

# JBoss 使用指南

Release v1.0

[jimmy (yang. kun), 于 2008-05-12 编写]

一. JBOSS 入门.....	3
1. 下载和安装 JBoss.....	3
2. JBoss 的目录结构.....	3
3. 启动服务器.....	4
4. JMX 控制台.....	5
5. 停止服务器.....	6
6. JBoss 中的部署.....	6
二. JBOSS 的配置.....	7
1. JBoss 日志设置.....	7
2. web 服务的端口号的修改.....	7
3. JBoss 的安全设置.....	8
3.1 jmx-console 登录的用户名和密码设置.....	8
3.2 web-console 登录的用户名和密码设置.....	10
4. JBoss 数据源的配置.....	13
5. JMS 使用和设置.....	15
5.1 JMS 消息的传递模型.....	15
5.2 JMS 的配置.....	17
三. JMX 原理和应用.....	21
1. 什么是 JMX.....	21
2. JMX 规范.....	22
3. 基于 JBoss 来写 MBean.....	22
3.1 HelloWorld 实例.....	22
3.2 程序代码.....	22
3.3 配置文件 jboss-service.xml.....	23
3.4 将实例部署到 JBOSS.....	23
3.5 MBean 的效果.....	24
s 四. EJB3.0 使用说明.....	26
1. Enterprise JavaBeans (EJB) 的概念.....	26
1.1 会话 Bean: .....	26
1.2 实体 Bean: .....	27
1.3 消息驱动 Bean (MDB): .....	27
2. 会话 Bean (Session Bean) .....	27
2.1 因为客户端需要通过 JNDI 查找 EJB, 那么 JNDI 是什么.....	27
2.2 Stateless Session Beans (无状态 bean) 开发.....	29
2.3 Stateless Session Bean 与 Stateful Session Bean 的区别.....	33
2.4 Session Bean 的生命周期.....	33
3. 消息驱动 Bean (Message Driven Bean) .....	34
3.1 Queue 消息的发送与接收 (PTP 消息传递模型) .....	34
3.2 Topic 消息的发送与接收 (Pub/sub 消息传递模型) .....	40
4. 实体 Bean (Entity Bean) .....	45
4.1 持久化 persistence.xml 配置文件.....	45
4.2 实体 Bean 发布前的准备工作.....	46

## 一. JBOSS 入门

### 1. 下载和安装 JBoss

在下载和安装 JBoss 之前，请开发者确认一下自己的机器是否安装了最新版的 JVM。为运行 JBoss 4.2.2GA，开发者必须提供 Java 5 虚拟机。在我们动身之前，请再次检查一下您是否安装了合适的 JDK，而且 JAVA\_HOME 环境变量是否已经设置好。

用户可以从 JBoss 网站 (<http://www.jboss.org/jbossas/downloads/>) 免费下载到 JBoss 应用服务器。其中，可用的二进制版本格式分别有 .zip、.tar.gz 以及 .bz2。JBoss 二进制发布版的具体内容与版本格式无关，用户需要根据各自的平台选择相应的二进制版本。在您下载完成 JBoss 后，将它解压到合适的机器位置上。有一点请注意，包含解压目录的完整路径（比如，Windows 操作系统中的 Program Files 目录）上不能够含有空格，因为这将导致错误的出现。

### 2. 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。
server	各种JBoss 配置。每个配置必须放在不同的子目录。子目录的名字表示配置的名字。 JBoss 包含3 个默认的配置：minimal，default 和 all，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，它含有大部分J2EE应用所需的标准服务。但是，它不含有JAXR服务、IIOP服务、或者其他任何群集服务。
server/minimal	这是启动JBoss服务器所要求的最低配置。minimal配置将启动日志服务、JNDI服务器以及URL部署扫描器，

	以找到待部署的（新）应用。对于那些不需要使用任何其他J2EE技术，而只是使用自定义服务的场合而言，则这种JMX/JBoss 配置最适合。它仅仅是服务器，而不包含Web容器、不提供EJB和JMS支持。
server/default/conf	含有指定JBoss核心服务的jboss-service.xml文件。同时，还包括核心服务的其他配置文件。
server/default/data	这一目录存储持久化数据，即使服务器发生重启其中的数据也不会丢失。许多JBoss服务将数据存储在这里，比如Hypersonic数据库实例。
server/default/deploy	用户将应用代码（JAR\WAR\EAR文件）部署在此处。同时， <b>deploy</b> 目录也用于热部署服务（即，那些能够从运行服务器动态添加或删除的服务）。因此，用户能够在 <b>deploy</b> 目录看到大量的配置文件。尤其是，用户能够看到 <b>JMX</b> 控制台应用（未打包的 <b>WAR</b> 文件），本书前面讨论过。JBoss服务器将定期扫描该目录，从而查找是否有组件更新或修改，从而自动完成组件的重新部署。
server/default/lib	服务器配置所需的JAR文件。用户可以添加自身的库文件，比如JDBC驱动，等等。
server/default/log	日志信息将存储到该目录。JBoss使用Jakarta Log4j包作为其日志功能。同时，用户可以在应用中直接使用Log4j日志记录功能。
server/default/tmp	供部署器临时存储未打包应用使用，也可以作为其他用途。
server/default/work	供编译JSP使用。

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

既然提到了 JBoss 中的热部署服务主题，接下来在探讨服务器配置问题前先来看看实际例子。如果还没有启动 JBoss，则请运行它。然后，请再次查看 deploy 目录（用户必须保证运行了 default 配置），然后删除 mail-service.xml 文件。通过运行 JBoss 服务器的控制台能够浏览到如下信息：

```
13:10:05,235 INFO [MailService] Mail service 'java:/Mail' removed from JNDI
```

然后，再次将 mail-service.xml 文件放回原处，用户将通过控制台再次发现 JBoss 重新部署了该服务。所以，这就是 JBoss 的热部署。

### 3. 启动服务器

首先，来看看如何运行 JBoss 服务器。用户可以在 JBoss 主安装目录的 bin 目录中找到若干个脚本文件。请执行 run 脚本（对于 Windows，则运行 run.bat；对于 Linux、OS X、UNIX 系统，则运行 run.sh）。默认 jboss 运行 default 配置，如果要运行 all 配置请执行命令 run -c all。其中，部署和启动 JBoss 组

件的具体日志信息能够在运行 JBoss 的控制台浏览到。如下消息表明，JBoss 服务器成功运行：

```
11:18:46,828 INFO [Server] JBoss (MX MicroKernel) [4.2.2.GA (build:SVNTag=JBoss_4_2_2_GA date=200710221139)] Started in 27s:94ms
```

用户可以通过 Web 浏览器验证 JBoss 应用服务器是否在运行，其 HTTP 监听端口为 8080（其中，必须保证在启动 JBoss 时，8080 端口并没有被其他应用或服务占用）。通过 Web 浏览器能够找到相关有用的 JBoss 资源（译者注：<http://localhost:8080>）。

## 4. JMX 控制台

通过 <http://localhost:8080/jmx-console>，即 JMX 控制台应用，用户能够浏览到服务器活动视图。图 1.1 给出了示例界面。上述界面给出了 JBoss 管理控制台，它提供了构成 JBoss 服务器的 JMX MBean 原始视图。我们暂时可以不用理会控制台的任何内容，但是需要知道控制台能够提供运行中的 JBoss 应用服务器的大量信息。另外，通过它，用户能够修改、启动、停止 JBoss 组件。比如，请找到 `service=JNDIView` 链接，然后单击。该特定 MBean 提供了如下服务内容，即能够浏览服务器中 JNDI 命名空间的结构信息。接下来，请在该 MBean 显示页面底端找到 `list` 操作，然后单击 `invoke` 按钮。`invoke` 操作将返回绑定到 JNDI 树中的当前名字列表，这对于获得 EJB 名字很有帮助，比如当 EJB 应用客户端不能够解析 EJB 名字时。

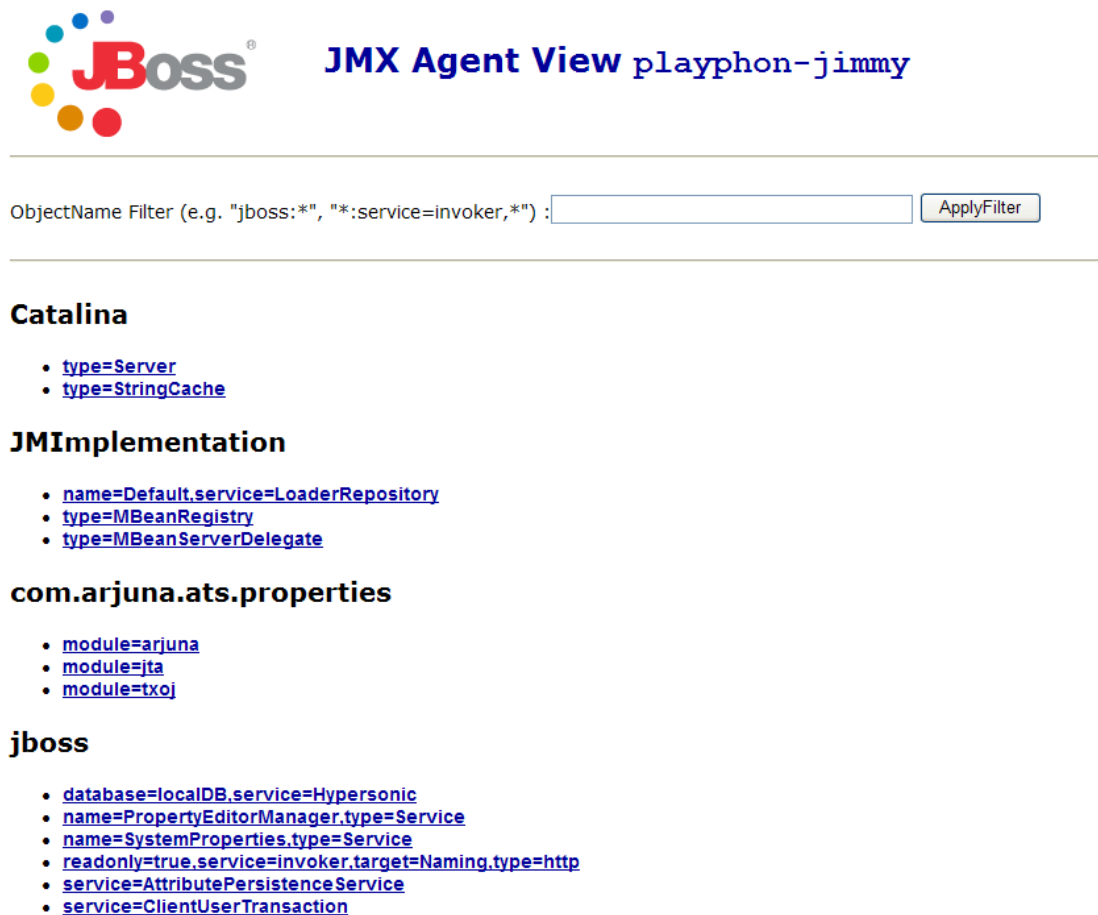


图 1-1 JMX 管理控制台 Web 应用视图

## 5. 停止服务器

为了能够停止 JBoss 服务器，用户可以敲入 Ctrl-C，或者从 bin 目录运行 shutdown 脚本。甚至，用户还可以使用管理控制台（请在 jboss.system 部分找到 type=Server，然后调用 shutdown 操作。）。

## 6. 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。

## 二. JBOSS 的配置

### 1. JBoss 日志设置

Log4j 是 JBoss 使用的日志功能包。通过 conf/jboss-log4j.xml 文件能够控制 JBoss 的日志功能。该文件定义了一套 Appender、指定了日志文件、具体消息 Category 类型的存储、消息格式以及消息的过滤级别。默认时, JBoss 会同时在控制台和日志文件 (位于 log/server.log 文件中) 中生成输出信息。一共存在 5 个基本的日志级别: DEBUG、INFO、WARN、ERROR 以及 FATAL。其中, 控制台的日志入口 (threshold) 为 INFO, 即用户通过控制台能够浏览到提示信息、警告信息、错误信息, 但是调试信息查看不到。相比之下, JBoss 并没有为 server.log 文件设置任何入口, 因此所有生成的消息将记录到 server.log 文件中。如果 JBoss 运行过程中出现了错误, 则通过控制台可能找不到用户有用的信息, 因此建议通过 server.log 文件查看是否有调试信息可供解决问题所用。然而, 请注意, 通过调整日志入口能够在控制台查看到调试信息, 但是这并没有保证所有的 JBoss 消息都将记录到 server.log 文件中。因此, 用户还需要为单个的 Category 设置不同的日志级别。比如, jboss-log4j.xml 给出了如下 Category。

```
<!-- Limit the org.apache category to INFO as its DEBUG is verbose -->
<category name="org.apache">
  <priority value="INFO"/>
</category>
```

### 2. web 服务的端口号的修改

这点在前文中有所提及, 即修改 JBoss 安装目录 server/default/deploy/jboss-web.deployer 下的 server.xml 文件, 内容如下:

```
<Connector port="8080" address="${jboss.bind.address}"
    maxThreads="250" maxHttpHeaderSize="8192"
    emptySessionPath="true" protocol="HTTP/1.1"
    enableLookups="false" redirectPort="8443" acceptCount="100"
    connectionTimeout="20000" disableUploadTimeout="true" />
```

将上面的 8080 端口修改为你想要的端口即可。重新启动 JBoss 后访问：  
<http://localhost/:新设置的端口>，可看到 JBoss 的欢迎界面。

### 3. JBoss 的安全设置

#### 3.1 jmx-console 登录的用户名和密码设置

默认情况访问 <http://localhost:8080/jmx-console> 就可以浏览 jboss 的部署管理的一些信息，不需要输入用户名和密码，使用起来有点安全隐患。下面我们针对此问题对 jboss 进行配置，使得访问 jmx-console 也必须要知道用户名和密码才可进去访问。步骤如下：

##### 3.1.1 修改 jboss-web.xml 文件

打开/server/default/deploy/jmx-console.war/WEB-INF/jboss-web.xml 文件，去掉<security-domain>java:/jaas/jmx-console</security-domain>的注释。修改后的该文件内容为：

```
<jboss-web>
  <!-- Uncomment the security-domain to enable security. You will
    need to edit the htmladaptor login configuration to setup the
    login modules used to authentication users.-->
  <security-domain>java:/jaas/jmx-console</security-domain>
</jboss-web>
```

##### 3.1.2 修改 web.xml 文件

与 3.1.1 中的 jboss-web.xml 同级目录下的 web.xml 文件，查找到<security-constraint/>节点，去掉它的注释，修改后该部分内容为：

```
<!-- A security constraint that restricts access to the HTML JMX console
to users with the role JBossAdmin. Edit the roles to what you want and
uncomment the WEB-INF/jboss-web.xml/security-domain element to enable
secured access to the HTML JMX console.-->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>HtmlAdaptor</web-resource-name>
    <description>An example security config that only allows users with the
```



```
role JBossAdmin to access the HTML JMX console web application
</description>
<url-pattern>/*</url-pattern>
<http-method>GET</http-method>
<http-method>POST</http-method>
</web-resource-collection>
<auth-constraint>
  <role-name>JBossAdmin</role-name>
</auth-constraint>
</security-constraint>
```

在此处可以看出，为登录配置了角色 JBossAdmin。

### 3.1.3 修改 login-config.xml 文件

在第一步中的 jmx-console 安全域和第二步中的运行角色 JBossAdmin 都是在 login-config.xml 中配置，我们在 JBoss 安装目录 /server/default/config 下找到它。查找名字为：jmx-console 的 application-policy：

```
<application-policy name = "jmx-console">
  <authentication>
    <login-module
      code="org.jboss.security.auth.spi.UsersRolesLoginModule"
      flag = "required">
      <module-option
        name="usersProperties">props/jmx-console-users.properties</module-option>
      <module-option
        name="rolesProperties">props/jmx-console-roles.properties</module-option>
    </login-module>
  </authentication>
</application-policy>
```

在此处可以看出，登录的角色、用户等的信息分别在 props 目录下的 jmx-console-roles.properties 和 jmx-console-users.properties 文件中设置，分别打开这两个文件。

其中 jmx-console-users.properties 文件的内容如下：

```
# A sample users.properties file for use with the UsersRolesLoginModule
```

```
admin=admin
```

该文件定义的格式为：用户名=密码，在该文件中，默认定义了一个用户名为 admin，密码也为 admin 的用户，读者可将其改成所需的用户名和密码。

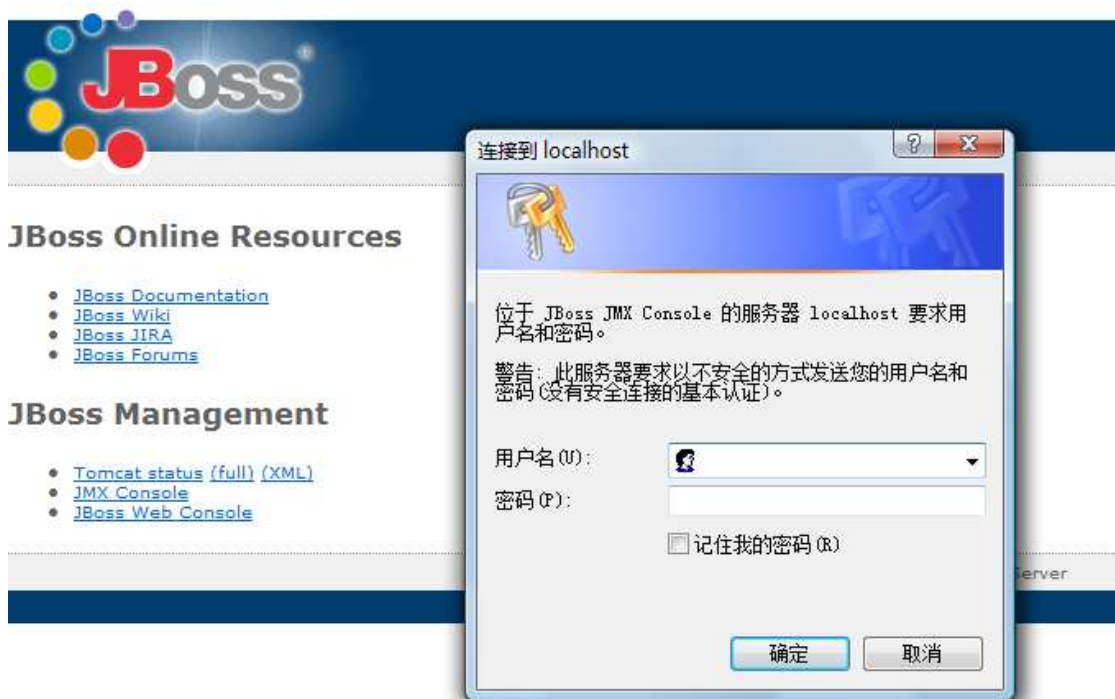
jmx-console-roles.properties 的内容如下：

```
# A sample roles.properties file for use with the UsersRolesLoginModule
```

```
admin=JBossAdmin, HttpInvoker
```

该文件定义的格式为：用户名=角色，多个角色以“,” 隔开，该文件默认为 admin 用户定义了 JBossAdmin 和 HttpInvoker 这两个角色。

配置完成后读者可以通过访问：<http://localhost:8088/jmx-console/>，输入 jmx-console-roles.properties 文件中定义的用户名和密码，访问 jmx-console 的页面。如图：



### 3.2 web-console 登录的用户名和密码设置

默认情况下，用户访问 JBoss 的 web-console 时，不需要输入用户名和密码，为了安全起见，我们通过修改配置来为其加上用户名和密码。步骤如下：

### 3.2.1 修改 jboss-web.xml 文件

打开 server/default/deploy/management/console-mgr.sar/web-console.war/WEB-INF/jboss-web.xml 文件，去掉 `<security-domain>java:/jaas/web-console</security-domain>` 的注释，修改后的文件内容为：

```
<?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>
  <!-- Uncomment the security-domain to enable security. You will
  need to edit the htmladaptor login configuration to setup the
  login modules used to authentication users.-->
  <security-domain>java:/jaas/web-console</security-domain>
  <!-- The war depends on the -->
  <depends>jboss.admin:service=PluginManager</depends>
</jboss-web>
```

### 3.2.2 修改 web.xml 文件

打开 3.2.1 中 jboss-web.xml 同目录下的 web.xml 文件，去掉 `<security-constraint>` 部分的注释，修改后的该部分内容为：

```
<!-- A security constraint that restricts access to the HTML JMX console
to users with the role JBossAdmin. Edit the roles to what you want and
uncomment the WEB-INF/jboss-web.xml/security-domain element to enable
secured access to the HTML JMX console.-->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>HtmlAdaptor</web-resource-name>
    <description>An example security config that only allows users with the
    role JBossAdmin to access the HTML JMX console web application
  </description>
  <url-pattern>/*</url-pattern>
```

```
<http-method>GET</http-method>
```

```
<http-method>POST</http-method>
```

```
</web-resource-collection>
```

```
<auth-constraint>
```

```
<role-name>JBossAdmin</role-name>
```

```
</auth-constraint>
```

```
</security-constraint>
```

### 3.2.3 修改 login-config.xml 文件

打开 JBoss 安装目录 server/default/conf 下的 login-config.xml 文件，搜索 web-console，可找到如下内容：

```
<application-policy name = "web-console">
```

```
  <authentication>
```

```
    <login-module
```

```
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
```

```
    flag = "required">
```

```
      <module-option
```

```
name="usersProperties">web-console-users.properties</module-option>
```

```
      <module-option
```

```
name="rolesProperties">web-console-roles.properties</module-option>
```

```
    </login-module>
```

```
  </authentication>
```

```
</application-policy>
```

在文件中可以看到，设置登录 web-console 的用户名和角色等信息分别在 login-config.xml 文件所在目录下的 web-console-users.properties 和 web-console-roles.properties 文件中，但因为该目录下无这两个文件，我们在 JBoss 安装目录 server/default/conf/props 三目录下建立这两个文件，文件内容可参考在“jmx-console 登录的用户名和密码设置”中的两个相应的配置文件的内容，web-console-users.properties 文件的内容如下：

```
# A sample users.properties file for use with the UsersRolesLoginModule
```

```
admin=admin
```

web-console-roles.properties 文件的内容如下：

```
# A sample roles.properties file for use with the UsersRolesLoginModule
```

```
admin=JBossAdmin,HttpInvoker
```

因为此时这两个文件不与 login-config.xml 同目录，所以 login-config.xml 文件需进行少许修改，修改后的<application-policy name = "web-console">元素的内容为：

```
<application-policy name = "web-console">
```

```
<authentication>
```

```
<login-module
```

```
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
```

```
flag = "required">
```

```
<module-option
```

```
name="usersProperties">props/web-console-users.properties</module-option>
```

```
<module-option
```

```
name="rolesProperties">props/web-console-roles.properties</module-option>
```

```
</login-module>
```

```
</authentication>
```

```
</application-policy>
```

## 4. JBoss 数据源的配置

在 jboss 路径\docs\examples\jca 目录下有各种数据库配置文件的样本。选择一个你合适的，copy 到 server\all\deploy 或 server\default\deploy 下，我们以 msserver 数据库为列，然后修改其中的如下内容：

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

```
<datasources>
```

```
<local-tx-datasource>

<jndi-name>DEV_MSSQLDS</jndi-name>

<connection-url>jdbc:sqlserver://192.168.14.201:1455;DatabaseName=Beta_Core_Ps
msg_Beta6</connection-url>

<driver-class>com.microsoft.sqlserver.jdbc.SQLServerDriver</driver-class>

<user-name>ppcn_beta6</user-name>

<password>ppcn_beta6</password>

<!--最小连接池数目 -->

<min-pool-size>5</min-pool-size>

<!--最大连接池数目 -->

<max-pool-size>800</max-pool-size>

<!--数据库连接空闲时间,单位为分钟,如果负载较大,可以设为 5, 如果
一般,可以设为 3-->

<idle-timeout-minutes>5</idle-timeout-minutes>

<!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml -->

<metadata>

<type-mapping>MS SQLSERVER2000</type-mapping>

</metadata>

</local-tx-datasource>

</datasources>
```

如果后台没有报异常,通过 <http://localhost:8080/jmx-console> 进入 jmx 控制台,在 jdbc 栏目可以看到以下内容:

## jboss.jdbc

- [datasource=DEV\\_MSSQLDS.service=metadata](#)
- [datasource=DefaultDS.service=metadata](#)
- [service=SQLExceptionProcessor](#)
- [service=metadata](#)

代码示例:

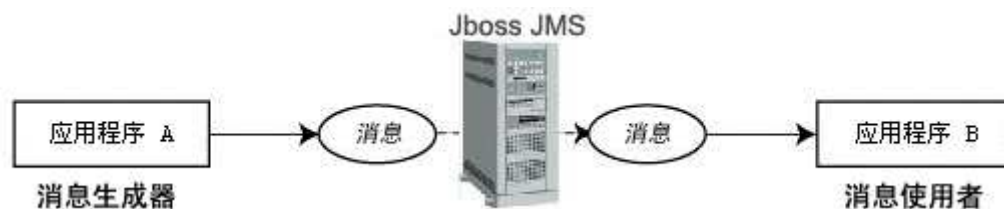
```
javax.naming.InitialContext ctx = new InitialContext();  
javax.sql.DataSource ds = (DataSource)ctx.lookup("java:/DEV_MSSQLDS");
```

另外请注意: 该配置文件可“随意”命名为 XXXX-ds.xml, 该命名的后缀请“确保”为“-ds.xml”。

## 5. JMS 使用和设置

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。



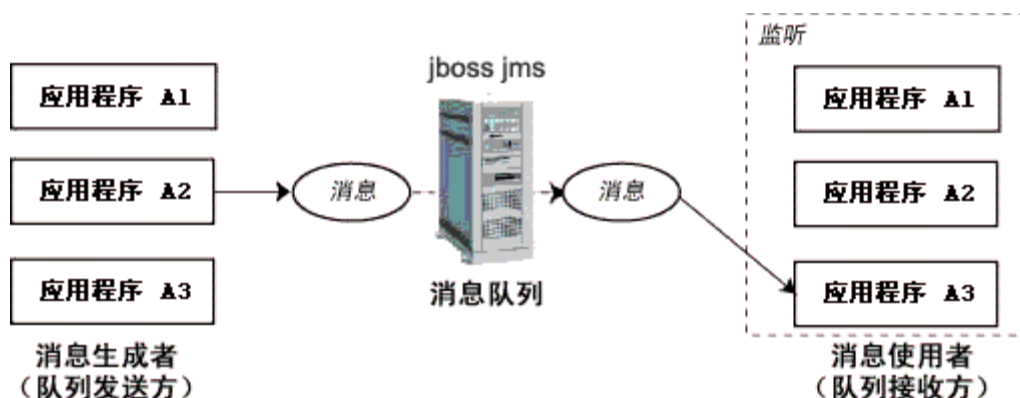
### 5.1 JMS消息的传递模型

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。

#### 5.1.1 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。

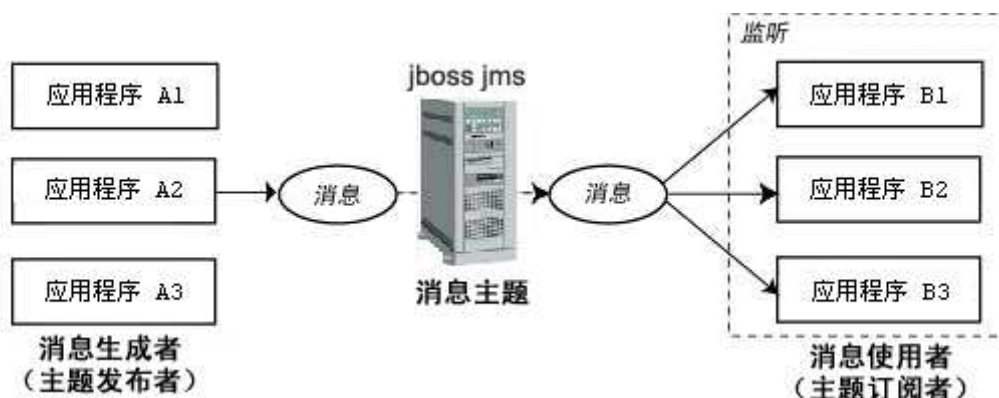


一个队列可以关联多个队列发送方和接收方，但一条消息仅传递给一个队列接收方。如果多个队列接收方正在监听队列上的消息，jboss JMS 将根据“先来者优先”的原则确定由哪个队列接收方接收下一条消息。如果没有队列接收方在监听队列，消息将保留在队列中，直至队列接收方连接队列为止。

### 5.1.2 发布/订阅消息传递

仅仅允通过发布/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。

## 5.2 JMS 的配置

当使用一个 JMS Provider 时，有三个 Provider-specific 因素：

- A 得到一个 JNDI 初始化上下文
- B 用到的连接工厂的名字。
- C 对目的地的管理和命名协定。

JBoss 同它的 JNDI 一起执行。为了简单的 JMS client，配置和查找初始化上下文，同实现其他 J2EE 客户端一样。JMS-specific 来约束 JBoss 的 JMS provider (JBossMQ)。JbossMQ 是通过 xml 文件 jbossmq-service.xml 进行配置的，该文件放在在 server/default/deploy/jms 下。

### 5.2.1 增加新的消息队列

修改文件 jbossmq-destinations-service.xml，文件在 jboss 目录 (server/default/deploy/jms/)。在文件中已经存在几个缺省的目的地，所以你比较容易明白怎样增加到文件中。例如你需要增加一个 TestQName 的 Queue，所以增加下面的语句到 jbossmq-destinations-service.xml 中。这种方式是长久存在的，不随着 JBoss 服务器关闭而消失。

```
<mbean code="org.jboss.mq.server.jmx.Queue"
name="jboss.mq.destination:service=Queue,name="TestQName">

<depends
optional-attribute-name="DestinationManager">jboss.mq:service=DestinationMa
nager</depends>

</mbean>
```

另外一种方法是可以通过 JMX HTML 管理界面。通过 <http://localhost:8080/jmx-console> 来访问。在 jboss.mq 下查找 service=DestinationManager 的连接。然后在 createTopic() 或 createQueue() 来建立。如图：

## void createQueue()

MBean Operation.

Param	ParamType	ParamValue	ParamDescription
p1	java.lang.String	TestQName	(no description)

Invoke

在 ParamValue 里输入 Queue 的名字，用鼠标点击 Invoke 在 jboss.mq.destination 下可以看到 [name=TestQName,service=Queue](#)，如图：

## jboss.mq.destination

- [name=A,service=Queue](#)
- [name=B,service=Queue](#)
- [name=C,service=Queue](#)
- [name=D,service=Queue](#)
- [name=DEV\\_TEST,service=Queue](#)
- [name=DLQ,service=Queue](#)
- [name=TestQ,service=Queue](#)
- [name=TestQName,service=Queue](#)
- [name=ex,service=Queue](#)
- [name=foshanshop,service=Queue](#)
- [name=securedTopic,service=Topic](#)
- [name=student,service=Topic](#)
- [name=testDurableTopic,service=Topic](#)
- [name=testQueue,service=Queue](#)
- [name=testTopic,service=Topic](#)

这种方法建立的目的地是临时性的，随着服务器开始存在，当 JBoss 服务器重新启动时，动态建立的目的地将会不存在。在 JbossMQ 中所有目的地都有一个目的地类型的前缀。对于 topic 前缀是 topic，对于 queue 前缀是 queue。例如查找一个 TestQName 的名字，需要查找名字为“queue/TestQName”。在此种方法中有 createTopic() 或 createQueue() 分别有两种方法：一是有两个参数，二是有一个参数的。两个参数分别是：建立的目的地名称和 JNDI 名称。一个参数的只是目的地名称，对于 JNDI 名称默认是：[目的地类型 (topic/queue)]/目的地名称。在这里我们使用的是第一种方法。直接修改 jbossmq-destinations-service.xml 文件。

### 5.2.2 连接工厂

JBossMQ 包括为 topic 和 queue 几个不同的连接工厂，每个连接工厂有自己特性。当通过 JNDI 来查找一个连接工厂时，需要知道此连接工厂的名称。所有可用连接工厂和它们的属性，名称都会在文件 jbossmq-service.xml 中。有三种类型连接工厂，依赖的通讯协议如下：

## OIL

快速双向 socket 通讯协议。它是缺省的。

## UIL

超过一个 socket 协议，可以使用在通过防火墙访问，或者当客户端不能正确的查找到服务器的 IP 地址。

## RMI

最早的协议，是稳定的，但是比较慢。

## JVM

在 JBoss 2.4 之后增加的一个新的连接工厂类型。不需要用 socket，当客户端和 JbossMQ 使用同样虚拟机时，可以使用。在 JBoss2.4.1 以后版本中，对于 topic- 和 queue-目的地，连接工厂使用同样的名字。下表列出了在 JBoss 中 JMS 连接工厂：

目的地类型	JNDI 名字	连接工厂类型
Topic/Queue	java:/ConnectionFactory	JVM
Topic/Queue	java:/XAConnectionFactory	JVM 支持 XA 事务
Topic/Queue	RMIConnectionFactory	RMI
Topic/Queue	RMIXAConnectionFactory	RMI 支持 XA 事务
Topic/Queue	ConnectionFactory	OIL
Topic/Queue	XAConnectionFactory	OIL 支持 XA 事务
Topic/Queue	UILConnectionFactory	UIL
Topic/Queue	UILXAConnectionFactory	UIL 支持 XA 事务

### 5.2.1 消息存储模式配置

在 jboss 路径\docs\examples\jms 目录下有各种 jms 数据源配置文件的样本。选择一个你合适的，copy 到 server\all\deploy\jms 或 server\default\deploy\jms 下，我们以 msserver 数据库为例，然后修改其中的如下内容：

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

```
<!-- $Id: mssql-jdbc2-service.xml 63369 2007-06-05 22:22:14Z dbhole $ -->
```

```
<server>
```

```
<!--
```

```
    | The destination manager is the core service within JBossMQ
```

```
-->
```

```
<mbean code="org.jboss.mq.server.jmx.DestinationManager"
name="jboss.mq:service=DestinationManager">
```

```
<depends
optional-attribute-name="MessageCache">jboss.mq:service=MessageCache</depend
s>
```

```
<depends
optional-attribute-name="PersistenceManager">jboss.mq:service=PersistenceManage
r</depends>
```

```
<depends
```

```
optional-attribute-name="StateManager">jboss.mq:service=StateManager</depends>
</mbean>
```

```
<mbean code="org.jboss.mq.server.MessageCache"
  name="jboss.mq:service=MessageCache">
  <attribute name="HighMemoryMark">50</attribute>
  <attribute name="MaxMemoryMark">60</attribute>
  <attribute
name="CacheStore">jboss.mq:service=PersistenceManager</attribute>
</mbean>
```

```
<mbean code="org.jboss.mq.pm.jdbc2.MSSQLPersistenceManager"
  name="jboss.mq:service=PersistenceManager">
  <depends
optional-attribute-name="ConnectionManager">jboss.jca:service=DataSourceBinding
,name=DEV_MSSQLDS</depends>
  <attribute name="SqlProperties">
    BLOB_TYPE=BINARYSTREAM_BLOB
    INSERT_TX = INSERT INTO JMS_TRANSACTIONS (TXID) values(?)
    INSERT_MESSAGE = INSERT INTO JMS_MESSAGES (MESSAGEID,
DESTINATION, MESSAGEBLOB, TXID, TXOP) VALUES(?,?,?,?)
    SELECT_ALL_UNCOMMITTED_TXS = SELECT TXID FROM
JMS_TRANSACTIONS
    SELECT_MAX_TX = SELECT MAX(TXID) FROM JMS_MESSAGES
    DELETE_ALL_TX = DELETE FROM JMS_TRANSACTIONS
    SELECT_MESSAGES_IN_DEST = SELECT MESSAGEID,
MESSAGEBLOB FROM JMS_MESSAGES WHERE DESTINATION=?
    SELECT_MESSAGE_KEYS_IN_DEST = SELECT MESSAGEID FROM
JMS_MESSAGES WHERE DESTINATION=?
    SELECT_MESSAGE = SELECT MESSAGEID, MESSAGEBLOB FROM
JMS_MESSAGES WHERE MESSAGEID=? AND DESTINATION=?
    MARK_MESSAGE = UPDATE JMS_MESSAGES SET TXID=?, TXOP=?
WHERE MESSAGEID=? AND DESTINATION=?
    UPDATE_MESSAGE = UPDATE JMS_MESSAGES SET
MESSAGEBLOB=? WHERE MESSAGEID=? AND DESTINATION=?
    UPDATE_MARKED_MESSAGES = UPDATE JMS_MESSAGES SET
TXID=?, TXOP=? WHERE TXOP=?
    UPDATE_MARKED_MESSAGES_WITH_TX = UPDATE
JMS_MESSAGES SET TXID=?, TXOP=? WHERE TXOP=? AND TXID=?
    DELETE_MARKED_MESSAGES_WITH_TX = DELETE FROM
JMS_MESSAGES WHERE TXID IN (SELECT TXID FROM
JMS_TRANSACTIONS) AND TXOP=?
    DELETE_TX = DELETE FROM JMS_TRANSACTIONS WHERE TXID
```

```
= ?
DELETE_MARKED_MESSAGES = DELETE FROM JMS_MESSAGES
WHERE TXID=? AND TXOP=?
DELETE_TEMPORARY_MESSAGES = DELETE FROM
JMS_MESSAGES WHERE TXOP='T'
DELETE_MESSAGE = DELETE FROM JMS_MESSAGES WHERE
MESSAGEID=? AND DESTINATION=?
CREATE_MESSAGE_TABLE = CREATE TABLE JMS_MESSAGES
(MESSAGEID INTEGER NOT NULL, DESTINATION VARCHAR(150) NOT
NULL, TXID INTEGER, TXOP CHAR(1), MESSAGEBLOB IMAGE)
CREATE_IDX_MESSAGE_TXOP_TXID = CREATE INDEX
JMS_MESSAGES_TXOP_TXID ON JMS_MESSAGES (TXOP, TXID)
CREATE_IDX_MESSAGE_DESTINATION = CREATE INDEX
JMS_MESSAGES_DESTINATION ON JMS_MESSAGES (DESTINATION)
CREATE_IDX_MESSAGE_MESSAGEID_DESTINATION = CREATE
UNIQUE CLUSTERED INDEX JMS_MESSAGES_IDX ON JMS_MESSAGES
(MESSAGEID, DESTINATION)
CREATE_TX_TABLE = CREATE TABLE JMS_TRANSACTIONS ( TXID
INTEGER, PRIMARY KEY (TXID) )
CREATE_TABLES_ON_STARTUP = TRUE
</attribute>
<!-- Uncomment to override the transaction timeout for recovery per
queue/subscription, in seconds -->
<!--attribute name="RecoveryTimeout">0</attribute-->
<!-- The number of blobs to load at once during message recovery -->
<attribute name="RecoverMessagesChunk">0</attribute>
</mbean>

</server>
```

我们只需要修 `jboss.jca:service=DataSourceBinding, name=DEV_MSSQLDS` 这行让 `name` 等于我们上面配置的数据源就可以了。这样我们就可以在数据库里看到我们的消息了。

## 三. JMX 原理和应用

### 1. 什么是 JMX

JMX (Java Management Extensions, 即 Java 管理扩展) 是一个为应用程序、设备、系统等植入管理功能的框架。JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议, 灵活的开发无缝集成的系统、网络和服务管理应用。

## 2. JMX 规范

JMX 是一份规范，SUN 依据这个规范在 JDK (1.3、1.4、5.0) 提供了 JMX 接口。而根据这个接口的实现则有很多种，比如 Weblogic 的 JMX 实现、MX4J、JBoss 的 JMX 实现。在 SUN 自己也实现了一份，不过在 JDK1.4 之前，这件 JMX 实现（一些 JAR 包）是可选的，你得去它的网站上下载。JDK5.0 则内嵌了进来，安装 JDK5.0 就可以开发基于 JMX 的代码了。

## 3. 基于 JBoss 来写 MBean

JMX 的实现不独 SUN 一家，JBoss 也有自己的 JMX 实现。如果你使用 JBoss 来做 WEB 服务器，那么基于 JBoss 的实现来写 MBean，是一个不错的选择，我们用一个简单的例子来说明一下。

### 3.1 HelloWorld 实例

#### 1、准备工作

JBoss 实现了 JMX 规范，这个实例是基于 JBoss 来实现的。请先去下载一个 JBoss，我是 jboss-4.2.2.GA，下载地址：  
<http://www.jboss.com/downloads/index#as>。

### 3.2 程序代码

假设我们有一个叫 message 的属性需要经常进行改动配置的，那么我们就把它写成一个 MBean。

#### 1、HelloWorldServiceMBean 接口

在写 MBean 之前，我们先需要写一个 MBean 接口，接口里的方法都是属性的 set/get 方法。这个接口必须继承接口 ServiceMBean。

```
import org.jboss.system.ServiceMBean;
public interface HelloWorldServiceMBean extends ServiceMBean {
    String getMessage();
    void setMessage(String message);
}
```

#### 2、HelloWorldService 实现类

然后写出 HelloWorldServiceMBean 接口的实现类 HelloWorldService，这个实现类还必须继承 ServiceMBeanSupport 类。

```
import org.jboss.system.ServiceMBeanSupport;

public class HelloWorldService extends ServiceMBeanSupport implements
HelloWorldServiceMBean {

    private String message;

    public String getMessage() {

        System.out.println("getMessage()" + message);

        return message;

    }

    public void setMessage(String message) {

        System.out.println("setMessage(" + message + ")");

        this.message = message;

    }

}
```

### 3.3 配置文件 jboss-service.xml

```
<? xml version="1.0" encoding="UTF-8"? >
<server>
    <mbean code="example.mbean.HelloWorldService"
name="www.yangkun.com.cn:service=Hello World">
        <attribute name="Message">Hello World</attribute>
    </mbean>
</server>
```

说明：

code 项指向 MBean 的实现类 HelloWorldService, name 项是一个名称, 格式一般是: [说明性文字]:service=[类名] attribute 是为属性设置初始值, 这样当 JBOSS 一加载 HelloWorldService 类时, message 属性就有了一个初始值 Hello World。注意 Message 的第一个字母必须是大写。

### 3.4 将实例部署到 JBOSS

在 jboss-4.2.2.GA \server\default\deploy 目录下创建一个 hello.sar 目录, 然后创建如下目录文件结构:

```
hello.sar
|----example
|          |----mbean
|          |----HelloWorldService.class
|          |----HelloWorldServiceMBean.class
|----META-INF
|          |----jboss-service.xml
```

### 3.5 MBean 的效果

打开网址：<http://127.0.0.1:8080/jmx-console/>，出现下图。



然后单击“service=HelloWorld”项打开详细页面如下：





## JMX MBean View

MBean Name:      **Domain Name:** www.yangkun.com.cn  
                         **service:**            HelloWorld  
MBean Java Class: com.jmx.mbean.HelloWorldService

[Back to Agent View](#)    [Refresh MBean View](#)

### MBean description:

Management Bean.

### List of MBean attributes:

Name	Type	Access	Value	Description
StateString	java.lang.String	R	Started	MBean Attribute.
State	int	R	3	MBean Attribute.
Message	java.lang.String	RW	<input type="text" value="Hello World !!"/>	MBean Attribute.
Name	java.lang.String	R	HelloWorldService	MBean Attribute.

单击“Apply Changes”应用修改，得到如下效果：

## s 四. EJB3.0 使用说明

### 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)。

#### 1.1 会话 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。

### 1.2 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 看作是用于存放数据的 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）。

### 1.3 消息驱动Bean(MDB):

消息驱动 Bean 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。

## 2. 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。

### 2.1 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。

## 2.2 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。

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

步骤如下：

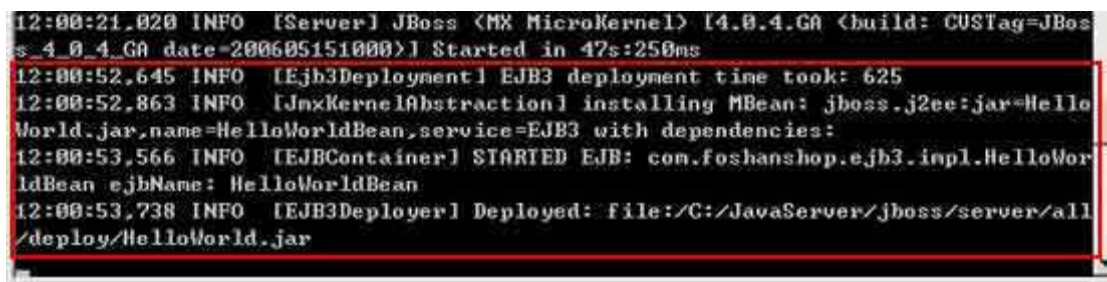
第一步：要定义一个会话 Bean，首先需要定义一个包含他所有业务方法的接口。这个接口不需要任何注释，就像普通的 java 接口那样定义。调用 EJB 的客户端通过使用这个接口引用从 EJB 容器得到的会话 Bean 对象 stub。接口的定义如下：

```
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, 文件夹下找到, 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。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: CUSTag=JBoss_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=HelloWorld.jar,name=HelloWorldBean,service=EJB3 with dependencies:
12:00:53.566 INFO [EJBContainer] STARTED EJB: com.foshanshop.ejb3.impl.HelloWorldBean 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.jb
```

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：

```
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;
}
}
```

```
package junit.debug;
```

```
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("Hello WorldBean/remote");
    }
}
```



```
static class Sayrun implements Runnable{
    public void run(){
        System.out.println(" 你好!世界,这是我的第一个 EJB3
哦."+helloworld.SayHello("西安人"));
    }
}

@Test
public void testSayHello() {
//    while(true){
//        Thread rh = new Thread(new Sayrun());
//        rh.start();
//    }
    System.out.println(" 你好!世界,这是我的第一个 EJB3
哦."+helloworld.SayHello("西安人"));
    //assertEquals(" 你好!世界,这是我的第一个 EJB3 哦.",
helloworld.SayHello("西安人"));
}
}
```

### 2.3 Stateless Session Bean 与 Stateful Session Bean 的区别

发这两种 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。

### 2.4 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。

### 3. 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。

#### 3.1 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.JMSEException;
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("no1", "北京和平里一号");
            producer.send(mapmsg);
            //发送 BytesMessage
            BytesMessage bmsg = session.createBytesMessage();
            bmsg.writeBytes("我是一个兵,来自老百姓".getBytes());
            producer.send(bmsg);
        } 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 MessageListenser 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

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  
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("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(content);
}
}
```

上面通过@MessageDriven 注释指明这是一个消息驱动 Bean，并使 @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。

### 3.2 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("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);
}
}
```

```
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.JMSEException;
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("no1", "北京和平里一号");
producer.send(mapmsg);
//发送BytesMessage
BytesMessage bmsg = session.createBytesMessage();
bmsg.writeBytes("我是一个兵,来自老百姓".getBytes());
producer.send(bmsg);
//发送StreamMessage
StreamMessage smsg = session.createStreamMessage();
smsg.writeString("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 Topic 消息的发送步骤和 Queue 消息相同。

## 4. 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。

### 4.1 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第四节“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。

#### 4.2 实体 Bean 发布前的准备工作

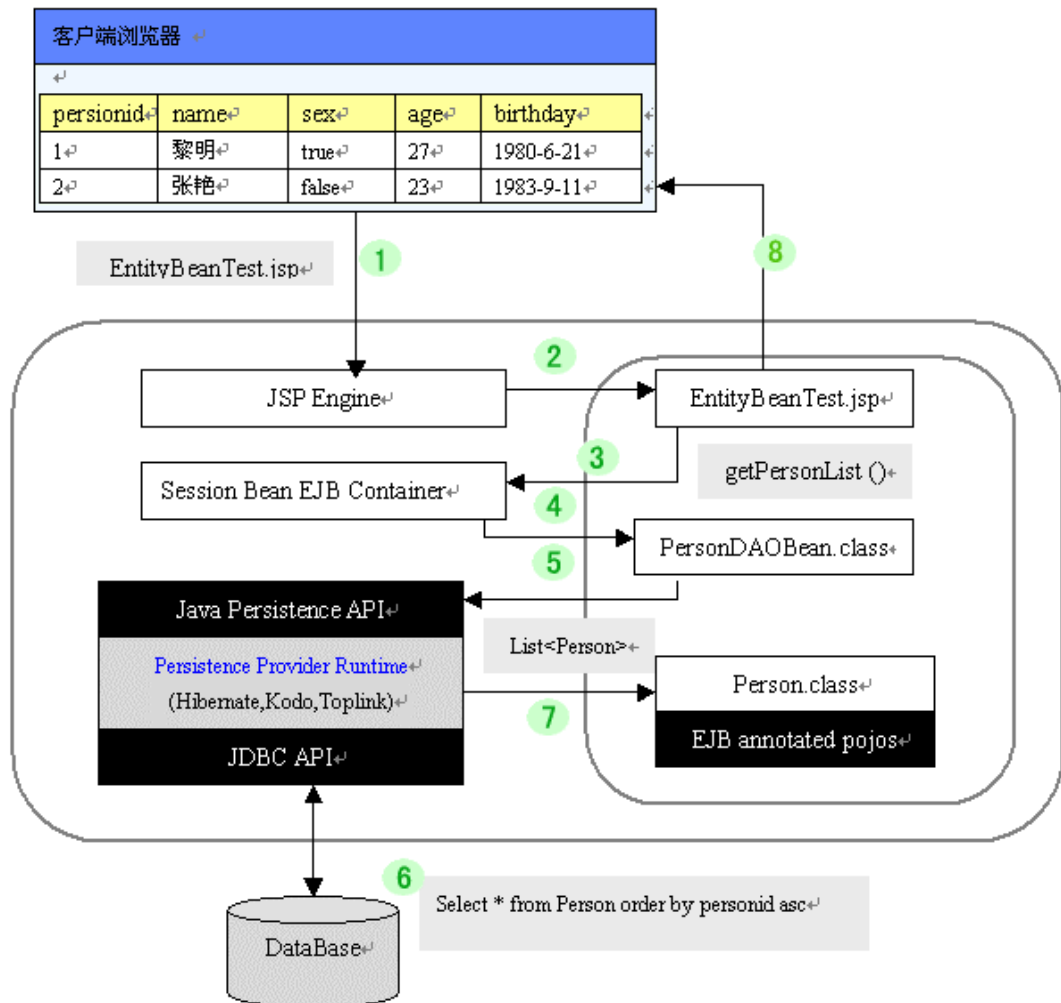
1 配置数据源并放置在 [jboss 安装目录]/server/default/deploy 目录, 把数据库驱动 Jar 包放置在 [Jboss 安装目录]\server\default\lib 目录下, 放置后需要重启 Jboss 服务器。如果数据源已经存在就不需要配置。

2 配置 `persistence.xml` 文件, 在文件中指定使用的源据源及各项参数。

3 把实体类和 `persistence.xml` 文件打成 Jar, `persistence.xml` 放在 jar 文件的 META-INF 目录

#### 4.3 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

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 IDENTIT 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.Pers
on,
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。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。通过 <http://localhost:8080/EJBTest/EntityBeanTest.jsp> 访问客户端。