

第十六章 开发 EJB 应用

第十六章 开发 EJB 应用	1
16.1 介绍	2
16.1.1 概述	2
16.1.2 Java EE 5 技术简介	5
16.1.3 Java EE打包和发布	7
16.1.3.1 JAR文件	7
16.1.3.2 WAR 文件	16
16.1.3.3 EAR文件	17
16.1.4 Java EE开发中的角色划分	19
16.1.5 JavaBean和EJB的区别	20
16.1.6 EJB和Spring能否共存	20
16.2 系统需求	20
16.3 准备工作	20
16.3.1 MyEclipse对EJB提供的开发支持	20
16.3.1 配置应用服务器	21
16.3.2 JBoss服务器目录结构简介	21
16.4 走进Java EE: JNDI	23
16.4.1 JNDI简介	23
16.4.2 如何查看 JBoss服务器的JNDI树	26
16.4.3 开发JNDI应用	28
16.4.3.1 简单JNDI开发	28
16.4.3.2 JNDI访问数据源	34
16.4.3.3 使用 JNDI 进行DNS, 邮件服务器, 主机信息查找	38
16.4.4 JBoss/Tomcat中的一个JNDI Bug问题解决	41
16.5 开发 Session Bean	41
16.5.1 Session Bean简介	41
16.5.2 开发无状态Session Bean	46
16.5.3 体验无状态Bean的混乱态	51
16.5.4 开发有状态 Session Bean	52
16.5.5 EJB发布描述符和JBoss JNDI 地址	56
16.5.5 EJB互访问和资源注入	58
16.5.6 拦截器	65
16.5.7 EJB 和 Web 服务	67
16.5.8 EJB最佳实践	69
16.6 开发实体Bean	70
16.5.1 使用反向工程生成 EJB 3 实体 Bean	70
16.5.2 调整生成的配置文件和实体类	81
16.5.3 编写并运行测试代码	81
16.6 消息驱动Bean	82
16.6.1 JMS简介	82

16.6.2 JMS编程模型	86
16.6.3 JMS点对点模式编程.....	87
16.6.4 JMS 发布订阅模式编程	96
16.6.5 MDB简介及MDB编程	99
16.7 可嵌入式的EJB引擎	103
16.8 小结.....	104
16.9 参考资料.....	106
16.10 术语表	107

16.1 介绍

注意：本章将会出现大量术语（英文缩写等），请您参考本章末尾的术语表。

16.1.1 概述

本章将会简要介绍 JNDI, EJB 及其开发。随着轻量级框架的流行, EJB 的开发相对不如以前那样受到大家的重视, 不过在某些场合, 以及大公司和大企业例如银行, 电信等机构中, 它的使用还是非常的广泛的。尤其在一些需要多台服务器集群的时候, 还是离不开的。这是因为 Spring 等轻量级框架不提供对于分布式服务的支持。那么什么是分布式服务呢? 我们来举一个简单的例子, 例如你承包了一片地, 需要找人来开发。第一种办法呢, 是你自己去开发, 来盖房子, 当然这种方式呢会有问题, 比如说你不是建筑方面的专家, 没有学习过相关的技术不会盖房子那怎么办呢? 这时候呢, 一般来说我们可以去委托给一些建筑队或者是建筑公司, 让他们来进行这些房子的设计和开发, 并且根据实际情况, 如果工程量比较大, 或者是进度要求比较快, 我们呢, 还可以联系多家建筑公司一块儿来进行开发。这些相当于我们是客户端, 而一个建筑公司呢是服务器, 虽然有多家建筑公司, 但是他们提供的服务是标准化的服务, 也就是都能够进行合乎标准的建筑服务, 这样呢他们就相当于所谓的服务器集群, 我们作为客户端需要的就是提出建筑的具体要求 (所谓的业务逻辑), 其他的功能由他们来提供。同时, 因为他们提供保证符合标准前提下, 又带有特色的服务, 所以呢, 可以方便的在工程的中间阶段根据情况更换服务公司, 这相当于更换服务器。

简单回顾一下EJB的历史。EJB 1.0 出现于 1997 年, 那时候中国人还没几个做 Java 的, 只可惜搞的太重量级了, 开发比较困难, 后来才有了轻量级的 Hibernate和Spring。在 Hibernate 之前还有好多其它的轻量级 ORM 框架, 不过那也是在 EJB 推广之后的事情了。有意思的发现 EJB 最早是 IBM 的发明, 参考: <http://zh.wikipedia.org/wiki/EJB>。EJB 最早于 1997 年由IBM提出, 旋即被太阳微系统采用并形成标准 (EJB 1.0 和EJB 1.1)。在 EJB 2.1 和以前的版本中, 每个EJB都由一个类和两个接口组成。EJB容器负责创建这个类的实例, 接口则供客户端调用。参考: <http://www.ibm.com/developerworks/cn/opensource/os-ag-renegade14/>, 毋庸置疑, 1997 年推出的 IBM® 原始 EJB 规范是 Java 技术领域最重要的开发成果之一。EJB 和包含 EJB 的 J2EE 应用服务器迅速地应用于企业开发。然而, 对 EJB 的批评之声也正如 J2EE 的采用一样快速涌至。在这些批评之中, 最主要的抱怨是 EJB 难于理解且开发起来繁琐乏味。

EJB 是一种服务器的组件体系结构, 它简化了用 java 开发企业级的分布式组件应用程

序的过程。通过 EJB，我们能写出可扩展的、健壮的和安全的应用程序，而不用自己去写复杂的分布式组件框架。EJB 用于快速开发服务器端应用程序，通过利用由业界提供的一些事先写好的基础结构，我们可以快速而轻松的利用 Java 构建服务器端组件。EJB 被设计为支持应用程序的可移植性和八这些特性是适用于任何厂商的企业中间件服务的。应用服务器一般是指这种提供 EJB, JMS, 安全, 事务等多种服务的服务器，而其中一种属于 EJB 容器；而 Servlet 容器则一般只是提供 JSP 和 Servlet 的功能，例如 Tomcat。图 16.1 列出了传统的 Java EE 应用的结构，包括客户机，Java EE 服务器和数据库服务器。当然，没有人规定必须将所有的技术都用上才算是一个 Java EE 应用，目前基于轻量级开发的 SSH 应用，其实也是一种基于 Web 容器的解决方案，离开了 Web 容器，Struts 是运行不了的。

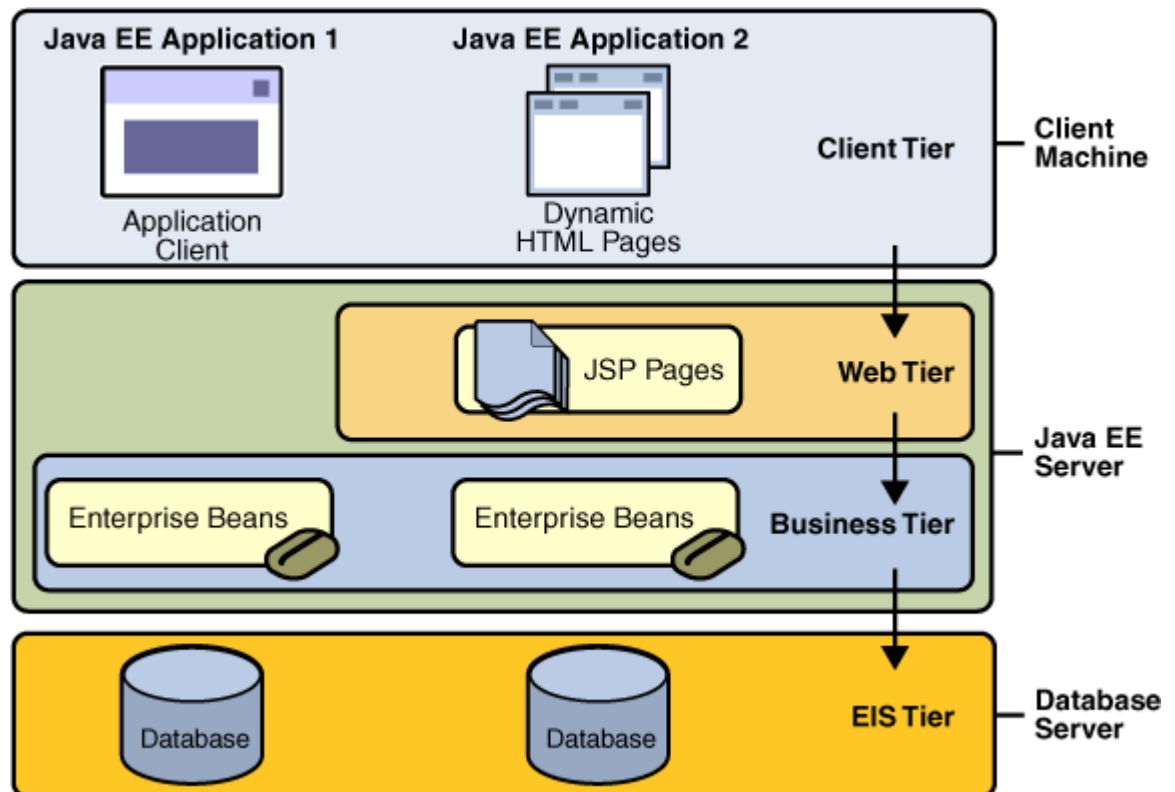


图 16.1 传统 Java EE 应用的结构

然而，实际上的应用服务器提供了多种技术，包括：JNDI, LDAP, JavaMail, JAF, JMS, JTA, Web 服务, JAXB, JAXP, JAXR, JCA, JAX-WS, SAAJ, RMI-IIOP, EJB, JSP, Servlet, JAAS, JSF, JSTL, Portal, SOA, JPA, 加密解密等等全套技术，而且一般会提供热发布，集群，管理控制台，性能监控，远程管理，各种各样的池（数据库连接池，线程池，对象池）等服务。就目前市场上的 Java EE 服务器来说，主流的仍然是国外厂商的商业服务器（即收费版本，一般都是单 CPU 多少万的卖，比 MyEclipse 这样的软件要贵的多，如果你的公司用了盗版的，那么千万要小心），这些服务器有 BEA WebLogic（WebLogic Express 只提供 Web 层的服务），IBM WebSphere Application Server, Oracle Application Server 等等。有人说开源的 JBoss 也不错，实际上 JBoss 提供付费版本，里面的组件和免费版大不一样，安装包也大多很多（商业版 JBoss 安装包大概 200 多 MB），而且最要命的是 JBoss 服务器的文档是收费的（开源版本没有带文档的）！不过目前 Sun 主导的开源 Java EE 服务器 GlassFish 呼声很高，据说性能比较好（全部使用 NIO 等技术实现）。Web 层的服务器，坦白说 Tomcat 是性能比较底下的一个，即使开源的 Resin 也比同样条

人员要求不高，国内的大企业，例如电信，银行，政府部门等（说实话，Java 不能用于实时操作的领域，这也是 Sun 的 JDK 下载里面说的很清楚的，所以一般都是提供外围接口服务的部分，例如服务网站，内部网等等），总之是有钱的大客户，大多数会采用这种方案。另一套是开源架构的，全部采用开源，不过问题也有，例如开源的 SSH 组合，往往一个地方配置出错，整个应用都会坏掉，甚至还有因为开源包之间互相冲突造成问题，也不支持集群，调优等功能，开发过程也比较缓慢。相对来说，JSF+JPA 的开发效率要稍高于 SSH。当然对于读者来说，到底用什么技术，是不能纯粹出于自己的喜好的，更多的是看单位的情况和要求。总而言之，本书的内容个个章节都是按照参考资料的方式提供的，读者可以根据项目需要来阅读。

由于 Java EE 领域的技术点是如此之多，因此如果要全面的介绍，那么光是技术点的内容就可以写一本千页的大书，例如我以前买过一本《J2EE 编程指南 1.3 版》。所以在本书中，我们不可能覆盖方方面面，本章，也主要集中于 EJB 3 的内容上。EJB 2.1 的内容，读者可以参考附录部分的 Mastering Enterprise JavaBeans Third Edition 这本书（中文出版名为《精通 EJB》），那本书是 EJB 2 的经典书籍，如果你看了那本书，就会发现仅仅是 EJB 就可以讲述那么多的内容。当然，对于我们开发人员来说，主要目的就是写例子，发布，运行即可。

16.1.2 Java EE 5 技术简介

接下来我们对 Java EE 技术进行简单的介绍。

为何推出 Java EE 5?

- Struts, Spring, Webwork, Hibernate 等等开源框架的冲击
- 传统的 EJB 2.1 及相关应用开发难度太大
- 开源 ORM 及 JBoss 等服务器的冲击
- 开源框架缺乏稳定性和商业支持
- 简化 ORM 开发人员负担（多种 ORM 框架互不兼容）

Java EE 5 有哪些新特性?

- 标注取代部署描述符
- 简化的 EJB 软件开发
- 使用依赖关系注入来访问资源
- Java 持久性 API 模型
- Web 服务

Java EE 5 大量采用标注简化开发，这种趋势已经影响到了很多开源框架例如 Struts 2（Struts 2 也支持标注方式的 Action 开发，并在开发完全无配置文件的版本）以及 Spring（Spring 2.5 大力增强了对标注方式的开发支持）。标注取代部署描述符，可以用来进行：

- 定义和使用 Web 服务
- 开发 EJB 软件应用程序
- 将 Java 技术类映射到 XML
- 将 Java 技术类映射到数据库
- 将方法映射到操作
- 指定外部依赖关系
- 指定部署信息，其中包括安全属性

标注格式：

标注使用 @ 字符来标记，例如下面的代码片段定义了一个无状态的会话 Bean：

```
import javax.ejb.*;
@Stateless
public class HelloWorldSessionBean implements mypackage.HelloWorldSessionLocal
{
}

```

简化了 EJB 软件开发。现在进行 EJB 的开发只需很少的类和接口。您不再需要 EJB 组件的 Home 接口和对象接口，因为现在容器负责公开必要的方法。您只需提供业务接口。您可以使用标注来声明 EJB 组件，并且通过容器来管理事务。

不再需要部署描述符。您可以在类中直接使用标注，为容器提供以前在部署描述符中定义的依赖关系和配置信息。如果没有任何特殊说明，容器将使用缺省规则来处理最常见的情况。

查找简单。您可以通过 `EJBContext` 直接在类中查找 JNDI 名称空间中的对象。

简化了对象关系映射。新的 Java 持久性 API 允许您使用 POJO 中的标注将 Java 对象映射到关系数据库，从而使对象关系映射变得更简单透明。

使用依赖关系注入来访问资源：

- 对象可以使用标注直接请求外部资源
- 使用 `@Resource` 标注或针对一些专用资源的 `@EJB` 和 `@WebServiceRef` 标注
- 可以注入以下资源：
 - `SessionContext` 对象
 - `DataSources` 对象
 - `EntityManager` 接口
 - 其他 Enterprise Beans
 - Web 服务
 - 消息队列和主题
 - 资源适配器的连接工厂

Java 持久性 API 模型

- 不仅用于 Java EE 服务器环境，也可用于 Java SE 和 Web 层
- 无缝集成/替换 Hibernate, TopLink 等 ORM 框架
- 使用标注开发(JDK 5 or later)
- 实体是 POJO
- 标准化的对象关系映射
- 命名查询
- 简单的打包规则
- 分离的实体
- `EntityManager` API

实体是 POJO。与使用容器管理持久性 (Container-Managed Persistence, CMP) 的 EJB 组件不同，使用新 API 的实体对象不再是组件，并且它们不再需要位于 EJB 模块中。

标准化的对象关系映射。新规范将对对象关系映射的处理方式进行标准化，从而使开发者不再需要了解特定于供应商的策略。Java 持久性 API 使用标注来指定对象关系映射信息，但它仍支持 XML 描述符。

命名查询。现在命名查询是用元数据表示的静态查询。查询可以是 Java 持久性 API 查询或本地查询。这样会使重用查询变得非常简单。

简单的打包规则。由于实体 Bean 是简单的 Java 技术类，因此几乎可以在 Java

EE 应用程序中的任意位置将其打包。例如，实体 Bean 可以是 EJB JAR、应用程序客户端 JAR、WEB-INF/lib、WEB-INF/classes 的一部分，甚至是企业应用程序归档 (Enterprise Application Archive, EAR) 文件中实用程序 JAR 的一部分。通过这些简单的打包规则，您不再需要创建 EAR 文件以使用来自 Web 应用程序或应用程序客户端的实体 Bean。

分离的实体。由于实体 Bean 是 POJO，因此可以对它们执行序列化，通过网络将其发送到其他地址空间，并在不识别持久性的环境中使用它们。这样，您就不再需要使用数据传输对象 (Data Transfer Object, DTO)。

EntityManager API。现在，应用程序编程人员可以使用标准 EntityManager API 来执行涉及实体的创建、读取、更新和删除 (Create Read Update Delete, CRUD) 操作。

Web 服务

- 使用标注显著改进和简化了 Web 服务支持
- JDK 6 直接内置/简化对 Web Service 的支持
- JAX-WS 2.0
 - JAX-WS 2.0 是 Java EE 5 平台中用于 Web 服务的新 API。
 - JAX-WS 2.0 保留了自然的 RPC 编程模型，同时在以下几个方面进行了改进：数据绑定、协议和传输的独立性、对 Web 服务的 REST 样式的支持以及易开发性。
- 异步 Web 服务
 - 在轮询模型中，发出调用。准备就绪后，请求结果。
 - 在回调模型中，注册处理程序。在响应到达后，立即向您发出通知。

关于 Web 服务的这些内容，我们已经在上一章做了探讨。

16.1.3 Java EE 打包和发布

刚刚接触到 Java EE 开发的读者，常常为 Java 中名目繁多的打包文件感到晕眩。常见的有 JAR, WAR, RAR, EAR，其实所有这些文件的格式都是 ZIP 格式的，可以使用 WinZIP 或者 WinRAR 等软件打开，当然也能用它们来创建。还可以用 JDK 自带的 JAR 工具或者开发工具例如 Eclipse 来创建 JAR。

16.1.3.1 JAR 文件

JAR 文件，Java ARchive，Java 档案文件。详细内容可以参考文末的资料 **JAR 文件揭密**。JAR 文件格式以流行的 ZIP 文件格式为基础，用于将许多个文件聚集为一个文件。我们可以自己制作 JAR 文件，将编译输出的类文件按照包结构压缩即可。与 ZIP 文件不同的是，JAR 文件不仅用于压缩和发布，而且还用于部署和封装库、组件和插件程序，并可被像编译器和 JVM 这样的工具直接使用。在 JAR 中包含特殊的文件，如 manifests 和部署描述符，用来指示工具如何处理特定的 JAR。图 16.4 列出了一个常见的 JAR 文件的目录结构。

注意：JAR 文件中的 *META-INF* 目录（大小写不能错）不是必须的，但是大多数时候还是加上为妙，因为笔者以前就发现在 Tomcat 下以 jar 文件的方式加入自己写的类的时候，没有加入此目录，结构 Tomcat 竟然不识别自己写的类。

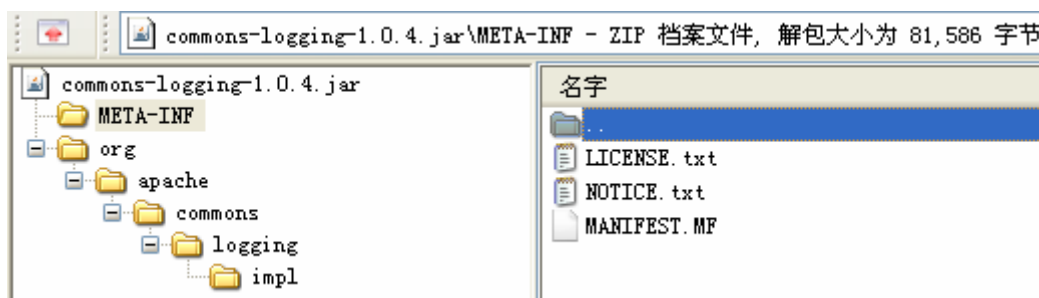


图 16.4 Commons Logging 包的 JAR 文件结构

一个 JAR 文件可以用于：

- 用于发布和使用类库
- 作为应用程序和扩展的构建单元
- 作为组件、applet 或者插件程序的部署单位
- 用于打包与组件相关联的辅助资源

JAR 文件格式提供了许多优势和功能，其中很多是传统的压缩格式如 ZIP 或者 TAR 所没有提供的。它们包括：

- **安全性。**可以对 JAR 文件内容加上数字化签名。这样，能够识别签名的工具就可以有选择地为您授予软件安全特权，这是其他文件做不到的，它还可以检测代码是否被篡改过。
- **减少下载时间。**如果一个 applet 捆绑到一个 JAR 文件中，那么浏览器就可以在一个 HTTP 事务中下载这个 applet 的类文件和相关的资源，而不是对每一个文件打开一个新连接。
- **压缩。**JAR 格式允许您压缩文件以提高存储效率。
- **传输平台扩展。**Java 扩展框架(Java Extensions Framework)提供了向 Java 核心平台添加功能的方法，这些扩展是用 JAR 文件打包的(Java 3D 和 JavaMail 就是由 Sun 开发的扩展例子)。
- **包密封。**存储在 JAR 文件中的包可以选择进行 *密封*，以增强版本一致性和安全性。密封一个包意味着包中的所有类都必须要在同一 JAR 文件中找到。
- **包版本控制。**一个 JAR 文件可以包含有关它所包含的文件的数据，如厂商和版本信息。
- **可移植性。**处理 JAR 文件的机制是 Java 平台核心 API 的标准部分 (java.util.jar 包下面)。

META-INF 目录

大多数 JAR 文件包含一个 META-INF 目录，它用于存储包和扩展的配置数据，如安全性和版本信息。Java 2 平台识别并解释 META-INF 目录中的下述文件和目录，以便配置应用程序、扩展和类装载器：

- **MANIFEST.MF。**这个 *manifest* 文件定义了与扩展和包相关的数据。
- **INDEX.LIST。**这个文件由 jar 工具的新选项 `-i` 生成，它包含在应用程序或者扩展中定义的包的位置信息。它是 JarIndex 实现的一部分，并由类装载器用于加速类装载过程。
- **xxx.SF。**这是 JAR 文件的签名文件。占位符 `xxx` 标识了签名者。
- **xxx.DSA。**与签名文件相关联的签名程序块文件，它存储了用于签名 JAR 文件的公共签名。

jar 工具

为了用 JAR 文件执行基本的任务，要使用作为 Java Development Kit 的一部分提供的 Java Archive Tool (jar 工具)。用 jar 命令调用 jar 工具。下表显示了一些常见的应用：

功能	命令
用一个单独的文件创建一个 JAR 文件	jar cf jar-file input-file...
用一个目录创建一个 JAR 文件	jar cf jar-file dir-name
创建一个未压缩的 JAR 文件	jar cf0 jar-file dir-name
更新一个 JAR 文件	jar uf jar-file input-file...
查看一个 JAR 文件的内容	jar tf jar-file
提取一个 JAR 文件的内容	jar xf jar-file
从一个 JAR 文件中提取特定的文件	jar xf jar-file archived-file...
运行一个打包为可执行 JAR 文件的应用程序	java -jar app.jar

可执行的 JAR

一个 *可执行的 jar* 文件是一个自包含的 Java 应用程序，它存储在特别配置的 JAR 文件中，可以由 JVM 直接执行它而无需事先提取文件或者设置类路径。要运行存储在非可执行的 JAR 中的应用程序，必须将它加入到您的类路径中，并用名字调用应用程序的主类。但是使用可执行的 JAR 文件，我们可以不用提取它或者知道主要入口点就可以运行一个应用程序。可执行 JAR 有助于方便发布和执行 Java 应用程序。

创建可执行 JAR

创建一个可执行 JAR 很容易。首先将所有应用程序代码放到一个目录中。假设应用程序中的主类是 `com.mycompany.myapp.Sample`。您要创建一个包含应用程序代码的 JAR 文件并标识出主类。为此，在某个位置(不是在应用程序目录中)创建一个名为 `manifest` 的文件，并在其中加入以下一行：

```
Main-Class: com.mycompany.myapp.Sample
```

然后，像这样创建 JAR 文件：

```
jar cmf manifest ExecutableJar.jar application-dir
```

所要做的就是这些了 -- 现在可以用 `java -jar` 执行这个 JAR 文件 `ExecutableJar.jar`。

一个可执行的 JAR 必须通过 `manifest` 文件的头引用它所需要的所有其他从属 JAR。如果使用了 `-jar` 选项，那么环境变量 `CLASSPATH` 和在命令行中指定的所有类路径都被 JVM 所忽略。

启动可执行 JAR

既然我们已经将自己的应用程序打包到了一个名为 `ExecutableJar.jar` 的可执行 JAR 中了，那么我们就可以用下面的命令直接从文件启动这个应用程序：

```
java -jar ExecutableJar.jar
```

首先需要声明的是，并不是所有类型的 Java 应用程序都能打包到单独的一个 JAR 文件中去。必须符合一定的条件，一个 Java 应用程序的所有文件才能打包到一个文件中去。另外，JAR 文件有两种，一种是有清单(`manifest`)文件的，一种是没有清单文件的(这种主要用来存放 Java 库文件)。清单文件有许多作用，我这里只讨论那种可以使用 Java 来运行 JAR 文件的清单文件。就是使用命令

```
java -jar MyJarFile.jar
```

可以直接运行的 JAR 文件。这种 JAR 文件的清单文件的内容如下所示:

```
Manifest-Version: 1.0
Main-Class: MainClassFile
Created-By: Your Company
```

。例如，假定你的程序里面的主程序是 `com.abc.MainFrame`，那么你的这个清单文件的内容是:

```
Manifest-Version: 1.0
Main-Class: com.abc.MainFrame
Created-By: Abc Company
```

。然后我想讨论一下打包到单独 JAR 文件的应用程序的编写需要注意的问题。

程序的资源文件(如*.gif, *.jpg, *.jpeg, *.properties 等等)不能使用物理路径，只能使用相对于当前类文件的相对资源路径，只有这样才能保证打包以后程序才能找到这些资源文件。那么，这些文件在打开的时候都要使用下列语句:

```
URL url = getClass().getResource(String name);
或者
InputStream in = getClass().getResourceAsStream(String name);
```

。例如:

```
javax.swing.ImageIcon icon = new javax.swing.ImageIcon("1.gif");
```

那么打包后的程序将不能找到这个图片，因为它引用了一个物理路径的图片，它应该改成这样:

```
javax.swing.ImageIcon icon = new
javax.swing.ImageIcon(getClass().getResource("1.gif"));
```

。其它的资源文件的使用也与此相似。不过属性文件(*.properties)的使用，Java 会自动会加载 JAR 文件中的属性文件，例如:

```
ResourceBundle.getBundle("com.abc.test");
```

那么，这个文件只要放在 JAR 文件目录结构下的 `com/abc/test.properties`，Java 就能自动找到这个文件。

接着我想讨论一下使用 JAR 工具如何来创建 JAR 文件，这个工具可以在 `<JDK_install_HOME>/bin` 下找到，文件名在 Win32 版下为 `jar.exe`，Unix 下为 `jar`。在命令行窗口下输入 `jar` 命令，将看到下列输出:

```
用法: jar {ctxu}[vfm0M] [jar-文件] [manifest-文件] [-C 目录] 文件名 ...
```

选项:

- c 创建新的归档
- t 列出归档内容的列表
- x 展开归档中的命名的（或所有的）文件
- u 更新已存在的归档
- v 生成详细输出到标准输出上
- f 指定归档文件名
- m 包含来自指定的清单（manifest）文件的清单（manifest）信息
- 0 只存储方式；未用 ZIP 压缩格式

- M 不产生所有项的清单 (manifest) 文件
- i 为指定的 jar 文件产生索引信息
- C 改变到指定的目录, 并且包含下列文件:

如果一个文件名是一个目录, 它将被递归处理。

清单 (manifest) 文件名和归档文件名都需要被指定, 按 'm' 和 'f' 标志指定的相同顺

示例 1: 将两个 class 文件归档到一个名为 'classes.jar' 的归档文件中:

```
jar cvf classes.jar Foo.class Bar.class
```

示例 2: 用一个存在的清单 (manifest) 文件 'mymanifest' 将 foo/ 目录下的所有文件归档到一个名为 'classes.jar' 的归档文件中:

```
jar cvfm classes.jar mymanifest -C foo/ .
```

假设你想将上面的 com.abc.MainFrame 为主程序的一个应用程序打包为一个单独的 JAR 文件, 那么首先在程序的根目录下建立内容如前所述的清单文件, 文件名为 MANIFEST.MF, 然后在命令行窗口下, 转变当前目录为程序根目录, 输入下列命令:

```
jar cvfm abcApp.jar MANIFEST.MF *.*
```

, 那么, 会使用现有的清单文件创建一个名为 abcApp.jar 的文件, 并将目录下的所有文件(包括子目录)加入此 JAR 文件中。

而如果希望使用非 JAR 工具建立 JAR 文件, 建议使用 WinRAR。首先还是需要建立一个清单文件(名字必须为 MANIFEST.MF), 向此文件中写入同使用 JAR 工具时一样的内容, 接着需要建立一个 META-INF 子目录, 然后将 MANIFEST.MF 放入此目录, 接着就可以保持目录结构将所有这些内容使用 WinRAR 压缩为 ZIP 格式的文件, 文件名取为 abcApp.jar 即可。如下所示, 目录结构为:

```
+
|
+-----/META-INF/MANIFEST.MF
|
+----com/abc/MainFrame.class
```

。选中程序根目录下的各个子目录及根目录下的文件, 点击鼠标右键, 选择"添加到压缩包..."命令, 然后指定文件名为 abcApp.jar, 格式为 ZIP, 最后点击"确定"即可完成。

一个更新 JAR 文件内容的批处理命令:

```
REM 更新 JAR 文件中的 classes 目录(做为根目录更新)
```

```
jar ufM tree.jar -C .\classes\ .
```

打包 classes 目录下文件为 jar 的命令(做为根目录创建):

```
@ECHO Compress classes to a jar file
jar cfm lib/rmc.jar MANIFEST.MF -C classes .
```

打包 WAR 文件:

```
jar cfM ROOT.war file1.jsp WEB-INF/ file2.jsp dir3/
```

将当前目录打包为一个 ZIP 文件:

```
jar cfM javarmc1_0beta.zip .
```

由于作者的表达力和能力所限, 因此这篇讨论的内容有一些不尽如人意的地方。因此您可以参考 Sun Microsystems 的 Java 教程里面的 JAR 部分作为补充(英文版), 地址: <http://java.sun.com/docs/books/tutorial/jar/>。

最后, 我们稍微介绍下如何在 Eclipse 中创建 JAR 文件。首先当然是打开我们的项目,

例如 *HelloWorld*，接着选择菜单 **File > Export**（导出），接着即可打开导出向导对话框，如图 16.5 所示。然后点击 **Next** 按钮进入第二步。同时在这一页，读者也可以看到 MyEclipse

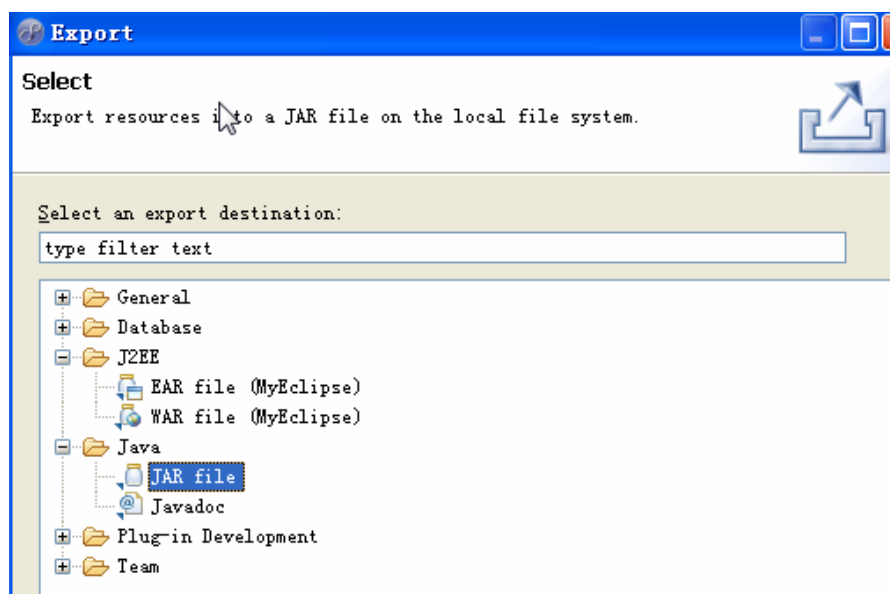


图 16.5 打包为 JAR 文件向导第一步：选择类型

的对 EAR 和 WAR 文件创建的支持，位于 J2EE 这个目录下。

接着在图 16.6 中选择要打包的内容，以及生成的 jar 文件的存放位置。一般发布的时候，不要打包根目录下的 `.classpath`，`.project` 这些文件，也不要选中 **Export java source files and resources**，否则源代码将会被打包，而这是源码放在 jar 里没有用处的，如果是外包项目的话，还会因此泄露源程序。点击 **Browse** 按钮可以选择 JAR 文件的保存位置。确认后，点击 **Next** 按钮进入下一页的设置。

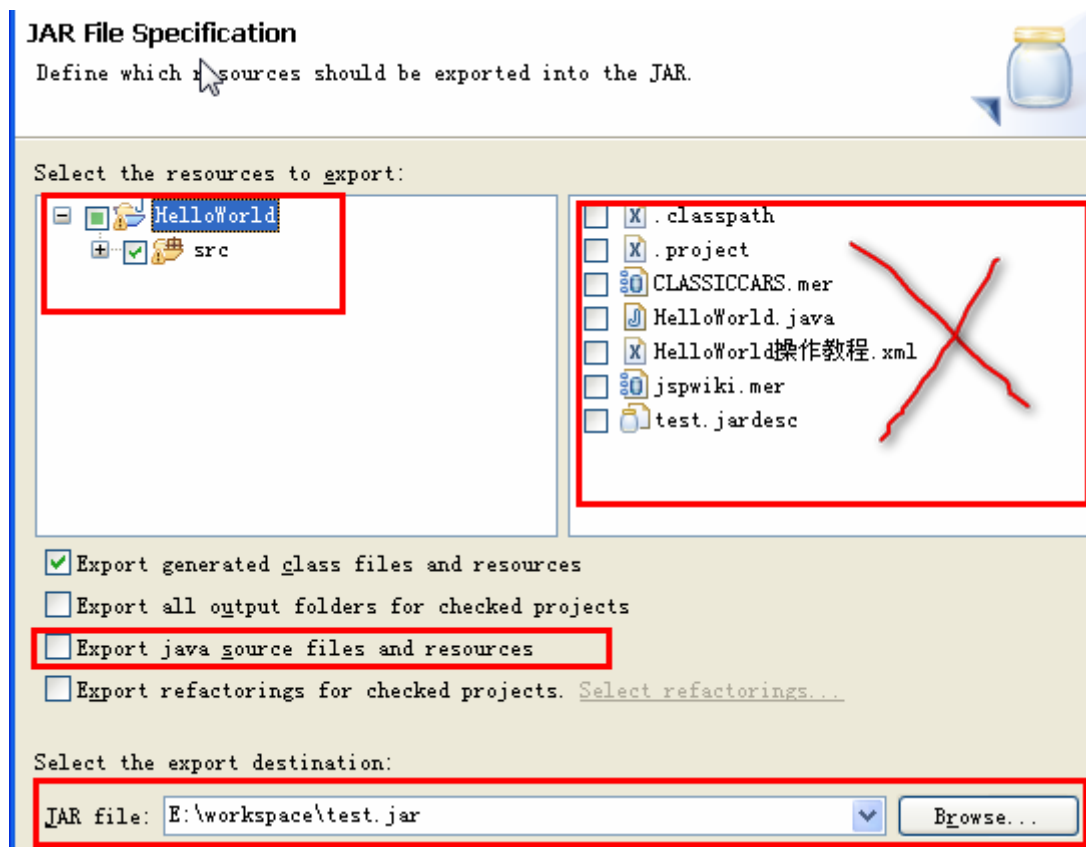


图 16.6 打包 jar 第二步：选择打包内容和 JAR 存放路径

接着我们看到的是图 16.7 所示的打包 jar 的第三步，设置一些 JAR 导出的选项。在这一步可以选中是否要把这些创建 jar 的相关设置信息保存进一个描述文件中，一般最好保存一下，这样以后重复生成 jar 文件时就不需要反复的进行选中操作了，这一选项的名称是 **Save the description of this JAR in the workspace**。另外可以设置的就是是否导出编译有错误或者警告的那些类(Export class files with compiler errors/warnings)。点击 **Browse** 按钮可以选择 JAR 描述信息文件的保存位置，如图中框起来的内容所示。确认无误后，点击 **Next** 按钮进入最后一页的设置。

如图 16.8 所示的是打包 jar 的最后一步，导出 jar 文件的清单文件 (manifest) 设置。在这一步，可以选择生成清单文件 (Generate the manifest file)，同时可以选择是否要将其保存在工作区中 (Save the manifest in the workspace)，以及重用并将此文件保存在工作区中 (Reuse and save the manifest in the workspace)。也可以选择直接使用工作区中已经建好的清单文件 (Use existing manifest from workspace)。另外还可以选择是否包密封 (Seal contents)，密封 JAR 文件中的一个包意味着在这个包中定义的所有类都必须在一个 JAR 文件中找到。这使包的作者可以增强打包类之间的版本一致性。密封还提供了防止代码篡改的手段。主类设置决定了此 jar 是否是可执行的 (Main class)，再次我们选中了可执行的主类 HelloWorld。一切准备完毕后，点击 **Finish** 按钮即可完成向导并创建 JAR 文件。

最后我们得到的是一个 jar 文件，双击此 jar 文件可以执行 (如果电脑上安装了 JRE 的话)，当然看不到任何输出，这是因为 Windows 下面默认是用 javaw.exe 来执行 jar 的，其命令为：

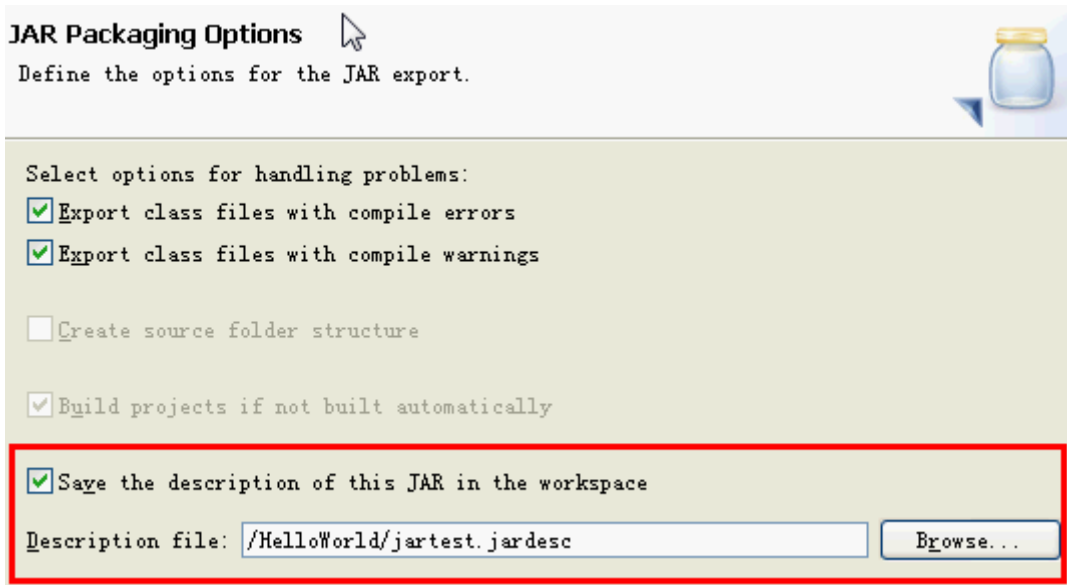


图 16.7 打包 jar 第三步：保存 jar 的创建文件（.jardesc）

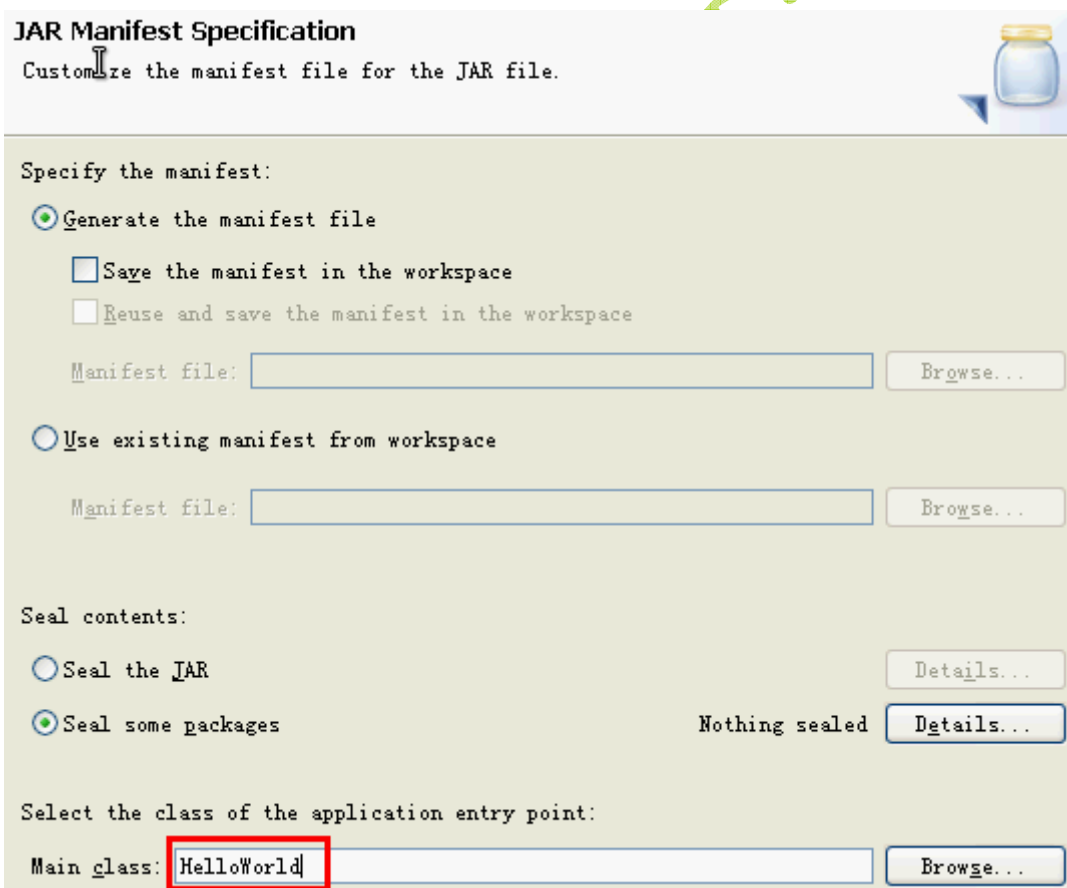


图 16.8 打包 jar 第四步：导出 jar 文件的清单文件设置

```
javaw -jar test.jar
```

。为了能看到输出，我们改用下面命令即可：

```
java -jar test.jar
```

。那么得到如下的输出：

你好 世界!

。此 jar 的文件结构如图 16.9 所示，其中根目录下放置着所有的类文件，而 META-INF/MANIFEST.MF 的文件内容就是我们所说的清单文件了，有一个 Main-Class 的属性，正是它指明了这个 jar 在是用上述命令时会自动调用的类。另外我们也可以用下面的方法来执行：

```
java -classpath test.jar HelloWorld
```

。这时候得到的输出内容是相同的，上面的命令将 test.jar 作为类路径的一部分加入 java 命令执行时候的虚拟机中。



图 16.9 test.jar 文件内容

由于 EJB 的默认打包方式也是 jar，那么此时其文件内容如图 16.10 所示（需要指出的是目前 Java EE 5 中已经不需要 META-INF 文件夹下的配置文件了）。要发布 EJB，简单的就是将 myejb.jar 文件放置到指定的目录，就可以了，或者是用散包的方式，将一个目录命名为 myejb.jar/，下面放入文件内容，也可以发布。当然也有的服务器必须用服务器提供的管理平台来手工发布，例如 WebLogic 服务器。

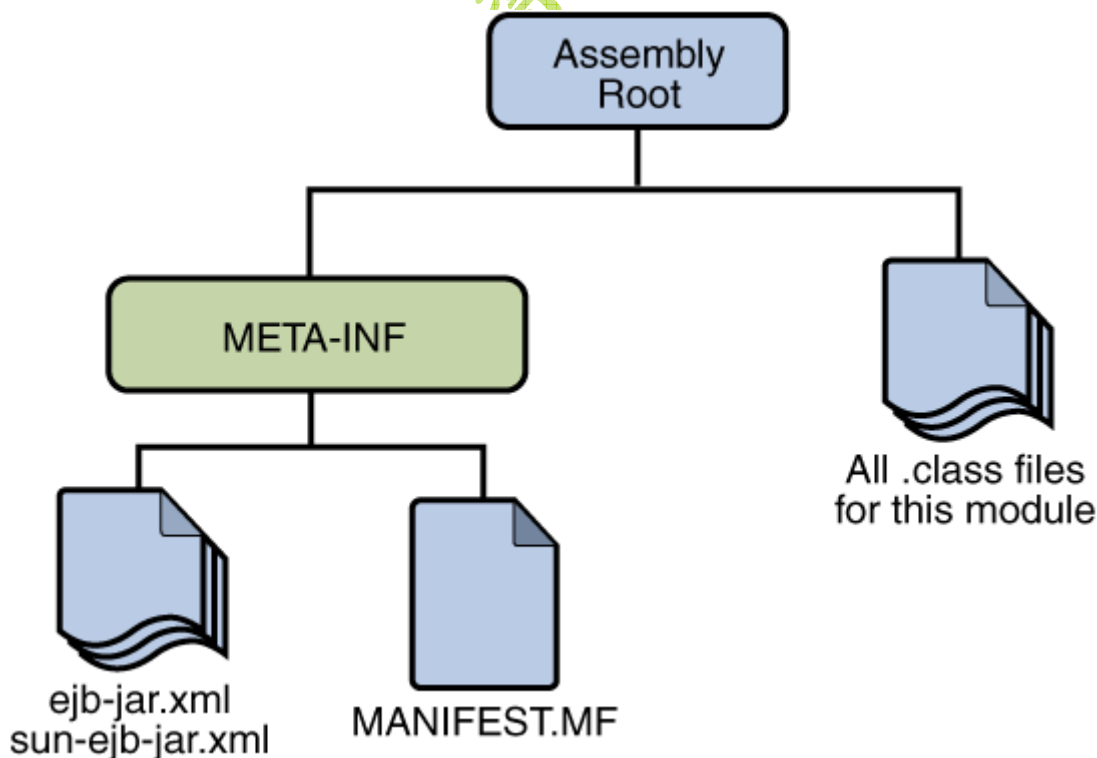


图 16.10 EJB 的 JAR 文件格式

另外，EJB 的客户端文件，也是 jar 格式的。

16.1.3.2 WAR 文件

WAR，就是Web ARchive的缩写（Web档案）。在 8.2 Web 项目和术语一节，我们已经简述了Web项目的结构，而WAR就是将所有的文件按照规定的目录结构打包成ZIP文件后所得到的压缩包，其结构如图 16.11 所示。每个WAR文件包含一个Web模块。例如我们可以随时将Tomcat下的webapps文件夹下的任意一个文件夹压缩成WAR格式，需要注意的是，不要在WAR里面加入一层目录，例如图 16.12 显示了两个将Tomcat的ROOT应用打包后所形成的WAR文件，左侧的是正确的打包方式，而右侧的则是错误的。有的服务器只支持WAR方式的Web应用。一般来说访问的应用路径和war文件的名字相对应，例如a.war对应访问地址为 <http://localhost:8080/a/>。

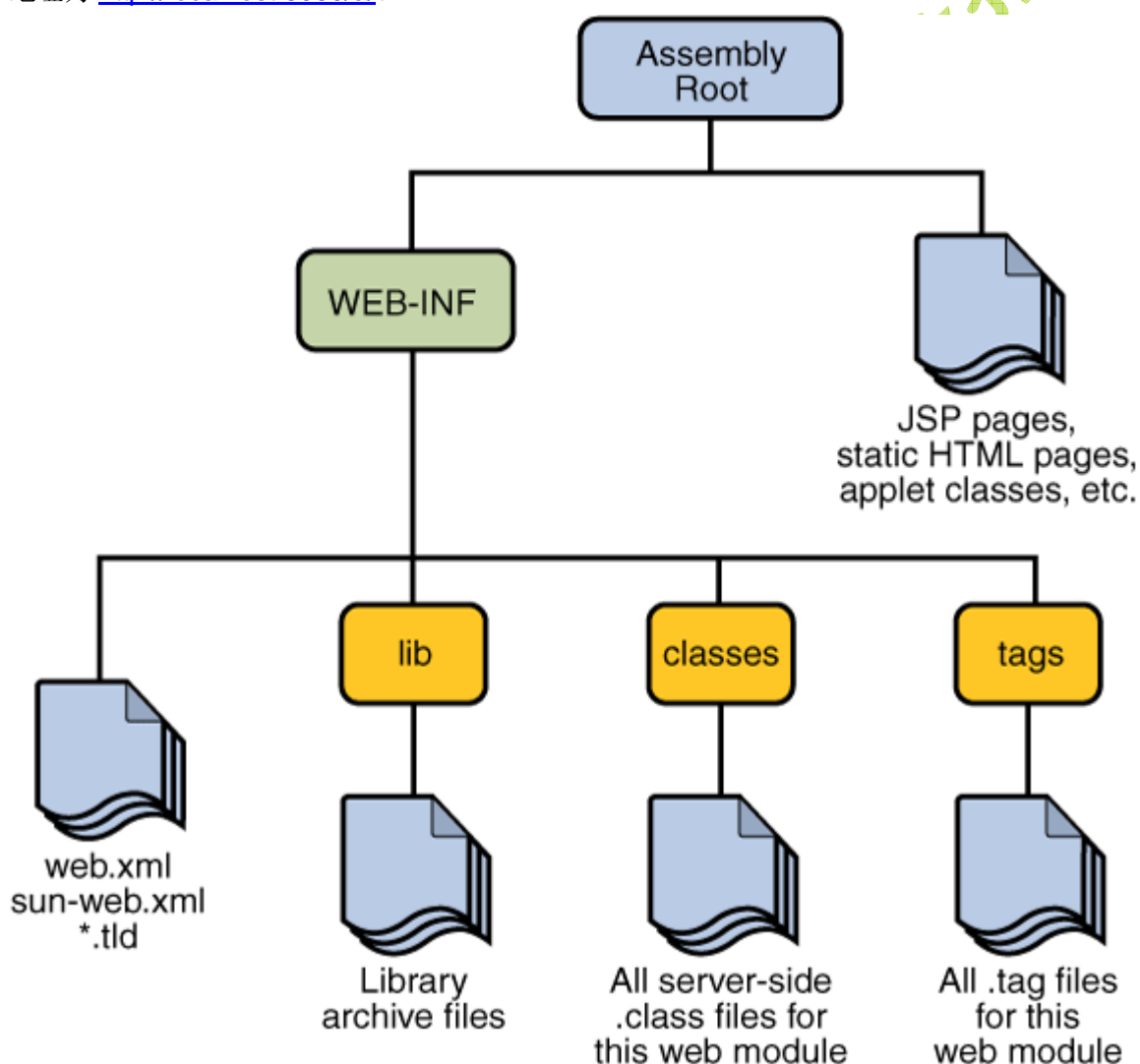


图 16.11 WAR 文件格式

关于 WAR 文件的制作，绝大多数开发工具都支持（大名鼎鼎的 Eclipse 无插件版本除外，Eclipse 3.3 的 Java EE 版本里面支持），最简便的当属是用 JDK 的 jar 命令或者 WinZIP，WinRAR 等软件直接添加目录内容为文件，例如下面的是将 Tomcat 的 webapps 下面的 ROOT 目录打包为 ROOT.war 时所用的命令（执行命令的目录在 TOMCAT 安装目录

(webapps 下):

```
jar cvf ROOT.war -C ROOT .
```

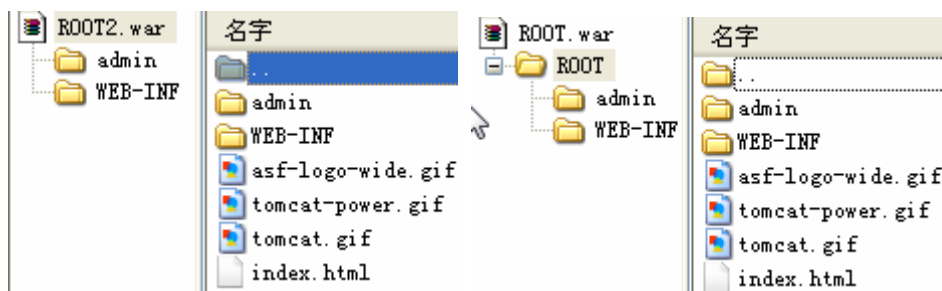


图 16.12 打包的 WAR 文件（左侧为正确格式）

至于 MyEclipse，则提供了导出 WAR 的向导，非常简单的内容，在 16.1.3.1 一节，我们执行命令 **File > Export** 时，已经可以在图 16.5 中看到此向导，点击 **Next** 后进入如图 16.13 所示的设置界面非常简单，就需要设置一个 WAR 文件名字。在这里我们打包的是 *HelloJSP* 这个小项目，点击 **Finish** 按钮后即可完成打包过程。

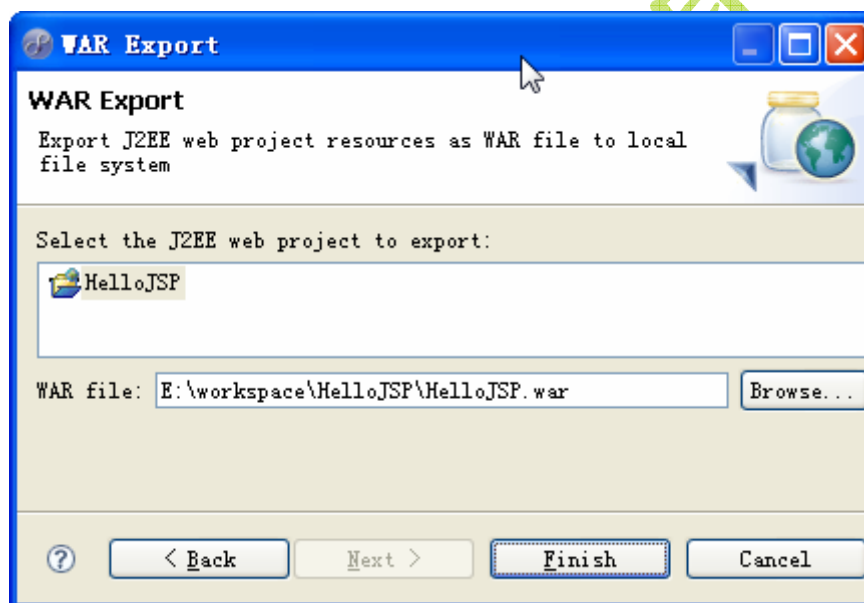


图 16.13 MyEclipse 中的打包 WAR 文件向导

同样的，WAR 文件是 Java EE 规范所固定的文件发布格式。支持自动发布的服务器需要将其放入指定目录即可，而手工的则需要通过控制台等方式进行发布。散包发布，即按照目录结构放好，则一般是开发的时候所采用的发布方式。

16.1.3.3 EAR 文件

一个 Java EE 应用通常以 Enterprise Archive (EAR) 文件的格式发布，内容是一个标准的 Java Archive (JAR) 文件，后缀为 .ear，其实就是 ZIP 格式。每个 EAR 里面又可以包含其它的一个或者多个独立模块例如 JAR, WAR, RAR 等等，构成一个完整的企业应用模块。每个 EAR 文件（参考图 16.14），都包括了 Java EE 模块以及发布描述符。发布描述符是 XML 文件，描述了应用的设置，或者单个模块或组件的设置。由于发布描述符是用声明方

式编写的，因此它可以在不需要修改源代码的方式进行修改。运行的时候，Java EE 服务器阅读发布描述符并进行必要的配置和操作。

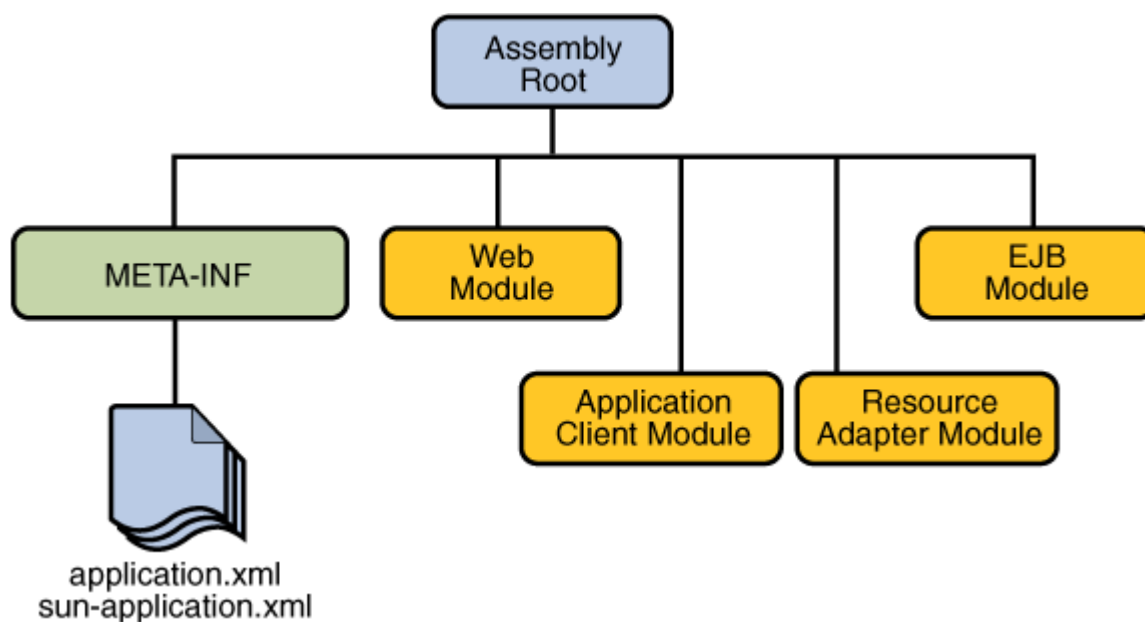


图 16.14 EAR 文件结构

发布描述符有两种: Java EE 标准的和运行时描述符。Java EE 发布描述符通过 Java EE 规范来规定，并且可以在任何 Java EE 兼容的服务器实现上进行设置。运行时描述符一般成为厂商特定描述符，这个是各个公司互不兼容的，甚至同一个公司的不同版本的之间也是互不通用的，它配置了运行的时候服务器的特定配置信息，是不可迁移的，需要开发人员根据不同的服务器版本进行配置。例如上图中的 sun-开头的文件，都是 Sun 的服务器 Sun Java System Application Server Platform Edition 9 里面的特定描述文件，而 JBoss 的文件名有和它不一样，内容也不同。

一个 EAR 包的内容可能如下所示：

MyApp.EAR

/myweb.war

/myejb.jar

/lib/mysql.jar

/myconnector.rar

，而其中的每个子文件又可以根据规范有自己的目录结构，层层嵌套。不过，一个 EAR 里面不能嵌套另一个 EAR。

有四种 Java EE 模块：

■ EJB 模块，包含了企业 bean 的类文件和 EJB 发布描述符。通常用 JAR 格式打包，文件后缀为 .jar。

■ Web 模块，包含了 servlet 类文件，JSP 文件以及附带的用户类文件，图片和静态 HTML 页面，以及 web 应用的发布描述符。其后缀一般是 .war (Web ARchive)。

■ 应用程序客户端模块，包括了类文件和应用程序客户端发布描述符。通常以 .jar 后缀打包。不过实用中这种模块很少见到，因为它其实是单独的访问 EJB 的客户端 Java 类。

■ 资源适配器 (Resource Adapter) 模块，包括了 Java 接口，类和本地类库，以及文档，和资源适配器发布文档。这些内容完整的实现了 Connector Architecture (连接器)规范，通常用于和老系统中的模块进行集成，将遗留系统整合进 Java EE 服务器中。打包内容也是

JAR 格式，文件后缀是.rar (resource adapter archive, 资源适配器档案)。注意这个 RAR 和 WinRAR 的不是同一种格式，千万不可混用。图 16.15 列出了 RAR 文件的格式。

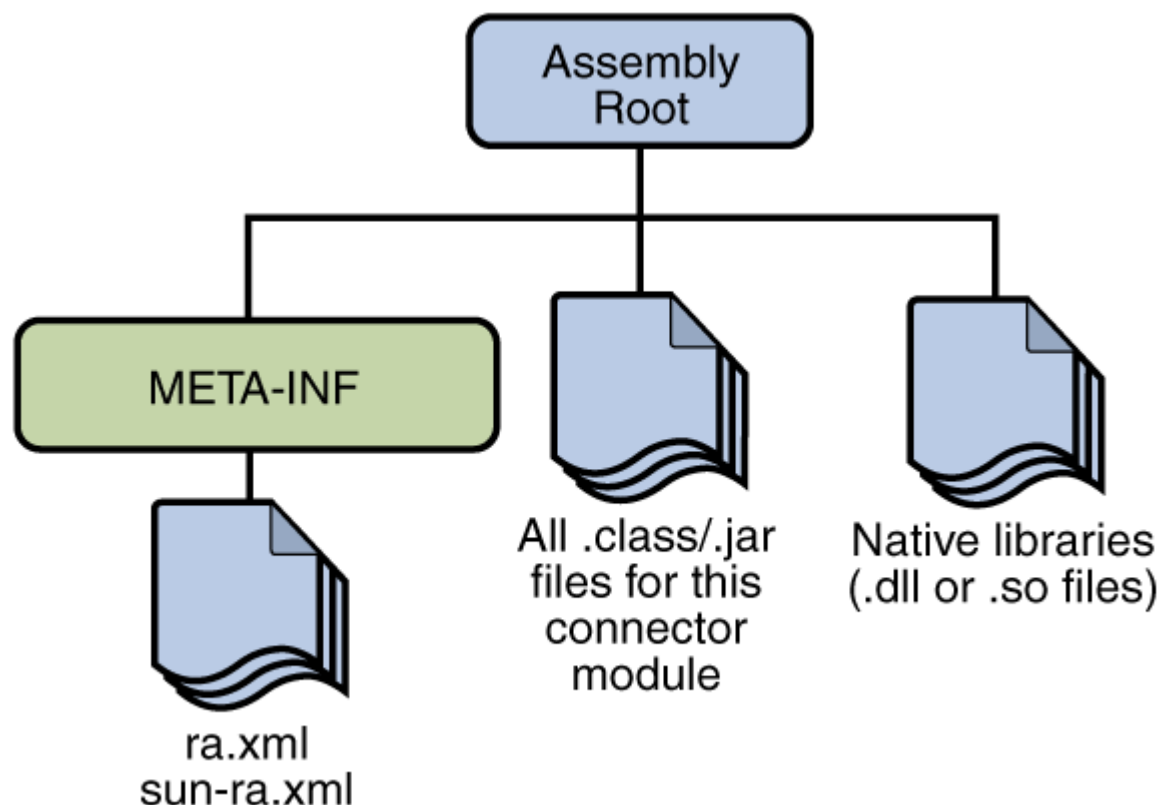


图 16.15 资源适配器文件格式

16.1.4 Java EE 开发中的角色划分

按照 Java EE 的规范，整个 Java EE 的开发分成了好几种角色，不过在实际开发中很可能好多角色是重复的。

Java EE Product Provider

Java EE 产品提供者一般都是商业性的大公司或者是像 Apache 这样的开源机构，例如 BEA, IBM, 金蝶, RedHat 等等，他们开发了服务器软件，一般商业服务器都以高价出售，国外的公司多从事这方面的工作。这些服务器软件都需要兼容 Java EE 规范，这样不管是哪个公司的产品，只要开发人员按照规范编写了程序，都可以运行。举个最简单的例子，为什么同样一个 JSP 页面放到不同的服务器上都能运行并得到同样的输出呢？这是因为他们符合了 Servlet 规范，所以基于 Spring 的开发并非不需要 Java EE 的容器技术，只不过只是用到了 Web 层而已。

Tool Provider

工具提供者提供开发工具，可以是公司或者个人，也就是像 MyEclipse, JBuilder 这样的软件，可以被组件开发人员或组装人员和发布人员使用。其实大部分服务器自身都带有一些这样的工具，例如可以发布或者删除应用。

Application Component Provider

应用程序组件提供者是一些公司或者个人，它们创建可以用于 Java EE 应用的组件，包括 web 组件, EJB, Applets 或者应用程序客户端。

下面还有很多内容，按照理论是可以划分成很多的。我们还是根据实际情况来做个介绍

吧，例如我们购买了 BEA 的 WebLogic 服务器，那么 BEA 就是 Java EE 产品提供者；如果你下载了 Tomcat，那么 Apache 也是 Java EE 产品提供者。之后你开发了个 JSP 的应用，那么你就是应用程序组件提供者。你把它打包成了 .war 文件，那么你就是应用程序组装者。一般的公司测试服务器和上线的生产服务器（Production）是分开的，服务器有专人管理，那么他就是服务器管理员，普通开发人员不能随便发布程序到生产机。如果再加上有专人负责发布程序并观察结果，那么此人就是应用程序发布者。一般情况下，服务器管理员和发布者是一组的，他们通常还要负责管理多台服务器集群，定时重启，调优性能等等，有一个专门的职业就是服务器管理员（和网管差不多，不过工作重点主要集中在服务器上）。也许有人奇怪了，为啥服务器还要重启呢？那是因为随着服务器上的程序越来越多，一般都会不堪重负，Java 的垃圾回收并非 100% 的，随着时间增长服务器进程所占内存越来越多，当到达一个极限后，服务器就崩溃了，换句话说就是叫宕机了（常说的服务器挂了），所以需要定时重启。

16.1.5 JavaBean 和 EJB 的区别

Java Bean 是可复用的组件，对 Java Bean 并没有严格的规范，理论上讲，任何一个 Java 类都可以是一个 Bean（例如 Spring 中就是这样，所有的类都可以配置为 bean）。但通常情况下，由于 Java Bean 是被容器所创建（如 Tomcat 的），所以 Java Bean 应具有一个无参的构造器，另外，通常 Java Bean 还要实现 Serializable 接口用于实现 Bean 的持久性。Java Bean 实际上相当于微软 COM 模型中的本地进程内 COM 组件，它是不能被跨进程访问的。Enterprise Java Bean 相当于 DCOM，即分布式组件。它是基于 Java 的远程方法调用（RMI）技术的，所以 EJB 可以被远程访问（跨进程、跨计算机）。但 EJB 必须被部署在诸如 Webspere、WebLogic 这样的容器中，EJB 客户从不直接访问真正的 EJB 组件，而是通过其容器访问。EJB 容器是 EJB 组件的代理，EJB 组件由容器所创建和管理。客户通过容器来访问真正的 EJB 组件。

16.1.6 EJB 和 Spring 能否共存

答案是肯定的，由于 Spring 容器本身就是普通的 JavaBean，所以它可以集成到 EJB 中来使用。而且由于 Spring 本身不提供分布式和集群服务，所以 Spring+EJB 有时候是一个非常好的解决方案。而且 Spring 本身也对客户端访问 EJB 提供了很好的支持。

16.2 系统需求

本章内容需要 JBoss 服务器，MyEclipse 6 以及 MySQL 或者 Derby 数据库即可。


16.3 准备工作

16.3.1 MyEclipse 对 EJB 提供的开发支持

坦白说 MyEclipse 对 EJB 的开发不是太好，它对 EJB 2.1 的开发是通过 XDoclet 的方式进行的，而 JBuilder 2006 中有 EJB 设计器，可以可视化的生成和修改 EJB。而它对 EJB

3.0 的开发相对好一点，有一个实体 Bean 的生成向导以及从数据库生成实体 Bean 的反向工程功能。开源免费的 Netbeans 6 对 EJB 的开发也支持的不错。

16.3.1 配置应用服务器

在第六章 [管理应用服务器](#)中，我们已经详细讨论了配置应用服务器的普遍方法，读者可以参考那一章的内容完成配置。那么在我们本章的例子中用的服务器是 JBoss 4.2.2GA，其配置步骤是：点击 **Servers** 视图的工具栏上的  按钮来打开服务器配置对话框，或者从主菜单中选择 **Window > Preferences**，这样就可以打开配置对话框。

当配置对话框打开后，可以展开左侧的树，选择 **MyEclipse > Servers**，然后就可以看到左侧列出的可用的应用服务器连接器了，选中 **JBoss 4.x**，接着在右侧的 **JBoss home directory** 处输入 JBoss 的主目录，例如 `C:\jboss-4.2.2.GA`，随后需要根据情况添加一个 JDK，因为 JBoss 必须使用 JDK 才能运行，最后点击 **Enable** 单选钮完成配置。对话框中的 **Server name** 可以取值为 `all`，`default` 和 `minimal`，对应着三个实际存在的目录，位于 `C:\jboss-4.2.2.GA\server` 下面（关于这三个目录差别的介绍，参考下一节内容）。另外，默认情况下 JBoss 只监听来自本地（localhost）的连接请求，如果读者希望能够通过局域网来测试的话，必须设置 **Optional program arguments** 为 `-b 0.0.0.0`，关于此问题在 [1.4 JBoss 服务器的下载, 安装和运行\(可选\)](#)一节中也有介绍。具体配置读者可以参考图 16.16。

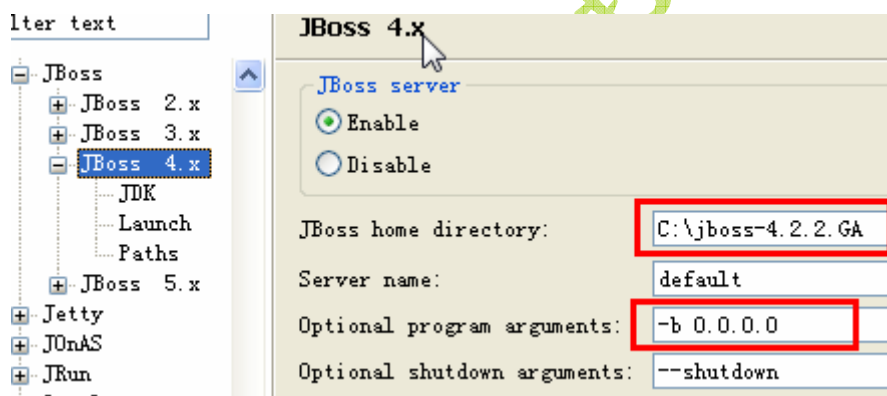


图 16.16 配置 JBoss 服务器

16.3.2 JBoss 服务器目录结构简介

开源版本的 JBoss 基于微内核和插件机制进行设计，因此其服务器的各种功能都是通过插件的方式实现的。目前来说 JBoss 的 Web 层是基于 Tomcat 内核实现的，不过 JBoss 的管理功能比较弱，其控制台只能浏览组件，无法修改，也无法实现人工重新部署。另外 JBoss 目前尚未通过 Java EE 5 全面认证，这意味着某些功能缺失，对 EJB 3 的支持尚可，其实体 Bean 是通过 Hibernate 实现的。很遗憾的 JBoss 的开源版本不带说明文档，因此很多功能只能参考配置文件中的注释来修改。不过，读者可以购买 JBoss 商业版本，带有完整的文档和全面的功能。

JBoss 4 安装之后，得到如下表所示的目录结构：

目录	描述
bin	启动和关闭 JBoss 的脚本。run.bat 启动服务器，有一些附加参数可用，例如 -b 和 -c；shutdown.bat 关闭服务器。

client	客户端访问 JBoss 服务（远程调用 EJB 等）所需的 Java 类库
docs	不是文档，只是一些配置的样本文件（数据源配置等）和 XML 的 DTD, Schema 文件。包括下列子目录：dtd, examples, licenses, schema, tests。
lib	包括 JBoss 启动时加载的全局类库（以及一些 JBoss 内核类库），并被所有 JBoss Server 配置共享（不要放置 JDBC 驱动等全局类库到此处）。
server	各种 JBoss 服务器配置（相当于 Windows 中每个用户拥有自己不同的桌面一样，但是大家用的系统都是一样的）。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，运行时可以切换，例如：run.bat -c all 将会运行 all 这个配置。可以创建自己的配置，但同一时刻默认情况下只能运行一个配置。minimal 包含了最简配置。
server/all	JBoss 的完全配置，启动所有服务，包括集群和 IIOP，如果需要全部功能，可以使用此配置。启用方式：run.bat -c all，或者在 MyEclipse 连接器中输入 all
server/default	JBoss 的默认配置目录。默认运行时即启用此配置，包含大多数功能。
server/配置/conf	JBoss 的配置文件，包括身份验证等。
server/配置/data	JBoss 的配置数据，不要删除或者手工修改，否则会出错。
server/配置/部署	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，包括 EJB、WAR、EAR、Web 服务，以及数据源等服务。
server/配置/lib	当前配置的共享 JAR 文件（例如 JDBC 驱动），JBoss 在启动特定配置时加载。
server/配置/log	JBoss 的日志文件。
server/配置/tmp	JBoss 的临时文件。
server/配置/work	JBoss 的 JSP 工作文件夹，包含了编译后的 JSP 对应的 Servlet 源代码文件和类文件，可在必要时清空以纠正某些页面无法刷新问题。

图 16.17 列出了 jboss 的目录结构，以及发布的应用列表。可以看到我们上一章所做的练习 JBossWS.war 也在其中，只要把 16.1.3 节中所说的文件复制到此处即可发布，或者是建立对应的目录也可，例如：myEJB.jar/。另外，数据源的发布也可以通过此处完成，此时只需将 XML 文件复制到此处即可。当新文件覆盖了老文件时，即可激活重新部署，不过个别时候会出错无法完成，此时需要重启服务器。

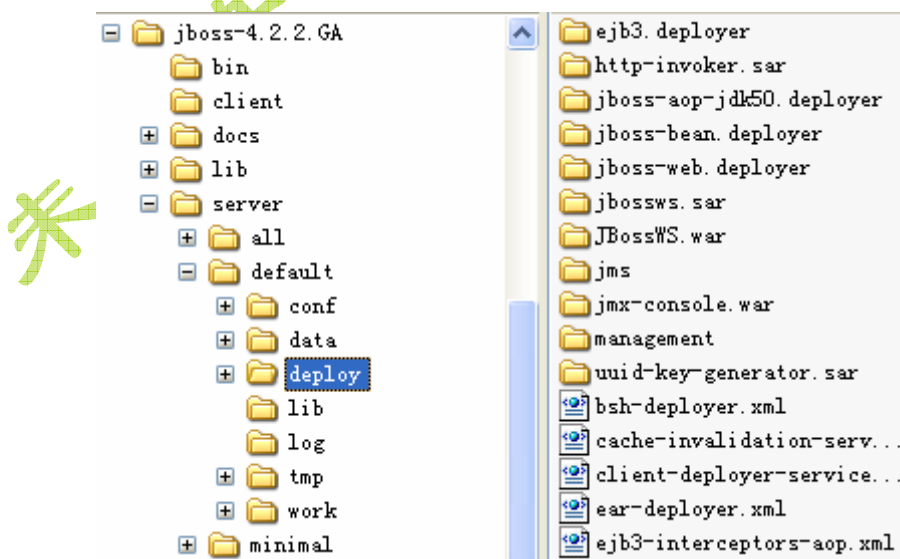


图 16.17 JBoss 的目录结构

16.4 走进 Java EE: JNDI

本节内容对 JNDI 进行介绍，并编写程序访问 JNDI。

16.4.1 JNDI 简介

俗话说中国人不知道长城，就像当于美国人不知道华盛顿一样可笑。不过，如果您学完了 EJB 仍然不知道 JNDI 是什么，那也是件相当严重的问题。JNDI 是打开 Java EE 大门的钥匙，有了它您才能理解数据源，EJB 等等种种神奇的东西。

JNDI 是什么呢？JNDI(Java Naming and Directory Interface)是指通过提供统一的 Java API 访问不同的命名和目录服务。包括下列功能：

- 把名字和对象联系起来
- 提供一个工具，根据名字寻找资源，我们称为查寻或搜索资源
- 类似于文件系统或者地址簿，一棵绑满了对象的树，不同的是树根在上

举个例子，大家的电脑上都有一棵文件树，大家找文件的时候，是从根节点我的电脑开始的，然后从上到下一步步的找。文件系统就是一个命名服务，它将我们能够理解的东西，文件名，和我们不能直接操作的东西：硬盘上的一格格的小磁铁所存储的数据，关联起来。虽然从本质上说，当我们打开一个电影文件的时候，是操作系统通过复杂的线路打开了硬盘或者光盘上的文件。这就是命名服务的功能所在：将文件名和磁盘上的文件关联；提供工具，文件浏览器，可以查找资源并操作资源。由于通常操作系统的开发商，例如微软，已经做好了文件浏览器，它相当于命名服务提供者，那么对于我们这些用户来说，只需要按照规定操作，点击鼠标，即可打开或者复制删除某个文件，而无须知道底层的细节（即使从来没听说过硬盘也无所谓，您要的只是数据，那个文件）。磁盘相当于一个服务器，存放着大量真实的二进制 0101 的数据，而我们，则通过微软的这个系统，来访问到想要的的数据。而且，文件一般都有根文件夹，根文件夹下面可以放文件，也可以继续放一个子文件夹，层层嵌套。图 16.18 已经很好的展示了 JNDI 的结构。所不同的是，Java 中的每个节点，存放的是 Java

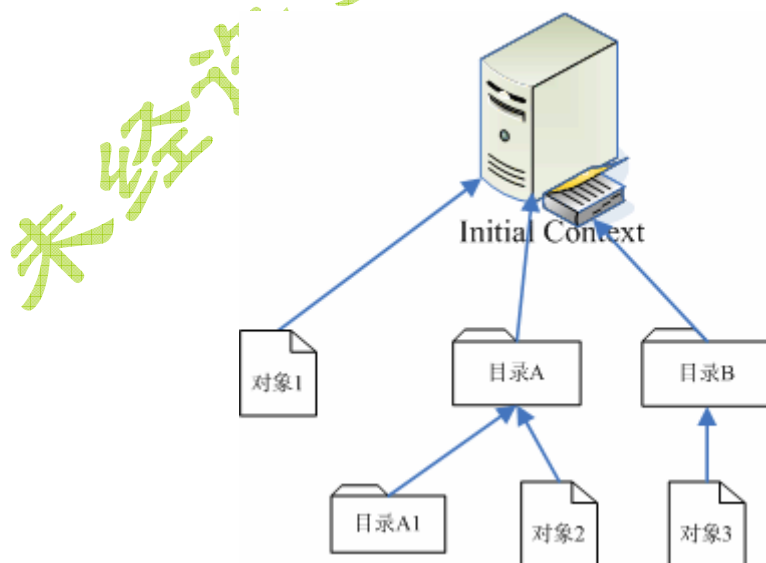


图 16.18 JNDI 树状结构

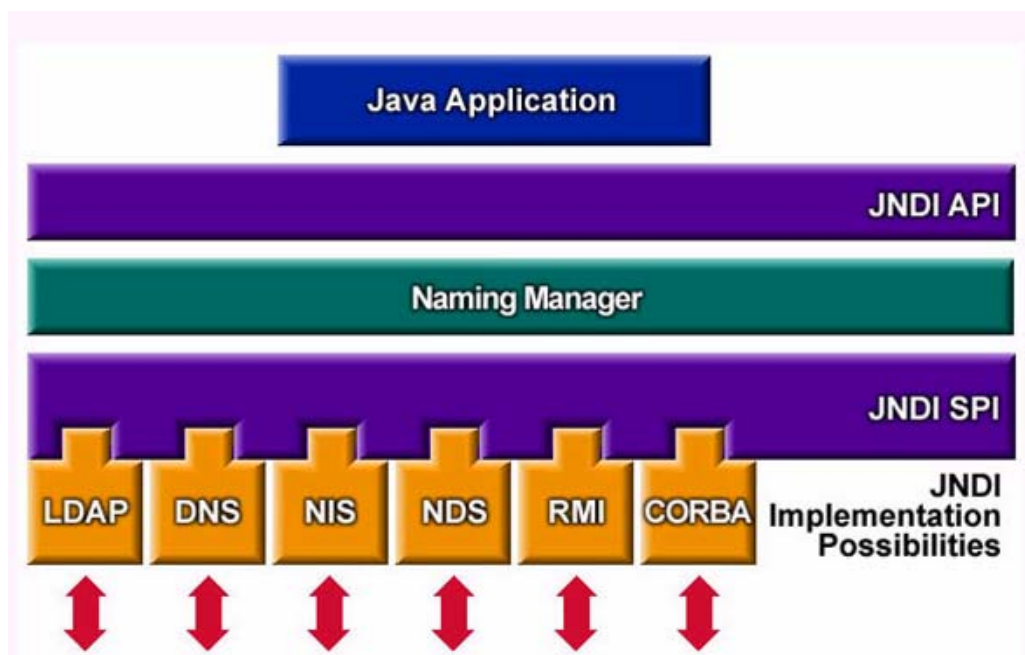


图 16.19 JNDI API 体系结构

对象，而不是磁盘文件，这个对象可以是任意类型，只要实现了序列化接口就可以了。包括但不限于：字符串，整数，EJB，Web 服务，数据源等等，这样一来开发人员就不用关心这些数据的来源，只需要使用即可，就相当于看电影文件时不需要关心到底是硬盘还是光盘的文件。JNDI 服务器就是一台没有预装数据的空白磁盘外加管理器。图 16.19 列出了其层次结构。由于 JNDI 规定了三层结构，对于程序员来说，只需要了解 JNDI 的 API，这就相当于大家只需知道怎么找到文件浏览器就行了，剩下的事情，交给 JNDI SPI(Service Provider Interface)，它们来实现具体的读写功能，包括读写光盘，软盘，硬盘。因此，JNDI 具有替代功能，例如可以用 Windows，也可以用 Linux，来读取同一张光盘上的内容而不会出现读取错误，这就是替换了服务提供者。正因为如此，除了标准 JNDI API 之外，我们还需要特定 JNDI 厂商所提供的驱动程序（一般叫做 JNDI 工厂类库），并指定连接参数，即可访问不同的服务器数据了。所幸的是 JBoss 已经提供了一个 JNDI 实现，我们可以进行一些测试。

和文件一样，每个对象所在的路径都必须是唯一的，也不允许有重复的上下文。对于对象来说，有的是只读的，有的是加密的，还有的是只有特定用户才能访问的。JNDI 支持类似的概念，有的对象属于服务器特定对象，用户无法访问；而有的对象则只能读取，无法修改和删除，例如 Tomcat 通过 JNDI 配置的连接池。同样的有的服务器也不支持用户设置自定义对象，这就相当于只读服务器。

JNDI 的客户端开发包括下列步骤：

1. 创建初始化上下文(InitialContext)。它相当于打开文件浏览器，就像要找文件先要打开 Windows 上的“我的电脑”一样。对本地客户端（例如同一台服务器上的 EJB 或者 Web 层的 Servlet 中），可以直接初始化上下文：

```
Context ctx = new InitialContext();
```

对任意客户端的话，则需要设置一些初始化的参数：

```
Hashtable ht = new Hashtable();
```

```
// 设置工厂类（指定 SPI），不同的服务器取值不同
```

```
ht.put
```

```
(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
```



```
// 指定要连接的服务器地址（相当于 JDBC 的 URL）
ht.put(Context.PROVIDER_URL,"jnp://localhost");//如果要加端口，地址为：jnp://host:port
// 可以设置服务器的用户名和密码（也和 JDBC 类似）
//ht.put(Context.SECURITY_PRINCIPAL,"user");
//ht.put(Context.SECURITY_CREDENTIALS,"password");
Context ctx = new InitialContext (ht);
```

当然编写这一步的时候需要确保放入了驱动类。不同的服务器，驱动程序类（Context Factory）和 URL 都是不一样的。

然而这样要写很多的代码才可以，另有一种办法就是在类路径根目录下放置配置文件 jndi.properties，文件代码清单如下：

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost
```

此时也可以直接编写代码：`Context ctx = new InitialContext();`来创建初始化上下文。

2.创建上下文（类似于创建目录）。代码如下：

```
ctx.createSubcontext("jdbc");//这相当于在 C 盘创建了个子目录
```

如需创建多层目录结构，那么还可以继续这样创建：

```
ctx.createSubcontext("jdbc/mysql");
```

结构相当于下列层次结构：

```
/jdbc/mysql/
```

。既然有创建上下文，自然也就有删除上下文，对应的代码是：

```
ctx.destroySubcontext("jdbc/mysql");
```

注意：目前 JBoss 并不支持这一销毁的操作，调用会报错：not supported。

3.绑定对象。因为对象不能凭空产生，需要服务器或者用户创建后，然后绑定到服务器（相当于上传文件到服务器上），之后才能被访问和使用。JBoss 绑定也是这样做的，它会启动时首先创建 EJB 等对象，然后绑定 EJB 等资源。被绑定的类唯一的要求就是实现 java.io.Serializable 接口。示意代码如下：

```
ctx.rebind("jdbc/MyName", "BeanSoft 刘长炯");// 我们可以绑定一个数据源对象等其它可序列化的对象
```

还有一个方法名为 `bind()`，其主要区别是 `rebind()`可以重复绑定（或者说更新老对象），而 `bind()`只能调用一次。

4.查找对象。这一步相当于从服务器下载文件，或者说读取文件，代码如下所示：

```
// 查找 JNDI 上的对象，
```

```
Object myName = ctx.lookup("jdbc/MyName");
```

```
// 打印读取的值
```

```
System.out.println(myName);
```

5.从服务器删除对象。这一步的操作叫 `unbind`，取消绑定，相当于从服务器上删除文件，后果就是服务器上指定位置的对象被删掉，不能再次访问到。代码如下：

```
ctx.unbind("jdbc/MyName");
```

6.关闭上下文。这一步相当于断开到服务器的连接，注意数据仍然在服务器上存在，可以在下次连接时再次取到。即使服务器关闭了，这些数据也应该可以通过下次启动时加载。

```
ctx.close();// 关闭
```

JNDI 的这一客户端访问过程如图 16.20 所示。

JNDI 服务器上的数据一般都是针对读取优化的，虽然也能做一些增删改查，但是这些数据多放置在内存中，并不经常做变动。因此，它不能作为数据库来使用，只能存储少量信

息, 常见的例如用户数据。此外, JNDI 还可以用来进行域名解析, 不过自然需要您提供 DNS 服务器。

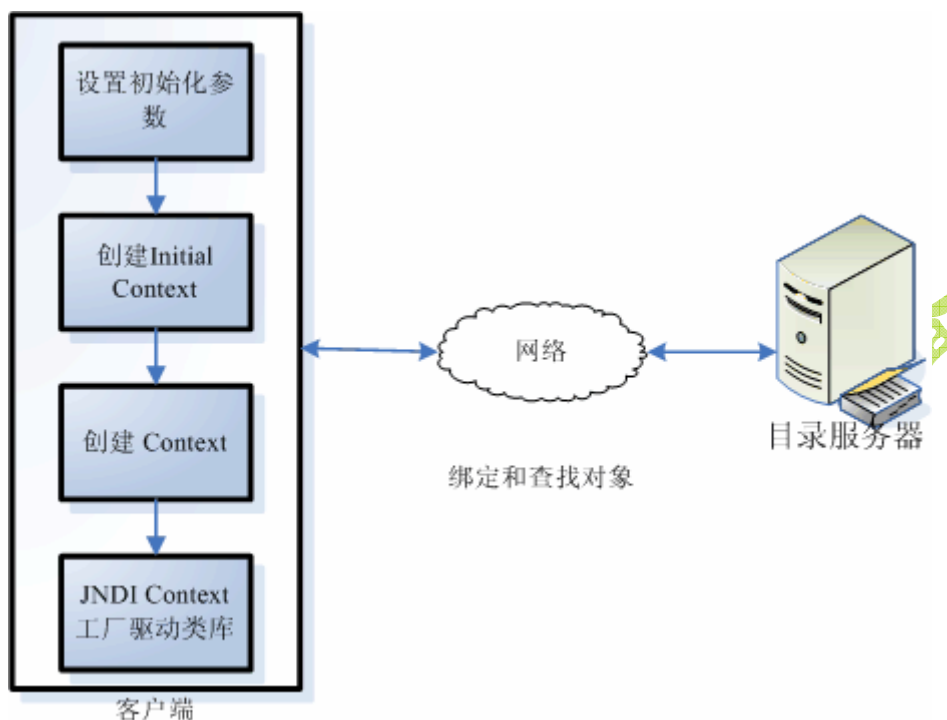


图 16.20 JNDI 客户端和服务端体系

16.4.2 如何查看 JBoss 服务器的 JNDI 树

首先自然是需要启动 JBoss 服务器了, 单独启动或者在 MyEclipse 中启动均可, 然后用任意浏览器打开地址:

<http://localhost:8080/jmx-console/HtmlAdaptor>, 这一步将打开 JBoss 的控制台, 然后找到页面中的下列内容:

jboss

- [database=localDB,service=Hypersonic](#)
- [name=PropertyEditorManager,type=Service](#)
- [name=SystemProperties,type=Service](#)
- [readonly=true,service=invoker,target=Naming,type=http](#)
- [service=AttributePersistenceService](#)
- [service=ClientUserTransaction](#)
- [service=JNDIView](#)

点击链接 `service=JNDIView`, 打开如下地址:

<http://localhost:8080/jmx-console/HtmlAdaptor?action=inspectMBean&name=jboss%3Aservice%3DJNDIView>, 此时即可看到 JBoss 的 JNDI 的管理 Bean, 再找到页面中的 List of MBean operations: 一栏, 如图 16.21 所示, 点击 `Invoke` 按钮后即可看到 JNDI 列表了。

JBoss 的 JNDI 大致分为三类。第一类是各个应用自带的局部 JNDI, 例如 JBossWS.war:

java:comp namespace of the JBossWS.war application:

```

+- UserTransaction[link -> UserTransaction] (class: javax.naming.LinkRef)
+- env (class: org.jnp.interfaces.NamingContext)
| +- security (class: org.jnp.interfaces.NamingContext)
| | +- realmMapping[link -> java:/jaas/other] (class: javax.naming.LinkRef)
| | +- subject[link -> java:/jaas/other/subject] (class: javax.naming.LinkRef)
| | +- securityMgr[link -> java:/jaas/other] (class: javax.naming.LinkRef)
| | +- security-domain[link -> java:/jaas/other] (class: javax.naming.LinkRef)

```

List of MBean operations:

java.lang.String list()

Output JNDI info as text

Param	ParamType	ParamValue	ParamDescription
verbose	boolean	<input checked="" type="radio"/> True <input type="radio"/> False	If true, list the class of each object in addition to its name

Invoke

图 16.21 调用输出 JNDI 信息为文本方式的操作

第二类是服务器提供的服务资源，例如数据源，事务管理器等等，这些内容只能通过服务器配置，用户不可自行添加和修改，而且只能在服务器内部访问，不可远程访问（例如可以在同一服务器的 JSP 或者 EJB 中访问，严格的说不能跨 JVM 或者跨网访问）。

java: Namespace

```

+- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
+- DefaultDS (class: org.jboss.resource.adapter.jdbc.WrapperDataSource)
+- SecurityProxyFactory (class: org.jboss.security.SubjectSecurityProxyFactory)
+- DefaultJMSProvider (class: org.jboss.jms.jndi.JNDIProviderAdapter)
+- comp (class: javax.naming.Context)

```

这些对象位于 java: 下的命名空间中，例如下面的数据源 DefaultDS 可以这样访问：访问的时候代码应该这样写：`lookup("java:/DefaultDS")` 或者 `lookup("java:DefaultDS")`；

第三类是服务器提供的全局 JNDI 资源，这些内容包括各种 EJB，JMS 服务等等，用户可以向其中加入自己的 JNDI 内容。我们开发的 EJB 和 JNDI 测试都位于这下面，例如：

Global JNDI Namespace

```

+- TopicConnectionFactory (class: org.jboss.naming.LinkRefPair)

```

一般来说这下面的资源是可以跨进程远程访问的，但是，也不尽然，例如 EJB 的 Local 接口就是不可访问的。我们可以认为它只是列出了访问时不需要加前缀 java: 的资源列表。

除此之外，JBoss 还带了一个 Web Console，其访问地址为 <http://localhost:8080/web-console/>，以树状列出了服务器的状态（基于 Java Applet），感兴趣的读者可以打开此地址浏览相关信息。

16.4.3 开发 JNDI 应用

16.4.3.1 简单 JNDI 开发

本节我们展示下列内容：

1. 向 JBoss JNDI 加入简单对象和自己的对象；
2. 从 JSP 中访问 JNDI；
3. 使用接口的方式访问 JNDI。

首先让我们创建一个普通的 Java 项目，名为 *JBossJNDITest*，接着，请读者将文件 *\$JBASS_HOME/client/jbossall-client.jar*（文件大小 4.7MB，包含了所有的远程访问 JBoss 服务所需的 Java 类库）在 Libraries 中通过 Add External Jar 加入到项目的 Build Path（相当于 CLASSPATH）中，或者将其复制到项目目录中，然后点击右键加入 Build Path。

接着我们建立一个测试类，来进行字符串的测试，该测试类名为 *jndi.JNDIBindTest*，代码清单如下：

```
package jndi;

import java.util.Hashtable;
import javax.naming.*;

/**
 * JNDI 测试程序.
 *
 * @author 刘长炯
 */
public class JNDIBindTest {
    InitialContext ctx; // Initial Context 对象

    // 初始化上下文对象
    void initJNDI() throws NamingException {
        Hashtable properties = new Hashtable();
        // 配置驱动程序, JBoss 特定的
        properties.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
        // 配置 URL, 这个 URL 格式没有规定, 各个服务器各自一套
        properties.put(Context.PROVIDER_URL, "jnp://localhost");
        ctx = new javax.naming.InitialContext(properties);
    }

    // 关闭 JNDI
    void closeJNDI() throws NamingException {
        ctx.close();
    }
}
```

```
// 创建层次结构
void createContext() throws NamingException {
    ctx.createSubcontext("beansoft");
    ctx.createSubcontext("beansoft/stringtest");
}

// 删除层次结构
void deleteContext() throws NamingException {
    ctx.destroySubcontext("beansoft/stringtest");
    ctx.destroySubcontext("beansoft");
}

// 绑定字符串
void bindString() throws NamingException {
    ctx.rebind("beansoft/stringtest/MyName", "BeanSoft 刘长炯");
}

// 查找字符串对象,相当于下载文件
void lookupString() throws NamingException {
    Object myName = ctx.lookup("beansoft/stringtest/MyName");

    // 打印读取的值
    System.out.println(myName);
}

// 取消 JNDI 绑定,相当于删除文件
void unbindString() throws NamingException {
    ctx.unbind("beansoft/stringtest/MyName");
}

// 专门针对字符串测试
void stringTest() throws NamingException {
    initJNDI();
    createContext();
    bindString();
    lookupString();
    //unbindString();
    //deleteContext();
    closeJNDI();
}

public JNDIBindTest() {
    try {
```

```

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

public static void main(String[] args) {
    new JNDIBindTest();
}
}

```

这段代码展示了我们在上一节提到的 JNDI 开发的步骤，运行后得到的输出仅仅是一段看似无关痛痒的：

BeanSoft 刘长炯

，这段内容是从 `lookupString()` 这个方法中执行得到的。然而，此时我们查看 JBoss 的 JNDI 树，会看到全局 JNDI 下面多出了下面所示的内容：

```

+- beansoft (class: org.jnp.interfaces.NamingContext)
/ +- stringtest (class: org.jnp.interfaces.NamingContext)
/ | +- MyName (class: java.lang.String)

```

。那么从此以后，这个对象就放在了 JBoss 服务器上了，但是如果只是这样存放的数据，关闭服务器后下次再打开，它们就消失了，具体错误为：

javax.naming.NameNotFoundException: beansoft not bound (一般来说 JBoss 是通过配置文件的方式来确保每次启动时都会绑定对象)。不过，在本次服务器未重启时，数据是一直可以看到的，那么第二次运行测试代码访问时，就不需要再次绑定数据就可以读取到了，此时的测试方法应修改为：

```

void stringTest() throws NamingException {
    initJNDI();
    lookupString();
    closeJNDI();
}

```

读者可以试试第二次运行，采用这样的几个方法，包括打开，读取和关闭，同样可以读到数据。

那么第二个小实验，就是保持上面服务器存放数据不变的情况下，从 Web 程序中访问 JNDI 数据，很简单，创建一个 Web 项目名为 *JBossJNDIWeb*，然后修改其 **index.jsp** 文件内容为：

```

<%@ page language="java" pageEncoding="GBK"%>

<html>
<head>
<title>JNDI Web 访问测试</title>
</head>
<body>
<%
javax.naming.InitialContext ctx = new javax.naming.InitialContext();

```

```
Object myName = ctx.lookup("beansoft/stringtest/MyName");
// 打印读取的值
out.println(myName);
%>
</body>
</html>
```

，发布项目到JBoss服务器，然后在地址栏键入 <http://localhost:8080/JBossJNDIWeb/> 刚问，可得到如下的输出：

BeanSoft 刘长炯

。此处没有设置上下文参数就可创建 `InitialContext` 是因为默认在 JBoss 中它连向 JBoss 自带的 JNDI 服务，Context 工厂类采用 JBoss 自带值，URL 则指向当前所在的 JSP 程序所在服务器。反过来如果需要连接到别的服务器，或者是用 Tomcat 来作为客户端访问远程的 JNDI 数据，就需要采用那种设置初始化参数的方式了。

接下来要测的第三个小项目就是在 Tomcat 中远程访问 JBoss 中的 JNDI 服务了，随便新建一个 Web 项目，例如我们把它起名为 `TomcatJNDIWeb`，接着将其 `index.jsp` 文件内容修改为：

```
<%@ page language="java" import="java.util.*,javax.naming.*"
pageEncoding="GBK"%>

<html>
<head>
<title>Tomcat 访问 JBoss JNDI 测试</title>
</head>
<body>
<%
Hashtable properties = new Hashtable();
// 配置驱动程序, JBoss 特定的
properties.put(Context.INITIAL_CONTEXT_FACTORY,
                "org.jnp.interfaces.NamingContextFactory");
// 配置 URL,这个 URL 格式没有规定,各个服务器各自一套
properties.put(Context.PROVIDER_URL, "jnp://localhost");

javax.naming.InitialContext ctx = new
javax.naming.InitialContext(properties);
Object myName = ctx.lookup("beansoft/stringtest/MyName");
// 打印读取的值
out.println(myName);
%>
</body>
</html>
```

接着我们还需要 JBoss 的客户端类库作为 JNDI 的初始化类库，把文件 `$JBASS_HOME/client/jbossall-client.jar` 复制到项目的 `WEB-INF/lib` 下即可。

要测试这个应用，首先要解决的问题就是 Tomcat 的 Web 监听端口也是 8080，这样就会和 JBoss 的冲突，为了方便测试，我们将 Tomcat 的监听端口改为 8090，具体办法是找

到 Tomcat 目录/conf/server.xml 中的如下片段，并将端口（port）改为 8090，如下所示：

```
<Connector port="8080" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443" />
```

将"8080"改为"8090"即可。

也许有读者反过来想的是修改 JBoss 的 Web 监听端口，那也很好办，打开文件 C:\jboss-4.2.2.GA\server\default\deploy\jboss-web.deployer\server.xml 中的如下片段，将端口改为需要的值就可以了，如下所示：

```
<Connector port="8080" address="{jboss.bind.address}"
    maxThreads="250" maxHttpHeaderSize="8192"
    emptySessionPath="true" protocol="HTTP/1.1"
```

然后我们启动此服务器，并将应用 TomcatJNDIWeb 发布过去，接着在浏览器键入地址 <http://localhost:8090/TomcatJNDIWeb/> 进行测试，没有问题的情况下我们可以看到输出正确的信息，即刚才绑定到 JBoss JNDI 的名字。

最后，我们要测试的是这样一种场景。假设我们和一家客户通过 JNDI 提供服务，我们编写了一个接口，告诉他们提供了什么样的服务，但是，我们不希望他们知道我们的核心技术，在这种情况下，自然不能给他们实现类的源代码了（再次只能模拟 EJB+客户端的一种情况）。那么客户只需要拿到接口和 JNDI 的地址，就可以调用此服务了，而类的源代码它们是拿不到的，那么，我们分两部分来开发。首先看提供服务的人，我们，需要开发一个接口和一个类，这个服务假设就是做汇率计算，从美元转换成人民币。继续刚才开发的项目 JBossJNDITest，新建两个类。

接口 bean.Calculator 代码清单如下：

```
package bean;

/**
 * 汇率计算器接口类。
 * @author BeanSoft
 */
public interface Calculator {
    double dollarToRMB(double input);
}
```

接下来是其实现类 bean.CalculatorImpl，这个类要实现序列化接口，代码清单如下：

```
package bean;

/**
 * 汇率计算器实现类。
 * @author BeanSoft
 */
public class CalculatorImpl implements Calculator, java.io.Serializable
{
    public double dollarToRMB(double input) {
        return input * 7.0026;
    }
}
```



```
}

```

。然后修改类 `jndi.JNDIBindTest`，新加入操作计算器的功能类和测试方法，并把构造器中的方法修改为新的内容，下面只列出修改的片段：

```
// 绑定计算器对象
void bindCalculator() throws NamingException {
    ctx.createSubcontext("Calculator");
    ctx.rebind("Calculator/remote", new bean.CalculatorImpl());
}

// 取消计算器 绑定
void unbindCalculator() throws NamingException {
    ctx.unbind("Calculator/remote");
}

// 专门针对计算器对象测试
void caculatorTest() throws NamingException {
    initJNDI();
    bindCalculator();
    //unbindCalculator();
    System.out.println(ctx.lookup("Calculator/remote"));
    closeJNDI();
}

public JNDIBindTest() {
    try {
        //stringTest();
        caculatorTest();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

。然后运行这个测试类后，即可在 JNDI 树上看到新绑定的对象：

```
+-- Calculator (class: org.jnp.interfaces.NamingContext)
/   +- remote (class: bean.CalculatorImpl)
.
```

随后，我们要在 Web 应用中访问此对象，在应用 `JBossJNDIWeb` 中编写一段代码来测试此 JNDI 对象。首先，因为我们这里并不是真正的 EJB，所以不得不把 `bean` 包下的两个类的代码都复制到 Web 应用的 `src` 目录下，否则在测试接下来提到的 `calculator.jsp` 时会得到这样的出错提示：

```
javax.servlet.ServletException: javax.naming.CommunicationException [Root exception is
java.lang.ClassNotFoundException: bean.CalculatorImpl (no security manager: RMI class
loader disabled)]
```

。不过，如果是真正的 EJB 应用的话，就不需要在 `src` 下面放这两个类的代码了。在 `WebRoot` 下新建测试页面 `calculator.jsp`，代码清单如下：

```

<%@ page language="java" pageEncoding="GBK"%>
<%@page import="bean.Calculator"%>

<html>
  <head>
    <title>JNDI Web 访问测试</title>
  </head>
  <body>
    <%
      javax.naming.InitialContext ctx = new
      javax.naming.InitialContext();
      Calculator calc = (Calculator)
      ctx.lookup("Calculator/remote");
      out.println(calc.dollarToRMB(1.23));
    %>
  </body>
</html>

```

随后，发布应用到 JBoss 服务器后，在浏览器中键入地址 <http://localhost:8080/JBossJNDIWeb/calculator.jsp> 即可测试，输出的结果应是：

8.613198

。读者需要注意上面的代码形式，可以看到 JNDI 时可以完全不用关心实现类的名字，只需要转换为接口即可。而 JNDI 的对象地址 *Calculator/remote* 实际上是我们自己通过 JNDI 提供的创建 Context 功能和绑定功能完成的。实际上 JBoss 发布 EJB 时，也会采用类似的调用过程，本节内容就是为了帮助大家理解即将出现的 JBoss EJB 客户端代码的访问方式的。

16.4.3.2 JNDI 访问数据源

数据源的开发在 Java EE 服务器中是必不可少的，相比较于 WebLogic 等商业服务器，JBoss 的数据源只能用编写配置文件的方式手工部署。首先我们介绍 MySQL 数据源的部署方式，首先大家在 JBoss 的安装目录下的 *docs\examples\jca* 中找到配置文件 **mysql-ds.xml**，将其内容修改为如下所示：

```

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

<!-- $Id: mysql-ds.xml 63175 2007-05-21 16:26:06Z rrajesh $ -->
<!-- Datasource config for MySQL using 3.0.9 available from:
http://www.mysql.com/downloads/api-jdbc-stable.html
-->

<datasources>
  <local-tx-datasource>
    <jndi-name>MySqlDS</jndi-name>
    <use-java-context>>false</use-java-context>
    <connection-url>jdbc:mysql://localhost:3306/test</connection-url>

```

```

<driver-class>com.mysql.jdbc.Driver</driver-class>
<user-name>root</user-name>
<password></password>

<exception-sorter-class-name>org.jboss.resource.adapter.jdbc.vendor.MySQLException
Sorter</exception-sorter-class-name>
  <!-- should only be used on drivers after 3.22.1 with "ping" support

<valid-connection-checker-class-name>org.jboss.resource.adapter.jdbc.vendor.MySQLV
alidConnectionChecker</valid-connection-checker-class-name>
  -->
  <!-- sql to call when connection is created
  <new-connection-sql>some arbitrary sql</new-connection-sql>
  -->
  <!-- sql to call on an existing pooled connection when it is obtained from pool -
MySQLValidConnectionChecker is preferred for newer drivers
  <check-valid-connection-sql>some arbitrary sql</check-valid-connection-sql>
  -->

  <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml -->
  <metadata>
    <type-mapping>mySQL</type-mapping>
  </metadata>
</local-tx-datasource>
</datasources>

```

。然后要做的，就是先启动 MySQL 数据库服务器，然后将其驱动程序，例如 *mysql-connector-java-3.1.11-bin.jar* 复制到目录 *C:\jboss-4.2.2.GA\server\default\lib* 下，随后，再将文件 *mysql-ds.xml* 复制到 *C:\jboss-4.2.2.GA\server\default\deploy* 下，这样即可发布此数据源，可以在 JBoss 服务器的输出信息中看到：

```

11:29:43,109 INFO [WrapperDataSourceService] Bound ConnectionManager
'jboss.jca:service=DataSourceBinding,name=MySqlDS' to JNDI name
'MySqlDS'

```

，这样即是将数据源发布了，查看 JBoss 的 JNDI 的话，位于 Global JNDI Namespace 下。

注意：默认情况下，JBoss 的数据源都是配置为仅服务器可见的（即服务器内部的 EJB，JSP 应用能够访问，外部例如独立运行的 Java 程序客户端不能访问），也就是位于 *java:Namespace* 下面。如果要做跨服务器的测试，请加入标签 **<use-java-context>false</use-java-context>**，例如我们的例子所示。一般情况下如果没有远程访问数据源的要求，则可以去掉此配置，就像 JBoss 自带的数据源 *java:DefaultDS* 一样。另外，SQL Server，Oracle 等的配置文件示例都可以在 *docs\examples\jca* 目录中找到。

如果读者想配置 MyEclipse 自带的 Derby 数据库作为数据源，那么需要两件事：把 JDBC 驱动 *derbyclient.jar* 放入目录 *C:\jboss-4.2.2.GA\server\default\lib* 下；然后就是配置一份文件 **derby-ds.xml** 复制到 *C:\jboss-4.2.2.GA\server\default\deploy* 下：

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

<!-- The Hypersonic embedded database JCA connection factory config -->

<!-- $Id: hsqldb-ds.xml 63175 2007-05-21 16:26:06Z rrajesh $ -->

<datasources>
  <local-tx-datasource>

    <!-- The jndi name of the DataSource, it is prefixed with java:/ -->
    <!-- Datasources are not available outside the virtual machine -->
    <jndi-name>DerbyDS</jndi-name>

    <!-- For server mode db, allowing other processes to use hsqldb over tcp.
    This requires the org.jboss.jdbc.HypersonicDatabase mbean.
    <connection-url>jdbc:hsqldb:hsq!://${jboss.bind.address}:1701</connection-url>
    -->
    <!-- For totally in-memory db, not saved when jboss stops.
    The org.jboss.jdbc.HypersonicDatabase mbean is required for proper db shutdown
    <connection-url>jdbc:hsqldb:./</connection-url>
    -->
    <!-- For in-process persistent db, saved when jboss stops.
    The org.jboss.jdbc.HypersonicDatabase mbean is required for proper db shutdown
    -->

    <connection-url>jdbc:derby://localhost:1527/myeclipse;create=true</connection-url>

    <!-- The driver class -->
    <driver-class>org.apache.derby.jdbc.ClientDriver</driver-class>

    <!-- The login and password -->
    <user-name>app</user-name>
    <password>app</password>

    <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml -->
    <metadata>
      <type-mapping>Derby</type-mapping>
    </metadata>

  </local-tx-datasource>
```

```
</datasources>
```

然后我们要测试此数据源，这次我们试试另一种办法来初始化 JNDI 上下文，即使用 JNDI 初始化配置文件的方式。在 Java 项目 *JBossJNDITest* 的 *src* 目录下新建文件 **jndi.properties**，代码清单如下：

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost
```

随后创建 Java 类 **jndi.DataSourceTest** 来访问此数据源：

```
package jndi;

import javax.naming.InitialContext;
import javax.sql.DataSource;

/**
 * MySQL 数据源测试。
 */
public class DataSourceTest {
    public DataSourceTest() {
        try {
            InitialContext ctx = new javax.naming.InitialContext();
            Object obj = ctx.lookup("MySqlDS");
            System.out.println(obj);
            DataSource ds = (DataSource)obj;
            java.sql.Connection conn = ds.getConnection();
            System.out.println(conn.getMetaData().getDatabaseProductName());
            java.sql.Statement stmt = conn.createStatement();
            java.sql.ResultSet rs = stmt.executeQuery("select * from
student");
            while(rs.next()) {
                System.out.println(rs.getString(2));
            }
            rs.close();
            stmt.close();
            conn.close();
            ctx.close();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new DataSourceTest();
    }
}
```

由于使用了 JNDI 配置文件，所以初始化上下文的时候，就不需要再设置参数了，这是个节省代码的好办法，推荐在普通 Java 类中访问 JNDI 时使用。这段代码运行后，输出内容如下：

```
org.jboss.invocation.jrmp.interfaces.JRMPInvokerProxy@140c281
MySQL
学生1
测试 Spring JPA
```

。分别打印出了数据源的类型，以及数据库版本和一些 Student 表的数据信息。

接着我们展示位于 java: 下面的服务器私有的数据源的访问方式，它们只能通过同一服务器上的 JSP 或者 EJB 来访问，所以在 Web 项目 JBossJNDIWeb 的 WebRoot 下新建 JSP 文件 **datasource.jsp**，代码清单如下：

```
<%@ page language="java" import="javax.sql.*, javax.naming.*"
    pageEncoding="GBK"%>

<html>
  <head>
    <title>JNDI 私有数据源 访问测试</title>
  </head>
  <body>
    <%
      InitialContext ctx = new javax.naming.InitialContext();

      Object obj = ctx.lookup("java:DefaultDS");

      DataSource ds = (DataSource) obj;

      java.sql.Connection conn = ds.getConnection();

      out.println(conn.getMetaData().getDatabaseProductName());
      conn.close();
      ctx.close();
    %>
  </body>
</html>
```

。项目发布到 JBoss 后，用浏览器打开页面 <http://localhost:8080/JBossJNDIWeb/datasource.jsp> 进行测试，可以看到以下输出：

```
HSQL Database Engine
```

。这说明 JBoss 默认的这个 DefaultDS 数据源，是 HSQL，也是个开源的嵌入式 Java 数据库，其官方网站地址是 <http://www.hsqldb.org/>。

16.4.3.3 使用 JNDI 进行 DNS，邮件服务器，主机信息查找

本节内容属于了解性质，它本质上和 JBoss 服务器没有任何关系。这个功能有什么用

呢？坦白说大多数垃圾邮件发送器，或者说是号称不经过中转直接发送邮件的软件，都会用 DNS 进行 MX 服务器查找，然后直接发送邮件，仅供大家作为了解吧。

先看如何找 DNS 服务器，在 Windows 下点击**开始 > 运行**，然后输入 `cmd` 后回车，接着输入 `ipconfig /all` 命令后回车，即可看到输出的网络连接信息中有 DNS 服务器（当然前提是你的电脑已经联网了），如下所示：

Ethernet adapter 本地连接:

```

Connection-specific DNS Suffix  .:
Description . . . . .: Realtek RTL8139/810x Family Fast Eth
ernet NIC
Physical Address. . . . .: 00-0F-EA-E0-D0-0B
Dhcp Enabled. . . . .: Yes
Autoconfiguration Enabled . . . .: Yes
IP Address. . . . .: 192.168.0.5
Subnet Mask . . . . .: 255.255.255.0
Default Gateway . . . . .: 192.168.0.1
DHCP Server . . . . .: 192.168.0.1
DNS Servers . . . . .: 192.168.0.1
                        202.106.46.151
Lease Obtained. . . . .: 2008 年 4 月 14 日 9:10:56
Lease Expires . . . . .: 2008 年 4 月 15 日 9:10:56

```

。这里取 202.106.46.151 作为 DNS 服务器就可以了。

然后，在 Java 项目 `JBossJNDITest` 下新建类 `jndi.JNDISearchDNS`，代码清单如下：

```

package jndi;

import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;

// 用 JNDI 进行 DNS 查询
public class JNDISearchDNS {

    public static void main(String args[]) {
        try {
            // Hashtable for environmental information
            Hashtable env = new Hashtable();

            // Specify which class to use for our JNDI provider
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.sun.jndi.dns.DnsContextFactory");
            env.put(Context.PROVIDER_URL, "dns://202.106.46.151/");///

```

把这个地址修改为你当前的

地址

```

// DNS
// 服务器的

String dns_attributes[] = { "MX", "A", "HINFO" };

// Get a reference to a directory context
DirContext ctx = new InitialDirContext(env);
Attributes attrs1 = ctx.getAttributes("google.com",
    dns_attributes);

if (attrs1 == null) {
    System.out.println("主机没有上述信息");
} else {

    for (int i = 0; i < dns_attributes.length; i++) {
        Attribute attr = attrs1.get(dns_attributes[i]);

        if (attr != null) {
            System.out.print(dns_attributes[i] + ": ");
            for (Enumeration vals = attr.getAll(); vals
                .hasMoreElements();) {
                System.out.println(vals.nextElement());
            }
        }

        System.out.println(" ");
    }
} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
}

```

这段代码目的是探测 Google.com 的邮件接受服务器地址，以及 IP 信息和主机操作系统信息。MX 对应邮件接受服务器，IP 信息对应 A（能列出此域名下对应的多个 IP），HINFO 对应主机信息（不过为了安全一般不会现实），运行后得到的输出如下：

```
MX: 10 smtp4.google.com.
```

```
10 smtp1.google.com.
```

```
10 smtp2.google.com.
```

```
10 smtp3.google.com.
```

```
A: 72.14.253.99
```


64.233.161.99

64.233.171.99

72.14.209.99

。喜欢发邮件的可以随便跳一个 smtp 服务器地址就可以给 Google 的人发送邮件了，不过，一般会有反垃圾服务器的过滤器，所以不能保证 100%成功。

16.4.4 JBoss/Tomcat 中的一个 JNDI Bug 问题解决

由于 JBoss 的 JNDI 部分的代码存在一个著名的 BUG，如果 JBoss 或者 Tomcat 安装到了带空格的路径中，例如：

C:\Apache Software\Tomcat 或者 C:\RadHet Inc\JBoss，那么在 Web 应用中初始化 JNDI 上下文的时候，总是会报类似于这样的一个错误：

```
javax.naming.CommunicationException[Root exception is  
java.net.MalformedURLException: no protocol: Files/Apache]
```

这个问题只有一个解决方案，就是安装的时候路径中不要带空格。

这个也一再验证了我们的建议：Java 软件不要安装在带空格的路径里，也不要安装在带中文字符的路径里。例如：c:\myapp 是个好路径，而 c:\my app 或者 c:\我的程序就很有可能导致不可知的问题。

这个 BUG 同样适用于单独的 Tomcat 来初始化 JNDI 然后尝试访问 JBoss EJB 时产生的问题（当然是使用 JBoss 的那个 JNDI 类库了）。

16.5 开发 Session Bean

16.5.1 Session Bean 简介

在 Java EE 5 中，只有两种 EJB: Session Bean(会话 Bean)和 Message Driven Bean (消息驱动 Bean, 简称 MDB)。而会话 Bean 又分为两类: Stateful Session Bean (有状态会话 Bean)和 Stateless Session Bean (无状态会话 Bean)。实体 Bean 呢，按照 Sun 公司官方的说法，应该并入 JPA 里面了。本节将会简介会话 Bean 的开发过程。

会话 Bean，顾名思义就是提供对话服务，类似于天气预报 112 电话，用户来这里，发起请求，得到自己想要的东西，结束。举个例子各位去医院看病，第一关，是要找到医院吧，然后，医院的保安还要对你略作评估（安全或者身份验证），至少要确信你不是来搞破坏的或者是付不起医药费的。接着，前台接待人员会热情的问你：是挂专家号还是普通号？专家号，自然，贵点，因为专家总是很忙，而且专家的工资也高，人数也少，所以你得多等会，而且，专家还会跟你很熟，记得你经常得哪些病，这样诊断起来就比较准确，甚至不需要你过多的描述，就能知道你的病情。普通号，很好挂，这样的医生也多，随治随走，也便宜，因为医生不需要记住你曾经得过什么病，也不需要认识你（人太多了，没有必要），这样每次你去，都得不厌其烦的陈述自己的病史。此处所列的专家和普通，前者就相当于有状态会话 Bean，后者则相当于无状态的会话 Bean。

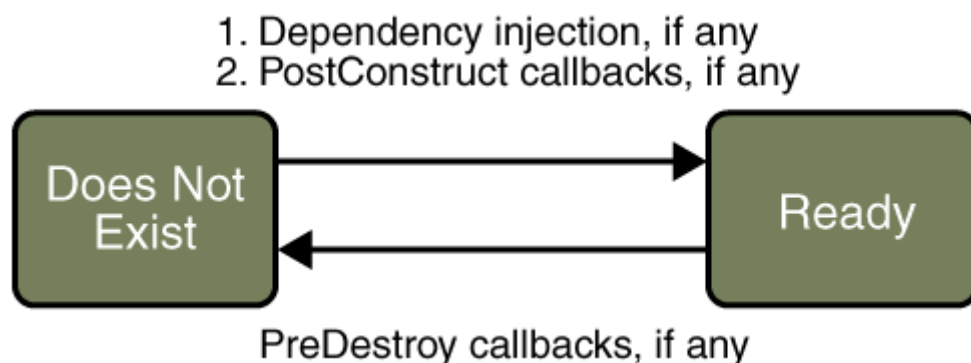


图 16.21 无状态的会话 Bean 的生命周期

图 16.21 列出了无状态的会话 Bean 的生命周期。这些生命周期，都是对于服务器（我们以医院为例）来说的。当你找一个普通门诊时，医院可以随意挑选一个闲着的医生对你进行诊治，所需的只是上岗证即可，普通医生看完一个人，就可以做一些必要的工作，例如销毁老药方等等，等待下一个病人上门。图 16.21 中，一个 Bean 需要首先进行依赖注入（Dependency injection），然后是一些初始化后的准备工作（PostConstruct callback，创建后回调方法），例如打开网络资源等，随后即可进入工作就绪状态。工作结束后可以随时进行预先销毁（Pre Destroy），例如释放网络资源等，接着进入不存在（Does Not Exist，也就是闲置状态）。这说明不能担保无状态会话 Bean 中的变量值是在访问时一直保持不变的，例如说你给其设置一局部变量，那么这个变量的值虽然你已经设置过一次，然而当你再次使用时，这个值很可能已经消失，就相当于你在同一医院，看完了眼睛之后，忽然想起还有点问题没问，然而你返回后，却发现屋里的医生已经换了另外一个（当然这情况有点夸张，不过无状态 Bean 就是这个意思）他自然不记得你以前的情况，他只根据你的病历来了解你过去的情况。所谓会话，即是当你初始化了 JNDI 后，一直到你断开 JNDI 这段时间。无状态 Bean 在即使你没有断掉 JNDI 时，也会出现忘掉你是谁的问题。

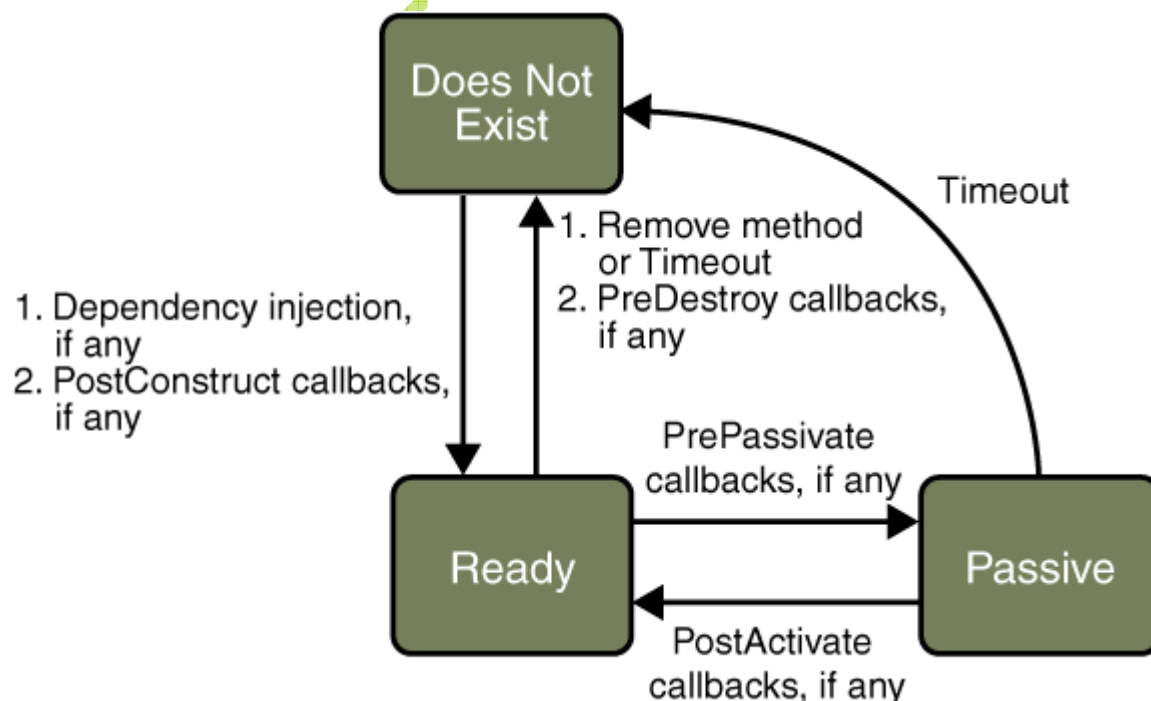


图 16.22 有状态的会话 Bean 的生命周期

图 16.22 列出了有状态的会话 Bean 的生命周期。它相当于专家，和普通的医生差不多的初始化过程，然而他却多了个额外状态：钝化（Passive）。这个状态怎么理解呢？专家因为要记住病人的情况，所以他不得不准备了一个笔记本，自己列下病人的病史，并妥善保管。当你来的时候，他拿出本子（PostActive，激活到 Ready 状态），根据你的名字找出你的既往病史，这样他就可以因人而异的进行看病了，而且给你的感觉是好像这专家一直认识你，其实他认识的是本子上的记录，因为来访的病人即是对专家来说，也超出了他们的记忆能力了。当你满意出门时，专家会对你的此次会诊情况进行笔录，然后存入笔记本，这样你的信息就处于 Passive 状态了，在你下次来的时候，再次激活。另外，也有可能出现超时这种意外情况（Timeout，专家的笔记本年代太久，有些内容丢了，那么下次你来了，他就不认识你了），或者医院对此专家不满（例如专家看病老出事），就把专家赶走了（Remove），那么下次来，你会经验的发现专家换人了，而这专家也不认识你了，不过，在你们彼此寒暄后，下次这个换了的专家依然会记得你的情况，而在普通门诊，这种事情是不会发生的。新版的 Java EE 5 多出了个定时器的服务。

会话 Bean 的定义主要通过 `@Stateless` 和 `@Stateful` 这两个位于 `javax.ejb` 包中的标注完成的，前者表明 Bean 是无状态，后者则是定义有状态 Bean。例如下面的两个代码片段：
无状态会话 Bean：

```
@Stateless
public class CalculatorImpl implements Calculator {
    public int calculate(int number) {
        return number;
    }
}
```

有状态会话 Bean：

```
@Stateful
public class CalculatorImpl implements Calculator {
    private int number;
    public int calculate(int number) {
        this.number += number;
        return this.number;
    }
}
```

这两个标注都有一些属性可以设置，如下表所示：

属性名	类型	说明	示例
name	String	此 Bean 的 ejb-name。	<code>@Stateful(name="MyCalc")</code>
description	String	描述信息	<code>@Stateful(description ="计算器 Bean")</code>
mappedName	String	此会话 Bean 应该映射到的特定产品名称（例如全局 JNDI 名称）。应用服务器不需要支持任何特殊形式或类型的映射名称，也不需要具有使用映射名称的能力。映射的名称与产品有关，并且通常与安装有关。任何对映射	<code>@Stateless(mappedName =" MyCalc")</code> 注意：JBoss 对此标注完全不支持！

	名称的使用都不可移植。	
--	-------------	--

一般来说，都需要先定义一个接口，然后 EJB 再实现它。不要接口的 EJB 是不能发布的，这样格式的 EJB 不能在 JBoss 上成功发布：

```
package session;

import javax.ejb.Stateless;

@Stateless
public class MyEJB {
    public int add(int a, int b) {
        return a + b;
    }
}
```

。一旦发布，JBoss 会抛出以下错误：

```
bean class has no local, webservice, or remote interfaces defined and does not implement at least one business interface: MyEJB
```

。这段话的意思是说，此 Bean 必须至少有一个接口。经过笔者在 Netbeans 6 和 GlassFish 服务器上进行测试，发现这样的 EJB 也是无法访问的，所以，EJB 必须要有一个接口。

另外，关于会话 Bean 的访问方式，还分成了远程 (Remote) 和本地 (Local) 方式。其主要区别是前者可以通过跨进程的 JNDI 访问，例如独立运行的 Java 程序，或者是另一台服务器 (Tomcat, JBoss 等都可)；那么默认情况下都是 Local 方式的，它们只能在和 EJB 所在的服务器相同的应用上访问，典型情况是发布在同一台服务器上的 EJB 或者 Web 应用可以访问，而跨进程则是不可访问的。那么 Local 的有什么用呢？由于大部分时候，EJB 和其访问类都位于同一服务器上，这样就不通过传统的远程模式走，而可以直接用类似于内部调用的方式，可以省略网络中间的通信延迟（举例：建设你和你家人都在一间屋子里，你还通过卫星电话聊天嘛？那肯定是直接面对面，快速高效还不用担心断线了），以及大量的其它消耗了。所以，如果没有远程访问的要求，推荐使用 Local 模式。

要创建一个允许远程访问的 EJB，必须进行至少下列操作之一：
在 bean 的接口类上，加上一个 @Remote 标注，代码片段如下：

```
@Remote
public interface 业务接口名 {
    ...
}
```

或者在 EJB 类上加上 @Remote 标注，不过此时必须指定业务接口（一个或者多个），代码片段如下：

```
@Remote(接口名.class)
public class Bean 名 implements 接口名 {
    ...
}
```

反过来，默认情况下，EJB 中的接口都是 Local 的，也就是说无须进行额外的配置。不过，一般为了清楚起见，还是可以加入一些标注来说明某些接口是 Local 的。有两种做法，可以在 bean 的接口类上，加上一个 @Local 标注，代码片段如下：

@Local

```
public interface 业务接口名 {
...
}
```

或者在 EJB 类上加上 @Local 标注，不过此时必须指定业务接口（一个或者多个），代码片段如下：

```
@Local (接口名.class)
public class Bean 名 implements 接口名 {
...
}
```

注意：一个 EJB 最少需要 Local 接口，或者是 Local + Remote 两个接口，一个接口也没有是会抛出错误的。换句话说，一个 EJB 至少对应一个 @Local 的业务接口。另外，对于标注来说，@标注 和 @标注() 是等价的，都表示不设置任何属性！

关于生命周期方法的标注（标注在方法上），有状态和无状态的会话 Bean 都支持：
@PostConstruct（在 Bean 创建之后调用），@PreDestroy（在 Bean 销毁之前调用）。

而下面这些标注只能用在有状态会话 Bean 上：
@PrePassivate（在 Bean 转入后台的钝化状态，把相关信息从内存移出，缓存到磁盘上时会调用，注意这时 Bean 并未消失，只是其变量值从内存消失了，换句话说一个 Bean 可以同时为多个调用者服务，只需要带有不同的状态即可）和 @PostActivate（从后台激活之后调用）。还有一个标注是 @Remove，此标注相当于 EJB 2.0 时候提供的 ejbRemove() 方法，调用此标注对应的方法后，表示用户希望结束会话，断开连接，销毁此 Bean，释放相关的资源，容器则会在合适的时候进行此操作（其实就是客户端可以提建议，但是容器保持最终的决定权在何时销毁），此标注对应的方法体可以为空。

另外，最新的 EJB 3 也支持拦截器机制。具体是在 Bean 上加入标注 @Interceptors({拦截器类名 1.class, 拦截器类名 2.class...})。拦截器类则为普通的 Java 类，需要进行拦截操作的方法需要加上标注 @AroundInvoke，例如下列代码：

```
@AroundInvoke
public Object 方法名(InvocationContext ctx) throws Exception {
    System.out.println(ctx.getMethod().getName() + "已经被调用!");
}
```

最后一个亮点是 Java EE 5 中简化了对外部资源的访问，也就是常说的依赖注入 (Dependency Injection)，这些特殊的标注用来声明注入一个存在与当前 JNDI 上下文的资源或者其它的实体，容器会自动注入这些值。包括下列的一些标注可用：

- **@Resource**

注入数据源，上下文变量， UserTransaction 和 EJBContext 等，例如获取数据源的代码片段：

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

- **@EJB**

注入 EJB 的业务接口。代码片段：

```
@EJB(beanName="A")
EJBA a;
```

- `@PersistenceContext`, `@PersistenceUnit`
注入 `EntityManager` 和 `EntityManagerFactory`。
- `@WebServiceRef`
注入 Web 服务的引用。Web service references。

注意：并非任意 JNDI 上的对象类型都可以被注入，目前只支持以下类型的资源：

`DataSource`, `JMS`, `Mail`, 环境属性, `EntityManager`, `EJB Context`, `UserTransaction` 和 `TimerService`。而通过 JNDI 访问则不受此限制。

在我们的接下来的测试中将会对提到的大部分内容进行展示,其它内容包括标注式的事务和安全, 定时器等我们不再讨论了, 读者可以阅读参考资料部分的电子书获取更多信息。

16.5.2 开发无状态 Session Bean

在本节, 我们会创建无状态的会话 Bean。要完成的功能很简单, 就是根据两个参数计算加法。

首先, 选择菜单 **File > New > EJB Project**, 来启动创建 EJB 项目的向导。在 **Project Name** 中输入 `EJB3Session`, 然后在 **J2EE Specification Level** 下选中单选按钮 `Java EE 5.0 - EJB 3`, 接着去掉复选框 `Add support for Entity beans` 的选中状态。最后点击对话框的 **Finish** 按钮结束向导。在刚才的操作中, 我们使用的是 EJB 3, 同时去掉了开发实体 Bean 的支持。如果读者希望自行开发 EJB 2, 那么选中对应的版本 J2EE 1.4 即可。此过程如图 16.23 所示。

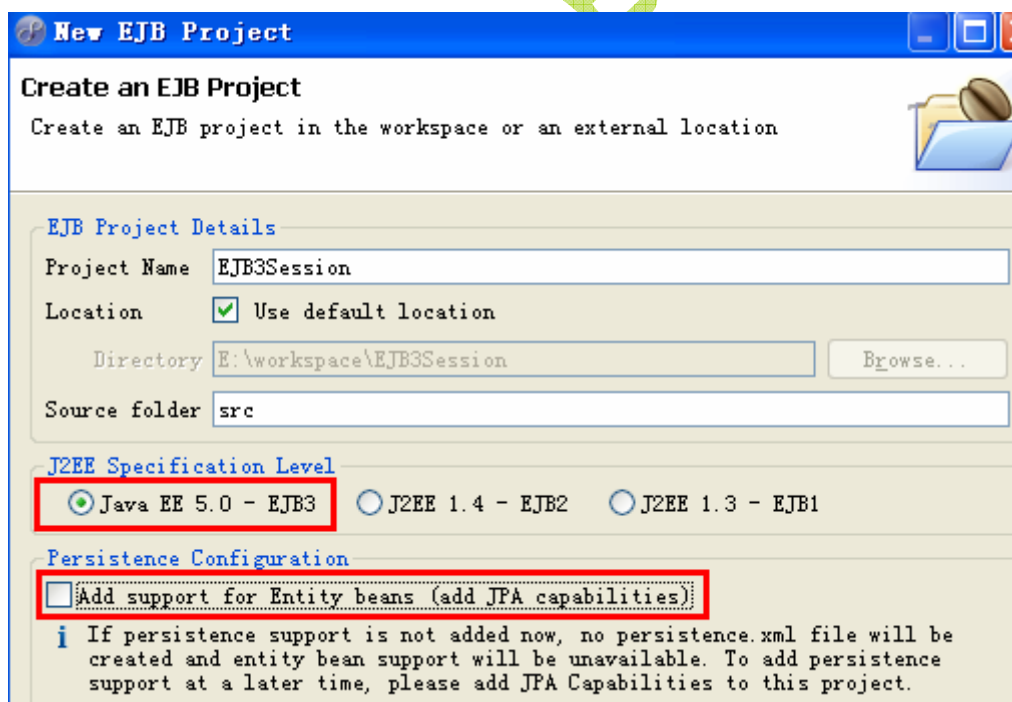


图 16.23 新建 EJB 项目对话框

接着选择菜单 **File > New > EJB3 Session Bean** 启动创建会话 Bean 的向导。**Package**(包)名中可以根据自己情况输入, 此处输入值 `session`, 接着 **Name** (类名) 处需要输入值 `StatelessCalculator`, **Session Type** (会话类型) 处选中单选按钮 `Stateless` (无状态), 而 **Create Interface** (创建接口) 处则需要选中复选框 `Local` 和 `Remote` (实际开发中只需要选中一处即可, 一般使用 `Remote`), 最后点击 **Finish** 按钮结束向导。此过程如图 16.24 所示。

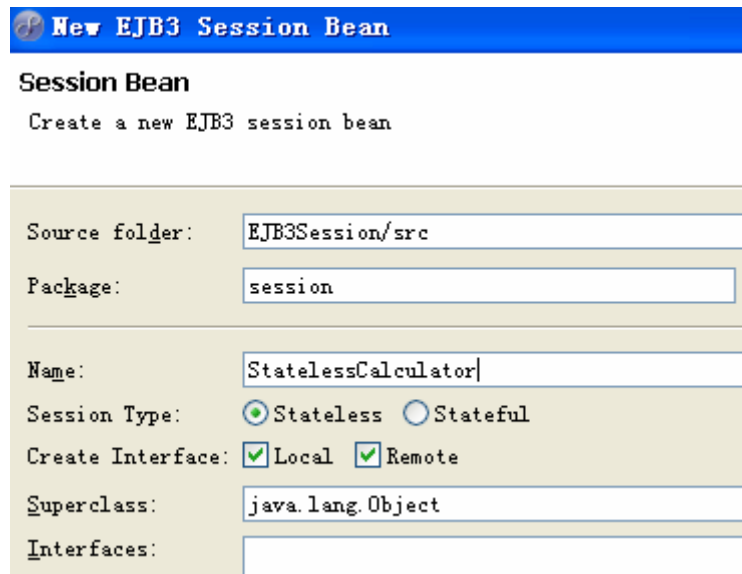


图 16.24 新建 EJB 会话 Bean 对话框

随后，将会出现三个类的代码：`StatelessCalculator`，`StatelessCalculatorLocal` 和 `StatelessCalculatorRemote`，我们将对这些代码略作修改。先来看两个业务接口，`Remote` 和 `Local` 的代码清单。

`StatelessCalculatorLocal.java`

```
package session;

import javax.ejb.Local;

@Local
public interface StatelessCalculatorLocal {
    int multiply(int a, int b);
}
```

`StatelessCalculatorRemote.java`

```
package session;

import javax.ejb.Remote;

@Remote
public interface StatelessCalculatorRemote {
    int add(int a, int b);
}
```

粗斜体部分列出了两个类中的新增功能代码，就是在远程接口中计算加法，而在本地接口中计算乘法。虽然这里把它们的方法写成了不一样的，实际上 `Remote` 和 `Local` 接口中的方法可以相同，分别服务于远程调用和本地调用。如果希望远程和本地的接口都提供同样的服务，除了这种写法之外，还有一种方法是让其中的一个接口继承自另一个，例如下面的写法：

```
package session;

import javax.ejb.Remote;
```

```
@Remote
```

```
public interface StatelessCalculatorRemote extends
StatelessCalculatorLocal {
}
```

这样也是可以的，可以避免大量的接口方法重复编写两次，如果您有这方面的需要，推荐用这种写法。

接着我们来看 EJB 类的代码，**StatelessCalculator.java** 的代码清单如下：

```
package session;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Stateless;

@Stateless
public class StatelessCalculator implements StatelessCalculatorLocal,
    StatelessCalculatorRemote {
    public int add(int a, int b) {
        return a + b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }

    @PreDestroy
    public void destory() {
        System.out.println("EJB StatelessCalculator 即将销毁");
    }

    @PostConstruct
    public void created() {
        System.out.println("EJB StatelessCalculator 已经创建完毕");
    }
}
```

这个类中，方法 `add(int a,int b)` 和 `multiply(int a,int b)` 是必须的，而后面的两个生命周期方法，则可以省略不写。而且，`@PreDestory` 后面的方法的名字是没有限制的，既可以写成 `destory()`，也可以写成 `abc()`。同理 `@PostConstruct` 后面的方法也一样。

好了，先把这个项目发布到 JBoss 服务器上，发布过程和 Web 应用是一样的。也可以在 Package Explorer 视图中选中项目节点 `EJB3Session`，然后选择菜单 **Run > Run As > 3 MyEclipse Server Application**，之后 MyEclipse 可能会显示一个可用的服务器列表，选中其中的 JBoss 服务器，点击 **OK** 按钮后，项目就会自动发布，对应的服务器会启动。

随后检查 JNDI 树，会看到 Global JNDI Namespace 下面列出了 EJB 的信息：

```
+- StatelessCalculator (class: org.jnp.interfaces.NamingContext)
/ +- local (proxy: $Proxy91 implements interface
session.StatelessCalculatorLocal, interface org.jboss.ejb3.JBossProxy)
/ +- remote (proxy: $Proxy90 implements interface
session.StatelessCalculatorRemote, interface
org.jboss.ejb3.JBossProxy)
```

可以看到默认情况下，JBoss 是把 Bean 的名字作为 JNDI 的上下文，然后 Bean 名字/local 上绑定 Local 接口，Bean 名字/remote 绑定远程接口。

注意：到底默认情况下该绑定到哪个地址，不同的服务器有不同的做法，具体情况请参考对应服务器的文档。例如有的是只绑定一个 HelloWorld.class.getName()，如 GlassFish 服务器。如果要改变绑定地址，可以改动其 name 属性。如：@Stateful(name="MyCalc")，这样设置或会绑定到 MyCalc/remote 或者 MyCalc/local。

此后，我们要写在 JSP 里面写代码调用 EJB 进行测试，在 Web 项目 JBossJNDIWeb 中，新建 JSP 文件 ejbstatelesscalc.jsp，其代码清单修改为如下所示：

```
<%@ page language="java" pageEncoding="GBK"%>

<html>
  <head>
    <title>StatelessCalculator 访问测试</title>
  </head>
  <body>
    Remote:
    <%
      javax.naming.InitialContext ctx = new
javax.naming.InitialContext();
      session.StatelessCalculatorRemote calc =
(session.StatelessCalculatorRemote)
ctx.lookup("StatelessCalculator/remote");
      out.println(calc.add(1, 5));
    %>

    Local:
    <%
      session.StatelessCalculatorLocal calcLocal =
(session.StatelessCalculatorLocal)
ctx.lookup("StatelessCalculator/local");
      out.println(calcLocal.multiply(2, 5));
      ctx.close();
    %>

  </body>
</html>
```

由上述代码可以看到调用 EJB 的过程相当的简单，就是通过 JNDI 访问，获取到对象后转换类型为远程或者本地的接口即可调用上面的方法得到结果了。其余问题都和 16.4.3 一节中介绍的 JNDI 部分相同。说白了，EJB 调用就是通过 JNDI 进行的。

注意：此处的 Web 程序中无须导入 EJB 的包和类即可调用，也无需将接口类复制到 src 下面。不过，这仅限于 JBoss 服务器中的情况，如果是单独的应用或者 Tomcat 中访问，请务必将接口 `StatelessCalculatorRemote` 复制到项目的 src 目录下。

发布运行此项目到 JBoss 服务器后，在 Web 浏览器中输入地址 <http://localhost:8080/JBossJNDIWeb/ejbstatelesscalc.jsp> 即可看到调用结果：

Remote: 6 Local: 10

很好，调用成功了！那么 JBoss 的服务器端还会打出下面的日志信息：

```
15:59:46,703 INFO [STDOUT] EJB StatelessCalculator 已经创建完毕
那么什么时候 Bean 会被销毁呢？当读者关闭 JBoss 服务器或者从 JBoss 中删除(undeploy, 反向发布)此项目的时候，Bean 就会被销毁了，此时控制台上显示如下信息：
```

```
17:22:46,046 INFO [STDOUT] EJB StatelessCalculator 即将销毁
```

感兴趣的读者可以在项目 `JBossJNDITest` 中测试在普通 Java 项目中调用 EJB，也可以在项目 `TomcatJNDIWeb` 然后发布到 Tomcat 测试访问，都可以发现 local 接口是无法被服务器之外的进程远程访问到的。不过，为了方便读者，我们已经在此项目中配置好了访问此无状态 Bean 的类和 JSP 页面，其中的 JSP 页面内容 `ejbstatelesscalc.jsp` 代码清单如下：

```
<%@ page language="java" import="java.util.*, javax.naming.*"
pageEncoding="GBK"%>
<html>
  <head>
    <title>StatelessCalculator 访问测试</title>
  </head>
  <body>
    Remote:
    <%
Hashtable properties = new Hashtable();
// 配置驱动程序, JBoss 特定的
properties.put(Context.INITIAL_CONTEXT_FACTORY,
    "org.jnp.interfaces.NamingContextFactory");
// 配置 URL, 这个 URL 格式没有规定, 各个服务器各自一套
properties.put(Context.PROVIDER_URL, "jnp://localhost");

javax.naming.InitialContext ctx = new
javax.naming.InitialContext(properties);
    session.StatelessCalculatorRemote calc =
(session.StatelessCalculatorRemote)
ctx.lookup("StatelessCalculator/remote");
    out.println(calc.add(1, 5));
    %>
  </body>
```

```
</html>
```

。测试访问地址为：<http://localhost:8090/TomcatJNDIWeb/ejbstatelesscalc.jsp>。

16.5.3 体验无状态 Bean 的混乱态

前面说到，容器提供无状态 Bean 的服务时，并不会保证 Bean 中的变量状态会保持一致，例如我们在项目 *EJB3Session* 中编写这样的一个无状态 Bean，**session.StatelessCalc2**（接口定义同此类的方法）：

```
package session;

import javax.ejb.Stateless;

@Stateless
public class StatelessCalc2 implements StatelessCalc2Local {
    private int a = 0;
    private int b = 0;

    public int add() {
        return a + b;
    }

    public void setA(int a) {
        this.a = a;
    }

    public void setB(int b) {
        this.b = b;
    }
}
```

，然后发布此项目，之后在 Web 应用中创建一个测试页面来进行测试，此处放在 Web 项目 *JBossJNDIWeb* 中，页面名为 *ejbstatelesscalc.jsp*，代码清单如下：

```
<%@ page language="java" pageEncoding="GBK"%>

<html>
  <head>
    <title>StatelessCalc2Local 访问测试</title>
  </head>
  <body>
    Local:
    <%
      javax.naming.InitialContext ctx = new
    javax.naming.InitialContext();
      session.StatelessCalc2Local calcLocal =
```

```

(session.StatelessCalc2Local) ctx.lookup("StatelessCalc2/local");
    calcLocal.setA(1);
    calcLocal.setB(2);
    System.out.println(calcLocal.add());
    calcLocal = (session.StatelessCalc2Local)
ctx.lookup("StatelessCalc2/local");
    calcLocal.setA(3);
    System.out.println(calcLocal.add());
    calcLocal = (session.StatelessCalc2Local)
ctx.lookup("StatelessCalc2/local");
    System.out.println(calcLocal.add());
    ctx.close();

%>

</body>
</html>

```

。咱们看看这段代码，先后获取了 3 个不同的 Bean（按照我们的理解它们应该是不同的，就相当于我们新建了 3 个对象一样，每个对象的初始化取值都为 0）。然后，第一次设置了两个参数，运算结果应该是 3，这没有疑问；第二次只设置了一个参数 3，那么加法结果也应该是 3，因为 $b=0, b+a=0+3=3$ ；第三次两个参数都没设置，加法结果应该是 0，因为 $a=0, b=0, a+b=0$ 。最后，发布 EJB 和 Web 项目，在浏览器中键入地址进行测试：

<http://localhost:8080/JBossJNDIWeb/ejbstatelesscalc2.jsp>

最后我们来看 JBoss 的控制台输出的结果：

```

18:25:38,609 INFO [STDOUT] 3
18:25:38,609 INFO [STDOUT] 5
18:25:38,609 INFO [STDOUT] 5

```

很不幸的，结果是 3, 5, 5。这充分证明了这种情况，服务器偷懒，对于每个要使用无状态 Bean 的人，都可能用的同一个对象，也不对对象上的变量进行重置状态操作（或者说使用之前先把原来的老对象变量值都清空）。这就像是去图书馆借书，第一次借的时候是本新书，读者可能会在书上做一些笔记，看完之后他把书还了；第二个借书的人，会发现书上已经有人留下了足迹；依次类推，这书也越来越旧，越来越破；也许偶然有一天，借书的人会发现自己又借到了一本新书，原来是人太多，图书馆不得不购买了新书给大家用。这就是说，如果你使用了这种设计，如果几十个人都来使用，很可能出现张冠李戴，例如甲设置参数为 $3+4$ ，乙设置为 $4+5$ ，那么结果可能出现： $3+5$ 或者 $4+4$ 这两种谁也不希望看到的错误结果，也就是服务器根本不管调用者是不是同一个人，只要是个闲置的给你用，就像我们说的专人配置的专家和随来随走的普通医生一样。稍后我们展示有状态 Bean 时就可以看到不会再出现这种问题。有状态 Bean，则相当于去书店买书，而不是借书看，虽然每次看到的都是新书，然而显然要比借书来看成本要高的多。

16.5.4 开发有状态 Session Bean

我们还在刚才使用的 Java 项目 *EJB3Session* 中创建有状态会话 Bean，同样的选择菜单 **File > New > EJB3 Session Bean** 启动创建会话 Bean 的向导。**Package**(包)名中可以根据自己情况输入，此处输入值 *session*，接着 **Name**（类名）处需要输入值

StatefullCalculator, **Session Type** (会话类型) 处选中单选钮 **Stateful** (有状态), 而 **Create Interface** (创建接口) 处则只需要选中复选框 **Remote** (实际开发中只需要选中一处即可, 一般使用 **Remote**), 最后点击 **Finish** 按钮结束向导。此过程如图 16.24 所示。

随后, 将会出现两个类的代码: **StatefullCalculator** 和 **StatefullCalculatorRemote**, 我们将对这些代码略作修改。在这里功能代码将和 16.5.3 一节中的内容一样, 就是先设置数据, 然后再计算和。先看远程接口的代码清单:

StatefullCalculatorRemote.java

```
package session;

import javax.ejb.Remote;

@Remote
public interface StatefullCalculatorRemote {
    public void setA(int a);
    public void setB(int b);
    public int add();
    public void remove();
}
```

在此接口中除了业务方法 **setA()**, **setB()**和 **add()**之外, 还加入了一个专门对外公开的用于请求删除 Bean 的回调方法 **remove()**, 为了 **@Remove** 标注准备。

为了多演示一些内容, 我们在 Bean 实现类中加入了一些生命周期方法:

StatefullCalculator.java

```
package session;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.PostActivate;
import javax.ejb.PrePassivate;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful
public class StatefullCalculator implements StatefullCalculatorRemote {
    private int a = 0;
    private int b = 0;

    public int add() {
        return a + b;
    }

    public void setA(int a) {
        this.a = a;
    }
}
```

```

public void setB(int b) {
    this.b = b;
}

@PreDestroy
public void destory() {
    System.out.println("EJB StatefullCalculator 即将销毁");
}

@PostConstruct
public void created() {
    System.out.println("EJB StatefullCalculator 已经创建完毕");
}

@PrePassivate
public void passive() {
    System.out.println("EJB StatefullCalculator 准备钝化");
}

@PostActivate
public void active() {
    System.out.println("EJB StatefullCalculator 已经激活");
}

@Remove
public void remove() {
    System.out.println("EJB StatefullCalculator 请求容器销毁当前 Bean
实例");
}
}

```

此 Bean 发布后，JNDI 上也将出现下列列表（不过和 **Stateless** 的稍微不同）：

```

+- StatefulCalculator (class: org.jnp.interfaces.NamingContext)
/ +- remote (class: java.lang.Object)
/ +- remoteStatefulProxyFactory (proxy: $Proxy117 implements
interface org.jboss.ejb3.ProxyFactory)

```

。随后让我们编写测试代码，老规矩还是放在 Web 项目 **JBossJNDIWeb** 中，代码清单如下：

```

<%@ page language="java" pageEncoding="GBK"%>

<html>
  <head>
    <title>StatefulCalculator 访问测试</title>
  </head>
  <body>
    Remote:

```

```

<%
    javax.naming.InitialContext ctx = new
javax.naming.InitialContext();
        session.StatefulCalculatorRemote calcLocal =
(session.StatefulCalculatorRemote)
ctx.lookup("StatefulCalculator/remote");

        calcLocal.setA(1);
        calcLocal.setB(2);
        System.out.println(calcLocal.add());
        calcLocal = (session.StatefulCalculatorRemote)
ctx.lookup("StatefulCalculator/remote");
        calcLocal.setA(3);
        System.out.println(calcLocal.add());
        calcLocal = (session.StatefulCalculatorRemote)
ctx.lookup("StatefulCalculator/remote");
        System.out.println(calcLocal.add());

        calcLocal.remove();

        ctx.close();
%>

</body>
</html>

```

和上一节的测试无状态Bean时类似，我们获取了 3 个有状态的计算器，并在最后一次调用了 `remove()` 方法，让我们发布项目后在浏览器键入地址 <http://localhost:8080/JBossJNDIWeb/ejbstatefulcalc.jsp>，看看服务器控制台的输出信息：

```

00:02:57,093 INFO [STDOUT] EJB StatefullCalculator 已经创建完毕
00:02:57,140 INFO [STDOUT] 3
00:02:57,156 INFO [STDOUT] EJB StatefullCalculator 已经创建完毕
00:02:57,156 INFO [STDOUT] 3
00:02:57,156 INFO [STDOUT] EJB StatefullCalculator 已经创建完毕
00:02:57,156 INFO [STDOUT] 0
00:02:57,156 INFO [STDOUT] EJB StatefullCalculator 请求容器销毁当前 Bean
实例

```

```
00:02:57,156 INFO [STDOUT] EJB StatefullCalculator 即将销毁
```

第一点，可以发现每个新获取的 **Session Bean** 都会重新执行创建方法，也就是没来一个客户，都要分配一个新对象；第二点，每个新获取的对象都会重置其状态为初始值，这样一来，执行结果就和我们预料的一致了，不过这样一来，服务器就会进行额外的许多操作了；第三点，调用 `remove()` 方法会促使容器销毁 **Bean**。至于何时会出现钝化和激活呢？访问一次 **EJB** 后长时间保持静止（不再访问），或者频繁获取多个 **EJB** 实例后，才能出现，换句话说说用户是没有能力去触发钝化或者激活的。

注意：在本步骤的开发中，我们发现了重新发布 EJB 项目后，需要接着重新发布 Web 项目才能正常测试，否则会抛出类型转换错误。另外如果 Web 开发中要保持一直连接到同一个有状态的会话 Bean，那么最好是放入 HTTP 的会话（session）中，并且不能调用 `ctx.close()`和 `remove()`方法。

16.5.5 EJB 发布描述符和 JBoss JNDI 地址

我们详细介绍下 JBoss 的 JNDI 绑定和命名规则，JBoss 有以下两种绑定规则：

- 缺省绑定

缺省情况下，会话 Bean 的 JNDI 绑定名称是：

- * local 接口: `<ejbName>/local`
- * remote 接口: `<ejbName>/remote`

当 EJB 被部署到某个 ear 中后，缺省绑定名称中会以 ear 包的名称作为前缀。比如，ear 文件是 `test.ear` 时，EJB 的绑定名称是：

- * local 接口: `test/<ejbName>/local`
- * remote 接口: `test/<ejbName>/remote`

- 用户自定义的绑定

JBoss 中，用户可以通过 `@LocalBinding` 或 `@RemoteBinding` 为某个 EJB 自定义 JNDI 绑定名称。注意：`@LocalBinding` 和 `@RemoteBinding` 是 JBoss 自提供的扩展注释，而非 EJB 3 规范的标准注释。建议不要使用！示例：

```
import org.jboss.annotation.ejb.LocalBinding;
import org.jboss.annotation.ejb.RemoteBinding;
```

```
@Stateless
@RemoteBinding (jndiBinding="aaa/Remote")
@LocalBinding (jndiBinding="mybean_Local")
public class MyBean implements .... { ... }
```

。

第二个话题，是 Java EE 5 中仍然支持正规的 XML 发布描述符，例如 Java EE 标准规定的 `ejb-jar.xml` 和服务器特定的描述符例如 `jboss.xml`。而且，如果配置文件中的值和标注中的发生了冲突，则以配置文件中的内容为准，这是为了让系统管理员发布程序时根据情况对一些参数进行调整。关于打包文件目录结构参考图 16.10 EJB 的 JAR 文件格式。

我们在 Java 项目 `EJB3Session` 中新建三个不带 EJB 标注的类。过程就不再赘述了，请看代码清单：

远程接口 `NewSessionRemote.java`

```
package conftest;

public interface NewSessionRemote {

    public String hello(String msg);

}
```

本地接口 `NewSessionLocal.java`

```
package conftest;

public interface NewSessionLocal extends NewSessionRemote {}
```


Bean 实现类 `NewSessionBean.java`

```

package conftest;

public class NewSessionBean implements NewSessionRemote {

    public String hello(String msg) {
        return "你好: " + msg;
    }
}

```

`ejb-jar.xml` 文件需要新建在 `src/META-INF` 目录下, 在此文件中, 我们主要通过配置文件定义了一个 EJB, 其中的 `ejb-name` 的取值对应 `@Stateless` 标注中的 `name` 属性, `business-local` 对应本地接口, `business-remote` 对应远程接口, `ejb-class` 对应作为 EJB 发布的 Bean 实现类, `session-type` 则对应了有状态或者无状态。代码清单如下:

```

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

<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee" version="3.0"
  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/ebj-jar_3_0.xsd">
  <enterprise-beans>
    <session>
      <ejb-name>MySessionBean</ejb-name>
      <business-local>conftest.NewSessionLocal</business-local>
      <business-remote>conftest.NewSessionRemote</business-remote>
      <ejb-class>conftest.NewSessionBean</ejb-class>
      <session-type>Stateless</session-type>
    </session>
  </enterprise-beans>
</ejb-jar>

```

`jboss.xml` 也位于同一目录下, 注意这个文件名完全是 JBoss 规定的, 属于厂商特定的配置文件 (换句话说别的服务器类似的文件名将会不一样), 代码清单如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>MySessionBean</ejb-name>
      <jndi-name>MySessionBeanTest</jndi-name>
      <local-jndi-name>MySession_Local</local-jndi-name>
      <!--
<remote-jndi-name>MySessionRemote</remote-jndi-name-->
    </session>
  </enterprise-beans>
</jboss>

```

此配置文件指定了给定 `ejb` 名字为 `MySessionBean` 的 bean 的 JNDI 名字以及本地接口的

JNDI 名字(也可指定远程接口的 JNDI 名字, 但是运行后看不到效果), 运行后您将会看到 JNDI 树上会出现下面的列表:

```
+-- MySessionBeanTest (proxy: $Proxy72 implements interface
confstest.NewSessionRemote, interface org.jboss.ejb3.JBossProxy)
+-- MySession_Local (proxy: $Proxy74 implements interface
confstest.NewSessionLocal, interface org.jboss.ejb3.JBossProxy)
```

那么, 这两个分别对应远程和本地接口, 从这里我们可以看到使用发布描述符定义的 EJB 和标注方式的是没有本质区别的; 而 `jboss.xml` 则定制了众多特定的和发布时候有关的配置信息, 例如 JNDI。对于 Java EE 5 来说, 大多数时候都不需要您了解 `ejb-jar.xml` 的配置文件格式。

16.5.5 EJB 互访问和资源注入

在 16.5.1 一节中, 我们已经简要介绍了 EJB 中除了可以通过 JNDI 来获得想要的对象之外, 还可以通过标注的方式实现依赖注入(这一功能的确大大简化了程序员的代码量, 以致于新版本的 Spring 2.5 也大大增强了标注注入功能)。我们先来看看 EJB 互访问的内容。EJB 的互访问需要用到一个名为 `@EJB` 的标注, 下面是这个标注类的源代码:

```
@java.lang.annotation.Target(value={java.lang.annotation.ElementType.TYPE,java.lang.
annotation.ElementType.METHOD,java.lang.annotation.ElementType.FIELD})
@java.lang.annotation.Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME)
public abstract @interface javax.ejb.EJB extends java.lang.annotation.Annotation {
    public abstract java.lang.String name() default "";
    public abstract java.lang.String description() default "";
    public abstract java.lang.String beanName() default "";
    public abstract java.lang.Class beanInterface() default java.lang.Object;
    public abstract java.lang.String mappedName() default "";
}
```

通过代码我们可以看到此标注既可以标在类型前面, 也可以标注在方法上, 还可以标注在变量上。关于标注在类型前面, 有一个这样的例子, 通过标注方式的 Servlet 来访问 EJB, 只可惜 JBoss 的 Java EE 5 实现并不完整(也可能是我开发的方式不正确, 貌似需要 EAR 模式的开发), 这个例子在 JBoss 上是没法运行的, 在 GlassFish 上运行良好, 下面是这个例子的代码片段:

```
package enterprise.servlet_stateless_war;

import java.io.*;
import javax.ejb.EJB;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import enterprise.servlet_stateless_ejb.*;
```

// 尽管在类前面标注对企业 bean 的依赖没有任何问题，但是对于无状态会话 Bean 来说
 // 这种操作却不是必须的，因此下面的一行内容注释掉了，我们将依赖容器来注入 bean
 // @EJB(name="StatelessSession", beanInterface=StatelessSession.class)

```
public class Servlet2Stateless
    extends HttpServlet {

    // 在这里注入无状态会话 Bean 仍然是线程安全，因为 ejb 容器会自动将每个请求调度
    // 到不同的 bean 实例上。然而，对于有状态会话 Bean 来说依赖的 bean 必须在类型
    // 前进行

    @EJB
    private StatelessSession sless;

    public void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();

        try {

            out.println(sless.sayHello("HelloTest"));

        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println("webclient servlet test failed");
            throw new ServletException(ex);
        }
    }
}
```

由于实际开发中发现 JBoss 和 Java EE 文档中的某些说法并不一致，因此我们这里只好按照 JBoss 的习惯来做了。

如果标在变量前，应该是类似于这样：

```
@EJB(beanName="StatefulCalculator")
StatefulCalculatorRemote stfulcalc;
```

标在方法前时，则应该是这样：

```
@EJB (beanName="StatefulCalculator ")
public void set StatefulCalculatorRemote (StatefulCalculatorRemote bean) {

    ...

}
```

除了这些标注位置之外，我们需要了解的第二个内容是 @EJB 标注中提供的属性：

属性名	类型	说明	示例
-----	----	----	----

beanInterface	Class<T>	保存以下目标 EJB 接口类型之一：[本地业务接口、远程业务接口、本地 Home 接口、远程 Home 接口]	@EJB(beanName="StatefulCalculator",beanInterface=StatefulCalculatorRemote.class)
beanName	String	此引用映射到的 EJB。仅适用于在同一应用程序或独立模块内将目标 EJB 定义为声明组件的情况。	@EJB (beanName="StatefulCalculator")
description	String	描述信息	@EJB (description ="引用一个 Bean")
mappedName	String	此 ejb 引用应该映射到的 EJB 组件的特定产品名称。此映射的名称通常是全局 JNDI 名称，但也可以是任何形式的名称。应用服务器不需要支持任何特殊形式或类型的映射名称，也不需要具有使用映射名称的能力。映射的名称与产品有关，并且通常与安装有关。任何对映射名称的使用都不可移植。	@EJB (mappedName ="StatefulCalculator/remote") 注：对于 JBoss 服务器来说，此地址一般对应 JNDI。
name	String	声明组件环境内 (java:comp/env) ejb 引用的逻辑名称。	@EJB(name="StatefulCalculator")

OK，第二个需要看的标注就是 **@Resource**。Resource 标注标记应用程序所需的资源。此标注可以应用于应用程序组件类，或者该组件类的字段或方法。如果将该标注应用于一个字段或方法，那么初始化应用程序组件时容器将把所请求资源的一个实例注入其中。如果将该标注应用于组件类，则该标注将声明一个应用程序在运行时将查找的资源。即使此注释没有被标记为可继承的，部署工具仍然需要检查任意组件类的所有超类，以发现这些超类中所有使用此标注的地方。所有此类的标注实例都指定了应用程序组件所需的资源。注意，此标注可能出现在超类的 **private** 字段和方法上；在这种情况下容器也需要执行注入操作。它所拥有的一些可配置的属性如下：

属性名	类型	说明	示例
authentication	Resource.AuthenticationType	用于此资源的验证类型。可以为表示任何受支持类型的连接工厂的资源指定此方法，不得为其他类型的资源指定。	@Resource(authentication=AuthenticationType.APPLICATION) 或者 @Resource (authentication=AuthenticationType.CONTAINER)
description	String	资源的描述。描述应该使用部署应用程序的系统的默认语言。该描述能够呈现给发布者以帮助帮助他们选择正确的资源。	@ Resource (description ="MySQL 数据源")
name	String	资源的 JNDI 名称。对于字	@ Resource

		段注释，默认值为字段名称。对于方法注释，默认值为与方法对应的 JavaBean 属性名称。对于类注释，没有默认值，必须指定此项。	(name="MySQLDS") 遗憾的发现，JBoss 上不可使用此属性来指定 JNDI 地址
mappedName	String	此资源应该映射到的特定于产品的名称。此资源的名称（由 name 元素定义或为默认值）是使用该资源的应用程序组件的本地名称。（它的名称位于 JNDI <code>java:comp/env</code> 名称空间中。）许多应用程序服务器都提供一种方式将这些本地名称映射到应用程序服务器已知的资源名称。此映射的名称通常是全局 JNDI 名称，但也可以是任何形式的名称。 应用程序服务器不需要支持任何特殊形式或类型的映射名称，也不需要具有使用映射名称的能力。映射的名称与产品有关，并且通常与安装有关。任何对映射名称的使用都不可移植。	@Resource (mappedName="java:DefaultDS") 注：对于 JBoss 服务器来说，此地址一般对应 JNDI。
shareable	boolean	指示是否能在此组件与其他组件之间共享此资源。可以为表示任何受支持类型的连接工厂的资源指定此方法，不得为其他类型的资源指定。	@Resource (shareable =true)
type	Class<T>	资源的 Java 类型。对于字段注释，默认值为字段的类型。对于方法注释，默认值为 JavaBean 属性的类型。对于类注释，没有默认值，必须指定此项。	@Resource(type = String.class , name="MyName")

OK，接下来我们就展示 EJB 互访问和资源注入的一个会话 Bean，它的名字是 `DependencyInjection`。依然在 Java 项目 `EJB3Session` 中新建这些类。我们先来看 EJB 的接口：

session.DependencyInjectionLocal.java

```
package session;

import java.sql.SQLException;
import javax.ejb.Local;
```

```

@Local
public interface DependencyInjectionLocal {
    int add(int a, int b);

    public String testDataSource() throws SQLException;

    public String getMyName();
}

```

。在这个接口中我们定义了三个方法，第一个加法运算将会委托给 `StatefulCalculator` 这个 Bean 来完成；第二个测试数据源则注入我们配置过的 MySQL 数据库；第三个获取名字则从环境属性中获取。

Bean 的实现类的代码清单：

session.DependencyInjection.java

```

package session;

import java.sql.SQLException;

import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.ejb.EJB;
import javax.naming.InitialContext;
import javax.naming.NamingException;

@Stateless
public class DependencyInjection implements DependencyInjectionLocal {
    @EJB(beanName="StatefulCalculator")
    // @EJB(mappedName="StatefulCalculator/remote")
    StatefulCalculatorRemote stfulcalc;

    @Resource(mappedName="MySqlDS")
    javax.sql.DataSource mysqlDS;

    @Resource(name="myName", type=String.class)
    String myName;

    public int add(int a, int b) {
        stfulcalc.setA(a);
        stfulcalc.setB(b);
        return a + b;
    }

    public String testDataSource() throws SQLException {
        java.sql.Connection conn = mysqlDS.getConnection();

```

```

String dsName = conn.getMetaData().getDatabaseProductName();
conn.close();
return dsName;
}

public String getMyName() {
    // EJB 中的 JNDI 访问
    try {
        InitialContext ctx = new InitialContext();
        Object myName = ctx.lookup("beansoft/stringtest/MyName");
        // 打印读取的值
        System.out.println(myName);
    } catch (NamingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return myName;
}
}
}

```

。在此类的开头，列出了三个通过依赖注入进行初始化的变量。`@EJB` 注入 EJB 的示例，既可以通过 `beanName` 来注入，也可以通过 JNDI 的 `mappedName` 来注入（不过不推荐此 `mappedName` 的方式）。`@Resource(mappedName="MySqlDS")` 将一个配置好的全局 JNDI 上的 MySQL 数据源注入下面的变量定义。`@Resource(name="myName", type=String.class)` 则将字符串类型的环境属性注入下面的 `String` 类型的变量。类中的三个方法则分别使用注入的这些变量来完成计算加法，获取数据库产品名称和获取本人名字这样的操作。另外，在 `getMyName()` 方法中，我们特意展示了如何在 EJB 中使用 JNDI。

关于数据源的配置，就不再赘述了。此处需要补充的是环境属性的配置方式，我们需要在 `ejb-jar.xml` 中完成环境属性的配置，代码清单如下所示：

```

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

<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee" version="3.0"
  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">
  <enterprise-beans>
    <session>
      <ejb-name>MySessionBean</ejb-name>
      <business-local>confctest.NewSessionLocal</business-local>

      <business-remote>confctest.NewSessionRemote</business-remote>
      <ejb-class>confctest.NewSessionBean</ejb-class>
      <session-type>Stateless</session-type>
    </session>
  </enterprise-beans>
</ejb-jar>

```

```

</session>

<session>
  <ejb-name>DependencyInjection</ejb-name>
  <env-entry>
    <env-entry-name>myName</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>BeanSoft</env-entry-value>
  </env-entry>
</session>
</enterprise-beans>
</ejb-jar>

```

粗斜体的部分，根据 EJB 的名字，配置了一个环境属性，定义了变量的名字，类型和取值，一般来说只有简单类型的变量才能通过这种方式进行配置。环境属性类似于可以修改的配置参数，通常用来定制 Bean 的行为或者是一些运行的时候才能确定的变量值。

之后我们要测试这个依赖注入的 EJB，在 Web 项目 JBossJNDIWeb 中新建 JSP 页面 **ejbdi.jsp**，代码清单如下：

```

<%@ page language="java" pageEncoding="GBK"%>

<html>
  <head>
    <title>DependencyInjection EJB 访问测试</title>
  </head>
  <body>
    <%
      javax.naming.InitialContext ctx = new
      javax.naming.InitialContext();
      session.DependencyInjectionLocal calc =
      (session.DependencyInjectionLocal)
      ctx.lookup("DependencyInjection/local");
      out.println(calc.add(1, 5) + "<br>");
      out.println(calc.testDataSource() + "<br>");
      out.println(calc.getMyName() + "<br>");
    %>
  </body>
</html>

```

由于依赖注入都是在 EJB 这一端完成的，所以客户端的代码很简单，和访问普通的 EJB 没有什么区别。

最后，先发布 EJB，再发布 Web 项目，启动 JBoss 服务器后在地址栏键入地址 <http://localhost:8080/JBossJNDIWeb/ejbdi.jsp> 进行测试，可以得到正确的输出：

6

MySQL

BeanSoft

。同时在 JBoss 的服务器后台也打印出了之前我们自行绑定的 JNDI 数据：

09:47:46,125 INFO [STDOUT] BeanSoft 刘长炯

这说明运行很成功，很正确，很强大。

16.5.6 拦截器

曾经在第九章 Spring 中，我们用泡 MM 被 FBI 监视的例子来展示 AOP。实际上 AOP 就是拦截器，Java EE 5 也支持拦截器功能。在本节，我们将 FBI 的例子用 EJB3 来重新实现。

OK，不必讲太多的废话，依然在 Java 项目 *EJB3Session* 中新建这些 Bean 类，先看远程接口：

IMan.java

```
package interceptor;

import javax.ejb.Remote;

@Remote
public interface IMan {
    String getName();
    void qq();
    void mm();
    String sayHelp();
}
```

此接口是 EJB 必需定义的，这一点和 Spring 中的 AOP 是不同的。功能还是包括返回姓名，聊 QQ，泡 MM 和发现被监视之时求救。

接下来的是 Bean 类的代码清单：

Man.java

```
package interceptor;

import javax.ejb.Stateful;
import javax.interceptor.Interceptors;

/**
 * 具有聊QQ和泡MM以及求救三个行为的人对象，还有一个用户名属性。
 * @author BeanSoft
 */
@Stateful
@Interceptors({FBI.class})
public class Man implements IMan {
    private String name = "BeanSoft";

    public void qq() {
        System.out.println("我在聊QQ");
    }
}
```

```

public void mm() {
    System.out.println("我在泡MM");
}

public String sayHelp() {
    return "救我, 我是" + getName();
}

public String getName() {
    return name;
}
}

```

很自然的这个 **Bean** 除了拦截器的标注 `@Interceptors({FBI.class})` 指明将会有个 **FBI** 对此类进行监视外, 没有其它更多的内容。

最后一个类就是我们的监视者: **FBI**, 代码清单如下:

FBI.java

```

package interceptor;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

/**
 * 联邦调查局的探员将您的所有行动都记录在案。
 * @author BeanSoft
 */
public class FBI {
    @AroundInvoke
    public Object before(InvocationContext ctx) throws Exception {
        Man man = (Man)ctx.getTarget();
        System.err.println("FBI 发现" + man.getName() + "即将正在进行 " +
            ctx.getMethod().getName() + " 活动。");
        // 禁止张三泡MM
        if(ctx.getMethod().getName().equals("mm")) {
            System.err.println("FBI 将阻止 " + man.getName() + " 泡MM。");
        } else if(ctx.getMethod().getName().equals("sayHelp")) {
            System.err.println("FBI 将欺骗 " + man.getName() + " 的朋友告
            诉他们他很好。");
            return "我是 " + man.getName() + " , 我现在过的很好。";
        } else {
            // proceed() 方法将使原来的方法能够继续执行
            Object object = ctx.proceed();
            System.err.println("FBI 发现" + man.getName() + "已经完成了 " +
                ctx.getMethod().getName() + " 活动。");
            return object;
        }
    }
}

```

```

    }
    return null;
}
}

```

此类和 Spring 中的 AOP 相比，除了类名略有不同外，其它功能似乎大同小异。需要提示大家的是，拦截器方法的签名必须 throws Exception，而不能像 Spring 中那样是 throws Throwable，否则 Bean 将无法发布。

之后我们要测试这个依赖注入的 EJB，在 Web 项目 *JBossJNDIWeb* 中新建 JSP 页面 *fbi.jsp*，代码清单如下：

```

<%@ page language="java" pageEncoding="GBK"%>

<html>
  <head>
    <title>拦截器 EJB 访问测试</title>
  </head>
  <body>
    <%
      javax.naming.InitialContext ctx = new javax.naming.InitialContext();
      interceptor.IMan man = (interceptor.IMan) ctx.lookup("Man/remote");
      man.qq();
      man.mm();
      System.out.println(man.sayHelp());
    %>
  </body>
</html>

```

最后，先发布 EJB，再发布 Web 项目，启动 JBoss 服务器后在地址栏键入地址 <http://localhost:8080/JBossJNDIWeb/fbi.jsp> 进行测试，可以在 JBoss 的 Server 控制台看到正确的输出：

```

23:51:31,343 ERROR [STDERR] FBI 发现BeanSoft即将正在进行 qq 活动。
23:51:31,343 INFO [STDOUT] 我在聊QQ
23:51:31,343 ERROR [STDERR] FBI 发现BeanSoft已经完成了 qq 活动。
23:51:31,343 ERROR [STDERR] FBI 发现BeanSoft即将正在进行 mm 活动。
23:51:31,343 ERROR [STDERR] FBI 将阻止 BeanSoft 泡MM。
23:51:31,343 ERROR [STDERR] FBI 发现BeanSoft即将正在进行 sayHelp 活动。
23:51:31,343 ERROR [STDERR] FBI 将欺骗 BeanSoft 的朋友告诉他们他很好。
23:51:31,343 INFO [STDOUT] 我是 BeanSoft ，我现在过的很好。

```

至此读者已经对拦截器有了一定的了解，实际上项目中也许很少会有机会用到它，而用它来做日志记录，更是不被推荐的，那样只会使服务器的性能严重下降。

16.5.7 EJB 和 Web 服务

在第十五章，我们已经讨论了 Java EE 5 支持的 JSR 标注方式的 Web 服务器开发，当时进行的例子是在 Web 应用中运行的。实际上，标注方式的 Web 服务也支持在 EJB 中开

发，而且不需要额外的配置 Servlet 信息，因此相比之下要简单的多。另外，Netbeans 6 这样的开发工具还支持可视化的 Web 服务设计器。

好了，依然在 Java 项目 *EJB3Session* 中新建这个 Bean 类，而且，按照 Java EE 规范，这个 Bean 类是不需要远程和本地接口的，下面就是我们的 Web 服务器 Bean 类的代码清单：

EJBWebService.java

```
package ejbws;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.ejb.Stateless;

/**
 * EJB 格式的 Web 服务。
 * @author BeanSoft
 */
@WebService
@Stateless
// 务必加上下面这句绑定方式配置，否则会无法传递参数
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJBWebService {

    @WebMethod(operationName = "hello")
    public String hello(@WebParam(name = "username")
        String username) {
        System.out.println(username);

        return "My name is Bean, James Bean. EJB Web 服务测试, 欢迎你: " +
            username;
    }
}
```

此段代码就是一个无状态的 EJB 外加 Web 服务标注，需要提示的是测试中发现如果不加入代码注释中提到的 `SOAPBinding` 的方式设置的话，最后生成的 Web 服务将无法远程调用！

那么此 EJB 发布后，JBoss 会自动创建一个 Web 应用，然后将其发布成 Web 服务，此服务的访问地址是：<http://localhost:8080/EJB3Session/EJBWebService?wsdl>。我们可以用 MyEclipse 的 Web Service Explorer 来对它进行测试，也可以生成客户端代码来访问。

注意：因为 JBoss 并不是严格的支持 Java EE 5 规范，所以测试这个例子的时候，会很遗憾的发现参数根本无法传递过去！也就是说调用将会失败！所以请读者不要用 JBoss 下的 EJB 方式的 Web 服务，当然，别的服务器还是可以的，此处仅仅展示概念。

大家可以看到这种方式的 Web 服务开发的确是非常简单的，这也是 Java EE 5 的一个优势所在。

16.5.8 EJB 最佳实践

第一个问题是：我们的开发中应该选择无状态还是有状态的会话 Bean？

由于我们之前的公司是基于 Weblogic 平台开发的，并根据实际中接触到的其它公司的情况，我个人认为是大多数公司还是倾向于使用无状态会话 Bean 的，因为这样高并发，高负载的情况下，服务器的负担会少很多。反过来有状态 Bean 则因为有时服务器并不能确定你是否还需要它，所以虽然你连一次然后关了，但是服务器还得缓存一段时间，直到一个超时后，才销毁（这和 Web 层的会话是一样的，Tomcat 不知道这个客户是否还会重新连接过来，所以 session 需要设置超时），这在大访问量时无疑会造成资源浪费，除非每个开发人员都在不用的时候调用 bean 的 remove 方法。因此，按照一般的建议，就是尽量使用无状态会话 Bean。这就跟医院一样，大多数病人来看的都是感冒发烧，但专家难求，而普通医生一抓一大把，服务器就是医院，您要减轻它们的负担，就只能多用普通医生，而不是简单的感冒也要动用专家来看。

第二个问题是：EJB 的代码有什么限制？

前面我们的例子过于简单，实际上 EJB 中的 Java 代码不是想写什么就写什么的，为了服务器的安全和资源分配着想，有很多东西是不可以在 EJB 中写的，例如在服务器上创建并显示一个 Swing 的窗口。以下是你应该回避使用的一些 Java 特色，并且在你的 EJB 组件的实现代码中要严格限制它们的使用：

- 1.使用 static，非 final 字段。建议你在 EJB 组件中把所有的 static 字段都声明为 final 型的。这样可以保证前后一致的运行期语义，使得 EJB 容器有可以在多个 Java 虚拟机之间分发组件实例的灵活性。

- 2.使用线程同步原语来同步多个组件实例的运行（也不要再在 EJB 中操作线程，或者创建线程，这样有可能会导导致服务器的线程管理混乱）。避免这个问题，你就可以使 EJB 容器灵活的在多个 Java 虚拟机之间分发组件实例。

- 3.使用 AWT 函数完成键盘的输入和显示输出。约束它的原因是服务器方的商业组件意味着提供商业功能而不包括用户界面和键盘的 I/O 功能。

- 4.使用文件访问/java.io 操作。EJB 商业组件意味着使用资源管理器如 JDBC 来存储和检索数据而不是使用文件系统 API。同时，部署工具提供了在部署描述器（descriptor）中存储环境实体，以至于 EJB 组件可以通过环境命名上下文用一种标准的方法进行环境实体查询。所以，使用文件系统的需求基本上是被排除了。

- 5.监听和接收 Socket 连接，或者用 socket 进行多路发送。EJB 组件并不意味着提供网络 socket 服务器功能，但是，这个体系结构使得 EJB 组件可以作为 socket 客户或是 RMI 客户并且可以和容器所管理的环境外面的代码进行通讯。

- 6.使用映象 API 查询 EJB 组件由于安全规则所不能访问的类。这个约束加强了 Java 平台的安全性。

- 7.欲创建或获得一个类的加载器，设置或创建一个新的安全管理器，停止 Java 虚拟机，改变输入、输出和出错流。这个约束加强了安全性同时保留了 EJB 容器管理运行环境的能力。

- 8.设置 socket 工厂被 URL's ServerSocket,Socket 和 Stream handler 使用。避免这个特点，可以加强安全性同时保留了 EJB 容器管理运行环境的能力。

- 9.使用任何方法启动、停止和管理线程。这个约束消除了与 EJB 容器管理死锁、线程和并发问题的责任相冲突的可能性。

通过限制使用 10—16 几个特点，你的目标是堵上一个潜在的安全漏洞：

- 10.直接读写文件描述符。

- 11.为一段特定的代码获得安全策略信息。
- 12.加载原始的类型。
- 13.访问 Java 一般角色所不能访问的包和类。
- 14.在包中定义一个类。
- 15.访问或修改安全配置对象（策略、安全、提供者、签名者和实体）。
- 16.使用 Java 序列化特点中的细分类和对象替代。
- 17.传递 `this` 引用指针作为一个参数或者作为返回值返回 `this` 引用指针。你必须使用 `SessionContext` 或 `EntityContext` 中的 `getEJBObjct()` 的结果。

16.6 开发实体 Bean

本节将对 EJB 3 中的实体 Bean 开发进行介绍。如前所述，Java EE 5 中已经正式去掉了实体 Bean，转而使用 JPA。因此我们现在所说的实体 Bean，是不能像会话 Bean 那样发布到 JNDI 上去的，而一般的做法则是使用 JPA + 会话 Bean 的方式来实现远程访问数据的能力。在第十三章中，我们已经详细介绍了 JPA 的开发，唯一不同的是当时没有进行容器环境下的开发模式的介绍。在容器环境下（一般是位于 EJB 服务器中，或者完整支持 Java EE 5 的服务器），创建 `EntityManagerFactory` 或者 `EntityManager` 可以不用通过 `Persistence` 类来进行，而使用标注即可，一般其代码如下所示：

```
@PersistenceContext
    EntityManager em;
@PersistenceUnit
    EntityManagerFactory emf;
```

因此，本章中的实体 Bean 开发，就是在会话 Bean 中注入这些持久化单元，然后进行实体的增删改查操作。所幸的是 MyEclipse 6 支持反向工程生成实体 Bean 代码的功能。

16.5.1 使用反向工程生成 EJB 3 实体 Bean

在进行开发之前，请按照 16.4.3.2 JNDI 访问数据源一节的内容，配置好数据源。当然，不同的服务器会有不同的配置方式，这里介绍的只是 JBoss 下面的方式。

在本节，我们将创建一个简单的实体 Bean。要完成的功能很简单，就是访问 MySQL 数据库中的 `Student` 表。

首先，选择菜单 **File > New > EJB Project**，来启动创建 EJB 项目的向导。在 **Project Name** 中输入 `EJB3Entity`，然后在 **J2EE Specification Level** 下点中单选钮 `Java EE 5.0 -EJB 3`，注意一定要保持复选框 `Add support for Entity beans` 的选中状态，这样才能开发实体 Bean，此过程如图 16.23 所示。在刚才的操作中，我们使用的是 EJB 3，同时添加了对开发实体 Bean 的支持。如果读者希望自行开发 EJB 2，那么选中对应的版本 `J2EE 1.4` 即可，不过具体过程比较繁琐，因为 MyEclipse 对 EJB 2 的开发实在支持的太差了，几乎就是手写代码！

接着点击对话框的 **Next** 按钮进入第二页的设置，如图 16.25 所示。上侧的 **JPA Persistence** 一栏用来设置 JPA 相关的信息。在 **Persistence Unit Name** 中输入持久化单元的名字，保持默认值即可，当然也可做一些修改，不过如果只有一个持久化单元的话，这个值的意义不是很大。**JNDI Data Source** 则输入数据源的 JNDI 地址，这里用的是我们先前配置好的 `MySqlDS`，注意大小写不要写错了，一定要保持一致。复选框 **Use Java**

Transaction APIs 则可以让我们使用服务器自带的 Java 事务服务。对话框下侧的 **Design-time Persistence Tools Configuration** 则是 MyEclipse 自带的设置信息，主要是为了将数据源和 Database Explorer 视图中的数据库连接关联起来，与 JPA 技术没有任何关系，它主要是便于反向工程生成 JPA 代码。首先选中 **Driver** 为 *mysql5*，接着需要点击按钮 **Update List**，之后才可以在 **Catalog/Schema** 下拉框选中 *test*，最后点击向导对话框的 **Finish** 按钮即可完成项目的创建。

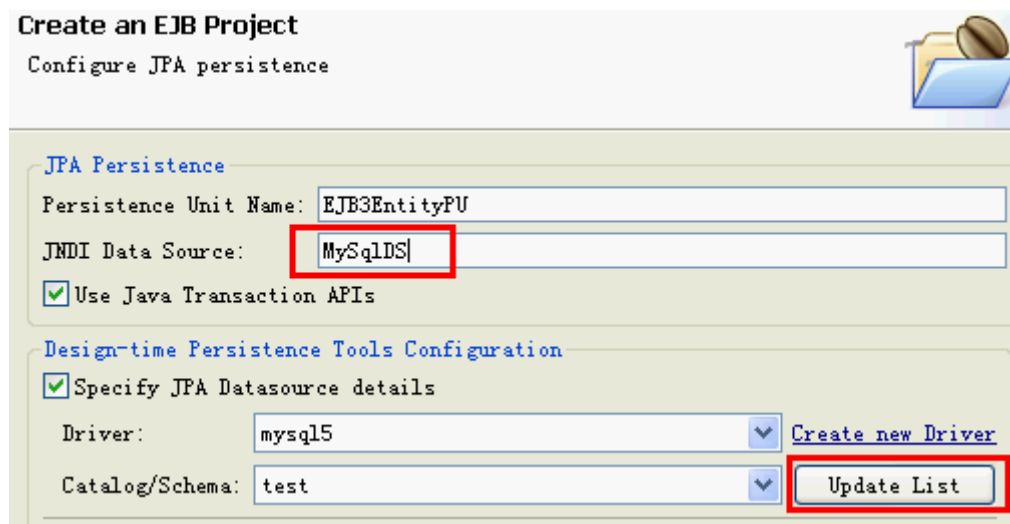


图 16.25 创建支持 JPA 的 EJB 项目向导第二页

项目创建完成后，即可看到目录结构如图 16.26 所示。里面唯一多的文件就是 META-INF/persistence.xml。其代码清单如下所示：

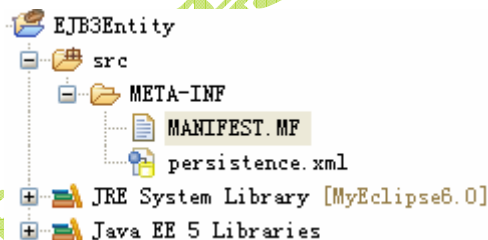


图 16.26 EJB 3 实体 Bean 项目目录结构

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">


  <persistence-unit name="EJB3EntityPU" transaction-type="JTA">
    <jta-data-source>MySqlDS</jta-data-source>
  </persistence-unit>

</persistence>
```

。将此代码和 JPA 一章的进行对比，我们可以发现几个不同，首先是事务类型

(transaction-type) 是 JTA，其次是数据库连接配置现在用的是 `jta-data-source` (只能用于 EJB 服务器中)，需要指定 JNDI 地址，最后，就是不需要在此处指定持久化提供者，也即 provider 了，因为每个支持 Java EE 5 的服务器都会默认提供一个 JPA 的默认实现，也就是 provider，例如 JBoss 中用的就是 Hibernate，当然我们也可以根据情况更换这里的提供者。

接着我们使用 MyEclipse 来反向工程生成实体 Bean，它可以让我们几乎不写一行代码就生成实体 Bean 以及对应的 Session Bean 所组成的 DAO 类，并更新配置文件。

首先打开 **MyEclipse Database Explorer** 透视图。切换透视图有两种办法，如何切换请参考 [3.1.3 透视图 \(Perspective\) 切换器](#)。一种比较快办法是如那一节介绍的，点击工具栏上的  按钮可以显示多个透视图供切换，如图 3.3 所示，然后单击其中的 **MyEclipse Database Explorer** 即可切换到此透视图；另一种办法是选择菜单 **Window > Open Perspective > Other > MyEclipse Database Explorer** 来显示打开透视图对话框，然后点击 **OK** 按钮。

接着选中 **DB Browser** 视图中的数据库连接 `mysql5` (根据您的情况，也可以使用 Derby, Oracle 等等其它数据库，但是必须和 [13.3.3](#) 节中 JPA 项目配置的数据库连接一致)，点击并展开数据库里面的树状表结构，直到看到你希望处理的数据库表为止，单击选中表。**注意：**你可以选中或者一个多个要处理的表。在这个例子中要找的表是 **Student**。接着点击右键在上下文菜单中选择 **EJB 3 Reverse Engineering...**，这将启动 **EJB3 Reverse Engineering** 向导，此过程如图 16.27 所示。

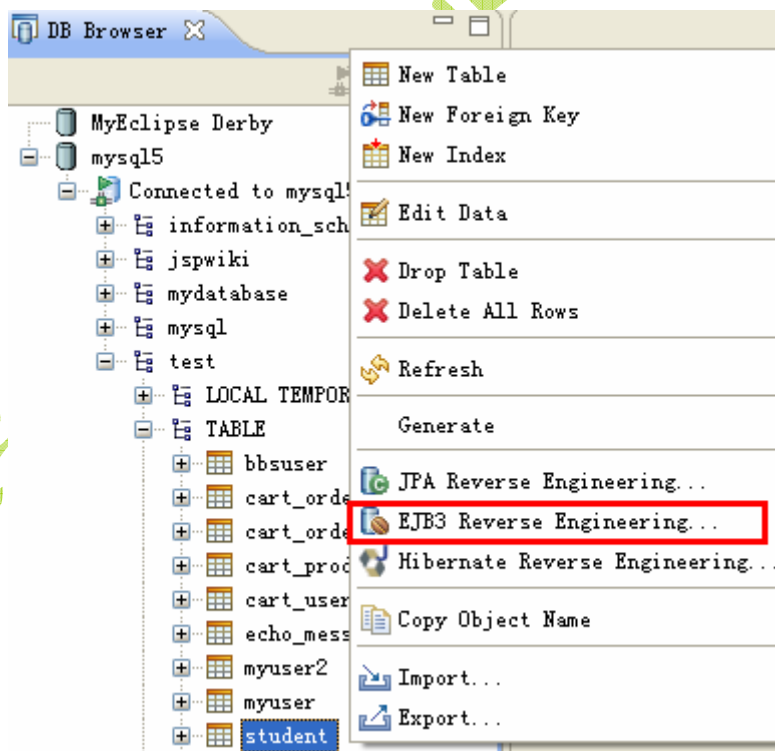


图 16.27 启动 EJB 3 反向工程向导

启动的反向工程向导第一页设置如图 16.28 所示，基本上和 [13.3.5 使用反向工程快速生成 JPA 实体类和 DAO](#) 一节的内容无甚差异。在此页需要设置的内容主要是多了个 **Entity Facade** (会话正面) 的选项，其实就是要不要生成一个无状态的会话 Bean 来提供远程访

问实体类的类似于 DAO 的功能，我们已经说过了，EJB 中的实体 Bean 已经消失，不再直接具有被远程访问的能力，只能通过 Session Bean + JPA 来实现远程访问功能了。此选项下有下述的几个复选框，其意义为：**Enable pagination for query results**，对查询结果启用分页；**Generate precise findBy methods**，生成精确的 findBy 查找方法；最后面的两个选项是和 EJB 有关的，分别是 **Generate local interface**——生成本地接口和 **Generate remote interface**——生成远程接口。读者按照本图进行设置即可，如果有兴趣的可以点击 Next 按钮进入后两页的高级设置，不过您会发现后两页设置和之前 JPA 一章的内容一模一样。在本例中，只需要在第一页点击 **Finish** 按钮，就可以完成向导。

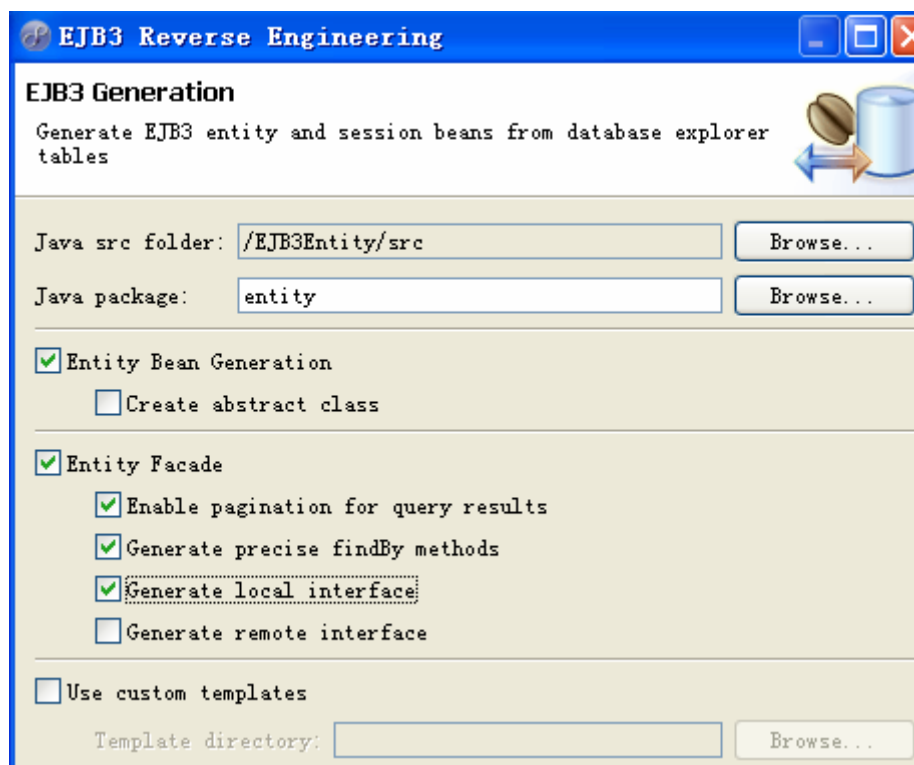


图 16.28 EJB 3 反向工程向导第一页

对话框关闭后，会显示一个正在生成代码的进度框，结束后，就可以看到如图 16.29 所示的项目目录结构了。在此目录中多了四个类，其中 **LogUtil** 是一个进行日志操作的辅助工具类；而 **Student** 则是实体类；**StudentFacade** 则是一个无状态的会话 Bean，它实现的接口是 **StudentFacadeLocal**，这二者合并形成了一个可供远程访问的 DAO 模式的类。由于是处于容器环境中，所以此向导结束后不会向 **persistence.xml** 中加入任何内容，因为运行的时候服务器会扫描发布的 EJB 的 jar 文件中是否有实体类，然后自动的会对它们进行处理。因为一个 EJB 的 jar 文件中的类是有限的，所以可以用这种方式很快的扫描完毕，而不必担心面临 Java SE 方式时，会面临成千上亿个 Java 类需要扫描的情况，就好像假设带有 JPA 标注的实体类是长有六个手指的人，那么在一个屋子（jar 文件）里判断有几个这样的人，显然要比跑遍全国找出所有六指人要方便快速的多。

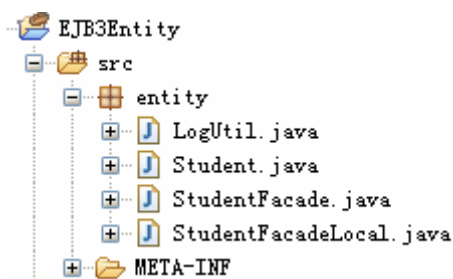


图 16.29 生成代码的目录结构

让我们依次简单的看看这些类的代码，首先是 **LogUtil.java**，这个类简单的对 JDK 自带的 **Logging**(日志功能)包进行了封装，提供了一个获取日志记录器 (**getLogger()**) 和打印日志 (**log(String,Level,Throwable)**) 的方法。代码清单如下：

```
package entity;

import java.util.logging.Level;
import java.util.logging.Logger;
/**
 * @author MyEclipse Persistence Tools
 */
public class LogUtil {

    private static final Logger logger;

    static {
        logger = Logger.getLogger("EJB3EntityPU");
        logger.setLevel(Level.ALL);
    }

    public static void log(String info, Level level, Throwable ex) {
        logger.log(level, info, ex);
    }

    public static Logger getLogger() {
        return logger;
    }
}
```

关于实体类 **Student.java**，它和第 13 中生成的实体类一模一样，就不再赘述了。

接着是本地接口（如果选中了生成远程接口的话，两者的接口定义内容都是相同的），它里面定义了一些方法，相当于第 13 中生成代码中的 **DAO**，至于类似于这样的参数的意义：**int... rowStartIdxAndCount**，读者回头去看 **13.3.5** 节的讨论即可。我们来看一下主要的方法列表，虽然这些方法和 **JPA** 一章的内容几乎相同：

方法	说明
----	----

save()	保存实体
delete()	删除实体
update()	更新实体
findById()	根据主键(ID)查找实体
findByProperty()	根据属性查找实体 (带分页功能)
findByUsername()	根据用户名查找实体 (带分页功能)
findByPassword()	根据密码查找实体 (带分页功能)
findByAge()	根据年龄查找实体 (带分页功能)
findAll()	找出所有实体 (带分页功能)

。为了便于读者对比，我在此处列出其代码并将其中的注释翻译成中文，接下来就是接口 **StudentFacadeLocal.java** 的代码清单：

```

package entity;

import java.util.List;
import javax.ejb.Local;

/**
 * StudentFacade的本地接口.
 *
 * @author MyEclipse Persistence Tools
 */
@Local
public interface StudentFacadeLocal {
    /**
     * 对先前未保存过的 Student 实体进行首次保存操作。
     * 所有后续的对此实体的持久化操作应该使用 #update()
     * 方法。
     *
     * @param entity
     *      需要持久化的 Student 实体
     * @throws RuntimeException
     *      当操作失败时抛出异常
     */
    public void save(Student entity);

    /**
     * 删除已经持久化的 Student 实体。
     *
     * @param entity
     *      需要删除的 Student 实体
     * @throws RuntimeException
     *      当操作失败时抛出异常
     */
    public void delete(Student entity);

```

```

/**
 * 将已经保存过的Student实体更新，并将它或者其副本返回调用者
 * .返回副本仅仅会在JPA持久化后台层没有跟踪此被更新的实体时发生。
 *
 * @param entity
 *         需要更新的 Student 实体
 * @returns Student 持久化过的 Student 实体实例，可能不是传入的对象
 * @throws RuntimeException
 *         当操作失败时抛出异常
 */
public Student update(Student entity);

public Student findById(Integer id);

/**
 * 查找出所有给定属性值的 Student 实体。
 *
 * @param propertyName
 *         需要查询的 Student 属性名字
 * @param value
 *         需要匹配的属性值
 * @param rowStartIdxAndCount
 *         可选的 int 动态参数。rowStartIdxAndCount[0] 指定了查询结果
 *         集合中读取数据的起始下标（分页开始数）。
 *         rowStartIdxAndCount[1] 指定了返回结果的最大数目（分页结束数）。
 * @return List<Student> 查询结果数据
 */
public List<Student> findByProperty(String propertyName, Object
value,
    int... rowStartIdxAndCount);

public List<Student> findByUsername(Object username,
    int... rowStartIdxAndCount);

public List<Student> findByPassword(Object password,
    int... rowStartIdxAndCount);

public List<Student> findByAge(Object age, int...
rowStartIdxAndCount);

/**
 * 列出所有的 Student 实体。
 *

```

```

* @param rowStartIdxAndCount
*     可选的 int 动态参数。rowStartIdxAndCount[0] 指定了查询结果
*     集合中读取数据的起始下标（分页开始数）。
*     rowStartIdxAndCount[1] 指定了返回结果的最大数目（分页结束数）。
* @return List<Student> 所有 Student 实体
*/
public List<Student> findAll(int... rowStartIdxAndCount);
}

```

。请读者此时打开第 [13.3.5 使用反向工程快速生成 JPA 实体类和 DAO](#) 一节的内容，将 StudentDAO 类的代码中的注释和此处的会话 Bean 接口的注释对比，可以看到一个最大的区别，就是注释中少了需要您进行事务操作代码的提示信息，这是为何？因为 EJB 的每个方法都会自动在事务中执行，因此我们使用 EJB 时，无须再考虑其中保存和操作数据时候的事务问题，容器会自动打开事务并进行提交，当然，出错时会自动回滚，这也是容器环境和普通 Java 程序区别的一个地方。

接下来，我们要看的就是实现类，也就是无状态 Bean 的代码，因为其中的注释和接口类中的重复，因此为了节省篇幅，我们就删掉了。下面就是 StudentFacade 的代码清单：

```

package entity;

import java.util.List;
import java.util.logging.Level;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

/**
 * Facade for entity Student.
 *
 * @see entity.Student
 * @author MyEclipse Persistence Tools
 */
@Stateless
public class StudentFacade implements StudentFacadeLocal {
    // property constants
    public static final String USERNAME = "username";
    public static final String PASSWORD = "password";
    public static final String AGE = "age";

    @PersistenceContext
    private EntityManager entityManager;

    public void save(Student entity) {
        LogUtil.log("saving Student instance", Level.INFO, null);
    }
}

```

```
try {
    entityManager.persist(entity);
    LogUtil.log("save successful", Level.INFO, null);
} catch (RuntimeException re) {
    LogUtil.log("save failed", Level.SEVERE, re);
    throw re;
}
}

public void delete(Student entity) {
    LogUtil.log("deleting Student instance", Level.INFO, null);
    try {
        entity = entityManager.getReference(Student.class,
entity.getId());
        entityManager.remove(entity);
        LogUtil.log("delete successful", Level.INFO, null);
    } catch (RuntimeException re) {
        LogUtil.log("delete failed", Level.SEVERE, re);
        throw re;
    }
}

public Student update(Student entity) {
    LogUtil.log("updating Student instance", Level.INFO, null);
    try {
        Student result = entityManager.merge(entity);
        LogUtil.log("update successful", Level.INFO, null);
        return result;
    } catch (RuntimeException re) {
        LogUtil.log("update failed", Level.SEVERE, re);
        throw re;
    }
}

public Student findById(Integer id) {
    LogUtil
        .log("finding Student instance with id: " + id, Level.INFO,
            null);
    try {
        Student instance = entityManager.find(Student.class, id);
        return instance;
    } catch (RuntimeException re) {
        LogUtil.log("find failed", Level.SEVERE, re);
        throw re;
    }
}
```

```

    }
}

@SuppressWarnings("unchecked")
public List<Student> findByProperty(String propertyName,
    final Object value, final int... rowStartIdxAndCount) {
    LogUtil.log("finding Student instance with property: " +
propertyName
        + ", value: " + value, Level.INFO, null);
    try {
        final String queryString = "select model from Student model
where model."
            + propertyName + "= :propertyValue";
        Query query = entityManager.createQuery(queryString);
        query.setParameter("propertyValue", value);
        if (rowStartIdxAndCount != null && rowStartIdxAndCount.length
> 0) {
            int rowStartIdx = Math.max(0, rowStartIdxAndCount[0]);
            if (rowStartIdx > 0) {
                query.setFirstResult(rowStartIdx);
            }

            if (rowStartIdxAndCount.length > 1) {
                int rowCount = Math.max(0, rowStartIdxAndCount[1]);
                if (rowCount > 0) {
                    query.setMaxResults(rowCount);
                }
            }
        }
        return query.getResultList();
    } catch (RuntimeException re) {
        LogUtil.log("find by property name failed", Level.SEVERE, re);
        throw re;
    }
}

public List<Student> findByUsername(Object username,
    int... rowStartIdxAndCount) {
    return findByProperty(USERNAME, username, rowStartIdxAndCount);
}

public List<Student> findByPassword(Object password,
    int... rowStartIdxAndCount) {
    return findByProperty(PASSWORD, password, rowStartIdxAndCount);
}

```

```

    }

    public List<Student> findByAge(Object age, int...
rowStartIdxAndCount) {
        return findByProperty(AGE, age, rowStartIdxAndCount);
    }

    @SuppressWarnings("unchecked")
    public List<Student> findAll(final int... rowStartIdxAndCount) {
        LogUtil.log("finding all Student instances", Level.INFO, null);
        try {
            final String queryString = "select model from Student model";
            Query query = entityManager.createQuery(queryString);
            if (rowStartIdxAndCount != null && rowStartIdxAndCount.length
> 0) {
                int rowStartIdx = Math.max(0, rowStartIdxAndCount[0]);
                if (rowStartIdx > 0) {
                    query.setFirstResult(rowStartIdx);
                }

                if (rowStartIdxAndCount.length > 1) {
                    int rowCount = Math.max(0, rowStartIdxAndCount[1]);
                    if (rowCount > 0) {
                        query.setMaxResults(rowCount);
                    }
                }
            }
            return query.getResultList();
        } catch (RuntimeException re) {
            LogUtil.log("find all failed", Level.SEVERE, re);
            throw re;
        }
    }
}

```

在这个类中，估计唯一需要解释的代码，就是这段：

```

@PersistenceContext
private EntityManager entityManager;

```

那么这段代码实际上就是调用 `Persistence` 类然后创建一个 `EntityManager` 的实例，还记得我们以前的代码嘛：

```
emf = Persistence.createEntityManagerFactory("HelloJPAPU");
```

因为现在是容器环境，而且在 `persistence.xml` 中只配置了一个持久化单元，因此容器分析配置文件后，会自动调用这样一段代码来创建持久化单元，不用指定持久化单元名字，那如

果有多个持久化单元配置时怎么办呢？很简单，因为 `@PersistenceContext` 标注有一个属性，专门可以用来创建指定名称的持久化单元。这个属性就是 `unitName`，类型为 `String`，表示持久性单元的名称。如果指定了 `unitName` 元素，则可以在 JNDI 中访问的实体管理器的持久性单元必须拥有相同的名称。例如上面的标注在我们的例子中也可以写成：

```
@PersistenceContext(unitName="EJB3EntityPU")
private EntityManager entityManager;
```

`EJB3EntityPU` 是位于配置文件中的创建的持久化单元的名字，这个是我们刚创建项目的时候就指定了的。至于这个类的代码的别的部分的内容，就不再赘述了，参考第 13 章的内容即可。

16.5.2 调整生成的配置文件和实体类

由于实体 `Bean` 除了运行环境外，别的方面和 `JPA` 没有任何区别，因此读者可以按照 `JPA` 的相关知识来进行修改。这里就不得不重复一下以前的内容了，需要修改主键生成器。

和第 13 章开发 `JPA` 的时候一样，默认生成的实体类代码很多时候都不符合我们的要求，必须加以修改。实体类的代码在上一节我们已经列出，现在我们考虑的是要对它的主键进行修改，加入自动生成主键值的功能，采用的生成策略是 `IDENTITY`，可以配合 `MySQL` 的自增字段 (`increment`) 来使用。下面已粗斜体列出新加入的代码，位于类 `entity.Student` 中：

```
// Property accessors
@Id
@Column(name = "id", unique = true, nullable = false, insertable =
true, updatable = true)
@javax.persistence.GeneratedValue(strategy=javax.persistence.GenerationType.I
DENTITY)
public Integer getId() {
    return this.id;
}
```

。关于此标注的详细意义，我们已经在 [13.1.3.3 实体类及标注](#) 一节详细讨论过了，读者可参考那一章的介绍。如果不设置的话，默认情况下，`JPA` 持续性提供程序选择最适合于基础数据库的主键生成器类型，其默认值为 `GenerationType.AUTO`，一般情况下不会出现问题，不过最好还是指定一下。

`EJB` 发布后，只能看到无状态 `Bean` 的 JNDI 地址：

```
+ StudentFacade (class: org.jnp.interfaces.NamingContext)
| + local (proxy: $Proxy71 implements interface
entity.StudentFacadeLocal, interface org.jboss.ejb3.JBossProxy)
```

。实体 `Bean` 是不会有通过 JNDI 访问的机会的。

16.5.3 编写并运行测试代码

之后我们要测试这个 `Student` 的实体 `EJB`，在 `Web` 项目 `JBossJNDIWeb` 中新建 JSP 页面 `entityclient.jsp`，只是简单的测试了一下保存数据，代码清单如下：

```
<%@ page language="java" pageEncoding="GBK"%>
<html>
```

```

<head>
  <title>StudentFacade EJB 访问测试</title>
</head>
<body>
  <%
    javax.naming.InitialContext ctx = new
javax.naming.InitialContext();
    entity.StudentFacadeLocal dao = (entity.StudentFacadeLocal)
ctx
        .lookup("StudentFacade/local");
    entity.Student entity = new entity.Student();
    entity.setUsername("实体 Bean test");
    entity.setPassword("ejb + jpa");
    entity.setAge(20);
    dao.save(entity);
    ctx.close();
  %>
</body>
</html>

```

最后，先发布 EJB，再发布 Web 项目，启动 JBoss 服务器后在地址栏键入地址 <http://localhost:8080/JBossJNDIWeb/entityclient.jsp> 进行测试，运行一次后检查数据库即可发现新插入的记录，同时还可以看到 JBoss 服务器的日志信息：

```

00:33:47,234 ERROR [STDERR] 2008-4-21 0:33:47 entity.LogUtil log
信息: saving Student instance
00:33:47,563 ERROR [STDERR] 2008-4-21 0:33:47 entity.LogUtil log
信息: save successful

```

当然，读者可以继续对 StudentFacade 中的其它类进行更进一步的测试，或者是在 Servlet 或者 Struts 的 Action 类中访问。

另外，由于 JPA 和 EJB3 的实体 Bean 本质上属于同样内容，所以，读者依然可以切换到 MyEclipse Java Persistence 这个透视图来进行相关的实体 Bean 的修改。

16.6 消息驱动 Bean

16.6.1 JMS 简介

在介绍消息驱动 Bean (Message Driven Bean, MDB) 之前，需要先熟悉下 JMS。JMS (Java Message Service, Java 消息服务) 是一组 Java 应用程序接口 (Java API)，它提供创建、发送、接收、读取消息的服务。由 Sun 公司和它的合作伙伴设计的 JMS API 定义了一组公共的应用程序接口和相应语法，使得 Java 程序能够和其他消息组件进行通信。

在前面大家已经接触到了会话和实体 Bean 的开发和运行，以及访问，然而，由于 EJB 本身的缺陷，有下面一些问题无法解决：

性能 当服务器处理客户端对 EJB 的请求时，客户端必须等待（或者说阻塞），只有当服务器端完成了处理后，客户端才能得到接过，之后才能继续工作。

可靠性 当调用 EJB 时，服务器必须是可连接且正常运行的。如果服务器崩溃或者网络故障，客户端就得不到预期的结果了。

对多方通话的支持 EJB 的限制是在任意时候一个客户端只能和一台服务端对话，不能视线多对多的连接功能。

然而，消息服务可以解决这些问题。当然，实际上消息服务在 Java 和 EJB 出现之前就已经存在了，Sun 的贡献就是把消息服务器的 API 规范化，标准化，减少程序员的负担。消息服务通过下列几个方式解决了这些问题：

性能 当执行请求时，消息服务的客户端不必等待或者阻塞。换句话说支持异步消息服务，当然也支持同步方式的。例如外地用户给您邮寄了一台笔记本电脑，显然在你收到他的通知后，你不必天天吃住在邮局等待这台笔记本的到来。您要做的，只是在一段时间后去邮局领取笔记本即可，邮局这个中间的消息服务器会帮你妥善保管收到的任何信息，当然，超时之后，它们有权利按照规定来对笔记本电脑（消息）进行合适的处理，例如：抛弃或者返回给投递者。

可靠性 如果消息中间件支持担保发送（guaranteed delivery）的话，那么可以发送消息，并明确知道此消息是否会到达目的地，即使在不能访问消息使用者时也如此。其实邮局就是这样的机构，投递者只需要去邮局查询投递情况即可，邮局可以告诉投递者消息是否到达，或者到达后无人领取等情况，他无须跑到收件人那里进行询问，否则那样投递者何不亲自送去，通过邮局岂不是多此一举。

对多方通话的支持 大多数消息中间件能够接受多个发送者的消息，然后将这些消息广播到多个接受者。这样就可以实现多对多的通信。

Java 消息服务分为两部分：一部分是应用编程接口（API），用来编写发送和接受消息的代码，消息类型包括简单文本（TextMessage）、可序列化的对象（ObjectMessage）、属性集合（MapMessage）、字节流（BytesMessage）、原始值流（StreamMessage）等；另一部分是服务供应商接口（SPI），在服务供应商接口中可以插入 Java 消息服务驱动程序。因此，对于开发人员来说，主要需要了解 API，不过，读者也要知道，虽然大部分的 Java EE 服务器都支持 JMS，但是，专业的 JMS 服务器一般都是单独购买或者安装，运行的，例如 BEA Tuxedo，IBM WebSphere MQ 等。当然，也有一些开源的 JMS 服务器，JBoss 就内置了一个简单的消息服务器，我们接下来的例子中就会使用它，还有其它的开源的服务器例如 Apache ActiveMQ 等。

Java 消息服务支持两种消息模型：Point-to-Point 消息(P2P)和发布订阅消息（Publish Subscribe messaging，简称 Pub/Sub）。JMS 规范并不要求供应商同时支持这两种消息模型，但开发者应该了解这两种消息模型的优势与缺点。

发布/订阅模式 此模式类似于看电视。很多电视台都在广播自己的信号，而同时又有很多人在观看这些节目。不过，需要区别的是，JMS 中通信是双向的，一般来说接收者同时也可能是一个发布者，因为它收到消息后需要回复。订阅者（收看者）注册自己感兴趣的特定主题，发布者则创建发布到所有订阅者的消息。这时候有个关键的术语就叫 Topic（主题），一般来说对应服务器的某个 JNDI 地址，发送者向里面发送信息，接受者则订阅此 Topic 即可。此模式如图 16.30 所示。

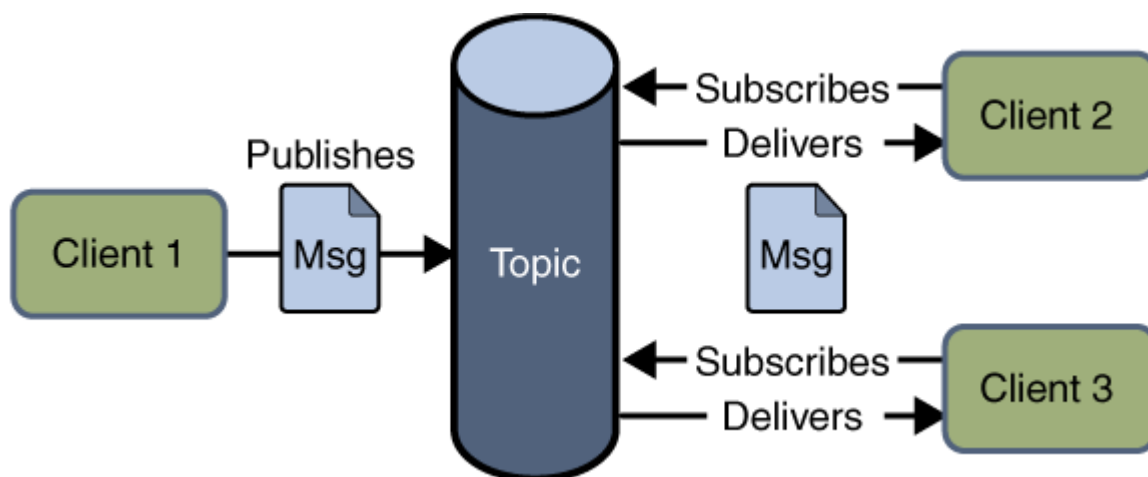


图 16.30 JMS 发布订阅模式

点对点方式 此模式类似于语音信箱，可以给此信息发送语音留言，接受者会阅读完毕后删除此信息。这种模式下，一条消息只能被一个人接受，而且也只能阅读一次。这可以认为是发布/订阅模式的简化为一个电台一个接收者时候的情形。此时对应的术语叫 Queue(队列)，两个人同时连接到同一队列，才能收发消息。此模式如图 16.31 所示。

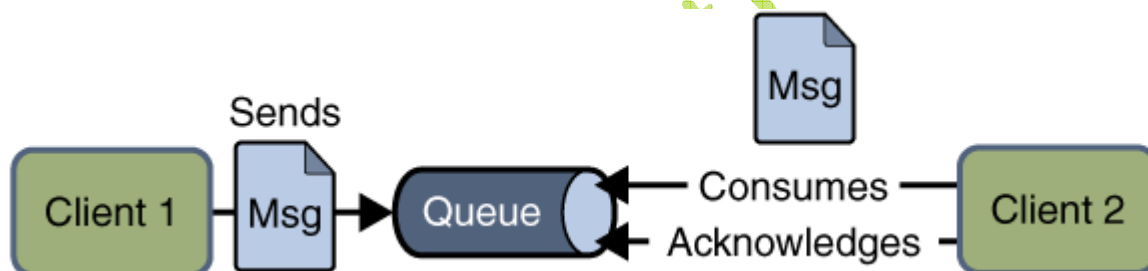


图 16.31 JMS 点对点模式

JMS 消息由以下部分组成：

- **Header (头)** - 所有消息都支持相同的头字段集合。头字段包含客户端和提供者用于标识和路由消息的值。
- **Property (属性)** - 每个消息都包含一个内置工具，用于支持应用程序定义的属性值。属性为支持应用程序定义的消息过滤提供了一种有效的机制。
- **Body (正文)** - JMS API 定义了几种消息正文类型，涵盖了当前使用的大多数消息传递样式。

消息正文

JMS API 定义了五种消息正文类型：

- **Stream (流)** - `StreamMessage` 对象的消息正文包含 Java 编程语言原始值流 (“Java 基本类型”)。按顺序填充和读取。
- **Map(映射)** - `MapMessage` 对象的消息正文包含一组名称-值对，其中名称是 `String` 对象，值是 Java 基本类型。可以根据名称按顺序或随机访问这些条目。条目的顺序是不确定的。
- **Text (文本)** - `TextMessage` 对象的消息正文包含 `java.lang.String` 对象。此消息类型可用于传输纯文本消息和 XML 消息。
- **Object (对象)** - `ObjectMessage` 对象的消息正文包含 `Serializable Java` 对象。

- **Byte** (字节) - **BytesMessage** 对象的消息正文包含未解释的字节流。此消息类型可以按字面意义编码正文，以匹配现有的消息格式。在大多数情况下，可以使用更易用的其他正文类型。尽管 **JMS API** 允许将消息属性用于字节消息，但一般不使用它们，因为包含属性可能会影响格式。

消息头

JMSCorrelationID 头字段用于连接一个消息与另一个消息。它通常连接应答消息与请求消息。

JMSCorrelationID 可以保存特定于提供者的消息 ID、特定于应用程序的 **String** 对象，或者提供者-本机 **byte[]** 值。

消息属性

Message 对象包含一个内置工具，用于支持应用程序定义的属性值。实际上，这提供了一种将特定于应用程序的头字段添加到消息的机制。

属性允许应用程序通过消息选择器让 **JMS** 提供者使用特定于应用程序的标准为自己选择或过滤消息。

属性名称必须遵守消息选择器标识符的规则。属性名称不得为 **null** 或空字符串。如果设置了属性名称，并且它为 **null** 或空字符串，则必定抛出 **IllegalArgumentException**。

属性值可以为 **boolean**、**byte**、**short**、**int**、**long**、**float**、**double** 和 **String**。属性值在发送消息前设置。客户端收到消息时，其属性值是只读模式。如果客户端尝试在此时设置属性，则抛出 **MessageNotWriteableException**。调用了 **clearProperties** 后便可以读取和写入属性。注意，头字段与属性不同。头字段永远不会是只读模式。

属性值可以复制消息正文中的值，也可以不复制。尽管 **JMS** 没有定义属性行为策略，但应用程序开发人员应该注意，**JMS** 提供者处理消息正文中的数据可能比处理消息属性中的数据更高效。要获得最佳性能，应用程序应该仅在需要自定义消息头时才使用消息属性。这样做的主要原因是为了支持自定义消息选择。

消息属性支持以下转换表格。必须支持做了记号的那些情况。没有做记号的那些情况必须抛出 **JMSEException**。如果基本类型的 **valueOf** 方法不接受 **String** 作为基本类型的有效表示形式，则 **String** 向基本类型的转换可能抛出运行时异常。作为行类型写入的值可以作为列类型读取。

	boolean	byte	short	int	long	float	double	String
boolean	X							X
byte		X	X	X	X			X
short			X	X	X			X
int				X	X			X
long					X			X
float						X	X	X
double							X	X
String	X	X	X	X	X	X	X	X

除了特定于类型的属性设置/获取方法外，**JMS** 还提供了 **setObjectProperty** 和 **getObjectProperty** 方法。这些方法都支持使用对象化原始值的同一属性类型集。这是为了能够在执行时（而不是编译时）确定属性类型。它们支持相同的属性值转换。

setObjectProperty 方法接受类 **Boolean**、**Byte**、**Short**、**Integer**、**Long**、**Float**、**Double** 和

String 的值。尝试使用任何其他类都会抛出 `JMSEException`。

`getObjectProperty` 方法仅返回类 `Boolean`、`Byte`、`Short`、`Integer`、`Long`、`Float`、`Double` 和 `String` 的值。

属性值的顺序是不确定的。若要迭代消息的属性值，请使用 `getPropertyNames` 获取属性名称枚举，然后使用各种属性获取方法获取它们的值。可使用 `clearProperties` 方法删除消息的属性。此方法将使消息的属性集为空。

获取未设置名称的属性值将返回 `null` 值。只有 `getStringProperty` 和 `getObjectProperty` 方法才会返回 `null` 值。尝试将 `null` 值作为基本类型读取将视同于调用基本类型相应的 `valueOf(String)` 转换方法，并以 `null` 值作为参数。

JMS API 为 JMS 定义的属性保留 JMSX 属性名称前缀。Java 消息服务规范中定义了完整的属性集合。新的 JMS 定义属性可能会添加到以后的 JMS API 版本中。对这些属性的支持是可选的。`String[] ConnectionMetaData.getJMSXPropertyNames` 方法返回连接支持的 JMSX 属性名称。

无论这些 JMSX 属性是否受连接支持，都可以在消息选择器中引用它们。如果消息中不存在这些属性，则处理它们的方式与任何其他不存在的属性相同。

在规范中定义为“在发送过程中由提供者设置”的 JMSX 属性对于消息生成方和使用方都适用。规范中定义为“在接收过程中由提供者设置”的 JMSX 属性只适用于使用方。

`JMSXGroupID` 和 `JMSXGroupSeq` 是客户端分组消息时应该使用的标准属性。所有提供者都必须支持它们。除非特别说明，否则 JMSX 属性的值和语义是未定义的。

JMS API 为特定于提供者的属性保留 `JMS_vendor_name` 属性名称前缀。每个提供者都定义其自己的 `vendor_name` 值。这是 JMS 提供者使每个 JMS 客户端能够使用其特殊的消息服务 (per-message services) 所采用的机制。

特定于提供者的属性的用途是，提供在单个 JMS 应用程序中集成 JMS 客户端与提供者-本机客户端时所需的特殊功能。它们不能用于 JMS 客户端之间的消息传递。

16.6.2 JMS 编程模型

在这里我们简单讨论下 JMS 的编程模型，JMS 的使用者一共要完成两个任务，一是发送消息，二是接收消息。此过程如图 16.32 所示。

客户端发送消息一般有以下步骤：

1. 初始化 JNDI;
2. 通过 JNDI 查找 `ConnectionFactory` (连接工厂);
3. 使用 `ConnectionFactory` 创建 `JMS Connection` (连接);
4. 使用连接创建会话 (`Session`);
5. 通过 JNDI 查找目的地(主题或者队列, `Topic/Queue`);
6. 创建消息生产者/队列发送者 (`TopicPublisher/QueueSender`);
7. 创建消息;
8. 给消息设置一些额外属性 (可选);
9. 发送消息;
10. 关闭会话。

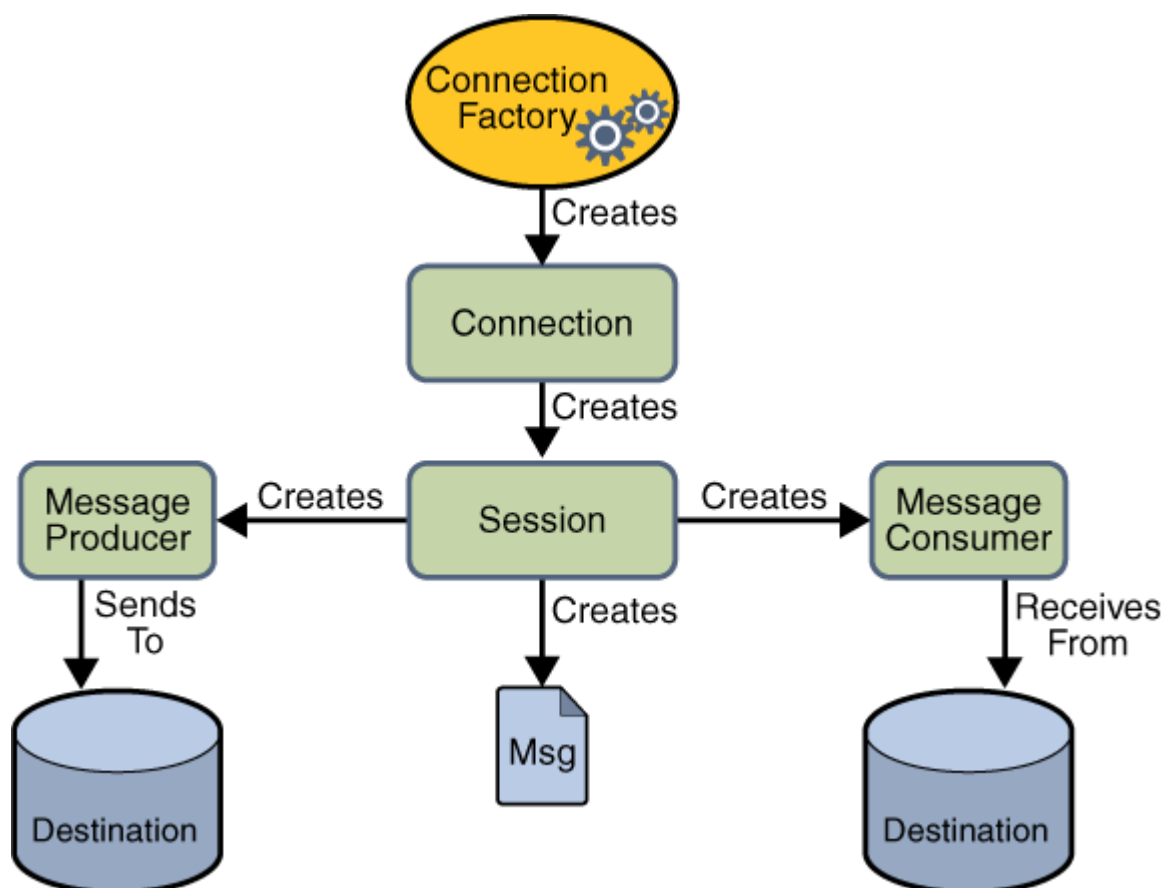


图 16.32 JMS 编程模型

如果要接收消息，其过程的前四步都是一样的，只有从第 6 步开始的过程不一样，如下所示：

6. 创建消息消费者或者主题订阅者（MessageConsumer/TopicSubscriber）；
7. 打开连接（connection.start()）；
8. 接收并处理消息（可以使用消息监听器 MessageListener）；
9. 关闭会话。

16.6.3 JMS 点对点模式编程

首先需要启动 JBoss 服务器才能完成我们的练习。启动 JBoss 服务器后，查看 JNDI 的 Global 下面可以看到下面几个和 JMS 有关的地址：

TopicConnectionFactory (class: org.jboss.naming.LinkRefPair) -> 此地址为主题连接工厂

ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory) -> 普通连接工厂

```

+- queue (class: org.jnp.interfaces.NamingContext)
|   +- A (class: org.jboss.mq.SpyQueue)
|   +- testQueue (class: org.jboss.mq.SpyQueue)
|   +- ex (class: org.jboss.mq.SpyQueue)
|   +- DLQ (class: org.jboss.mq.SpyQueue)
|   +- D (class: org.jboss.mq.SpyQueue)
|   +- C (class: org.jboss.mq.SpyQueue)
  
```

```
| +- B (class: org.jboss.mq.SpyQueue)
```

此处列出的都是队列，用于点对点模式的发送，一般使用 `testQueue` 即可。

```
+ topic (class: org.jnp.interfaces.NamingContext)
| +- testDurableTopic (class: org.jboss.mq.SpyTopic)
| +- testTopic (class: org.jboss.mq.SpyTopic)
| +- securedTopic (class: org.jboss.mq.SpyTopic)
```

此处列出的是主题，注意默认情况下只能使用 `testTopic`。

接着我们要在项目 `JBossJNDITest` 中完成测试（当然读者也可以使用发布到 JBoss 的 Web 项目中的 JSP 页面来进行测试，例如项目 `JBossJNDIWeb`），按照顺序，首先来编写并运行消息的发送端，代码清单如下：

P2PSender.java

```
package jms;

import javax.naming.*;
import javax.jms.*;

/**
 * 点对点方式消息发送演示程序，JBoss 4.2 服务器版本
 *
 * @author 刘长炯
 * @version 1.0 2007-04-22
 */
public class P2PSender {

    public static void main(String[] args) throws Exception {
        // 初始化 JNDI
        Context ctx = new InitialContext();

        // 1: 通过 JNDI 查找 ConnectionFactory (连接工厂)
        QueueConnectionFactory factory = (QueueConnectionFactory) ctx
            .lookup("ConnectionFactory");

        // 2: 使用 ConnectionFactory 创建 JMS 连接
        QueueConnection connection = factory.createQueueConnection();

        // 3: 使用连接创建会话
        QueueSession session = connection.createQueueSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);

        // 4: 通过 JNDI 查找目的地(主题或者队列)
        Queue queue = (Queue) ctx.lookup("queue/testQueue");

        // 5: 创建消息生产者/队列发送者 (TopicPublisher/QueueSender)
        QueueSender publisher = session.createSender(queue);
```



```

// 6: 创建文本消息
TextMessage msg = session.createTextMessage();
msg.setText("简单的点对点文本消息:欢迎阅读!");
// 7. 可以给消息设置一些额外属性(可选)
msg.setStringProperty("BookName", "MyEclipse6 Java 开发中文教程");

// msg.setJMSType("first");//额外属性: 设置JMS Type 等

// 8. 发送消息
publisher.send(msg);
// 9. 关闭会话
session.close();
}
}

```

这段代码已经加入了足够多的注释,我想不需要做过多的解释了。不过还是提醒大家注意一下,连接工厂对应的 JNDI 地址是 *ConnectionFactory*,而队列的对应 JNDI 地址是 *queue/testQueue*,千万不要写入其它的地址了。

在这段代码的第 3 部分,通过 *QueueConnection* 创建 *QueueSession* 处,它和下面提到的 *Connection* 类中的 *createSession()* 的参数是相同的,方法签名如下:

QueueConnection.createQueueSession(boolean transacted, int acknowledgeMode) throws JMSEException

或者 **Connection.createSession(boolean transacted, int acknowledgeMode) throws JMSEException**

创建 *QueueSession* 对象。

参数 *transacted* — 指示会话是否是事务性的

参数 *acknowledgeMode* — 指示使用方或客户端是否将确认收到的任何消息;如果会话是事务性的,则忽略此参数。合法值包括 *Session.AUTO_ACKNOWLEDGE*、*Session.CLIENT_ACKNOWLEDGE* 和 *Session.DUPS_OK_ACKNOWLEDGE*。这几个常量的解释如下:

AUTO_ACKNOWLEDGE

通过此确认模式,当会话从对 *receive* 的调用成功返回时,或在会话已调用的用于处理消息的消息侦听器成功返回时,会话会自动确认客户端的消息接收。

CLIENT_ACKNOWLEDGE

通过此确认模式,客户端通过调用消息的 *acknowledge* 方法确认已使用的消息。确认已使用的消息将确认该会话已使用的所有消息。

使用客户端确认模式时,客户端可能会在尝试处理消息时构建大量的未确认消息。*JMS* 提供者应向管理员提供一种方法来限制客户端超限,这样客户端就可以避免资源耗尽和由于其使用的某些资源临时阻塞而导致的失败。

DUPS_OK_ACKNOWLEDGE

此确认模式指示会话延迟确认消息的传送。这可能在 *JMS* 提供者失败的情况下导致传送某些重复消息,因此只有能允许重复消息的使用方才应使用此模式。使用此模式可以通过最大限度地减少会话为防止重复所做的工作,从而减少会话开销。

另外这段代码中的第 7 部分,给消息设置额外属性,有一个方法:

setStringProperty(String name, String value) throws JMSEException

说明如下：

将带指定名称的 `String` 属性值设置到消息中。

参数 `name` — `String` 属性的名称

参数 `value` — 要设置的 `String` 属性值

那么根据 JBoss 下的实践，发现 `name` 不能为中文，否则将不合法，读者请注意。这个取值有什么用呢？可以用来做消息过滤，类似于加入了频道信息。关于消息过滤接下来的代码部分会介绍到。另外，如果没有特殊需要，也可以完全不用设置属性信息。

出了上面这段发送消息的代码外，还可以用另一种方式来发送，代码示例如下：

Producer.java

```
package jms;

import javax.jms.*;
import javax.naming.*;

public class Producer {

    public static void main(String[] args) throws NamingException,
JMSEException {
        Context ctx = new InitialContext();
        ConnectionFactory cf = (ConnectionFactory) ctx
            .lookup("ConnectionFactory");
        Destination des = (Destination) ctx.lookup("queue/testQueue");

        Connection con = cf.createConnection();
        Session session = con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(des);
        TextMessage msg = session.createTextMessage();

        msg.setJMSType("first");

        msg.setText("发送测试2");
        producer.send(msg);

        con.close();
        System.out.println("发送结束!");
    }
}
```

这段代码也可以发送成功，相比起来代码相对简洁一些，不用特意使用 `Queue` 相关的类。大家可以根据自己的喜好选择发送的方式。

然后我们要编写接收消息的代码，这段代码相对复杂一些，清单如下：

P2PReceiver.java

```
package jms;

import javax.naming.*;
import javax.jms.*;

/**
 * 点对点方式消息接收演示程序, JBoss 4.2 服务器版本。
 * 请先运行接收程序, 然后再发送消息。
 * @author 刘长炯
 * @version 1.0 2007-04-22
 */
public class P2PReceiver {

    public static void main(String[] args) throws NamingException,
    JMSEException {
        // 初始化 JNDI
        Context ctx = new InitialContext();

        // 1: 通过 JNDI 查找 ConnectionFactory (连接工厂)
        QueueConnectionFactory factory = (QueueConnectionFactory) ctx
            .lookup("ConnectionFactory");

        // 2: 使用 ConnectionFactory 创建 JMS 连接
        QueueConnection connection = factory.createQueueConnection();

        // 3: 使用连接创建会话
        Session session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);

        // 4: 通过 JNDI 查找目的地(主题或者队列)
        Destination des = (Destination) ctx.lookup("queue/testQueue");

        // 5: 创建消息消费者 (MessageConsumer), 带有消息过滤器
        MessageConsumer consumer = session.createConsumer(des,
            "BookName LIKE '%MyEclipse6%'",
            false);

        // 也可使用这样的代码接收所有消息
        // consumer = session.createConsumer(des);

        // 6: 打开连接, 开始接收
        connection.start();

        // 7: 处理收到的消息
        Message message = consumer.receive();
    }
}
```

```

        if(message instanceof TextMessage){
            TextMessage msg =(TextMessage) message;
            String text=msg.getText();
            System.out.println(text);
        }

        // 也可以使用消息监听器方式来处理
// consumer.setMessageListener(new MyMessageListener());

        // 8: 关闭连接
        connection.close();
    }
}

```

这段代码中，需要解释的第一个地方，就是 `MessageConsumer consumer = session.createConsumer(des, "BookName LIKE '%MyEclipse6%'", false);`

这段代码，这里的 `BookName LIKE '%MyEclipse6%'` 类似于一段 SQL 的查询条件，然而却不完全相同，它的正式称呼叫 `MessageSelector`——消息选择器。`BookName` 这个属性是通过前面发送消息时使用代码片段 `msg.setStringProperty("BookName", "MyEclipse6 Java 开发中文教程");` 来定义的；如果是没有这样定义的消息，就不会被接收到。我们参考 Sun 的文档对消息选择器做一些介绍，仅供大家参考，因为专业的 JMS 开发还需要参考相关的服务器说明文档。

JMS 消息选择器 允许客户端通过头字段引用和属性引用指定感兴趣的消息。只传送头和属性值匹配选择器的那些消息。这意味着消息是否被传送取决于使用的 `MessageConsumer`（请参考 `QueueReceiver` 和 `TopicSubscriber`）。

消息选择器不能引用消息正文值。消息的头字段值和属性值替换了消息选择器中对应的标识符后，如果选择器求值为 `true`，则消息选择器匹配该消息。

消息选择器是一个 `String`，其语法是以 SQL92 条件表达式语法子集为基础的。如果消息选择器的值是一个空字符串，则该值视为 `null`，表示没有任何消息选择器可供消息使用。

消息选择器的求值顺序是按优先级从左到右计算。可以使用括号来更改此顺序。预定义选择器的字面值和操作符名称在这里用大写表示；实际上它们是不区分大小写的。

选择器可以包含：

- 字面值：
 - 字符串字面值使用单引号括起，单引号则用两个单引号表示；例如，`'literal'` 和 `'literal's'`。这些字面值与 Java 编程语言中的字符串字面值一样，都使用 `Unicode` 字符编码。
 - 一个精确的数字字面值是一个不带小数点的数值，如 `57`、`-957` 和 `+62`；支持 `long` 型的数值范围。精确的数字字面值使用 Java 编程语言中的整数字面值语法。
 - 一个近似的数字字面值是使用科学计数法表示的数值，如 `7E3` 和 `-57.9E2`，或者是带小数点的数值，如 `7.`、`-95.7` 和 `+6.2`；支持 `double` 型

的数值范围。近似的数字字面值使用 Java 编程语言中的浮点字面值语法。

- 布尔字面值为 TRUE 或 FALSE。
- 标识符:
 - 标识符是一个任意长度的字母数字序列，其首字符必须是字母。字母是对其调用 `Character.isJavaLetter` 方法可返回 `true` 的任何字符。这包括 '_' 和 '\$'。字母或数字是对其调用 `Character.isJavaLetterOrDigit` 方法可返回 `true` 的任何字符。
 - 标识符不能使用名称 NULL、TRUE 和 FALSE。
 - 标识符不能是 NOT、AND、OR、BETWEEN、LIKE、IN、IS 或 ESCAPE。
 - 标识符是头字段引用或属性引用。消息选择器中属性值的类型对应于设置属性时使用的类型。如果引用了消息中不存在的属性，则其值为 NULL。
 - 当属性用于消息选择器表达式时，不可应用适用于属性获取方法的转换。例如，假设将属性设置为一个字符串值，如下所示：

```
myMessage.setStringProperty("NumberOfOrders", "2");
```

以下消息选择器中的表达式将求值为 `false`，因为字符串不能用于算术表达式：

```
"NumberOfOrders > 1"
```

- 标识符是区分大小写的。
- 消息头字段引用限于 `JMSDeliveryMode`、`JMSPriority`、`JMSMessageID`、`JMSTimestamp`、`JMSCorrelationID` 和 `JMSType`。`JMSMessageID`、`JMSCorrelationID` 和 `JMSType` 值可以为 `null`，如果为 `null`，则按 NULL 值处理它们。
- 以 'JMSX' 开头的任何名称是一个 JMS 定义的属性名称。
- 以 'JMS_' 开头的任何名称是一个特定于提供者的属性名称。
- 不以 'JMS' 开头的任何名称是一个特定于应用程序的属性名称。
- 空白字符与 Java 编程语言中的定义相同：空格、水平制表符、换页符和行结束符。
- 表达式:
 - 选择器是一个条件表达式；求值为 `true`，则选择器匹配；求值为 `false` 或“未知”，则选择器不匹配。
 - 算术表达式由其他算术表达式、算术运算、标识符（其值视为数字字面值处理）和数字字面值构成。
 - 条件表达式由其他条件表达式、比较运算和逻辑运算构成。
- 支持使用标准括号 () 对表达式的求值顺序进行排序。
- 按优先级从高到低排序的逻辑运算符：NOT、AND、OR
- 比较运算符：=、>、>=、<、<=、<>（不等于）
 - 只能比较相同类型的值。有一种例外情况，即比较精确数字字面值和近似数字字面值是有效的；所需的类型转换由 Java 编程语言中的数值提升 (numeric promotion) 规则定义。如果尝试比较不同类型的值，运算的值为 `false`。如果两个类型值求值都为 NULL，则表达式的值未知。
 - 字符串和布尔值的比较只能使用 = 和 <>。当且仅当两个字符串具有相同字符序列时它们才相等。
- 按优先级从高到低排序的算术运算符:
 - +、-（一元）
 - *、/（乘和除）
 - +、-（加和减）

- 算术运算必须使用 Java 编程语言中的数值提升。
- *arithmetic-expr1* [NOT] BETWEEN *arithmetic-expr2* AND *arithmetic-expr3* (比较运算符)
 - "age BETWEEN 15 AND 19" 等于 "age >= 15 AND age <= 19"
 - "age NOT BETWEEN 15 AND 19" 等于 "age < 15 OR age > 19"
- *identifier* [NOT] IN (*string-literal1*, *string-literal2*,...) (比较运算符, 其中 *identifier* 具有 String 或 NULL 值)
 - "Country IN ('UK', 'US', 'France')" 对于 'UK' 为 true, 对于 'Peru' 为 false ; 它等效于表达式 "(Country = 'UK') OR (Country = 'US') OR (Country = 'France')"
 - "Country NOT IN ('UK', 'US', 'France')" 对于 'UK' 为 false, 对于 'Peru' 为 true ; 它等效于表达式 "NOT ((Country = 'UK') OR (Country = 'US') OR (Country = 'France'))"
 - 如果 IN 或 NOT IN 运算的标识符为 NULL, 则运算的值未知。
- *identifier* [NOT] LIKE *pattern-value* [ESCAPE *escape-character*] (比较运算符, 其中 *identifier* 具有 String 值。 *pattern-value* 是一个字符串字面值, 其中 '_' 表示任何单个字符, '%' 表示任何字符序列, 包括空序列, 所有其他字符都表示自身。可选的 *escape-character* 是一个单字符字符串字面值, 其字符用于转义 *pattern-value* 中 '_' 和 '%' 的特殊含义。)
 - "phone LIKE '12%3'" 对于 '123' 或 '12993' 为 true, 对于 '1234' 为 false。
 - "word LIKE 'l_se'" 对于 'lose' 为 true, 对于 'loose' 为 false。
 - "underscored LIKE '_%' ESCAPE \" 对于 '_foo' 为 true, 对于 'bar' 为 false。
 - "phone NOT LIKE '12%3'" 对于 '123' 或 '12993' 为 false, 对于 '1234' 为 true。
 - 如果 LIKE 或 NOT LIKE 运算的 *identifier* 为 NULL, 则运算的值未知。
- *identifier* IS NULL (比较运算符, 测试 null 头字段值或缺失的属性值)
 - "prop_name IS NULL"
- *identifier* IS NOT NULL (比较运算符, 测试非 null 头字段值或属性值是否存在)
 - "prop_name IS NOT NULL"

JMS 提供者必须在出现消息选择器时验证其语法的正确性。提供语法不正确的选择器的方法必定导致抛出 `JMSEException`。JMS 提供者还可以在出现选择器时有选择地提供一些语义检查。并非所有语义检查都可以在出现消息选择器时执行, 因为属性类型是未知的。以下消息选择器选择消息类型为 car、颜色为 blue、重量超过 2500 磅的消息:

```
"JMSType = 'car' AND color = 'blue' AND weight > 2500"
```

Null 值

如上所述, 属性值可以为 NULL。包含 NULL 值的选择器求值表达式由 SQL92 NULL 语法定义。下面简要描述这些语法。

SQL 将 NULL 值视为“未知”。比较或计算一个未知值总是得出一个未知值。

IS NULL 和 IS NOT NULL 运算符将一个未知值分别转换为 TRUE 和 FALSE 值。

布尔运算符使用下表定义的三值逻辑:

AND 运算符的定义

AND	T	F	U
-----	---	---	---

T	T	F	U
F	F	F	F
U	U	F	U

OR 运算符的定义

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

NOT 运算符的定义

NOT	
T	F
F	T
U	U

特别注意

在消息选择器中使用 `JMSDeliveryMode` 头字段被视为具有值 'PERSISTENT' 和 'NON_PERSISTENT'。

日期和时间值应使用标准 `long` 型毫秒值。消息选择器中包含日期和时间字面值时，它的毫秒值应该是整数字面值。生成毫秒值的标准方法是使用 `java.util.Calendar`。

尽管 `SQL` 支持定点十进制的比较和算术运算，但 `JMS` 消息选择器不支持。所以要将精确数字字面值限制为没有小数点的数字字面值(将带小数点的数字作为近似数值的可选表示形式)。最后，`JMS` 选择器的表达式不支持 `SQL` 注释。

关于消息的处理，除了可以使用 7 的方式先调用 `receive()` 方法，然后再进行判断和处理外，还可以使用消息监听器的方式，这时候需要删除 7 的代码端，改为这样的调用：

```
consumer.setMessageListener(new MyMessageListener());
```

这时候就需要编写一个消息监听器，代码清单如下：

MyMessageListener.java

```
package jms;
import javax.jms.*;

// JMS 监听器
public class MyMessageListener implements MessageListener {

    public void onMessage(Message message) {
        if(message instanceof TextMessage){
            TextMessage msg =(TextMessage) message;
```

```

String text;
try {
    text = msg.getText();
    System.out.println(text);
} catch (JMSEException e) {
    e.printStackTrace();
}
}
}
}
}

```

由此代码接收过程可以看到，使用 JMS 来接收并处理消息的确是非常费劲的，包括建立各式各样的连接，以及设置监听器，等等，因此才出现了后来才介绍的 MDB。

最后，我们就可以来测试收发消息了。这两个类可以不分先后顺序分别运行，其中接收方 *P2PReceiver* 必定能够收到 JMS 的消息，具体过程读者可以自行测试，对每个类选择菜单 **Run > Run** 来运行它即可。之所以能够收到离线消息，是因为 JMS 本身支持这样的离线收发消息的功能，就像大家使用 QQ 聊天时所受到的离线消息一样，不过，虽然发送方可以一次发送多条信息，接收方却只能一条条的接收（包括离线消息），在这里，我们就只能是运行一次又一次 *P2PReceiver* 这个类了。

16.6.4 JMS 发布订阅模式编程

鉴于上一节已经大量介绍了相关的概念和内容，本节内容就简单的多了。在这个例子中，我们只能先打开电视机，然后准备接收电台的节目（因为实际中的情况是电视台不会给单个用户重放节目的），所以需要先编写一个接受者，代码清单如下：

TopicReceiver.java

```

package jms;

import javax.naming.*;
import javax.jms.*;

// 发布订阅模式接受者测试代码
public class TopicReceiver {

    public static void main(String[] args) throws Exception {
        // 初始化 JNDI
        Context ctx = new InitialContext();

        // 1: 通过 JNDI 查找 ConnectionFactory (连接工厂)
        TopicConnectionFactory factory = (TopicConnectionFactory) ctx
            .lookup("TopicConnectionFactory");

        // 2: 使用 ConnectionFactory 创建 JMS 连接
        TopicConnection connection = factory.createTopicConnection();
    }
}

```



```

// 3: 使用连接创建会话
TopicSession session = connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);

// 4: 通过 JNDI 查找目的地(主题)
Topic topicReceiver = (Topic) ctx.lookup("topic/testTopic");

// 5: 创建消息消费者或者主题订阅者 (MessageConsumer/TopicSubscriber)
TopicSubscriber topicSubscriberMyEclipse =
    session.createSubscriber(topicReceiver,
        "BookName LIKE '%MyEclipse6%'",
        false);

// 也可使用这样的代码接收所有消息
//    topicSubscriberMyEclipse =
session.createSubscriber(topicReceiver);

// 6: 开始接收消息
connection.start();
// 7: 处理接收到的消息
Message received = topicSubscriberMyEclipse.receive();
printMessage(received);

// 8: 关闭连接
connection.close();
}
// 打印接收到的消息
static void printMessage(Message msg) throws Exception {
    if(msg instanceof TextMessage) {
        TextMessage message =(TextMessage) msg;
        System.out.println("接收到消息: " + message.getText());
    }
}
}
}

```

然后马上运行此接收程序，等待发送后打印收到的消息退出。需要提示的仍然是 JNDI 地址，以及消息的处理可以使用消息监听器。

接下来，我们需要创建消息发布者，代码清单如下：

TopicSender.java

```

package jms;

import javax.naming.*;
import javax.jms.*;
//发布订阅模式发送者测试代码
public class TopicSender {

```

```

public static void main(String[] args) throws Exception {
    // 初始化 JNDI
    Context ctx = new InitialContext();

    // 1: 通过 JNDI 查找 ConnectionFactory (连接工厂)
    TopicConnectionFactory factory = (TopicConnectionFactory) ctx
        .lookup("TopicConnectionFactory");

    // 2: 使用 ConnectionFactory 创建 JMS 连接
    TopicConnection connection = factory.createTopicConnection();

    // 3: 使用连接创建会话
    TopicSession session = connection.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);

    // 4: 通过 JNDI 查找目的地(主题)
    Topic topic = (Topic) ctx.lookup("topic/testTopic");

    // 5: 创建消息生产者
    TopicPublisher publisher = session.createPublisher(topic);

    // 6: 创建并发送文本消息
    TextMessage msg = session.createTextMessage();
    msg.setText("欢迎阅读书籍 MyEclipse6 Java 开发中文教程!");
    msg.setStringProperty("BookName", "MyEclipse6 Java 开发中文教程");

    publisher.publish(msg);

    // 7: 关闭连接
    connection.close();
}
}

```

。好了，运行此 Java 类发送消息后，即可看到在 *TopicReceiver* 的控制台中打印出如下的信息：

接收到消息：欢迎阅读书籍 MyEclipse6 Java 开发中文教程！

如果收不到消息，*TopicReceiver* 会一直等待，除非手工结束进程内容。

最后，虽然 JBoss 也支持离线的广播（相当于提供视频点播的电视台），然而，很遗憾的是需要进行复杂的设置，此处我们就不再多做讨论了。感兴趣的读者自行阅读示例代码中的 *DurableTopicReceiver.java*，虽然测试通不过，但是大致的接收过程却是如此的（此代码留作如果使用一些容易配置的 JMS 服务器时用）。发送时，也需要发送到对应的 JNDI 地址 *topic/testDurableTopic* 即可。

16.6.5 MDB 简介及 MDB 编程

如前所述，虽然 JMS 的消息发送相对比较容易，然而，如何接收并处理 Message，则相对显得要复杂的多。为此，EJB 推出了消息驱动 Bean (Message Driven Bean，简称 MDB)。MDB 不绑定到 JNDI 地址，也不能被别的 Bean 调用，也不能被客户端调用执行。它的功能，就是在收到消息后自动调用处理方法，简单的说就是一个服务器自动管理消息接收的 MessageListener，其代码结构如下所示：

```
@MessageDriven
```

```
public class MDBImpl implements MessageListener {
    public void onMessage(Message message) {
        System.out.println("A message arrived!");
    }
}
```

。除此之外不需要别的内容。

MDB 的生命周期和无状态会话 Bean 非常相似，其生命周期如图 16.33 所示。其生命周期除了依赖注入外，就是等待接收消息并处理了。右图可以看到 MDB 也支持 @PostConstruct, @PreDestroy 这样的标注。

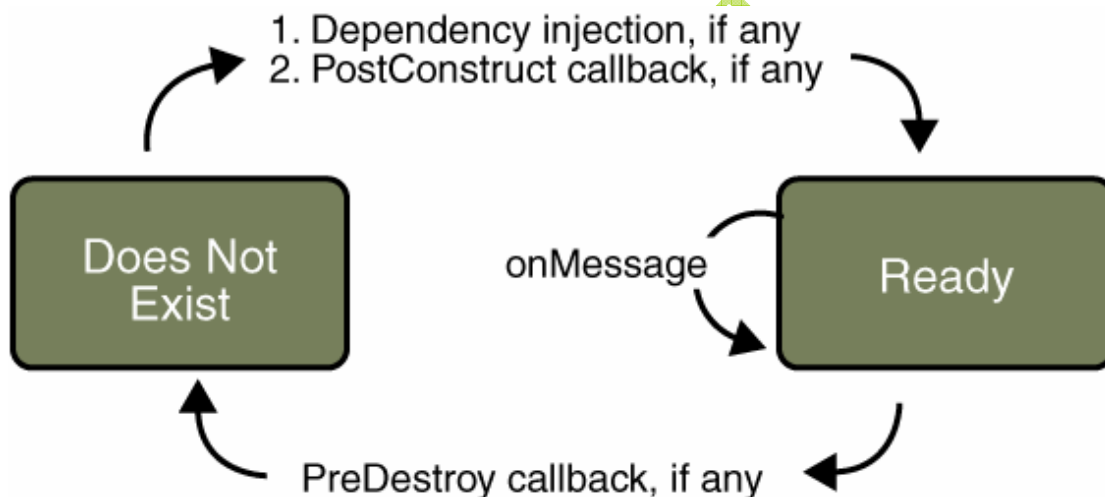


图 16.33 消息驱动 Bean 生命周期

接下来，让我们来开发一个消息驱动 Bean。首先，请参照 16.5.2 开发无状态 Session Bean 一节的操作步骤新建一个 EJB 项目，名为 *EJB3MessageDriven*。接着，选择菜单 **File > New > EJB3 Message Driven Bean**，启动新建消息驱动 Bean 的向导，如图 16.34 所示。按照插图所示设置各个选项即可。在 **Package** 中可以输入自己希望的包名；**Name** 中输入类名，这里设置为 *MyMDB*；**Destination Type** 中选择目标类型为 *Queue* (队列)，也可以根据实际情况选择 *Topic* (主题)。选择完成后，点击 **Finish** 按钮关闭对话框即可结束创建过程。随后我们可以看到生成的 MDB 代码如下所示：

MyMDB.java

```
package mdb;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
```

```

import javax.jms.MessageListener;

@MessageDriven(mappedName = "jms/MyMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
        propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Queue") })
public class MyMDB implements MessageListener {

    public void onMessage(Message arg0) {
        // TODO Auto-generated method stub
    }
}

```

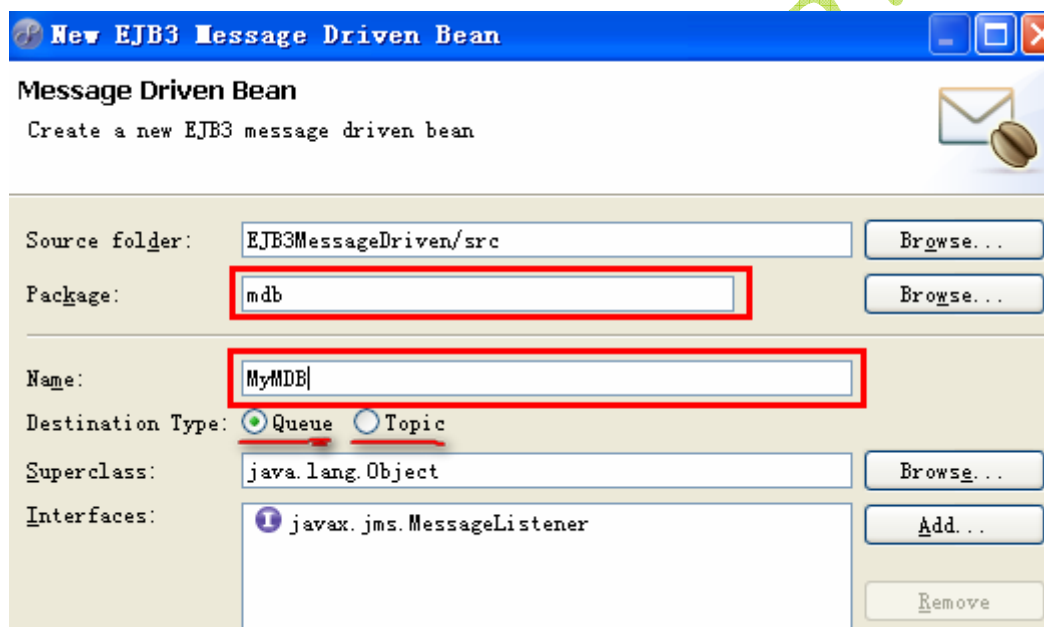


图 16.34 新建消息驱动 Bean 向导

对于这个代码来说，可以解释的地方不多。首先 `mappedName` 这个属性，大家回顾一下 16.5.5 EJB 互访问和资源注入一节的内容即可。关键点在 `activationConfig` 这个属性上，顾名思义，就是激活配置的意思，或者说是激活消息驱动 Bean 的条件。我想经过上面几个小节对 JMS 的介绍，大家已经能够明白已经存在的两个属性的含义了。不过，要想这个 Bean 运行起来，还缺了一个最重要的属性，就是 Topic 的 JNDI 地址，其配置属性为：

```

@ActivationConfigProperty(propertyName = "destination" , propertyValue =
"queue/testQueue"), queue/testQueue 即可指定监听地址。

```

另外，还可以设置消息选择器（这一步是可选的，如果不加则会接收所有发布到此地址的 JMS 消息），其配置方式为：

```

@ActivationConfigProperty(propertyName = "messageSelector" , propertyValue="消息选择表达式")。消息选择表达式可以为这样的值：BookName LIKE '%MyEclipse6%'或者 JMSType='first'等等，详细请参考 16.6.3 JMS 点对点模式编程一节的内容。

```

好了，最后再加上处理 JMS 消息的代码（简单的就是打印出来取值即可），这个修改过的消息驱动 Bean 的代码清单如下所示：

```

package mdb;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(mappedName = "jms/MyMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination",
propertyValue="queue/testQueue"),
    @ActivationConfigProperty(propertyName = "messageSelector",
propertyValue="BookName LIKE '%MyEclipse6%'")
})
public class MyMDB implements MessageListener {

    public void onMessage(Message message) {
        if(message instanceof TextMessage){
            TextMessage msg =(TextMessage) message;
            String text;
            try {
                text = msg.getText();
                System.out.println("收到文本消息: " + text);
            } catch (JMSEException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("收到其它消息: " + message);
        }
    }
}
}

```

现在可以发布这个 MDB 到 JBoss 服务器上，发布完毕后 JNDI 上也是看不到地址的。那么该怎么测试这个 Bean 呢？唯一的办法就是向 queue/testQueue 这个地址发消息，可以运行我们之前在项目 JBossJNDITest 中编写的 P2PSender 类，或者在 JBossJNDIWeb 这个项目中创建一个 jsp 页面用来发送 JMS：

p2pJMSSender.jsp

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="javax.naming.*, java.text.*, javax.jms.*,
java.util.Properties"%>
<%
    // 初始化 JNDI
    Context ctx = new InitialContext();

    // 1: 通过 JNDI 查找 ConnectionFactory (连接工厂)
    QueueConnectionFactory factory = (QueueConnectionFactory) ctx
        .lookup("ConnectionFactory");

    // 2: 使用 ConnectionFactory 创建 JMS 连接
    QueueConnection connection = factory.createQueueConnection();

    // 3: 使用连接创建会话
    QueueSession queueSession =
connection.createQueueSession(false,
        QueueSession.AUTO_ACKNOWLEDGE);

    // 4: 通过 JNDI 查找目的地(主题或者队列)
    Queue queue = (Queue) ctx.lookup("queue/testQueue");

    // 5: 创建消息生产者/队列发送者 (TopicPublisher/QueueSender)
    QueueSender publisher = queueSession.createSender(queue);

    // 6: 创建文本消息
    TextMessage msg = queueSession.createTextMessage();
    msg.setText("简单的点对点文本消息:欢迎阅读!");
    // 7. 可以给消息设置一些额外属性 (可选)
    msg.setStringProperty("BookName", "MyEclipse6 Java 开发中文教程
");
    // msg.setJMSType("first");//额外属性: 设置JMS Type 等

    // 8. 发送消息
    publisher.send(msg);
    // 9. 关闭会话
    queueSession.close();
%> <%@ page contentType="text/html; charset=GBK"%>
<%@ page import="javax.naming.*, java.text.*, javax.jms.*,
java.util.Properties"%>
<%
    // 初始化 JNDI
    Context ctx = new InitialContext();
```

```

// 1: 通过 JNDI 查找 ConnectionFactory (连接工厂)
QueueConnectionFactory factory = (QueueConnectionFactory) ctx
    .lookup("ConnectionFactory");

// 2: 使用 ConnectionFactory 创建 JMS 连接
QueueConnection connection = factory.createQueueConnection();

// 3: 使用连接创建会话
QueueSession queueSession =
connection.createQueueSession(false,
    QueueSession.AUTO_ACKNOWLEDGE);

// 4: 通过 JNDI 查找目的地(主题或者队列)
Queue queue = (Queue) ctx.lookup("queue/testQueue");

// 5: 创建消息生产者/队列发送者 (TopicPublisher/QueueSender)
QueueSender publisher = queueSession.createSender(queue);

// 6: 创建文本消息
TextMessage msg = queueSession.createTextMessage();
msg.setText("简单的点对点文本消息:欢迎阅读!");
// 7. 可以给消息设置一些额外属性(可选)
msg.setStringProperty("BookName", "MyEclipse6 Java 开发中文教程");

// msg.setJMSType("first");//额外属性: 设置JMS Type 等

// 8. 发送消息
publisher.send(msg);
// 9. 关闭会话
queueSession.close();
%>

```

。然后发布此项目后，通过访问 JSP 页面 <http://localhost:8080/JBossJNDIWeb/p2pJMSSender.jsp>，即可发送消息。不管用那种办法，只要消息发出后，都可以看到JBoss控制台的命令行输出下列信息（当然，发送多条会出现多条这样的信息）：

```
17:02:51,562 INFO [STDOUT] 收到文本消息: 简单的点对点文本消息:欢迎阅读!
```

。OK，这样就测试通过了，我们的消息 Bean 也开发成功了。

请读者完成一个小练习，要求 MDB 来接收发送给 Topic 中的消息，并进行测试。

16.7 可嵌入式的 EJB 引擎

随着形势的发展，有的公司推出了单独的可嵌入式（或者说是独立运行的 EJB 引擎），比较著名的有 OpenEJB 和 JBoss 的 Embed EJB 3.0，可以让我们在 Tomcat 这样的本身不

带 EJB 功能的服务器中使用 EJB 技术。

OpenEJB 是一个嵌入式，轻量级 EJB3.0 实现。既可以作为单独服务器使用，也可以嵌入到 Tomcat、JUnit、Eclipse、Intellij、Maven、Ant 和其它任何 IDE 与应用程序中。OpenEJB 被用于 Apple 的 WebObjects 与 Apache 的 Geronimo 应用服务器中。

OpenEJB 问世于 2000 年，其创建者是 David Blevins 和 Richard Monson-Haefel。Blevins 也是 Geronimo 的创建者之一，而 OpenEJB 是 Geronimo 中 EJB 实现的首选。OpenEJB 是 EJB 1.1 规范的第一批开源实现之一。它直接提供了一个远程会话 bean 的实现，并使用 Castor 作为它的容器管理持久性（CMP）实体 bean 的实现。它的下载和使用都是相当的简单，只需要下载 <http://apache.mirror.phpchina.com/openejb/3.0/openejb.war> 这个 war 文件，然后放入 Tomcat 的 webapps 下面即可。相关的文档请参考其官方网站的操作过程即可。

JBoss 的嵌入式 EJB 3 可以通过这个地址下载：http://sourceforge.net/project/showfiles.php?group_id=22866&package_id=132063，然而从其网站我们看到最新版本是 2006 年 12 月 13 日推出的 EJB 3 RC9 Patch1，版本相当的旧，不推荐读者使用，还是使用官方的 JBoss 好了。

16.8 小结

今天，以 Spring+Hibernate 为代表的开源解决方案和以 EJB+JPA 为代表的 Java EE 5 解决方案构成竞争关系，当然，实际上也是可以并存的。读者可以根据自己项目的需要来选择。总的来说，前者开发稍嫌复杂，配置文件多切容易出错，而且高并发高负载条件下的一些问题不容易解决，也不支持集群，这样的例子有参考资料中提到的“将 Spring 用于高并发环境的隐忧”，也有我们之前公司使用 Hibernate 的遇到的问题，当时出了性能问题了，虽然调整解决了，但不管怎么说，因为这些开源软件之前开发的时候并没有考虑高并发和集群的情况，还是比较容易出现问题，尤其是没有经过严格的压力测试。我个人认为，目前做的比较好的软件，依然是商业的。不过，开源软件的好处就是免费，所以如果企业资金不足，可以考虑使用 Linux + Tomcat/Resin + Spring + Hibernate + MySQL 这样的免费搭档来构造公司网站。然而，目前来说，大企业中多数还是走了商业服务器+EJB+Oracle 这样的路子，尤其是需要集群和高并发的情况下，如果技术水平达不到，购买商业服务器还是个比较不错的办法。不过，实际上如果各位是去从事开发，基本上用什么样的开发环境，公司都已经定好了。所以，我的个人建议仅供大家作为参考。

另外，本章内容没有提到 EJB 2.0 的开发。如果读者有需要，可以阅读 MyEclipse 的帮助文档即可完成开发过程，或者使用 JBuilder 2006 这样的开发工具，带有可视化的 EJB 设计器，非常的快捷。另外，也可以使用 Netbeans 6 这样的开发工具。那么 MyEclipse 6 中的 EJB 帮助文档怎么看呢？选择菜单 **Help > Help Contents**，然后在帮助文档浏览窗口的 **Contents** 栏目中选择 **MyEclipse Learning Center > EJB Development > Getting Started > EJB 2.x Tutorial (Using XDoclet)** 即可浏览学习。当然，此处也有 EJB 3 开发方面的官方教程，只不过是内容都是英文的。此文档的访问过程请参考图 16.35。

本章虽然页码比较多，然而需要提醒大家的是完整的 EJB 教程是远远多于此处的内容的。不过，对于普通的开发来说，本章的内容足够大家的使用了，因为我一直都是站在一个开发人员的角度上来介绍 EJB 的开发的，并不是事无巨细的把所有理论点展示一遍。

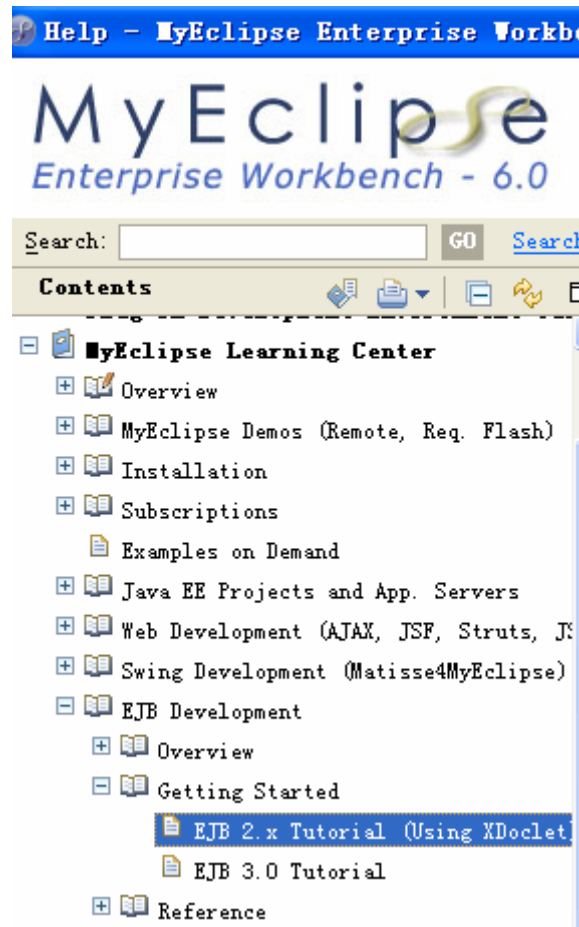


图 16.35 MyEclipse 中的 EJB 开发帮助文档

16.9 参考资料

<http://www.foshanshop.net/> 黎活明著免费电子书《EJB3.0 实例教程》官方网，基于JBoss和Eclipse，包括PDF书籍下载和源代码下载

<http://doc.java.sun.com/DocWeb/> 推荐：Java EE 中文在线JavaDOC

<http://www.blogjava.net/beansoft/archive/2008/04/17/193691.html> Java EE 5 中文 API 文档在线阅读和操作视频

<http://www.blogjava.net/beansoft/archive/2007/08/31/141695.html> MyEclipse 5.5 + JBoss 开发视频：配置,开发并运行第一个 EJB 3 项目

<http://openejb.apache.org/> OpenEJB的首页

<http://www.ibm.com/developerworks/cn/java/j-jar/index.html> IBM的教程：JAR 文件解密探索 JAR 文件格式的强大功能

<http://www.theserverside.com/tt/books/wiley/masteringEJB/> 精通EJB英文原版PDF和源代码下载，Mastering Enterprise JavaBeans Third Edition ，Ed Roman 著，需要指出讲的是EJB 2.1，不是 EJB 3

<http://geronimo.apache.org/> 开源免费Java EE服务器 Apache Geronimo

<http://www.ibm.com/developerworks/websphere/zones/was/wasce.html> 免费的 WebSphere Application Server Community Edition，支持Java EE 5

<http://www.bea.com.cn/products/beawebtuxe/index.jsp> BEA Tuxedo

<http://activemq.apache.org/> Apache ActiveMQ

<http://docs.sun.com/app/docs/doc/819-3660/6n5s7klp7?a=view> 英文：Chapter 1 Assembling and Deploying Applications

<http://java.sun.com/javaee/overview/compatibility.jsp> 支持Java EE 5 的服务器列表

http://www.sun.com/software/products/message_queue/index.xml Sun Java System Message Queue

<http://java.sun.com/products/jms/> Java Message Service (JMS) 技术首页

<http://java.sun.com/products/ejb/> Enterprise JavaBeans 技术首页

<http://java.sun.com/javaee/index.jsp> Java EE 技术首页

<http://java.sun.com/j2ee/connector/> J2EE Connector Architecture

<http://java.sun.com/products/indi/> JNDI技术首页

<http://java.sun.com/products/indi/tutorial/> SUN的JNDI教程（英文）

<http://java.sun.com/javaee/5/docs/tutorial/doc/JavaEETutorial.pdf> Sun的官方Java EE教程 (PDF版本)

<http://java.sun.com/javaee/5/docs/tutorial/doc/> 在线阅读Sun的Java EE教程

<http://www.openldap.org/> Open LDAP 开源的LDAP服务器（仅限Linux）

<http://java.sun.com/products/jts/> JTS技术首页

<http://java.sun.com/developer/technicalArticles/WebServices/jaxb/> JAXB 教程（英文）

http://www.blogjava.net/security/archive/2008/04/19/spring_bug.html 将Spring用于高并发环境的隐忧

16.10 术语表

缩写	全称	中文对照
API	Application Programming Interface	应用程序接口（编程接口）
EJB	Enterprise Java Bean	企业 Java Bean
EIS	Enterprise Infomation System	企业信息系统
JNDI	Java Naming and Directory Interface	Java 命名和目录接口
LDAP	Lightweight Directory Access Protocol	轻量级目录访问协议
JSP	JavaServer Page	Java 服务器页面
JSTL	JavaServer Pages Standard Tag Library	JSP 标准标签库
JSF	JavaServer Faces	这个比较难翻译: Face, 脸
JMS	Java Message Service	Java 消息服务
JTA	Java Transaction API	Java 事务服务编程接口
JAF	JavaBeans Activation Framework	JavaBean 激活框架
JAXP	Java API for XML Processing	Java XML 处理编程接口
JAX-WS	Java API for XML Web Services	基于 XML 的 Web 服务 Java 编程接口
JAXB	Java Architecture for XML Binding	Java XML 绑定架构
SAAJ	SOAP with Attachments API for Java	带附件的 SOAP 编程接口
JAXR	Java API for XML Registries	XML 注册器
JCA	Java EE Connector Architecture	Java EE 连接器架构
JDBC	Java Database Connectivity	Java 数据库连接
JPA	Java Persistence API	Java 持久化编程接口
JAAS	Java Authentication and Authorization Service	Java 验证和授权 API