameter("index");
        if (index!=null && !"".equals(index.trim())){
            InitialContext ctx = new InitialContext(props);
            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 EJB 源代码在 Query 文件夹（源代码下载：<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 Query 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据源配置文件是 mysql-ds.xml，你可以在下载的文件中找到。数据库名为 foshanshop

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

本例子的客户端代码在 EJCTest 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/EJCTest/QueryTest.jsp?index=1> (index 参数值可以是 1 到 29) 访问客户端。

6.9.1 大小写敏感性(Case Sensitivity)

除了 Java 类和属性名称外，查询都是大小写不敏感的。所以，SeLeCT 和 sELEct 以及 SELECT 相同的，但是 com.foshanshop.ejb3.bean.Person 和 com.foshanshop.ejb3.bean.PERSon 是不同的， person.name 和 person.NAME 也是不同的。

6.9.2 命名查询

你可以在实体 bean 上预先定义一个或多个查询语句，减少每次因书写错误而引起的 BUG。通常把经常使用的查询语句定义成命名查询，代码如下：

```
@NamedQuery(name="getPerson", query="FROM Person WHERE personid=?1")
@Entity
@Table(name = "Person")
public class Person implements Serializable{
```

上面@NamedQuery.name() 为查询定义一个名称，@NamedQuery.query() 定义你的查询语句。如果你需要定义多个命名查询，应在@javax.persistence.NamedQueries 注释里定义@NamedQuery，代码如下：

```
@NamedQueries({
    @NamedQuery(name="getPerson", query="FROM Person WHERE personid=?1"),
    @NamedQuery(name="getPersonList", query="FROM Person WHERE age>?1")
})
@Entity
@Table(name = "Person")
public class Person implements Serializable{
```

当命名查询定义好了之后，我们就可以通过名称执行其查询。代码如下：

```
Query query = em.createNamedQuery("getPerson");
query.setParameter(1, 1);
```

6.9.3 排序(order by)

下面是一个简单查询的例子，可以看到 EJB3 QL 和 SQL 的使用方法很类似。"ASC"和"DESC"分别为升序和降序，如果不显式注明，EJB3 QL 中默认为 asc 升序。（例子的源代码在 Query 文件夹）

```

@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryOrderBy 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            Person person= (Person)iterator.next();
            out.append(person.getName()+ "<BR>");
        }
    }
    return out.toString();
}

```

6.9.4 查询部分属性

在前面的例子中，都是对针对 Entity 类的查询，返回的也是被查询的 Entity 类的实体。EJB3 QL 也允许我们直接查询返回我们需要的属性，而不是返回整个 Entity。在一些 Entity 中属性特别多的情况，这样的查询可以提高性能。(例子的源代码在 Query 文件夹)

```

@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryPartAttribute 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            //取每一行
            Object[] row = ( Object[]) iterator.next();
            //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+ "; Person Name="+PersonName+ "<BR>");
        }
    }
}

```

```

    }
}
return out.toString();
}

```

6.9.5 查询中使用构造器(Constructor)

EJB3 QL 支持将查询的属性结果直接作为一个 java class 的构造器参数，并产生实体作为结果返回。(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。(例子的源代码在 Query 文件夹)

```

@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryConstructor 结果打印
*****<BR>");
    if (result!=null){

```

```

        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            SimplePerson simpleperson = (SimplePerson) iterator.next();
            out.append("人员介绍: "+ simpleperson.getDescription()+ "<BR>");
        }
    }
    return out.toString();
}

```

6.9.6 聚合查询(Aggregation)

象大部分的 SQL 一样, EJB3 QL 也支持查询中的聚合函数。目前 EJB3 QL 支持的聚合函数包括:

1. AVG()
2. SUM()
3. COUNT() , 返回类型为 Long, 注意 count(*)语法在 hibernate 中可用, 但在 toplink 其它产品中并不可用
4. MAX()
5. MIN()

(例子的源代码在 Query 文件夹)

```

@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)

```

@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryGroupBy 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            //取每一行
            Object[] row = (Object[]) iterator.next();
            //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)

```

@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(*)

```

```

>?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 result = query.getResultList();
StringBuffer out = new StringBuffer("***** QueryGroupByHaving 结果打印
*****<BR>");
if (result!=null){
    Iterator iterator = result.iterator();
    while( iterator.hasNext() ){
        //取每一行
        Object[] row = (Object[]) iterator.next();
        //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();
}

```

6.9.7 关联(join)

在 EJB3 QL 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, 都是允许符合条件的右边表达式中的 Entities 为空。(例子的源代码在 Query 文件夹)

```

@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryLeftJoin 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        Integer orderid = null;

```

```

        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            if (orderid==null || !orderid.equals(order.getOrderid())){
                orderid = order.getOrderid();
                out.append("订单号: "+ orderid+ "<BR>");
            }
        }
    }
    return out.toString();
}

```

这个句 EJB3 QL 编译成以下的 SQL。

```

select order0_.orderid as orderid18_, order0_.amount as amount18_, order0_.person_id as person4_18_,
order0_.createdate as createdate18_ from Orders order0_ left outer join OrderItems orderitems1_ on
order0_.orderid=orderitems1_.order_id, Person person2_ where order0_.person_id=person2_.personid and
person2_.age=26 order by order0_.orderid

```

需要显式使用 left join/left outer join 的情况会比较少。

inner join 要求右边的表达式必须返回 Entities。(例子的源代码在 Query 文件夹)

```

@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryInnerJoin 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        Integer orderid = null;
        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            if (orderid==null || !orderid.equals(order.getOrderid())){
                orderid = order.getOrderid();
                out.append("订单号: "+ orderid+ "<BR>");
            }
        }
    }
    return out.toString();
}

```

这个句 EJB3 QL 编译成以下的 SQL。


```
select order0_.orderid as orderid18_, order0_.amount as amount18_, order0_.person_id as person4_18_,
order0_.createdate as createdate18_ from Orders order0_ inner join OrderItems orderitems1_ on
order0_.orderid=orderitems1_.order_id, Person person2_ where order0_.person_id=person2_.personid and
person2_.age=26 order by order0_.orderid
```

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。在默认查询中，Entity 中的集合属性默认不会被关联，集合属性默认是缓加载(lazy-load)。如下：(例子的源代码在 Query 文件夹)

```
@PersistenceContext
protected EntityManager em;
...
private String QueryInnerJoinLazyLoad(){
    // 默认 EJB3 QL 编译后不关联集合属性变量(orderItems)对应的表
    Query query = em.createQuery("select o from Order o inner join o.orderItems where o.ower.age=26
order by o.orderid");
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryInnerJoinLazyLoad 结果打印
*****<BR>");
    if (result!=null){
        if( result.size(>0){
            //这时获得 Order 实体中 orderItems( 集合属性变量 )为空
            Order 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

            Set<OrderItem> list = order.getOrderItems();
            Iterator<OrderItem> iterator = list.iterator();
            if (iterator.hasNext()){
                OrderItem orderItem =iterator.next();
                out.append("订购产品名: "+ orderItem.getProductname()+ "<BR>");
            }
        }
    }
    return out.toString();
}
```

在执行"select o from Order o inner join o.orderItems where o.ower.age=26 order by o.orderid"时编译成的 SQL 如下：他不包含集合属性变量(orderItems)对应表的字段

```
select order0_.orderid as orderid6_, order0_.amount as amount6_, order0_.person_id as person4_6_,
order0_.createdate as createdate6_ from Orders order0_ inner join OrderItems orderitems1_ on
order0_.orderid=orderitems1_.order_id, Person person2_ where order0_.person_id=person2_.personid and
person2_.age=26 order by order0_.orderid
```

当执行到 Set<OrderItem> list = order.getOrderItems();时才会执行一条 SQL 语句来加载属于当前 Order 的 OrderItems，编译成的 SQL 如下：

```
select orderitems0_.order_id as order4_1_, orderitems0_.id as id1_, orderitems0_.id as id7_0_,
orderitems0_.order_id as order4_7_0_, orderitems0_.productname as productn2_7_0_, orderitems0_.price as
```

```
price7_0_ from OrderItems orderitems0_ where orderitems0_.order_id=? order by orderitems0_.id ASC
```

这样的查询性能上有不足的地方。为了查询 N 个 Order，我们需要一条 SQL 语句获得所有的 Order 的原始对象属性，但需要另外 N 条语句获得每个 Order 的 orderItems 集合属性。为了避免 N+1 的性能问题，我们可以利用 join fetch 一次过用一条 SQL 语句把 Order 的所有信息查询出来。如下：(例子的源代码在 Query 文件夹)

```
@PersistenceContext
protected EntityManager em;
...
private String QueryJoinFetch(){
    //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 fetch o.orderItems where
o.ower.age=26 order by o.orderid");
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryJoinFetch 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        Integer orderid = null;
        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            if (orderid==null || !orderid.equals(order.getOrderid())){
                orderid = order.getOrderid();
                out.append("订单号: "+orderid+ "<BR>");
            }
        }
    }
    return out.toString();
}
```

这个句 EJB3 QL 编译成以下的 SQL。

```
select order0_.orderid as orderid18_0_, orderitems1_.id as id19_1_, order0_.amount as amount18_0_,
order0_.person_id as person4_18_0_, order0_.createdate as createdate18_0_, orderitems1_.order_id as
order4_19_1_, orderitems1_.productname as productn2_19_1_, orderitems1_.price as price19_1_,
orderitems1_.order_id as order4_0_, orderitems1_.id as id0__ from Orders order0_ inner join OrderItems
orderitems1_ on order0_.orderid=orderitems1_.order_id, Person person2_ where
order0_.person_id=person2_.personid and person2_.age=26 order by order0_.orderid, orderitems1_.id ASC
```

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 N+1 条 SQL 语句在性能上有了极大的提升。

6.9.8 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 "
```

版权所有：黎活明

当有 N 个 orderItem 时就会产生 N 个 Order,而有些 Order 对象往往是相同的,这时我们需要使用 DISTINCT 关键字来排除掉相同的对象。

```
@PersistenceContext
protected EntityManager em;

...

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

6.9.9 比较 Entity

在查询中使用参数查询时,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 Entity 的实例。(例子的源代码在 Query 文件夹)

```
@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryEntityParameter 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        Integer orderid = null;
        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            if (orderid==null || !orderid.equals(order.getOrderid())){
                orderid = order.getOrderid();
                out.append("订单号: "+ orderid+ "<BR>");
            }
        }
    }
    return out.toString();
}
```

6.9.10 批量更新(Batch Update)

EJB3 QL 支持批量更新。(例子的源代码在 Query 文件夹)

```
@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();
}
```

6.9.11 批量删除(Batch Remove)

EJB3 QL 支持批量删除。(例子的源代码在 Query 文件夹)

```
@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(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();
}
```

6.9.12 使用操作符 NOT

(例子的源代码在 Query 文件夹)

```
@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryNOTOperate 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        Integer orderid = null;
        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            if (orderid==null || !orderid.equals(order.getOrderid())){
                orderid = order.getOrderid();
                out.append("订单号: "+ orderid+ "<BR>");
            }
        }
    }
    return out.toString();
}

```

6.9.13 使用操作符 BETWEEN

(例子的源代码在 Query 文件夹)

```

@PersistenceContext
protected EntityManager em;
...
private String QueryBETWEENOperate(){
    //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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryBETWEENOperate 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        Integer orderid = null;
        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            if (orderid==null || !orderid.equals(order.getOrderid())){

```

```

       orderid = order.getOrderid();
        out.append("订单号: "+orderid+ "<BR>");
    }
}
return out.toString();
}

```

6.9.14 使用操作符 IN

(例子的源代码在 Query 文件夹)

```

@PersistenceContext
protected EntityManager em;
...
private String QueryINOperate(){
    //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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryINOperate 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            Person person= (Person)iterator.next();
            out.append(person.getName()+ "<BR>");
        }
    }
    return out.toString();
}

```

6.9.15 使用操作符 LIKE

(例子的源代码在 Query 文件夹)

```

@PersistenceContext
protected EntityManager em;
...
private String QueryLIKEOperate(){
    //查找以字符串"li"开头的 Person
    Query query = em.createQuery("select p from Person as p where p.name like 'li%'");
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryLIKEOperate 结果打印
*****<BR>");
}

```

```

if (result!=null){
    out.append("----- 查找以字符串\"li\"开头的 Person -----<BR>");
    Iterator iterator = result.iterator();
    while( iterator.hasNext() ){
        Person person= (Person)iterator.next();
        out.append(person.getName()+ "<BR>");
    }
}
//可以结合 NOT 一起使用，比如查询所有 name 不以字符串"ming"结尾的 Person
query = em.createQuery("select p from Person as p where p.name not like '%ming'");
result = query.getResultList();
if (result!=null){
    out.append("----- 查询所有 name 不以字符串\"ming\"结尾的 Person -----<BR>");
    Iterator iterator = result.iterator();
    while( iterator.hasNext() ){
        Person person= (Person)iterator.next();
        out.append(person.getName()+ "<BR>");
    }
}
return out.toString();
}

```

6.9.16 使用操作符 IS NULL

(例子的源代码在 Query 文件夹)

```

@PersistenceContext
protected EntityManager em;
...
private String QueryISNULLOperate(){
    //查询含有购买者的所有 Order
    Query query = em.createQuery("select o from Order as o where o.ower is not null order by
o.orderid");
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryISNULLOperate 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        out.append("----- 查询含有购买者的所有 Order -----<BR>");
        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            out.append("订单号: "+ order.getOrderid()+ "<BR>");
        }
    }
}

```

```

//查询没有购买者的所有 Order
query = em.createQuery("select o from Order as o where o.ower is null order by o.orderid");
result = query.getResultList();
if (result!=null){
    Iterator iterator = result.iterator();
    out.append("----- 查询没有购买者的所有 Order -----<BR>");
    while( iterator.hasNext() ){
        Order order = (Order) iterator.next();
        out.append("订单号: "+ order.getOrderid()+ "<BR>");
    }
}

return out.toString();
}

```

6.9.17 使用操作符 IS EMPTY

IS EMPTY 是针对集合属性(Collection)的操作符。可以和 NOT 一起使用。(例子的源代码在 Query 文件夹)。注：低版权的 Mysql 不支持 IS EMPTY

```

@PersistenceContext
protected EntityManager em;
...
private String QueryISEMPTYOperate(){
    //查询含有订单项的所有 Order
    Query query = em.createQuery("select o from Order as o where o.orderItems is not empty order by o.orderid");
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryISEMPTYOperate 结果打印 *****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        out.append("----- 查询含有订单项的所有 Order -----<BR>");
        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            out.append("订单号: "+ order.getOrderid()+ "<BR>");
        }
    }
    //查询没有订单项的所有 Order
    query = em.createQuery("select o from Order as o where o.orderItems is empty order by o.orderid");
    result = query.getResultList();
    if (result!=null){
        Iterator iterator = result.iterator();
        out.append("----- 查询没有订单项的所有 Order -----<BR>");
    }
}

```



```

        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            out.append("订单号: "+ order.getOrderid()+ "<BR>");
        }
    }

    return out.toString();
}

```

6.9.18 使用操作符 EXISTS

[NOT]EXISTS 需要和子查询配合使用。(例子的源代码在 Query 文件夹)。注：低版权的 Mysql 不支持 EXISTS

```

@PersistenceContext
protected EntityManager em;
...
private String QueryEXISTSOperate(){
    //如果存在订单号为 1 的订单，就获取所有 OrderItem
    Query query = em.createQuery("select oi from OrderItem as oi where exists (select o from Order o
where o.orderid=1)");
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryEXISTSOperate 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        out.append("----- 如果存在订单号 1，就获取所有 OrderItem -----<BR>");
        while( iterator.hasNext() ){
            OrderItem item = (OrderItem) iterator.next();
            out.append("所有订购的产品名: "+ item.getProductname()+ "<BR>");
        }
    }
    //如果不存在订单号为 10 的订单，就获取 id 为 1 的 OrderItem
    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)");
    result = query.getResultList();
    if (result!=null){
        Iterator iterator = result.iterator();
        out.append("----- 如果不存在订单号 10，就获取 id 为 1 的 OrderItem
-----<BR>");
        if( iterator.hasNext() ){
            OrderItem item = (OrderItem) iterator.next();
            out.append("订单项 ID 为 1 的订购产品名: "+ item.getProductname()+ "<BR>");
        }
    }
}

```

```

        return out.toString();
    }

```

6.9.19 字符串函数

EJB3 QL 定义了内置函数方便使用。这些函数的使用方法和 SQL 中相应的函数方法类似。EJB3 QL 中定义的字符串函数包括：

1. CONCAT 字符串拼接
2. SUBSTRING 字符串截取
3. TRIM 去掉空格
4. LOWER 转换成小写
5. UPPER 转换成大写
6. LENGTH 字符串长度
7. LOCATE 字符串定位

(例子的源代码在 Query 文件夹)

```

@PersistenceContext
protected EntityManager em;
...
private String QueryStringOperate(){
    //查询所有人员，并在姓名后面加上字符串"_foshan"
    Query query = em.createQuery("select p.personid, concat(p.name, '_foshan') from Person as p");
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryStringOperate 结果打印
*****<BR>");
    if (result!=null){
        out.append("----- 查询所有人员，并在姓名后面加上字符串\"_foshan\" -----<BR>");
        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            //取每一行
            Object[] row = ( Object[]) iterator.next();
            //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+ "; Person Name="+PersonName+ "<BR>");
        }
    }

    //查询所有人员,只取姓名的前三个字符
    query = em.createQuery("select p.personid, substring(p.name,1,3) from Person as p");
    result = query.getResultList();
    if (result!=null){
        out.append("----- 查询所有人员,只取姓名的前三个字符 -----<BR>");
    }
}

```

```

        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            //取每一行
            Object[] row = ( Object[]) iterator.next();
            //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+ "; Person Name="+PersonName+ "<BR>");
        }
    }
    return out.toString();
}

```

6.9.20 计算函数

EJB3 QL 中定义的计算函数包括:

ABS 绝对值

SQRT 平方根

MOD 取余数

SIZE 取集合的数量

(例子的源代码在 Query 文件夹)

```

@PersistenceContext
protected EntityManager em;
...
private String QueryMathLOperate(){
    //查询所有 Order 的订单号及其订单项的数量
    Query query = em.createQuery("select o.orderid, size(o.orderItems) from Order as o group by
o.orderid");
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryMathLOperate 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        out.append("----- 查询所有 Order 的订单号及其订单项的数量 -----<BR>");
        while( iterator.hasNext() ){
            //取每一行
            Object[] row = ( Object[]) iterator.next();
            out.append("订单号: "+ row[0].toString()+ "; 订单项共"+row[1].toString()+ "项<BR>");
        }
    }
    //查询所有 Order 的订单号及其总金额/10 的余数
    query = em.createQuery("select o.orderid, mod(o.amount, 10) from Order as o");
    result = query.getResultList();
}

```

```

if (result!=null){
    Iterator iterator = result.iterator();
    out.append("----- 查询所有 Order 的订单号及其总金额/10 的余数 -----<BR>");
    while( iterator.hasNext() ){
        //取每一行
        Object[] row = ( Object[]) iterator.next();
        out.append("订单号: "+ row[0].toString()+ "; 总金额/10 的余数:"+row[1].toString()+
"<BR>");
    }
}
return out.toString();
}

```

6.9.21 子查询

子查询可以用于 WHERE 和 HAVING 条件语句中。(例子的源代码在 Query 文件夹)。注：低版权的 Mysql 不支持子查询

```

@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 result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QuerySubQueryOperate 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        out.append("----- 查询年龄为 26 岁的购买者的所有 Order -----<BR>");
        while( iterator.hasNext() ){
            Order order = (Order) iterator.next();
            out.append("订单号: "+ order.getOrderid()+ "<BR>");
        }
    }
    return out.toString();
}

```

6.9.22 结果集分页

有些时候当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（假如返回的记录有 3 条，容器会自动为记录编上索引，索引从 0 开始，依次为 0，1，2）

单表映射项目（EntityBean）中 `PersonDAOBean` 的代码片断：

```
public List getPersonList(int max, int whichpage) {
    try {
        int index = (whichpage-1) * max;
        Query query = em.createQuery("from Person p order by personid asc");
        List list = query.setMaxResults(max).
            setFirstResult(index).
            getResultList();
        em.clear(); //分离内存中受EntityManager管理的实体bean，让VM进行垃圾回收
        return list;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

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.Properties,
    java.util.List,
    java.util.Iterator"%>

<%
    Properties props = new Properties();
    props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
    props.setProperty("java.naming.provider.url", "localhost:1099");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx = new InitialContext(props);
    try {
        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 list = persondao.getPersonList(max, whichpage);
        if (list!=null) {
```

```

        Iterator it = list.iterator();
        while (it.hasNext()) {
            Person p = (Person)it.next();
            out.println("人员编号:" + p.getPersonid() + " 姓名: " + p.getName() + "<br>");
        }
    }
} catch (Exception e) {
    out.println(e.getMessage());
}
%>

```

上面代码中设置每页记录数为 2，如果数据库存在 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)等等。

如果我们需要显示大量的实体 Bean，假如记录成千上万，通过简单的循环处理，我们将很快消耗完内存，为此我们需要在显示了一部分实体 bean 之后调用 EntityManager.clear()及时把这些实体 bean 从 EntityManager 分离出来，让 Java VM 对他们进行垃圾回收。

在分页中，为了数据排列有序，建议在查询中针对某些字段进行排序。

6.10 调用存储过程

要调用存储过程，我们可以通过 EntityManager 对象的 createNativeQuery()方法执行 SQL 语句(注意：这里说的是 SQL 语句，不是 EJB3 QL)，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的调用方法。

6.10.1 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，他的 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;

```

下面的代码片断展示了无返回值存储过程的调用方法：

```

@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，在运行本例子，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。

6.10.2 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，他有一个 `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，你也可以这样写：`select 'foshanren'`

下面的代码片断展示了返回单值的存储过程的调用方法：

```

@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，在运行本例子，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。

6.10.3 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，他的 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;
```

下面的代码片断展示了返回表全部列的存储过程的调用方法，我们可以让 EJB3 Persistence 运行环境将列值直接填充入一个 Entity 的实例（本例填充进 `Person` 对象），并将实例作为结果返回。

```
@PersistenceContext
protected EntityManager em;
...
private String QueryStoreProcedure(){
    //调用返回 Person 全部列的存储过程
    Query query = em.createNativeQuery("{call GetPersonList()}", Person.class);
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryStoreProcedure 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            Person person= (Person)iterator.next();
            out.append(person.getName()+ "<BR>");
        }
    }
    return out.toString();
}
```

例子的源代码在 Query 文件夹，在运行本例子，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。

6.10.4 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，他的 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'

下面的代码片断展示了返回部分列的存储过程的调用方法。

```

@PersistenceContext
protected EntityManager em;
...
private String QueryPartColumnStoreProcedure(){
    //调用返回部分列的存储过程
    Query query = em.createNativeQuery("{call GetPersonPartProperties()}");
    List result = query.getResultList();
    StringBuffer out = new StringBuffer("***** QueryPartColumnStoreProcedure 结果打印
*****<BR>");
    if (result!=null){
        Iterator iterator = result.iterator();
        while( iterator.hasNext() ){
            //取每一行
            Object[] row = ( Object[]) iterator.next();
            //数组中的第一个值是 personid
            int personid = Integer.parseInt(row[0].toString());
            String PersonName = row[1].toString();
            out.append("人员 ID="+ personid+ "; 姓名="+PersonName+ "<BR>");
        }
    }
    return out.toString();
}

```

例子的源代码在 Query 文件夹，在运行本例子，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。

6.11 事务管理服务

最有用的容器服务可能就是事务管理服务，当应用出现失败或异常时，它保证了数据库的完整性。你可以简单地将为一个 POJO 方法申明它的事务属性。这样容器就可以在合适的上下文中运行这个方法。最常见的事务是定义在 session bean 的方法上，方法中所有的数据库操作只有在方法正常退出时才会提交，如果方法抛出未捕获的异常，事务管理将回滚所有的变更。

@TransactionAttribute 注释用作定义一个需要事务的方法。它可以有以下参数：

1.REQUIRED：方法在一个事务中执行，如果调用的方法已经在一个事务中，则使用该事务，否则将创建一个新的事务。

2.MANDATORY：如果运行于事务中的客户调用了该方法，方法在客户的事务中执行。如果客户没有关联到

事务中，容器就会抛出 `TransactionRequiredException`。如果企业 bean 方法必须用客户事务则采用 `Mandatory` 属性。

3.`REQUIRESNEW`:方法将在一个新的事务中执行，如果调用的方法已经在事务中，则暂停旧的事务。在调用结束后恢复旧的事务。

4.`SUPPORTS`:如果方法在一个事务中被调用，则使用该事务，否则不使用事务。

5.`NOT_SUPPORTED`: 如果方法在一个事务中被调用，容器会在调用之前中止该事务。在调用结束后，容器会恢复客户事务。如果客户没有关联到一个事务中，容器不会在运行入该方法前启动一个新的事务。用 `NotSupported` 属性标识不需要事务的方法。因为事务会带来更高的性能支出，所以这个属性可以提高性能。

6.`Never`: 如果在一个事务中调用该方法，容器会抛出 `RemoteException`。如果客户没有关联到一个事务中，容器不会在运行入该方法前启动一个新的事务。

如果没有指定参数，`@TransactionAttribute` 注释使用 `REQUIRED` 作为默认参数。

下面的代码展示了事务管理的开发：

TransactionDAOBean.java

```
package com.foshanshop.ejb3.impl;
import java.util.List;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import com.foshanshop.ejb3.TransException;
import com.foshanshop.ejb3.TransactionDAO;
import com.foshanshop.ejb3.bean.Product;

@Stateless
@Remote ({TransactionDAO.class})
public class TransactionDAOBean implements TransactionDAO {

    @PersistenceContext
    protected EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void insertProduct(String name, Float price, boolean error) {
        try {
            for(int i=0;i<3; i++){
                Product product = new Product(name+i,price*(i+1));
                em.persist(product);
            }
            if (error) new Float("kkk"); //制造一个例外
        } catch (Exception e) {
            throw new RuntimeException ("应用抛出运行时例外,为了使事务回滚,外部不要用try/catch包围");
        }
    }
}
```

```

    }
}

@Transactional(TransactionalType.REQUIRED)
public void ModifyProductName(String newname, boolean error) throws Exception {
    Query query = em.createQuery("select p from Product p");
    List result = query.getResultList();
    if (result!=null) {
        for(int i=0;i<result.size();i++) {
            Product product = (Product)result.get(i);
            product.setName(newname+ i);
            em.merge(product);
        }
        if (error && result.size()>0) throw new TransException ("抛出应用例外");
    }
}
}

```

上面定义了两个需要事务的方法，容器在运行这两个方法时将会创建一个事务，方法里的所有数据库操作都在此事务中运行，当这两个方法正常退出时，事务将会提交，所有更改都会同步到数据库中。如果方法抛出 `RuntimeException` 例外或 `ApplicationException` 例外，事务将会回滚。方法 `ModifyProductName` 中使用的 `TransException` 类是一个自定义 `ApplicationException` 例外。代码如下：

`TransException.java`

```

package com.foshanshop.ejb3;
import javax.ejb.ApplicationException;

@SuppressWarnings("serial")
@ApplicationException(rollback=true)
public class TransException extends Exception {
    public TransException (String message) {
        super(message);
    }
}

```

`@ApplicationException` 注释定义了在外例抛出时将回滚事务。

下面是 `TransactionDAOBean` 的接口

`TransactionDAO.java`

```

package com.foshanshop.ejb3;
public interface TransactionDAO {
    public void insertProduct(String name, Float price, boolean error);
    public void ModifyProductName(String newname, boolean error) throws Exception ;
}

```

下面是 `TransactionDAOBean` 使用的实体 Bean

`Product.java`

```

package com.foshanshop.ejb3.bean;

```

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "Products")
public class Product {
    private int hashCode = Integer.MIN_VALUE;
    private Integer productid;
    private String name;
    private Float price;

    public Product() {}

    public Product(String name, Float price) {
        this.name = name;
        this.price = price;
    }

    @Id
    @GeneratedValue
    public Integer getProductid() {
        return productid;
    }

    public void setProductid(Integer productid) {
        this.productid = productid;
    }

    @Column(name = "ProductName", nullable=false, length=50)
    public String getName() {
        return name;
    }

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

    @Column(nullable=false)
    public Float getPrice() {
        return price;
    }

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

```

    }

    public boolean equals (Object obj) {
        if (null == obj) return false;
        if (!(obj instanceof Product)) return false;
        else {
            Product mObj = (Product) obj;
            if (null == this.getProductid() || null == mObj.getProductid()) return false;
            else return (this.getProductid().equals(mObj.getProductid()));
        }
    }

    public int hashCode () {
        if (Integer.MIN_VALUE == this.hashCode) {
            if (null == this.getProductid()) return super.hashCode();
            else {
                String hashStr = this.getClass().getName() + ":" + this.getProductid().hashCode();
                this.hashCode = hashStr.hashCode();
            }
        }
        return this.hashCode;
    }
}

```

下面是 Session Bean 的 JSP 客户端代码:

TransactionTest.jsp

```

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

<%

    Properties props = new Properties();
    props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
    props.setProperty("java.naming.provider.url", "localhost:1099");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    try {
        InitialContext ctx = new InitialContext(props);
        TransactionDAO transactiondao = (TransactionDAO)
ctx.lookup("TransactionDAOBean/remote");
        transactiondao.insertProduct("电脑", new Float("1200"), false);
        transactiondao.ModifyProductName("数码相机", true);
        out.println("执行完成");
    }
}

```

```

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

```

%>

本例子的 EJB 源代码在 TransactionService 文件夹（源代码下载:<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 TransactionService 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据库配置文件是 mysql-ds.xml，你可以在下载的文件中找到。数据库名为 foshanshop。本例子的客户端代码在 EJCTest 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/EJCTest/TransactionTest.jsp> 访问客户端。

6.12 Entity 的生命周期和状态

在 EJB3 中定义了四种 Entity 的状态：

1. 新实体(new)。Entity 由应用产生，和 EJB3 Persistence 运行环境没有联系，也没有唯一的标示符(Identity)。
2. 持久化实体(managed)。新实体和 EJB3 Persistence 运行环境产生关联（通过 persist(), merge()等方法），在 EJB3 Persistence 运行环境中存在和被管理，标志是在 EJB3 Persistence 运行环境中有一个唯一的标示(Identity)。
3. 分离的实体(detached)。Entity 有唯一标示符，但它的标示符不被 EJB3 Persistence 运行环境管理，同样的该 Entity 也不被 EJB3 Persistence 运行环境管理。
4. 删除的实体(removed)。Entity 被 remove()方法删除，对应的纪录将会在当前事务提交的时候从数据库中删除。

当你执行持久化方法，如：persist(), merge(), remove(), find()或 JPQL 查询的时候，实体的生命周期事件将会触发。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。有时实体类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，用来记录数据库每条记录发生的操作（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，这个外部类称作实体监听者。通过@EntityListeners 注释绑定到实体 Bean。

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者。

6.12.1 生命周期回调事件

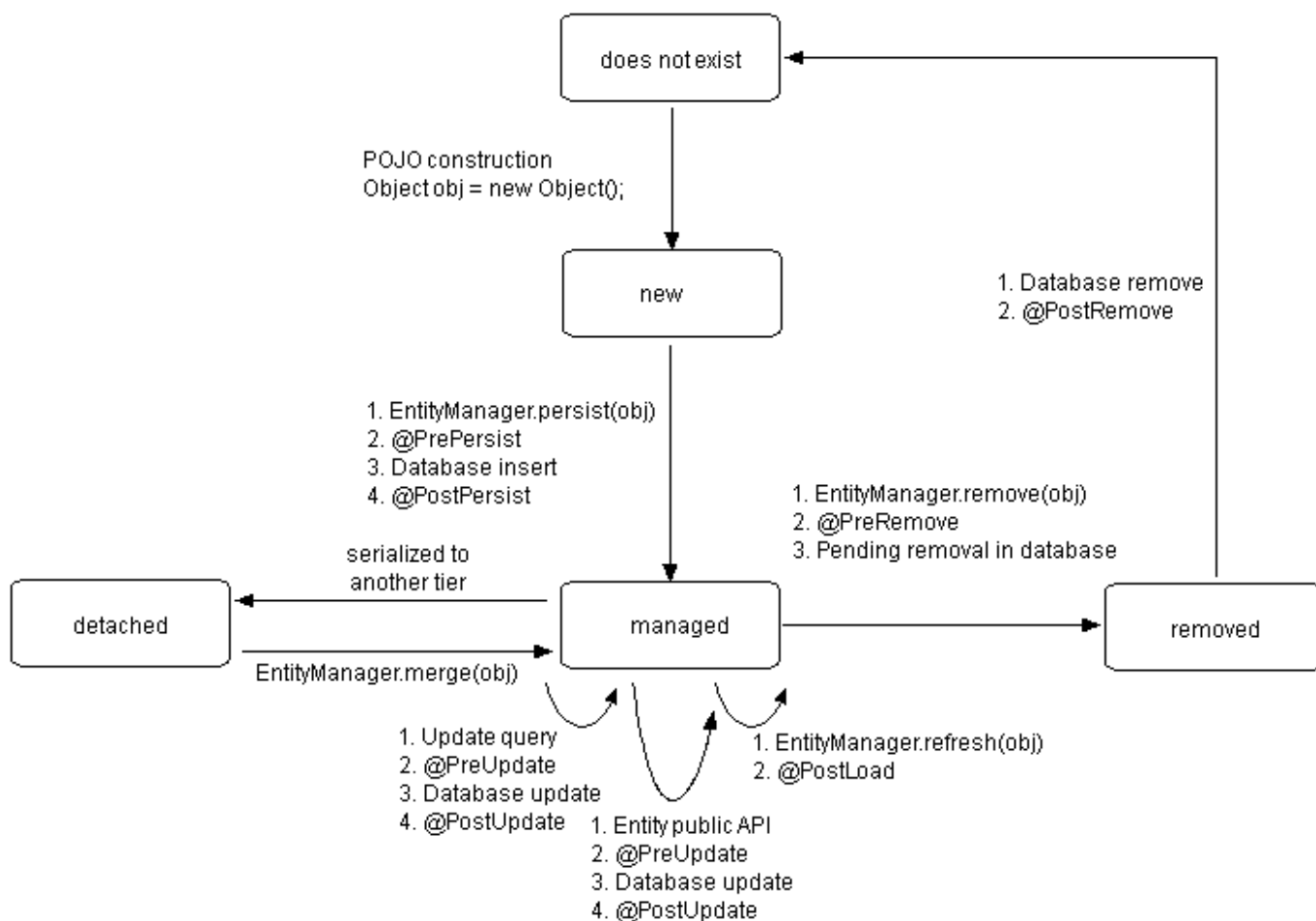
如果需要在生命周期事件期间执行自定义逻辑，请使用以下生命周期事件注释关联生命周期事件与回调方法，EJB 3.0 允许你将任何方法指定为回调方法。这些方法将会被容器在实体生命周期的不同阶段调用。

```

@javax.persistence.PostLoad
@javax.persistence.PrePersist
@javax.persistence.PostPersist
@javax.persistence.PreUpdate
@javax.persistence.PostUpdate
@javax.persistence.PreRemove
@javax.persistence.PostRemove

```

下图演示了实体生命周期事件之间的关系:



@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()` 方法被调用后

@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, 级联保存也会发生此事件, 此时的数据还没有真实插入进数据库。@PostPersist 事件在数据已经插入进数据库后发生。

@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, 同步在事务提交时发生。

@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。

作者对 @PostPersist 和 @PostUpdate 事件触发的时机有点怀疑, 文档上说 @PostPersist 事件在数据真实插入进数

数据库后发生，但作者测试的结果是：在数据还没有真实插入进数据库时，此事件就触发了。在本书的配套例子中，专门让当前线程睡眠 5 秒，在 5 秒内，因为事务还未提交（事务默认在堆栈末尾提交），所以数据还没有进入数据库，但 `@PostPersist` 事件在这之前就已经触发。所以作者感觉 `@PostPersist` 事件在调用了 JDBC 相关 API 后就会触发，不管此时的事务是否提交。如果是这样的话，`@PostPersist` 事件就不应该说成是在数据已经插入进数据库后触发。`@PostUpdate` 事件的情况一样，就 `@PostRemove` 事件触发的时机和文档上说的一样，在数据从数据库中删除后触发。（这也不排除作者对英文文档理解有误，欢迎指正）

下面我们将学习怎样实现这些生命周期事件的回调方法。

6.12.2 在外部类中实现回调

Entity listeners（实体监听者）用作拦截实体回调事件，通过 `@javax.persistence.EntityListeners` 注释可以把他们绑定到一个实体类。在 Entity listeners 类里，你可以指定一个方法拦截实体的某个事件，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。下面是一个例子

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("载入了实体Bean{" + entity.getClass().getName() + "}");
    }

    @PrePersist
    public void PreInsert(Object entity) {
        System.out.println("对实体Bean{" + entity.getClass().getName() + "}调用了
EntityManager.persist()或级联保存");
    }

    @PostPersist
    public void postInsert(Object entity) {
        System.out.println("在JDBC API层对实体Bean{" + entity.getClass().getName() + "}执行了插入操作,但事务还未提交");
    }
}
```



```

    }

    @PreUpdate
    public void PreUpdate(Object entity) {
        System.out.println("对实体Bean{" + entity.getClass().getName() + "}调用了
EntityManager.merge()或级联更新");
    }

    @PostUpdate
    public void PostUpdate(Object entity) {
        System.out.println("在JDBC API层对实体Bean{" + entity.getClass().getName() + "}执行了更
新操作,但事务还未提交");
    }

    @PreRemove
    public void PreRemove(Object entity) {
        System.out.println("对实体Bean{" + entity.getClass().getName() + "}调用了
EntityManager.remove()或级联删除");
    }

    @PostRemove
    public void PostRemove(Object entity) {
        System.out.println("实体Bean{" + entity.getClass().getName() + "}已经从数据库中删除");
    }
}

```

我们使用@javax.persistence.EntityListeners 注释把上面的实体监听类绑定到下面的实体中，@javax.persistence.EntityListeners 注释可以绑定多个实体监听类，之间用逗号分隔，监听的顺序按照定义的先后顺序执行。

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 javax.persistence.Table;
import com.foshanshop.ejb3.bean.listener.EntityListenerLogger;

@Entity
@Table(name = "EntityLifecycle")
@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;
}
}
```

为了使用上面的实体 Bean，我们定义一个 Session Bean 作为他的使用者。下面是 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})
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); //此处将会触发@PrePersist事件
        //下面让线程等待5秒, 在五秒时间内你可以查看数据是否已经插入进数据库
        try {
            System.out.println("***当前线程睡眠5秒, 在五秒时间内你可以查看数据是否已经插入进数据库***");
            Thread.sleep(5*1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void Remove() {
        Query query = em.createQuery("from EntityLifecycle e");
        List result = query.getResultList();
        if (result!=null && result.size()>1) {
            EntityLifecycle entitylifecycle = (EntityLifecycle)result.get(result.size()-1);
            em.remove(entitylifecycle); //此处将会触发@PreRemove事件
            //下面让线程等待5秒, 在五秒时间内你可以查看数据是否已经删除
            try {
                System.out.println("***当前线程睡眠5秒, 在五秒时间内你可以查看数据是否已经删除***");
                Thread.sleep(5*1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

    }
}

public void Update() {
    Query query = em.createQuery("from EntityLifecycle e");
    List result = query.getResultList();
    if (result!=null && result.size()>0) {
        EntityLifecycle entitylifecycle = (EntityLifecycle)result.get(result.size()-1);
        entitylifecycle.setName("张权");
        em.merge(entitylifecycle); //此处将会触发@PreUpdate事件
        //下面让线程等待5秒，在五秒时间内你可以查看数据是否已经更新
        try {
            System.out.println("***当前线程睡眠5秒，在五秒时间内你可以查看数据是否已经更新***");

            Thread.sleep(5*1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
}

```

下面是 JSP 客户端

EntityListenerTest.jsp

```

<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.EntityLifecycleDAO,
                com.foshanshop.ejb3.bean.EntityLifecycle,
                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");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx = new InitialContext(props);
    try {
        out.println("输出信息将打印在Jboss控制台, 每个方法的执行都会停留5秒");
        out.println(", 以便你更好地观察生命周期回调方法的运行效果");

        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 EJB 源代码在 EntityListeners 文件夹（源代码下载:<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 EntityListeners 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据库配置文件是 mysql-ds.xml，你可以在下载的文件中找到。数据库名为 foshanshop
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。

6.12.3 在 Entity 类中实现回调

除了外部类可以实现生命周期事件的回调，你也可以把回调方法写在 Entity 类中。要注意：直接写在 Entity 类中的回调方法不需带任何参数，格式如下：

void <MethodName>()

EntityLifecycle.java 程序片断

```

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

    @PostLoad
    public void postLoad() {
        System.out.println("载入了实体Bean{" + this.getClass().getName() + "}");
    }

    @PrePersist
    public void PreInsert() {
        System.out.println("对实体Bean{" + this.getClass().getName() + "}调用了
EntityManager.persist()或级联保存");
    }

    @PostPersist
    public void postInsert() {
        System.out.println("在JDBC API层对实体Bean{" + this.getClass().getName() + "}执行了插入
操作,但事务还未提交");
    }

    @PreUpdate

```

```

public void PreUpdate() {
    System.out.println("对实体Bean{" + this.getClass().getName() + "}调用了
EntityManager.merge()或级联更新");
}

@PostUpdate
public void PostUpdate() {
    System.out.println("在JDBC API层对实体Bean{" + this.getClass().getName() + "}执行了更新
操作,但事务还未提交");
}

@PreRemove
public void PreRemove() {
    System.out.println("对实体Bean{" + this.getClass().getName() + "}调用了
EntityManager.remove()或级联删除");
}

@PostRemove
public void PostRemove() {
    System.out.println("实体Bean{" + this.getClass().getName() + "}已经从数据库中删除");
}
}

```

当一个生命周期事件发生时，将会调用与该事件关联的回调方法。

6.13 复合主键(Composite Primary Key)

当我们需要使用多个属性变量（表中的多列）联合起来作为主键，我们需要使用复合主键。复合主键要求我们编写一个复合主键类(Composite Primary Key Class)。复合主键类需要符合以下一些要求：

- 复合主键类必须是 public 和具备一个没有参数的构造函数
- 复合主键类的每个属性变量必须有 getter/setter，如果没有，每个属性变量则必须是 public 或者 protected
- 复合主键类必须实现 java.io.Serializable
- 复合主键类必须实现 equals()和 hashCode()方法
- 复合主键类中的主键属性变量的名字必须和对应的 Entity 中主键属性变量的名字相同
- 一旦主键值设定后，不要修改主键属性变量的值

本节以航线为例，介绍复合主键的开发过程，航线以出发地及到达地作为联合主键，航线与航班存在一对多的关联关系，下面是他们的数据库表

airline

| 字段名称 | 字段类型属性 | 描述 |
|-----------------|------------------|-----------|
| leavecity (主键) | char(3) not null | 出发地 |
| arrivecity (主键) | char(3) not null | 到达地 |
| onoff | tinyint(1) null | 航线打开及关闭标志 |

flight

| 字段名称 | 字段类型属性 | 描述 |
|------|--------|----|
|------|--------|----|

| | | |
|-------------|----------------------|------|
| Id (主键) | int(11) not null | 流水号 |
| flightno | varchar(10) not null | 航班号 |
| arrivetime | varchar(10) null | 到达时间 |
| leavetime | varchar(10) null | 起飞时间 |
| Leave_City | char(3) not null | 出发地 |
| Arrive_City | char(3) not null | 到达地 |

按照复合主键类的要求，我们编写一个复合主键类 AirtLinePK.java,代码如下：

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Embeddable;

@SuppressWarnings("serial")
@Embeddable
public class AirtLinePK implements Serializable {
    private String leavecity;
    private String arrivecity;

    public AirtLinePK() {}

    public AirtLinePK(String leavecity, String arrivecity) {
        this.leavecity = leavecity;
        this.arrivecity = arrivecity;
    }

    @Column(nullable=false, length=3, name="LEAVECITY")
    public String getLeavecity() {
        return leavecity;
    }

    public void setLeavecity(String leavecity) {
        this.leavecity = leavecity;
    }

    @Column(nullable=false, length=3, name="ARRIVECITY")
    public String getArrivecity() {
        return arrivecity;
    }

    public void setArrivecity(String arrivecity) {
        this.arrivecity = arrivecity;
    }

    public boolean equals(Object o) {
        if (this == o) return true;
    }
}
```

```

        if (!(o instanceof AirtLinePK)) return false;

        final AirtLinePK airtLinePK = (AirtLinePK) o;

        if (leavecity==null || !leavecity.equals(airtLinePK.getLeavecity())) return false;
        if (arrivecity==null || !arrivecity.equals(airtLinePK.getArrivecity())) return false;
        return true;
    }

    public int hashCode() {
        int result;
        result = leavecity.hashCode();
        result = 29 * result + arrivecity.hashCode();
        return result;
    }
}

```

复合主键使用一个可嵌入的类作为主键表示，@Embeddable 注释指明这是一个可嵌入的类。下面 AirLine.java 用复合主键类 AirtLinePK 作为其主键。@EmbeddedId 注释指明复合主键类作为主键。

AirLine.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.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.OneToOne;
import javax.persistence.OrderBy;
import javax.persistence.Table;

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

    private AirtLinePK id;
    private Boolean onoff;
    private Set<Flight> flights = new HashSet<Flight>();

    public AirLine() {}

    public AirLine(AirtLinePK id, Boolean onoff) {

```



```
        this.id = id;
        this.onoff = onoff;
    }

    @EmbeddedId
    public AirtLinePK getId() {
        return id;
    }

    public void setId(AirtLinePK id) {
        this.id = id;
    }

    public Boolean getOnoff() {
        return onoff;
    }

    public void setOnoff(Boolean onoff) {
        this.onoff = onoff;
    }

    @OneToMany(mappedBy="airline", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @OrderBy(value = "id ASC")
    public Set<Flight> getFlights() {
        return flights;
    }

    public void setFlights(Set<Flight> flights) {
        this.flights = flights;
    }

    public void addFlight(Flight flight) {
        if (!this.flights.contains(flight)) {
            this.flights.add(flight);
            flight.setAirline(this);
        }
    }

    public void removeFlight(Flight flight) {
        flight.setAirline(null);
        this.flights.remove(flight);
    }
}
```

Flight.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.JoinColumns;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Table(name = "Flight")
public class Flight implements Serializable {
    private Integer id;
    private String flightno;//航班号
    private String leavetime;//起飞时间
    private String arrivetime;//到达时间
    private AirLine airline;

    public Flight() {}

    public Flight(String flightno, String leavetime, String arrivetime) {
        this.flightno = flightno;
        this.leavetime = leavetime;
        this.arrivetime = arrivetime;
    }

    @Id
    @GeneratedValue
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Column(nullable=false, length=10)
    public String getFlightno() {
        return flightno;
    }
}
```

```

public void setFlightno(String flightno) {
    this.flightno = flightno;
}

@Column(length=10)
public String getArrivetime() {
    return arrivetime;
}

public void setArrivetime(String arrivetime) {
    this.arrivetime = arrivetime;
}

@Column(length=10)
public String getLeavetime() {
    return leavetime;
}

public void setLeavetime(String leavetime) {
    this.leavetime = leavetime;
}

@ManyToOne(cascade=CascadeType.REFRESH, optional=true)
@JoinColumns ({
    @JoinColumn(name="Leave_City", referencedColumnName = "LEAVECITY", nullable=false),
    @JoinColumn(name="Arrive_City", referencedColumnName = "ARRIVECITY", nullable=false)
}) //TopLink产品中, referencedColumnName 属性值大小写敏感
public AirLine getAirline() {
    return airline;
}

public void setAirline(AirLine airline) {
    this.airline = airline;
}
}

```

Flight 与 AirLine 存在多对一关系, Flight 含有两个外键指向 AirLine, 用@JoinColumns 注释定义多个 JoinColumn 的属性, Flight 对应表中的 Leave_City 列作为外键指向 AirLine 对应表中的 leavecity 列, Flight 对应表中 Arrive_City 列指向 AirLine 对应表中的 arrivcity 列。

为了使用上面的实体 Bean, 我们定义一个 Session Bean 作为他的使用者。下面是 Session Bean 的业务接口, 他定义了两个业务方法 insertAirLine 和 getAirLineByID, 两个方法的业务功能是:

insertAirLine 添加一条航线 (含有三个航班) 进数据库

getAirLineByID 获取指定出发地及到达地的航线信息。

下面是 Session Bean 的业务接口及实现类

AirLineDAO.java

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

public interface AirLineDAO {
    public void insertAirLine();
    public AirLine getAirLineByID(String leavecity, String arrivecity);
}
```

AirLineDAOBean.java

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

import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import com.foshanshop.ejb3.AirLineDAO;
import com.foshanshop.ejb3.bean.AirLine;
import com.foshanshop.ejb3.bean.AirtLinePK;
import com.foshanshop.ejb3.bean.Flight;

@Stateless
@Remote({AirLineDAO.class})
public class AirLineDAOBean implements AirLineDAO {
    @PersistenceContext
    protected EntityManager em;

    public void insertAirLine() {
        Query query = em.createQuery("select count(a.id.leavecity) from AirLine a where a.id.leavecity=?1 and a.id.arrivecity=?2");
        //在Jboss4.2.0版本之前,hibernate对联合主键使用count运算时存在一个BUG,当通过count(a)统计记录数时,转译成的SQL为select count(airline0_.LEAVECITY, airline0_.ARRIVECITY) as col_0_0_ from AirLine airline0_,而TopLink即转译成count(*),因此在jboss中遇到对联合主键使用count运算时可以统计联合主键中某一个字段
        query.setParameter(1, "PEK");
        query.setParameter(2, "CAN");
        int result = Integer.parseInt(query.getSingleResult().toString());
        if (result==0){
            AirLine airLine = new AirLine(new AirtLinePK("PEK", "CAN"), true);
            //PEK为首都机场三字码, CAN为广州白云机场三字码
            airLine.addFlight(new Flight("CA1321", "08:45", "11:50"));
            airLine.addFlight(new Flight("CZ3102", "12:05", "15:05"));
        }
    }
}
```

```

        airLine.addFlight(new Flight("HU7801", "15:05", "17:45"));
        em.persist(airLine);
    }
}

public AirLine getAirLineByID(String leavecity, String arrivecity) {
    AirLine airLine = em.find(AirLine.class, new AirLinePK(leavecity, arrivecity));
    airLine.getFlights().size();
    //因为是延迟加载，通过执行size()这种方式获取航线下的所有航班
    return airLine;
}
}

```

下面是 Session Bean 的 JSP 客户端代码：

CompositePKTest.jsp

```

<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.AirLineDAO,
                com.foshanshop.ejb3.bean.*,
                javax.naming.*,
                java.util.*"%>

<%
    Properties props = new Properties();
    props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
    props.setProperty("java.naming.provider.url", "localhost:1099");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx = new InitialContext(props);
    try {
        AirLineDAO airlinedao = (AirLineDAO) ctx.lookup("AirLineDAOBean/remote");
        airlinedao.insertAirLine();

        AirLine airLine = airlinedao.getAirLineByID("PEK", "CAN");
        out.println("航线: " + airLine.getId().getLeavecity() + "--" +
airLine.getId().getArrivecity() + "<br>");
        out.println("===== 存在以下航班 =====<br>");
        if (airLine!=null) {
            Iterator iterator = airLine.getFlights().iterator();
            while (iterator.hasNext()) {
                Flight flight = (Flight) iterator.next();
                out.println("航班:" + flight.getFlightno() + "<br>");
            }
        } else {
            out.println("没有找到相关航线");
        }
    } catch (Exception e) {
        out.println(e.getMessage());
    }
}
%>

```

```
    }

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

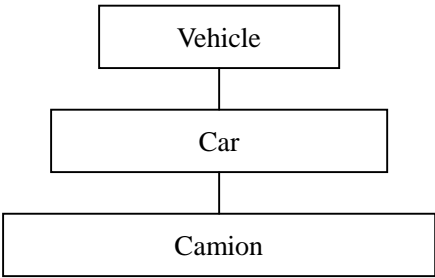
    %>
```

本例子的 EJB 源代码在 CompositePK 文件夹（源代码下载:<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 CompositePK 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据库源配置文件是 mysql-ds.xml，你可以在下载的文件中找到。数据库名为 foshanshop

本例子的客户端代码在 EJCTest 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/EJCTest/CompositePKTest.jsp> 访问客户端。

6.14 实体继承

在本章我们定义几个类，用来完成实体继承的介绍。三个类分别是：Vehicle（交通工具），Car（汽车），Camion（卡车），他们的继承关系如下：



因为关系数据库的表之间不存在继承关系，Entity 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)

三种继承映射策略的特性

| 继承策略 (Inheritance strategy) | 多态多 对一 | 多态一 对一 | 多态一对 多 | 多态多对 多 | 多态 find()或 getReference() | 多态查询 | 多态连接 (join) | 外连接 (Outer join)抓 取 |
|---|-------------------|------------------|-------------------|--------------------|--------------------------------|-----------------|-----------------------------------|------------------------------|
| 每个类分 层结构一 张表(table per class hierarchy) | <many-to -one> | <one-to -one> | <one-to- many> | <many-to -many> | em.find(Vehicle.cl ass, id) | from ower ow | from Order o join o.ower ow | 支持 |
| 每个子类 | <many-to | <one-to | <one-to- | <many-to | em.find(Vehicle.cl | from ower | from Order | 支持 |

| | | | | | | | | |
|--|---------------|--------------|---|----------------|--------------------------------|-----------------|-----------------------------------|----|
| 一张表 (table per subclass) | -one> | -one> | many> | -many> | ass, id) | ow | o join o.ower ow | |
| 每个具体类一张表 (table per concrete class) | <many-to-one> | <one-to-one> | <one-to-many> (仅对于 inverse="true"的情况) | <many-to-many> | em.find(Vehicle.cl ass, id) | from ower ow | from Order o join o.ower ow | 支持 |

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。

6.14.1 每个类分层结构一张表(table per class hierarchy)

这种映射方式只需为基类 Vehicle 创建一个表(Vehicle_Hierarchy)即可。在表中不仅提供 Vehicle 所有属性对应的字段，还要提供所有子类属性对应的字段，此外还需要一个字段用于区分子类的具体类型。

Vehicle_Hierarchy 表

| 字段名称 | 字段类型属性 | 描述 |
|---------------|------------------|--------------------|
| id (主键) | Int(11) not null | ID |
| speed | Smallint(6) null | 速度(基类 Vehicle 的属性) |
| engine | Varchar(30) null | 发动机(子类 Car 的属性) |
| container | Varchar(30) null | 集装箱(子类 Camion 的属性) |
| Discriminator | Varchar(30) null | 鉴别字段，用于区分各个类 |

要使用每个类分层结构一张表(table per class hierarchy) 策略，需要把@javax.persistence.Inheritance 注释的 strategy 属性设置为 InheritanceType.SINGLE_TABLE。除非你要改变子类的映射策略，否则@Inheritance 注释只能放在继承层次的基类。通过鉴别字段的值，持久化引擎可以区分出各个类，并且知道每个类对应那些字段。鉴别字段通过@javax.persistence.DiscriminatorColumn 注释进行定义，name 属性定义鉴别字段的列名，discriminatorType 属性定义鉴别字段的类型（可选值有：String, Char, Integer），如果鉴别字段的类型为 String 或 Char，可以用 length 属性定义其长度。@DiscriminatorValue 注释为继承关系中的每个类定义鉴别值，如果不指定鉴别值，默认采用类名。

Vehicle.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;
```

```

@SuppressWarnings("serial")
@Entity
@Table(name="Vehicle_Hierarchy")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="Discriminator",
                    discriminatorType = DiscriminatorType.STRING,
                    length=30)
@DiscriminatorValue("Vehicle")
public class Vehicle implements Serializable{
    private Long id;
    private Short speed;//速度

    @Id
    @GeneratedValue
    @Column(columnDefinition="integer")//指定使用适配Integer长度的数据类型
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }

    public Short getSpeed() {
        return speed;
    }
    public void setSpeed(Short speed) {
        this.speed = speed;
    }
}

```

Car.java (Car 继承 Vehicle, 并有自己的属性 engine)

```

package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

```



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

import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@SuppressWarnings("serial")
@Entity
@DiscriminatorValue("Car")
public class Car extends Vehicle{
    private String engine;//发动机

    @Column(nullable=true,length=30)
    public String getEngine() {
        return engine;
    }

    public void setEngine(String engine) {
        this.engine = engine;
    }
}
```

Camion.java (Camion 继承 Car，并有自己的属性 container)

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

import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@SuppressWarnings("serial")
@Entity
@DiscriminatorValue("Camion")
public class Camion extends Car{
    private String container;//集装箱

    @Column(nullable=true,length=30)
    public String getContainer() {
        return container;
    }

    public void setContainer(String container) {
        this.container = container;
    }
}
```

可以看出, 每个子类没有单独的映射, 在数据库中没有对应的表存在。而只有一个记录所有自身属性和子类所有属性的表, 在基类为 **Vehicle** 的时候, **Discriminator** 字段的值将为 **Vehicle**, 在子类为 **Car** 的时候, **Discriminator** 字段的值将为 **Car**, 子类为 **Camion** 的时候, **Discriminator** 字段的值将为 **Camion**。那么, 如果业务逻辑要求 **Car** 对象的 **engine** 属性不允许为 **null**, 显然无法在 **Vehicle_Hierarchy** 表中为 **engine** 字段定义 **not null** 约束, 可见这种映射方式无法保证关系数据模型的数据完整性。

为了使用上面的实体 **Bean**, 我们定义一个 **Session Bean** 作为他的使用者。下面是 **Session Bean** 的业务接口, 他定义了五个业务方法 **initializeData**, **getLastVehicle**, **getLastCar**, **getLastCamion** 和 **deleteVehicle**, 五个方法的业务功能是:

initializeData 添加一些测试数据进数据库

getLastVehicle 获取最后一条 **Vehicle** 记录

getLastCar 获取最后一条 **Car** 记录

getLastCamion 获取最后一条 **Camion** 记录

deleteVehicle 删除所有 **Vehicle** 对应的记录, 执行该操作除了会删除自身对应的记录之外, 还会删除所有继承 **Vehicle** 的记录, 因为他是继承树中的根类, 所以相当于清除整个表的数据。

下面是 **Session Bean** 的实现类及业务接口

EntityInheritanceDAOBean.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.EntityInheritanceDAO;
import com.foshanshop.ejb3.bean.Car;
import com.foshanshop.ejb3.bean.Vehicle;
import com.foshanshop.ejb3.bean.Camion;

@Stateless
@Remote ({EntityInheritanceDAO.class})
public class EntityInheritanceDAOBean implements EntityInheritanceDAO {
    @PersistenceContext
    protected EntityManager em;

    public void initializeData() {
        //添加Vehicle
        Vehicle vehicle = new Vehicle();
        vehicle.setSpeed((short)100);
        em.persist(vehicle);
        //添加Car
        Car car = new Car();
        car.setSpeed((short)300);
```

```
        car.setEngine("A发动机");
    em.persist(car);
    //添加Camion
    Camion camion = new Camion();
    camion.setSpeed((short)200);
    camion.setEngine("B发动机");
    camion.setContainer("2吨集装箱");
    em.persist(camion);
}

public Vehicle getLastVehicle() {
    //查询所有Vehicle时，因为他是最继承树中的根，查询结果会得到所有继承于Vehicle类的记录
    //转译成的SQL片断: select * from Vehicle_Hierarchy
    Query query = em.createQuery("from Vehicle v");
    List result = query.getResultList();
    if (result!=null && result.size()>0)
        return (Vehicle)result.get(result.size()-1);
    return null;
}

public Car getLastCar() {
    //查询所有Car时，除了得到Car类的记录，也会得到所有继承于Car类的记录
    //转译成的SQL片断: where Discriminator in ('Car', 'Camion')
    Query query = em.createQuery("from Car c");
    List result = query.getResultList();
    if (result!=null && result.size()>0)
        return (Car)result.get(result.size()-1);
    return null;
}

public Camion getLastCamion() {
    Query query = em.createQuery("from Camion c");
    List result = query.getResultList();
    if (result!=null && result.size()>0)
        return (Camion)result.get(result.size()-1);
    return null;
}

public void deleteVehicle() {
    //执行该操作会删除自身对应记录，还会删除所有继承Vehicle的记录，
    //因为他是最继承树中的根，就相当于清除整个表的数据
    Query query = em.createQuery("delete from Vehicle v");
    query.executeUpdate();
}
```

```
}
```

EntityInheritanceDAO.java 业务接口

```
package com.foshanshop.ejb3;
import com.foshanshop.ejb3.bean.Camion;
import com.foshanshop.ejb3.bean.Car;
import com.foshanshop.ejb3.bean.Vehicle;

public interface EntityInheritanceDAO {
    public void initializeData();
    public Vehicle getLastVehicle();
    public Car getLastCar();
    public Camion getLastCamion();
    public void deleteVehicle();
}
```

下面是 Session Bean 的 JSP 客户端代码:

EntityInheritanceTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.EntityInheritanceDAO,
                com.foshanshop.ejb3.bean.*,
                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");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

    InitialContext ctx = new InitialContext(props);
    try {
        EntityInheritanceDAO entityinheritancedao = (EntityInheritanceDAO)
ctx.lookup("EntityInheritanceDAOBean/remote");
        entityinheritancedao.initializeData();
        Vehicle vehicle = entityinheritancedao.getLastVehicle(); //取最后一个Vehicle
        out.println("Vehicle类: 速度=");
        out.println(vehicle.getSpeed());

        Car car = entityinheritancedao.getLastCar(); //取最后一个Car
        out.println("<br>Car类: 速度=");
        out.println(car.getSpeed()+"; 发动机="+ car.getEngine());

        Camion camion = entityinheritancedao.getLastCamion(); //取最后一个Camion
        out.println("<br>Camion类: 速度=");
    }
}
```

```

        out.println(camion.getSpeed()+ "； 发动机="+ camion.getEngine()+ "； 集装箱="+
camion.getContainer());
        /* 下面语句会删除所有记录
entityinheritancedao.deleteVehicle();
*/
    } catch (Exception e) {
        out.println(e.getMessage());
    }
}
%>

```

该策略的优点：

SINGLE_TABLE 映射策略在所有继承策略中是最简单的，同时也是执行效率最高的。他仅需对一个表进行管理 & 操作，持久化引擎在载入 entity 或多态连接时不需要进行任何的关联，联合或子查询，因为所有数据都存储在一个表。

该策略的缺点：

这种策略最大的一个缺点是需要对关系数据模型进行非常规设计，在数据库表中加入额外的区分各个子类的字段，此外，不能为所有子类的属性对应的字段定义 **not null** 约束，此策略的关系数据模型完全不支持对象的继承关系。

选择原则：查询性能要求高，子类属性不是非常多时，优先选择该策略。

本例子的 EJB 源代码在 EntityInheritance 文件夹（源代码下载：<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 EntityInheritance 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据库源配置文件是 **mysql-ds.xml**，你可以在下载的文件中找到。数据库名为 **foshanshop**。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/EntityInheritanceTest.jsp> 访问客户端。

6.14.2 每个子类一张表(table per subclass)

这种映射方式为每个类创建一个表。在每个类对应的表中只需包含和这个类本身的属性对应的字段，子类对应的表参照父类对应的表。

Vehicle 表

| 字段名称 | 字段类型属性 | 描述 |
|---------|------------------|--------------------|
| id (主键) | Int(11) not null | ID |
| speed | Smallint(6) null | 速度(基类 Vehicle 的属性) |

Car 表

| 字段名称 | 字段类型属性 | 描述 |
|------------|------------------|-----------------|
| CarID (主键) | Int(11) not null | Vehicle 的外键 |
| engine | Varchar(30) null | 发动机(子类 Car 的属性) |

Camion 表

| 字段名称 | 字段类型属性 | 描述 |
|---------------|------------------|---------|
| CamionID (主键) | Int(11) not null | Car 的外键 |

| | | |
|-----------|------------------|--------------------|
| container | Varchar(30) null | 集装箱(子类 Camion 的属性) |
|-----------|------------------|--------------------|

要使用每个子类一张表(table per subclass)策略, 需要把@javax.persistence.Inheritance 注释的 strategy 属性设置为 InheritanceType.JOINED。

Vehicle.java

```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name="Vehicle")
public class Vehicle implements Serializable{
    private Long id;
    private Short speed;//速度

    @Id
    @GeneratedValue
    @Column(columnDefinition="integer")
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Short getSpeed() {
        return speed;
    }

    public void setSpeed(Short speed) {
        this.speed = speed;
    }
}
```

Car.java

```
package com.foshanshop.ejb3.bean;
import javax.persistence.Column;
```

```
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Table(name="Car")
@PrimaryKeyJoinColumn(name="CarID")//把主键对应的列名更改为CarID
public class Car extends Vehicle{
    private String engine;//发动机

    @Column(nullable=true, length=30)
    public String getEngine() {
        return engine;
    }

    public void setEngine(String engine) {
        this.engine = engine;
    }
}
```

Camion.java

```
package com.foshanshop.ejb3.bean;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Table(name="Camion")
@PrimaryKeyJoinColumn(name="CamionID")//把主键对应的列名更改为CamionID
public class Camion extends Car{
    private String container;//集装箱

    @Column(nullable=true, length=30)
    public String getContainer() {
        return container;
    }

    public void setContainer(String container) {
        this.container = container;
    }
}
```

Session Bean 及 JSP 客户端我们使用上节的文件，这里不再列出。

该策略的优点：

这种映射方式支持多态关联和多态查询，而且符合关系数据模型的常规设计规则。在这种策略中你可以对子类的属性对应的字段定义 not null 约束。

该策略的缺点：

它的查询性能不如上面介绍的映射策略。在这种映射策略下，必须通过表的内连接或左外连接来实现多态查询和多态关联。

选择原则：子类属性非常多，需要对子类某些属性对应的字段进行 not null 约束，且对性能要求不是很严格时，优先选择该策略。

本例子的 EJB 源代码在 EntityInheritanceSubclass 文件夹（源代码下载：<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 EntityInheritanceSubclass 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据源配置文件是 mysql-ds.xml，你可以在下载的文件中找到。数据库名为 foshanshop。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/EntityInheritanceTest.jsp> 访问客户端。

6.14.3 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个表。在每个类对应的表中包含和这个类所有属性（包括从超类继承的属性）对应的字段。

Vehicle 表

| 字段名称 | 字段类型属性 | 描述 |
|---------|------------------|--------------------|
| id (主键) | Int(11) not null | ID |
| speed | Smallint(6) null | 速度(基类 Vehicle 的属性) |

Car 表

| 字段名称 | 字段类型属性 | 描述 |
|---------|------------------|--------------------|
| id (主键) | Int(11) not null | ID |
| speed | Smallint(6) null | 速度(基类 Vehicle 的属性) |
| engine | Varchar(30) null | 发动机(子类 Car 的属性) |

Camion 表

| 字段名称 | 字段类型属性 | 描述 |
|-----------|------------------|--------------------|
| id (主键) | Int(11) not null | ID |
| speed | Smallint(6) null | 速度(基类 Vehicle 的属性) |
| engine | Varchar(30) null | 发动机(子类 Car 的属性) |
| container | Varchar(30) null | 集装箱(子类 Camion 的属性) |

要使用每个具体类一张表(table per concrete class)策略，需要把 @javax.persistence.Inheritance 注释的 strategy 属性设置为 InheritanceType.TABLE_PER_CLASS。

Vehicle.java


```
package com.foshanshop.ejb3.bean;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@Table(name="Vehicle")
public class Vehicle implements Serializable{
    private Long id;
    private Short speed;//速度

    @Id
    @Column(columnDefinition="integer")
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Short getSpeed() {
        return speed;
    }

    public void setSpeed(Short speed) {
        this.speed = speed;
    }
}
```

注意：一旦使用这种策略就意味着你不能使用 AUTO generator 和 IDENTITY generator，即主键值不能采用数据库自动生成。

Car. java

```
package com.foshanshop.ejb3.bean;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
```

```

@Table(name="Car")
public class Car extends Vehicle{
    private String engine;//发动机

    @Column(nullable=true,length=30)
    public String getEngine() {
        return engine;
    }

    public void setEngine(String engine) {
        this.engine = engine;
    }
}

```

Camion.java

```

package com.foshanshop.ejb3.bean;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@SuppressWarnings("serial")
@Entity
@Table(name="Camion")
public class Camion extends Car{
    private String container;//集装箱

    @Column(nullable=true,length=30)
    public String getContainer() {
        return container;
    }

    public void setContainer(String container) {
        this.container = container;
    }
}

```

Session Bean 与上节的代码有些改动，代码如下：

```

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.EntityInheritanceDAO;
import com.foshanshop.ejb3.bean.Car;
import com.foshanshop.ejb3.bean.Vehicle;
import com.foshanshop.ejb3.bean.Camion;

@Stateless
@Remote ({EntityInheritanceDAO.class})
public class EntityInheritanceDAOBean implements EntityInheritanceDAO {
    @PersistenceContext
    protected EntityManager em;

    public void initializeData() {
        //添加Vehicle
        Vehicle vehicle = new Vehicle();
        vehicle.setId((long)1);
        vehicle.setSpeed((short)100);
        em.persist(vehicle);
        //添加Car
        Car car = new Car();
        car.setId((long)2);
        car.setSpeed((short)300);
        car.setEngine("A发动机");
        em.persist(car);
        //添加Camion
        Camion camion = new Camion();
        camion.setId((long)3);
        camion.setSpeed((short)200);
        camion.setEngine("B发动机");
        camion.setContainer("2吨集装箱");
        em.persist(camion);
    }

    public Vehicle getLastVehicle() {
        //查询所有Vehicle时，因为他是最继承树中的根，查询结果会得到所有继承于Vehicle类的记录
        //转译成的SQL片断：select * from Vehicle_Hierarchy
        Query query = em.createQuery("from Vehicle v");
        List result = query.getResultList();
        if (result!=null && result.size()>0)
            return (Vehicle)result.get(result.size()-1);
        return null;
    }

    public Car getLastCar() {
        //查询所有Car时，除了得到Car类的记录，也会得到所有继承于Car类的记录
    }

```

```

        //构造的SQL Where部分: where Discriminator in ('Car', 'Camion')
        Query query = em.createQuery("from Car c");
        List result = query.getResultList();
        if (result!=null && result.size()>0)
            return (Car)result.get(result.size()-1);
        return null;
    }

    public Camion getLastCamion() {
        Query query = em.createQuery("from Camion c");
        List result = query.getResultList();
        if (result!=null && result.size()>0)
            return (Camion)result.get(result.size()-1);
        return null;
    }

    public void deleteVehicle() {
        //执行该操作会删除自身对应记录，还会删除所有继承Vehicle的记录，
        //因为他是最终继承树中的根，就相当于清除整个表的数据
        Query query = em.createQuery("delete from Vehicle v");
        query.executeUpdate();
    }
}

```

Session Bean 的接口及 JSP 客户端可以使用上节的文件，这里不再列出。

该策略的优点:

在这种策略中你可以对子类的属性对应的字段定义 **not null** 约束。

该策略的缺点:

不符合关系数据模型的常规设计规则，每个表中都存在属于基类的多余的字段。同时，为了支持策略的映射，持久化管理者需要决定使用什么方法，一种方法是在 **entity** 载入或多态关联时，容器使用多次查询去实现，这种方法需要对数据库做几次来往查询，非常影响执行效率。另一种方法是容器通过使用 **SQL UNION** 查询来实现这种策略。

选择原则：除非你的现实情况必须使用这种策略，一般情况下不要选择。

本例子的 EJB 源代码在 **EntityInheritanceConcreteclass** 文件夹（源代码下载:<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 **lib** 文件夹下。要恢复 **EntityInheritanceConcreteclass** 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子 EJB（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。例子使用的数据库源配置文件是 **mysql-ds.xml**，你可以在下载的文件中找到。数据库名为 **foshanshop**。

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/EntityInheritanceTest.jsp> 访问客户端。

第七章 Web 服务(Web Service)

7.1 Web Service 的创建

在本章节,我们将创建一个基于 JSR-181 规范的 Web Service 及其对应的客户端。开发一个 JSR-181 POJO Endpoint 的 Web Service 应遵守下面几个步骤:

- 1> 建立一个 POJO endpoint
- 2> 把 endpoint 定义成一个 servlet
- 3> 把 endpoint 打包成一个 Web 应用(war 文件)

下面让我们看看如何建立一个 POJO endpoint。

```
package com.foshanshop.ws;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(name = "HelloWorld",
            targetNamespace = "http://com.foshanshop.ws", serviceName = "HelloWorldService")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class HelloWorldService {
    @WebMethod
    public String SayHello(String name) {
        return name+ "说:这是我的第一个 web 服务";
    }
}
```

@WebService 这个注释放置在 Java 类的前面,声明这个类的部分方法可以被发布为 Web 服务。

@WebService 的属性用于设置 Web 服务被发布时的一些配置信息,常用的属性说明如下

1. name

Web 服务的名字, WSDL 中 wsdl:portType 元素的 name 属性和它保持一致,默认是 Java 类或者接口的名字。

2. serviceName

Web 服务的服务名, WSDL 中 wsdl:service 元素的 name 属性和它保持一致,默认是 Java 类的名字+” Service”。

3. targetNamespace

WSDL 文件所使用的 namespace, 该 Web 服务中所产生的其他 XML 文档同样采用这个作为 namespace。

@SOAPBinding()表示这个服务可以映射到一个 SOAP 消息中。Style 用于指定 SOAP 消息请求和回应的编码方式。

@WebMethod 这个注释放在需要被发布成 Web 服务的方法前面。

下面把 POJO endpoint 定义成一个 servlet。

Web.xml

```
<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">

<servlet>
  <servlet-name>HelloWorldService</servlet-name>
  <servlet-class>com.foshanshop.ws.HelloWorldService</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>HelloWorldService</servlet-name>
  <url-pattern>/HelloWorldService/*</url-pattern>
</servlet-mapping>

</web-app>

```

把 endpoint 打包成一个 web 应用(*.war),下面是 Ant 配置文件 build.xml 的片断:

```

<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/foshanshop/ws/HelloWorldService.class" />
    </classes>
  </war>
</target>

```

注意: POJO endpoint 文件及 web.xml 都是必须的。

经过上面的步骤,完成了一个 Web Service 的开发,下面我们通过 Jboss 管理平台查看刚才发布的 Web Service ,我们可以输入 http://localhost:8080/jbossws/进入 JbossWS 的查看界面,如下:



点击“view”连接后,可以查看已经发布的 web services, 如下图:

JBossWS/Services

Registered Service Endpoints

ServiceEndpointID jboss.ws:context=Services,endpoint=HelloWorldService
ServiceEndpointAddress <http://localhost:8080/Services/HelloWorldService?wsdl>

| StartTime | StopTime | |
|------------------------------|-------------------|-------------------|
| Mon Apr 09 19:14:52 CST 2007 | | |
| RequestCount | ResponseCount | FaultCount |
| 0 | 0 | 0 |
| MinProcessingTime | MaxProcessingTime | AvgProcessingTime |
| 0 | 0 | 0 |

在上图中你可以点击 ServiceEndpointAddress 下的路径 <http://localhost:8080/Services/HelloWorldService?wsdl> 访问他的 wsdl 描述, wsdl 描述文件在应用发布时由容器自动生成, 输出如下:

```
<definitions name='HelloWorldService' targetNamespace='http://com.foshanshop.ws'
xmlns='http://schemas.xmlsoap.org/wsdl/' xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
xmlns:tns='http://com.foshanshop.ws' xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <types></types>
  <message name='HelloWorld_SayHello'>
    <part name='arg0' type='xsd:string'></part>
  </message>
  <message name='HelloWorld_SayHelloResponse'>
    <part name='return' type='xsd:string'></part>
  </message>
  <portType name='HelloWorld'>
    <operation name='SayHello' parameterOrder='arg0'>
      <input message='tns:HelloWorld_SayHello'></input>
      <output message='tns:HelloWorld_SayHelloResponse'></output>
    </operation>
  </portType>
  <binding name='HelloWorldBinding' type='tns:HelloWorld'>
    <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'>
    <operation name='SayHello'>
      <soap:operation soapAction=''>
      <input>
        <soap:body namespace='http://com.foshanshop.ws' use='literal'>
      </input>
      <output>
        <soap:body namespace='http://com.foshanshop.ws' use='literal'>
      </output>
    </operation>
  </binding>
</service name='HelloWorldService'>
```

```

<port binding='tns:HelloWorldBinding' name='HelloWorldPort'>
  <soap:address location='http://localhost:8080/Services/HelloWorldService'/>
</port>
</service>
</definitions>

```

本例子 Ant 的配置文件 (build.xml) 如下:

```

<?xml version="1.0"?>
<!-- ===== -->
<!-- JBoss build file -->
<!-- ===== -->

<project name="JBoss" 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="${basedir}/lib">
      <include name="**/*.jar" />
    </fileset>
    <pathelement location="${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="编译 web 服务">
    <javac srcdir="${src.dir}" destdir="${build.classes.dir}" debug="on" deprecation="on"
optimize="off" includes="**">

```



```

        <classpath refid="build.classpath" />
    </javac>
</target>
<!--
Build the test deployments
-->
<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/foshanshop/ws/HelloWorldService.class" />
        </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}" />
</target>
</project>

```

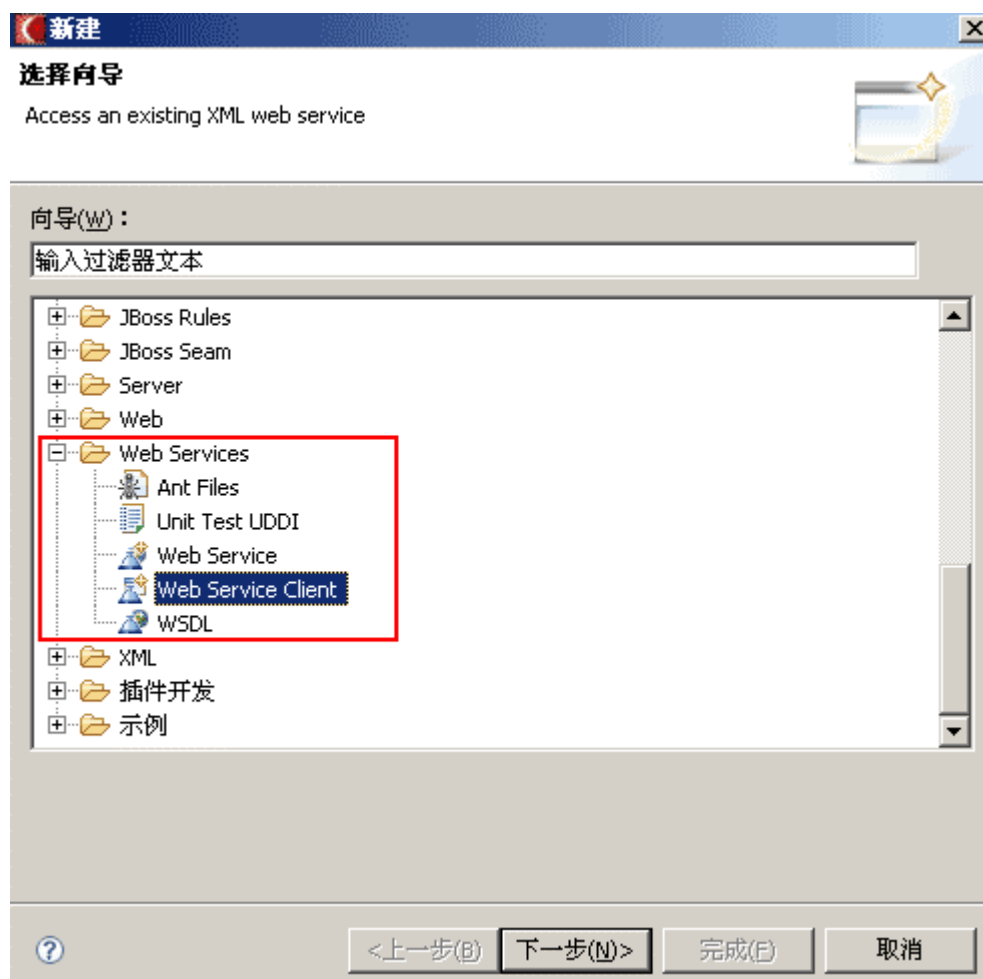
本例子的源代码在 JWS 文件夹（源代码下载：<http://www.foshanshop.net/>），项目中使用到的类库在上级目录 lib 文件夹下。要恢复 JWS 项目的开发环境请参考第三章“如何恢复本书配套例子的开发环境”，要发布本例子（确保配置了环境变量 JBOSS_HOME 及启动了 Jboss），你可以执行 Ant 的 deploy 任务。

7.2 Web Service 的客户端调用

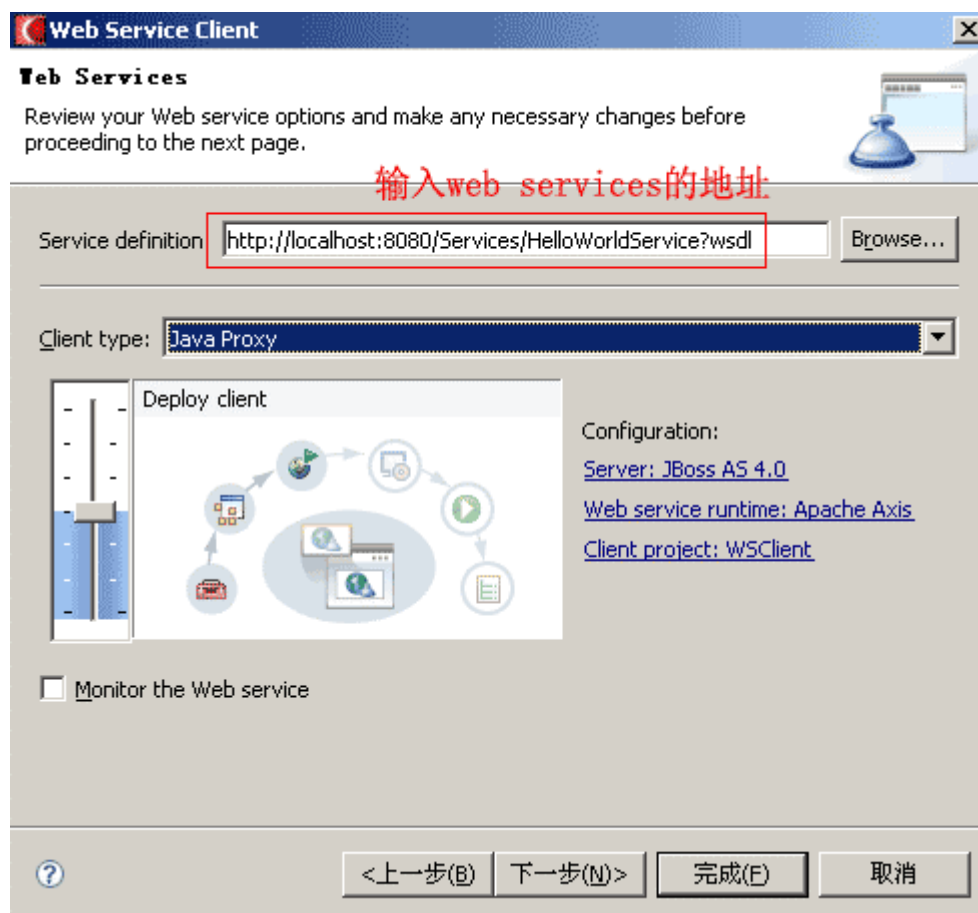
作为 Web Service 的客户端，你可以采用自己喜欢的语言进行开发。本节将介绍如何用 java 和 Asp 两种语言进行客户端开发，当然你也可以采用 C# 开发。

7.2.1 用 java 语言调用 Web Service

JbossIDE 提供了生成 Web Service 客户端代码的功能，我们用该功能生成 HelloWorldService 的客户端代码。首先我们新建一个名为 WSClient 的 java 项目，然后在项目名称上右击鼠标，在出现的属性菜单上点击“新建”——“其他”，出现下面窗口：



打开 Web Services 文件夹，选择 “Web Service Client”，点 “下一步”，在出现的窗口里输入 “http://localhost:8080/Services/HelloWorldService?wsdl”。



点击“下一步”直到完成。

生成的文件列表如下：

```
ws.foshanshop.com
├── HelloWorld.java
├── HelloWorldBindingStub.java
├── HelloWorldProxy.java
├── HelloWorldService.java
└── HelloWorldServiceLocator.java
```

现在我们就通过生成的代码调用 Web Services，我们新建一个名为 TestHelloWorld 的 java 文件，代码如下：

```
package com.foshanshop.AppTest;
import ws.foshanshop.com.HelloWorld;
import ws.foshanshop.com.HelloWorldProxy;

public class TestHelloWorld {

    public static void main(String[] args) {
        try {
            HelloWorldProxy proxy = new HelloWorldProxy();
            HelloWorld port = (HelloWorld) proxy.getHelloWorld();
            String out = port.sayHello("黎明");
            System.out.println("结果:" + out);
        } catch (Exception e) {
```

```

        e.printStackTrace();
    }
}
}

```

直接运行该文件就可以看到调用结果。

本例子的源代码在 WSCClient 文件夹(源代码下载:<http://www.foshanshop.net/>), 项目中使用到的类库在上级目录 lib 文件夹下。

7.2.2 用 asp 调用 Web Service

使用 ASP 调用 Web Service 的方法, 可以用 XMLHTTP 组件发送 SOAP 请求, 下面是调用 HelloWorldService 服务 SayHello 方法的 SOAP message

```

<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header/>
<env:Body>
  <ns1:SayHello xmlns:ns1='http://com.foshanshop.ws'>
    <String_1>佛山人</String_1>
  </ns1:SayHello>
</env:Body>
</env:Envelope>

```

下面的 asp 代码把上面的 SOAP message 发送到 <http://localhost:8080/Services/HelloWorldService>

```

<%
postUrl = "http://localhost:8080/Services/HelloWorldService"
Set xmlhttp = server.CreateObject("Msxml2.XMLHTTP")
xmlhttp.Open "POST",postUrl,false

SoapRequest ="<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header/><env:Body>"
SoapRequest = SoapRequest & "  <ns1:SayHello xmlns:ns1='http://com.foshanshop.ws'>"
SoapRequest = SoapRequest & "    <String_1>佛山人</String_1>"
SoapRequest = SoapRequest & "  </ns1:SayHello>"
SoapRequest = SoapRequest & "</env:Body></env:Envelope>"

xmlhttp.setRequestHeader "Content-Length",LEN(SoapRequest)
xmlhttp.setRequestHeader "SOAPAction", postUrl
xmlhttp.Send(SoapRequest)
str = xmlhttp.responseText
response.write str
%>

```

执行上面的 asp 后输出的 SOAP message 如下:

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Header/>
<env:Body>
  <ns1:SayHelloResponse xmlns:ns1='http://com.foshanshop.ws'>
    <result>佛山人说:这是我的第一个 web 服务</result>
  </ns1:SayHelloResponse>
</env:Body>
</env:Envelope>
```

result 节点的值就是 SayHello 方法的返回值。

第八章 使用 EJB3.0 构建轻量级应用框架

本章是一个独立的知识点,上面章节介绍的都是 EJB3.0 的分布式应用,而本章介绍的却是 EJB3.0 非分布式应用。JBoss EJB 3.0 除了可以运行在 Jboss 服务器中之外,可嵌入 (embeddable) 版本的 Jboss EJB 3.0 还支持独立运行在 junit tests, Tomcat 或其他应用服务器中,当然并不是所有 Jboss 中间件都可用,用可的功能如下:

- Local JNDI
- Transaction Manager
- JMS
- Local TX datasource/connection pool
- Stateful, Stateless, Service, Consumer, Producer, and MDBs
- EJB 3 Persistence
- Hibernate integration
- EJB Security

不可用的功能如下(引用文档原话):

- XA Connection pool is not available yet.
- When embedding into Tomcat, you still require a JBoss specific JNDI implementation. Tomcat's JNDI is read-only.
- You still must use the JBoss transaction manager even when embedding in another app server vendor. This will be remedied in the future when the JBoss AOP/Microcontainer integration is complete.
- Distributed remote communication is not supported yet.
- EJB Timer service not supported
- Even though @Remote interfaces are local, you can only communicate through local connections.
- You cannot access JMS remotely. Only locally. Thus, you have to lookup the "java:/ConnectionFactory".
- JNDI is not available remotely

在 EJB3 产品没有出来前,主流的应用框架要算 Spring,他和 EJB3.0 的区别是。前者不是标准,但非常流行,它基于依赖注入模式,大量使用 XML 配置;后者是 JCP 规定的标准,可以预料所有主要的 J2EE 厂商都会支持,它大量使用 annotation 记录配置信息。他们两者之间可以互相替代。

8.1 在 WEB 中使用 EJB3.0 框架

介绍如何在 WEB 中使用 EJB3.0 框架时,我们需要下载可嵌入 (embeddable) 版本的 Jboss EJB 3.0,下载网址:

http://sourceforge.net/project/showfiles.php?group_id=22866&package_id=132063, 文件名称:

jboss-EJB-3.0_Embeddable_ALPHA_8.zip(注: 不要下载 jboss-EJB-3.0_Embeddable_ALPHA_9.zip, 该版本目前在 tomcat 下启动不了)。

下载完后, 我们解压文件, 把 lib 文件夹下所有文件拷贝到 Web 应用的 WEB-INF\lib 下面 (建议拷贝到 Tomcat 的 shared/lib 文件夹下, 这样所有的 WEB 应用都可以共享), 把 conf 文件夹下的所有配置文件拷贝到 WEB-INF\classes 下。接着我们需要对 WEB 应用的 web.xml 文件进行修改, 加入下面的配置项:

```
<context-param>
  <param-name>jboss-kernel-deployments</param-name>
  <param-value>embedded-jboss-beans.xml, jboss-jms-beans.xml</param-value>
</context-param>

<listener>
  <listener-class>org.jboss.ejb3.embedded.ServletBootstrapListener</listener-class>
</listener>
```

embedded-jboss-beans.xml, jboss-jms-beans.xml 会自动被 ServletBootstrapListener 类载入, 如果你不需要 JMS 服务, 可以把 jboss-jms-beans.xml 配置文件去掉。org.jboss.ejb3.embedded.ServletBootstrapListener 用作启动 EJB3.0 框架, 如果你的监听器 (listener) 使用了 EJB3.0 框架的东西, 请确保 EJB3.0 框架在你的监听器之前启动, 否则将会抛出异常。

ServletBootstrapListener 会自动扫描并发布 /WEB-INF/lib 文件夹下的所有 EJB 和 Entity beans. 如果你想关掉自动扫描发布功能, 可以在 web.xml 文件中加入下面的配置项:

```
<context-param>
  <param-name>automatic-scan</param-name>
  <param-value>>false</param-value>
</context-param>
```

到目前为止, 在 WEB 中使用 EJB3.0 框架的配置已经完成, 接下来我们就开始学习在这种框架中如何进行开发。在实际的项目开发上, 作者有几点建议:

1. 为了日后移植到 JBOSS, EJB 组件最好同时实现他的 Local 及 Remote 接口, 并且指向同一个业务接口。
2. 对于 Local 及 Remote 接口的 JNDI 访问最好做成可配置方式, 在本章介绍的 WEB 框架中尽量访问 EJB 的 Local 接口, 日后移植到 JBOSS 时, 可以切换成 Remote 接口访问。
3. 把 EJB 访问的上下文环境独立出来。

8.1.1 如何使用 Session Bean

为了说明 EJB 可在两种 EJB 3.0 版本 (一种是本章介绍的可嵌入版本, 另一种是前面章节使用的运行在 Jboss 中的版本) 中通用, 本节直接使用第四章介绍的 HelloWorld 例子, 把 HelloWorld.jar 放入 WEB 应用的 WEB-INF\lib 文件夹下, EJB 3.0 容器将会自动发布该 EJB。

EJB 的 JSP 客户端代码如下:

Test.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.HelloWorld, javax.naming.*, com.foshanshop.conf.Constants"%>
<%

    try {
```

```

        InitialContext ctx = Constants.getInitialContext();
        HelloWorld helloworld = (HelloWorld) ctx.lookup("HelloWorldBean/remote");
        out.println(helloworld.SayHello("佛山人"));
    } catch (NamingException e) {
        out.println(e.getMessage());
    }
}

```

%>

Test.jsp 中使用的 Constants 类

```

package com.foshanshop.conf;
import java.util.Properties;
import javax.naming.InitialContext;

public class Constants {
    private static InitialContext ctx = null;

    public static InitialContext getInitialContext() {
        if (ctx!=null){
            return ctx;
        }else{
            try {
                Properties props = new Properties();
                props.load(Thread.currentThread().getContextClassLoader().getResourceAsStream("jndi.properties"));
                ctx = new InitialContext(props);
            } catch (Exception e) {
                ctx = null;
            }
            return ctx;
        }
    }
}

```

jndi.properties 文件内容

```

java.naming.factory.initial=org.jnp.interfaces.LocalOnlyContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces

```

本例子的源代码在 EmbeddedEJB3 文件夹（源代码下载：<http://www.foshanshop.net/>）。要打包成*.war 文件，你可以执行 Ant 的 web-war 任务。运行本例子时需要把 EJB3 可嵌入版本的三个 jar 文件(jboss-ejb3-all.jar, thirdparty-all.jar, hibernate-all.jar)放置在 Tomcat 的 shared/lib 文件夹下(或放置在 WEB 应用的 WEB-INF/lib 下),把 EmbeddedEJB3.war 文件拷贝到 Tomcat 的 webapps 目录，通过 <http://localhost:8080/EmbeddedEJB3/Test.jsp> 访问客户端。

注意：由于后面章节共用本例子，发布前请检查你的数据源参数与例子所设参数是否一致，数据源在 `embedded-jboss-beans.xml` 文件中配置，关于各项参数的具体配置请参考后面章节“如何使用 Entity Bean”。

8.1.2 如何使用 Message Driven Bean

在使用 Message Driven Bean 之前需要先初始化 JBoss MQ 核心服务，方法是在 web.xml 文件的配置参数 jboss-kernel-deployments 加入 jboss-jms-beans.xml 配置文件

```
<context-param>
  <param-name>jboss-kernel-deployments</param-name>
  <param-value>embedded-jboss-beans.xml, jboss-jms-beans.xml</param-value>
</context-param>
```

另外还要配置你自己使用的 Queue/Topic，下面是队列名为 foshanshop 的 Queue 队列配置：

foshanshop-jms.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
  xmlns="urn:jboss:bean-deployer">
  <bean name="jboss.mq.destination:service=Queue,name=foshanshop" class="org.jboss.mq.kernel.Queue">
    <property name="destinationManagerPojo"><inject
bean="jboss.mq:service=DestinationManager"/></property>
    <property name="initialContextProperties"><inject bean="InitialContextProperties"/></property>
    <property name="destinationName"><value>foshanshop</value></property>
  </bean>
</deployment>
```

配置好的 foshanshop-jms.xml 文件放在 WEB-INF/classes 下，同时在 web.xml 文件的配置参数 jboss-kernel-deployments 加入 foshanshop-jms.xml，内容如下：

```
<context-param>
  <param-name>jboss-kernel-deployments</param-name>
  <param-value>
    embedded-jboss-beans.xml, jboss-jms-beans.xml, foshanshop-jms.xml
  </param-value>
</context-param>
```

本节使用第五章的“消息驱动 Bean”例子，把 MessageDrivenBean.jar 放入 WEB 应用的 WEB-INF/lib 文件夹下，EJB 3.0 容器将会自动发布该消息驱动 Bean。

消息驱动 Bean 的 JSP 客户端代码如下：

MessageDrivenBeanTest.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="javax.naming.*, java.text.*, javax.jms.*, com.foshanshop.conf.Constants"%>
<%
  QueueConnection cnn = null;
  QueueSender sender = null;
  QueueSession sess = null;
  Queue queue = null;
```



```

try {
    InitialContext ctx = Constants.getInitialContext();
    QueueConnectionFactory factory = (QueueConnectionFactory)
ctx.lookup("java:/ConnectionFactory");
    cnn = factory.createQueueConnection();
    sess = cnn.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
    queue = (Queue) ctx.lookup("queue/foshanshop");
} catch (Exception e) {
    out.println(e.getMessage());
}

TextMessage msg = sess.createTextMessage("佛山人您好，这是我的第一个消息驱动Bean");
sender = sess.createSender(queue);
sender.send(msg);
sess.close ();
out.println("消息已经发送出去了，你可以到Tomcat控制台查看Bean的输出");
%>

```

本例子的源代码在 EmbeddedEJB3 文件夹（源代码下载：<http://www.foshanshop.net/>）。发布方式同上，通过 <http://localhost:8080/EmbeddedEJB3/MessageDrivenBeanTest.jsp> 访问客户端。

8.1.3 如何使用依赖注入(dependency injection)

本节使用第四章的“依赖注入”例子，把 MessageDrivenBean.jar 放入 WEB 应用的 WEB-INF\lib 文件夹下，EJB 3.0 容器将会自动发布该 EJB。

JSP 客户端代码如下：

InjectionTest.jsp

```

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

```

本例子的源代码在 EmbeddedEJB3 文件夹（源代码下载：<http://www.foshanshop.net/>）。发布方式同上，通过 <http://localhost:8080/EmbeddedEJB3/InjectionTest.jsp> 访问客户端。

8.1.4 如何使用 Entity Bean

在使用 Entity Bean 之前需要先配置数据源，数据源在 embedded-jboss-beans.xml 文件中配置，本例采用与前面章节一样的配置参数（数据库名：foshanshop 用户名：root 密码：123456 数据源名称：DefaultMySqlDS），加入的内容片断如下：

```
<bean name="DefaultMySqlDSBootstrap" class="org.jboss.resource.adapter.jdbc.local.LocalTxDataSource">
    <property name="driverClass">org.gjt.mm.mysql.Driver</property>
    <property name="connectionURL">
        jdbc:mysql://localhost:3306/foshanshop?useUnicode=true&characterEncoding=GBK
    </property>
    <property name="userName">root</property>
    <property name="password">123456</property>
    <property name="jndiName">java:/DefaultMySqlDS</property>
    <property name="minSize">0</property>
    <property name="maxSize">10</property>
    <property name="blockingTimeout">1000</property>
    <property name="idleTimeout">100000</property>
    <property name="transactionManager"><inject bean="TransactionManager"/></property>
    <property name="cachedConnectionManager"><inject
bean="CachedConnectionManager"/></property>
    <property name="initialContextProperties"><inject bean="InitialContextProperties"/></property>
</bean>

<bean name="DefaultMySqlDS" class="java.lang.Object">
    <constructor factoryMethod="getDatasource">
        <factory bean="DefaultMySqlDSBootstrap"/>
    </constructor>
</bean>
```

数据源配置好之后，我们还需把数据库驱动 Jar 文件放置在 WEB 应用的 WEB-INF\lib 文件夹下(或放置在 Tomcat 的 shared/lib 文件夹下)。

本节使用第六章的“单表映射的实体 Bean”例子,把 EntityBean.jar 放入 WEB 应用的 WEB-INF\lib 文件夹下, EJB 3.0 容器将会自动发布该 EJB。

EJB 的 JSP 客户端代码如下：

EntityBeanTest.jsp

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

<%
    try {
        InitialContext ctx = Constants.getInitialContext();
```

```
PersonDAO persondao = (PersonDAO) ctx.lookup("PersonDAOBean/remote");
SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
persondao.insertPerson("黎活明", true, (short)26, formatter.parse("1980-9-30"));
```

添加一个人

```
out.println(persondao.getPersonNameByID(1)); //取personid为1的人
} catch (Exception e) {
    out.println(e.getMessage());
}
```

%>

本例子的源代码在 EmbeddedEJB3 文件夹（源代码下载:<http://www.foshanshop.net/>）。发布方式同上，通过 <http://localhost:8080/EmbeddedEJB3/EntityBeanTest.jsp> 访问客户端。

注意：一定别忘了把数据库驱动 Jar 文件放置在 WEB 应用的 WEB-INF\lib 文件夹下，或放置在 Tomcat 的 shared/lib 文件夹下。

其中“事务管理”例子你可以通过 <http://localhost:8080/EmbeddedEJB3/TransactionTest.jsp> 访问

“多对多”例子你可以通过 <http://localhost:8080/EmbeddedEJB3/ManyToManyTest.jsp> 访问