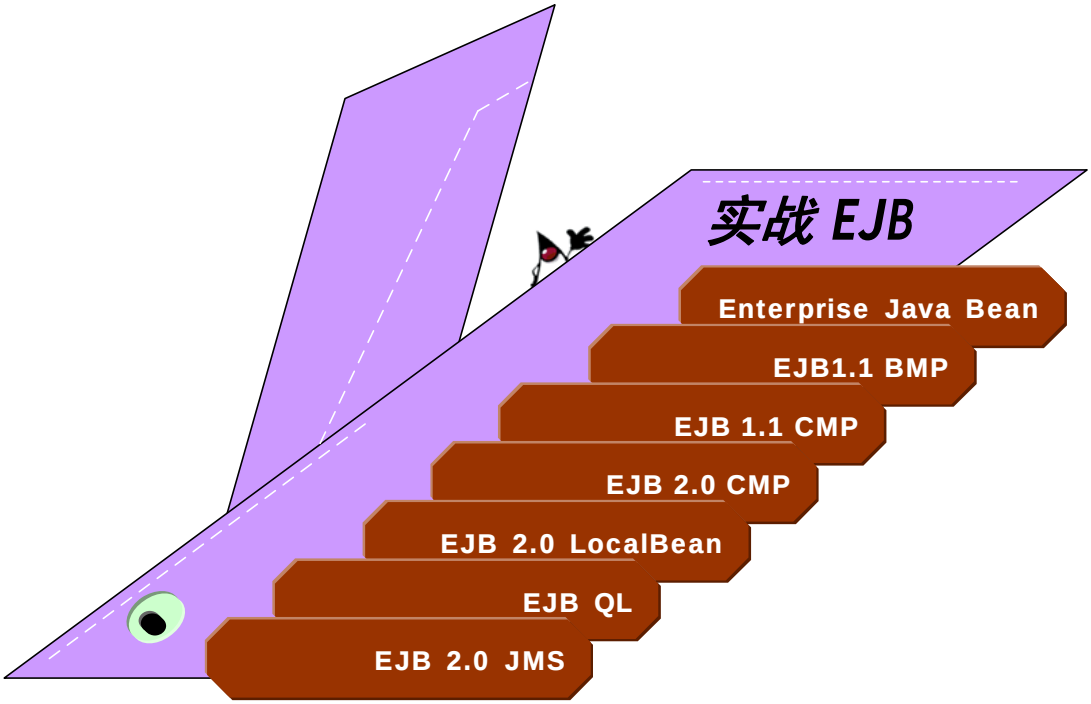


发布：中文 Java 技术网

网址：www.cn-java.com

日期：<草稿>

作者：Jackliu

索引

实战 EJB 之— 开发和部署你的第一个 Enterprise JavaBean *3*
什么是企业 JavaBeans 技术? 3
EJB 体系结构 4
开发人员的角色分配 4
编写第一个 EJB 程序 5
部署到应用服务器 7
开发和部署测试程序 9
运行测试程序 12
实战 EJB 之二 开发会话 Bean（无状态会话 Bean） *13*
什么是无状态 Session Bean? 13
无状态 Session Bean 寿命周期 13
编写一个无状态的 Session Bean 程序 14
部署到应用服务器 20
开发和部署测试程序 21
运行测试程序 24
实战 EJB 之三 开发会话 Bean（有状态会话 Bean） *26*
什么是有状态 Session Bean? 26
有状态 Session Bean 寿命周期 26
编写一个有状态 Session Bean 程序 28
部署到应用服务器 33
开发和部署测试程序 34
运行测试程序 38
实战 EJB 之四 开发实体 CMP（EJB 1.1 规范） *39*
EJB 1.1 规范中的 CMP 39
Entity Bean 的寿命周期 41
编写一个 EJB 1.1 的 CMP 程序 42
部署到应用服务器 48
开发和部署测试程序 51
运行测试程序 56
实战 EJB 之五 开发实体 BMP（EJB 1.1 规范） *57*
EJB 1.1 规范中的 BMP 57
编写一个 EJB 1.1 的 BMP 程序 60
部署到应用服务器 68
开发和部署测试程序 69
运行测试程序 71
实战 EJB 之六 开发 EJB 2.0 的 CMP(本地接口例程) *72*
实战 EJB 之七 开发 EJB 2.0 的 CMP(EJB QL) *72*
实战 EJB 之八 开发 EJB 2.0 的 JMS *72*

实战 EJB 之— 开发和部署你的第一个 Enterprise JavaBean

企业 JavaBeans(EJB)的组件结构是以作为可重复使用的服务器端组件而设计的，它使企业能够建立可升级、安全可靠、可运行于多重平台且以商务为重点的应用程序。实战 EJB 系列文章将以实战例程向大家全面系统介绍 EJB 的几种 Enterprise JavaBean 概念。

在本节中你将了解到：

- 什么是企业 JavaBeans 技术?
- EJB 体系结构
- EJB 开发人员的角色分配
- 编写第一个 EJB 程序
- 部署到应用服务器
- 开发和部署测试程序
- 运行测试程序

什么是企业 JavaBeans 技术?

EJB 结构是 JavaTM 平台上的服务器端组件模型。设计 EJB 结构的目的是，通过使企业开发人员将注意力只集中于编写商务逻辑，从而解决上面所提出的问题。EJB 技术取消了编写"全程(plumbing)" 码的要求。例如，企业开发人员不再需要编写那些处理事务行为、安全、连接共享或线程的代码，因为 EJB 体系结构将这些任务委托给服务器厂商完成了。

对用户和这一技术的实现者来说，将会获得如下收益:

- **生产效率:** 使用这一技术，企业开发人员将会进一步提高生产效率。他们不仅能够获得在 Java 平台上的开发成果，而且能够将注意力集中于商务逻辑，从而使效率倍增。
- **业内支持:** 试图建立 EJB 系统的客户会获得一系列可供选择的解决方案。企业 JavaBeans 技术已经被多达 25 个公司所接受、支持和应用。
- **投资保护:** 企业 JavaBeans 技术建立在企业现存系统之上。事实上，许多 EJB 产品都将建立在已有的企业系统之上。今天企业所使用的系统，明天将会运行企业 JavaBeans 组件。
- **结构独立:** 企业 JavaBeans 技术将开发人员和底层中间件相隔离; 开发人员看到的仅仅是 Java 平台。 这一点除下面将要谈到的交叉平台的好处外，还将使得 EJB 服务器厂商在不干扰用户的 EJB 应用程序的前提下，有机会改进中间件层。
- 服务器端仅写一次，即可随处运行（Server-Side Write Once, Run AnywhereTM）：通过对 Java 平台的支持, EJB 技术将"仅写一次，随处运行"的概念提高到了一个新的水平。它可以保证一个 EJB 应用程序可运行于任何服务器，只要这个服务器能够真正提供企业 JavaBeans APIs。

EJB 体系结构

理解 EJB 体系结构可以从两个方面分析：客户端和服务器端。

EJB 体系结构的客户端包含了调用 EJB 组件特定业务所需的 EJB 接口（扩展了 EJBHome 和 EJBObject 接口的组件接口），同时还包含了服务器端处理对象的句柄（部署组件时生成的残根类）。在 EJB 体系结构的服务器端则包含了 EJB 组件实现的实例，同时还包含了用来在客户端与 EJB 组件之间进行映射的容器代码。结构如下图 1-1：

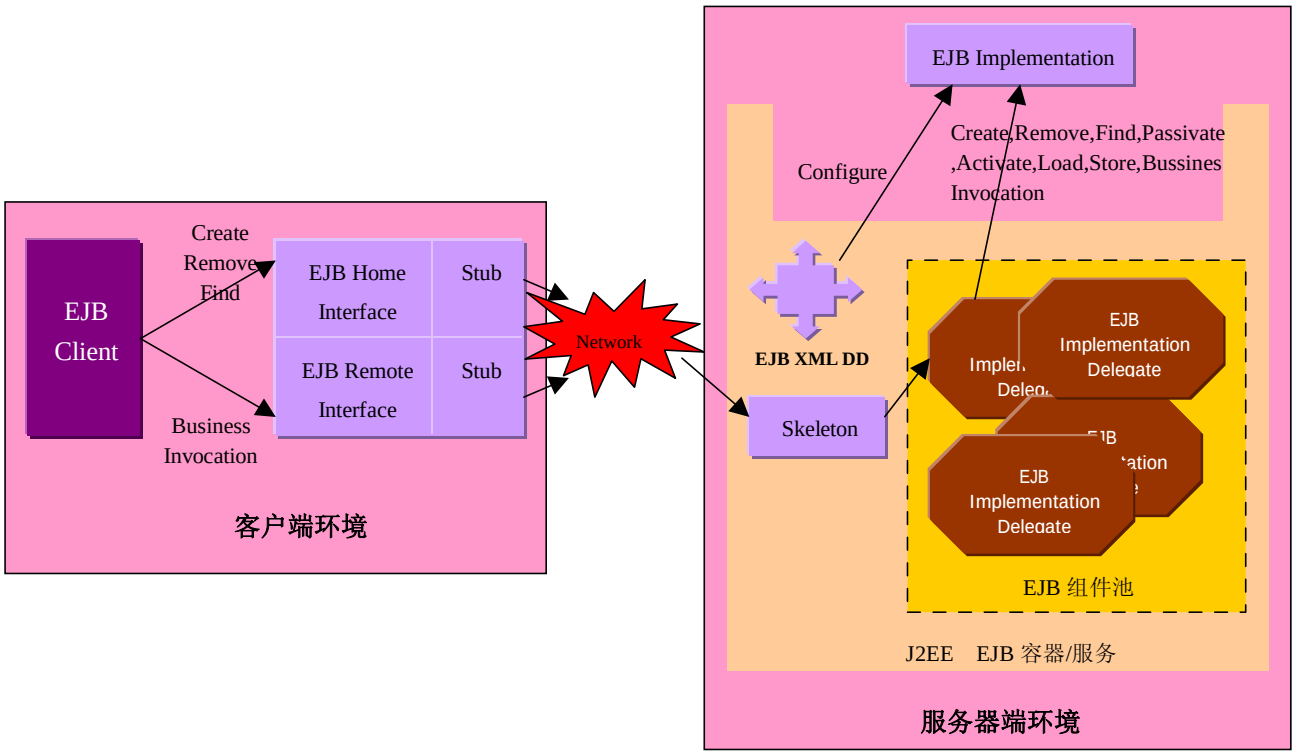


图 1-1

EJB 客户应用程序使用 JNDI 来查找对本地接口的实例引用，通过本地 EJB 接口和远程 EJB 主接口调用和引用 EJB 组件的全部方法和属性。

开发人员的角色分配

EJB 技术将开发人员分成固有的五种角色: 服务器提供者、 容器提供者、 企业 Beans 提供者、 应用程序装配者和部署者。对上述五种角色描述如下:

- 服务器提供者是分布式事务管理方面的专家，主要负责处理分布式对象和低层次系统服务。数据库和 TP 监控器厂商可典型地充当该角色。
- 容器提供者一般是系统编程方面的专家，由于容器有能力将 EJB 环境与现存应用程序(如 SAP R/3 和 CICS)桥接起来，因而这些专家有可能具备某一应用领域的经验。由于

容器为 Bean 提供了安全、可升级和事务性的环境，因而容器提供者需具备这些领域的经验。数据库和事务服务器厂商也适合这一角色，并可提供标准容器。

- 企业 Beans 提供者为 EJB 应用程序提供"积木"，他们是典型的以 Bean 的形式编写商务逻辑的域专家,而他们不一定是数据库或系统编程方面的专家。他们生成包括所有组件在内的 EJB JAR 文件。对象库厂商适合这一角色。

- 应用程序装配者是域专家，他们的工作是用第三方 Beans 建立应用程序，他们也有可能建立客户端 GUI。典型的应用程序装配者通常是程序员，他们建立应用程序来可访问已部署的组件。

- 部署者通常熟悉企业的操作环境，他们利用应用程序包并设置部分或全部应用程序的安全和事务描述符。部署者也有可能使用工具来修正 Bean 的商务逻辑。

编写第一个 EJB 程序

开发一个 EJB 至少包括四个步骤：

- 开发主接口
- 开发组件接口
- 开发 Bean 实现类
- 编写部署文件

我们将编写一个最简单 Session Bean 来实现这四个元素，这个例子将通过一个 getHello()方法返回一个“Hello, EJB”字符串，并为这个 Bean 起一个好听的名字：Hello

注意：假设你使用的 Windows 操作系统，这与程序代码和部署文件内容无关，但这些仅可能影响到存储路径和命令行。

1.开发主接口：

是由 Bean 开发人员编写的一个 Bean 的主接口（interface）程序，负责控制一个 Bean 的生命周期（生成、删除、查找 Bean）。只需要开发人员给出一个主接口类，类方法的实现由容器来完成。所以开发一个 Bean 的接口程序是非常的简单的。

一般情况下，习惯将主接口的命名规则规定为<bean-name>Home，所以我们把这个主接口类起名为 **HelloHome**

HelloHome.java 代码：

```
public interface HelloHome extends javax.ejb.EJBHome{
    public Hello create()
        throws java.rmi.RemoteException,
                javax.ejb.CreateException;
}
```

假设我们保存到 D:\ejb\Hello\src\HelloHome .java

接口类 HelloHome 扩展了 javax.ejb.EJBHome 类，这个类不在标准的 SDK 开发包中，需要你的开发机安装有 J2ee 的 SDK。所有深入的解说我们将略去，留在以后章节的例子中讲述。

2.开发组件接口：

当远程用户调用主接口类生成方法（create()）时，客户要得到一个组件的远程引用，因此 EJB 容器要求你为这个 Bean 的所有方法提供一个接口类，类的实现则与主接口一样由容器在部署时自动生成。

一般情况下，习惯将组件接口的命名规则规定为<bean-name>，所以我们把这个组件接口类起名为 **Hello**

Hello.java 代码：

```
public interface Hello extends javax.ejb.EJBObject{
    public String getHello() throws java.rmi.RemoteException;
}
```

假设我们保存到 D:\ejb\Hello\src\Hello.java

3.开发 Bean 实现类：

包含了业务类的所有详细设计细节，在这里我们将通过 getHelloEjb()方法返回一个“Hello, EJB”字符串。

一般情况下，习惯将 Bean 的实现类命名规则规定为<bean-name>EJB，所以我们把这个类起名为 **HelloEJB**

HelloEJB.java 代码：

```
import javax.ejb.*;
public class HelloEJB implements SessionBean{
    public void ejbCreate(){}
    public void ejbRemove(){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void setSessionContext(SessionContext ctx){}

    public String getHello() {
        return new String(“Hello,EJB”);
    }
}
```

假设我们保存到 D:\ejb\Hello\src\HelloEJB.java

到此为止我们的 Bean 程序 Hello 已经编写完毕了，使用如下命令进行编译：

```
cd bean\Hello
```

```
mkdir classes
cd src
javac -classpath %CLASSPATH%;../classes -d ../classes  *.java
```

如果顺利你将可以在..\Hello\classes 目录下发现有三个类文件，恭喜你，你已经迈出了第一步！不过后面还要有很多的事情要做，因为我们的 Bean 还没有部署到容器里，现在只是普普通通的三个 Java 类。因为 Bean 是没有界面的，所以我们还要为它编写一个测试这个 Bean 的测试程序。刚松了口气就又要下面的旅程了，不过，我们期待着的 Hello,EJB 在向我们靠近。

4．编写部署文件：

一个完整的 ejb 是由 java 类和一个描述其特性的 ejb-jar.xml 文件组成，部署工具将根据这些文件部署到容器中，并自动生成容器所需的类。

按照下面个格式编写一个 ejb-jar.xml 文件

ejb-jar.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <description>
        This is Hello EJB example
    </description>
    <display-name>HelloBean</display-name>
    <enterprise-beans>
        <session>
            <display-name>Hello</display-name>
            <ejb-name>Hello</ejb-name>
            <home>HelloHome</home>
            <remote>Hello</remote>
            <ejb-class>HelloEJB</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

假设我们将文件保存到 D:\ejb\Hello\classes\META-INF\ejb-jar.xml(注意 META-INF 必须大写)

部署到应用服务器

在部署之前我们需要将这些类文件和 xml 文件做成一个 jar 文件，EJB JAR 文件代表一个可

被部署的 JAR 库，在这个库里，包含了服务器代码与 EJB 模块的配置。ejb-jar.xml 文件被放置在 JAR 文件所指定的 META-INF 目录中。我们可以使用如下命令得到 EJB JAR 文件：

```
cd d:\ejb\Hello\classes      (要保证类文件在这个目录下，且有一个 META-INF 子目录存放 ejb-jar.xml 文件)
jar -cvf   Hello.jar *.*
```

部署工具一般由 Java 应用服务器的制造商提供，在这里我使用了 Apusic 应用服务器，并讲解如何在 Apusic 应用服务器部署这个 Hello 组件。

注意，如果使用其他部署工具，原理是一样的，要使用 Apusic 应用服务器，可以到 www.apusic.com 上下载试用版。

确定你的 Apusic 服务器已经被启动。

打开“部署工具”应用程序，点击文件—>新键工程：

第一步：选择“新建包含一个 EJB 组件打包后的 EJB-jar 模块”选项

第二步：选择一个刚才我们生成的 Hello.jar 文件，

第三步：输入一个工程名，可以随意，这里我们输入 Hello

第四步：输入工程存放的地址，这里我们假设被存放到 D:\ejb\Hello\deploy 目录下

完成四个步骤后，如果没有问题将出现 HellBean 的部署界面，因为这个例子非常的简单，所以不需要任何的配置，点击部署—>部署到 Apusic 应用服务器完成部署工作。

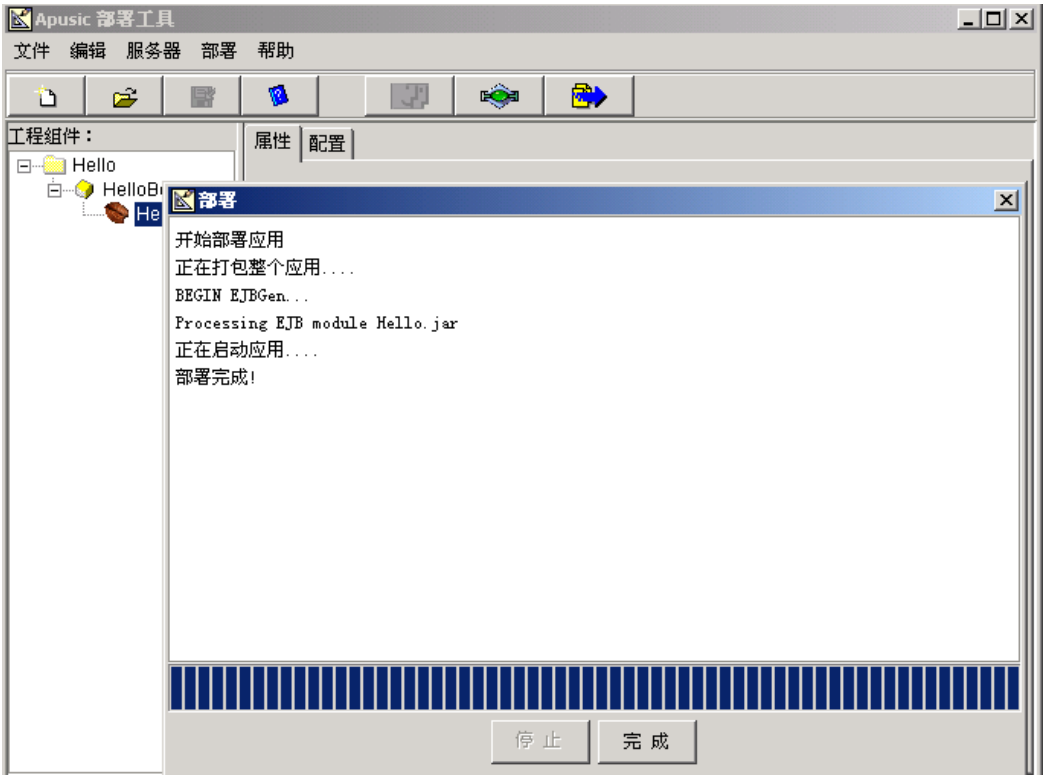


图 1-2

如果看到上述画面，恭喜你，你的 Bean 已经被部署到 EJB 容器中，其他部署工具不再详述。

开发和部署测试程序

一个 EJB 组件是没有任何运行界面的，所有组件的实例都被容器所管理，所以我们要测试这个 Bean 组件，需要写一段测试程序，简单期间，我们写一段小服务程序（Java Servlet）。

关于如何编写 Servlet 我们这里不做介绍，下面是提供的代码：

HelloServelt.java 文件：

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import javax.ejb.*;
import javax.naming.InitialContext;

public class HelloServlet extends HttpServlet{

        public void service(HttpServletRequest req,HttpServletResponse res) throws
IOException {
                res.setContentType("text/html");
                PrintWriter out =res.getWriter();
                out.println("<html><head><title>the first EJB</title></head>");
                try{
                        InitialContext ctx=new InitialContext();
                        Object objRef = ctx.lookup("java:comp/env/ejb/Hello");
                        //主接口
                        HelloHome
home=(HelloHome)javax.rmi.PortableRemoteObject.narrow(
                                objRef,HelloHome.class);
                        //组件接口
                        Hello bean =home.create();
                        out.println(bean.getHello());

                }catch(javax.naming.NamingException ne){
                        out.println("Naming Exception caught:"+ne);
                        ne.printStackTrace(out);
                }catch(javax.ejb.CreateException ce){
                        out.println("Create Exception caught:"+ce);
                        ce.printStackTrace(out);
                }catch(java.rmi.RemoteException re){
                        out.println("Remote Exception caught:"+re);
                        re.printStackTrace(out);
                }
```

```
                    out.println("</body></html>");

                }


}
```

假设我们将文件保存到 D:\ejb\Hello\src\HelloServlet.java

回到 src 目录下，使用如下命令编译 Servlet

```
javac -classpath %CLASSPATH%;../classes -d ../classes    HelloServlet.java
```

成功编译后，将这个 servlet 一同部署到 Hello 工程中，我们回到“部署工具”，点击编辑 **à** 填加一个 Web 模块。

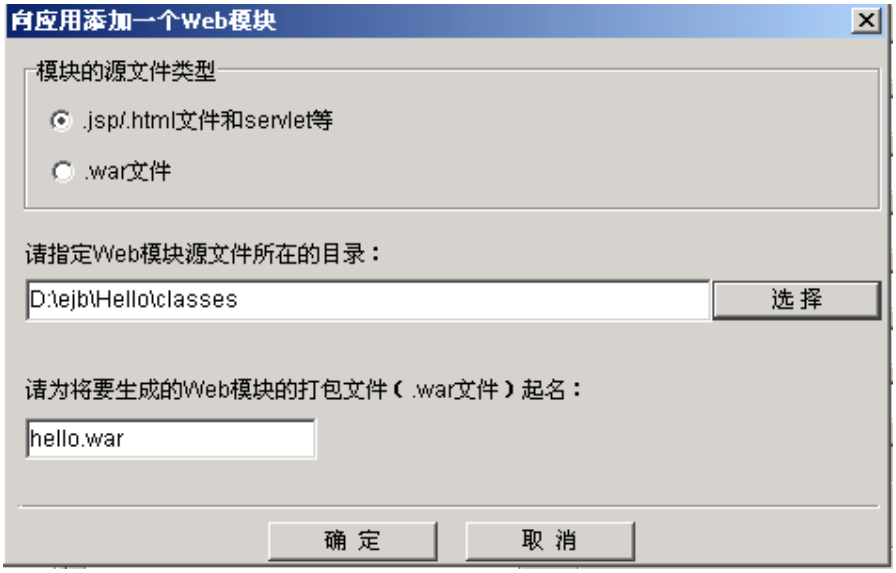


图 1-3

部署注意事项：

在 hello 属性页的“www 根路径”是访问这个 servlet 的相对路径，切记。

在 hello 内容属性一页中，展开 WEB-INF，选择 classes，单击“添加类”按钮按照下图添加这个 Servlet 类

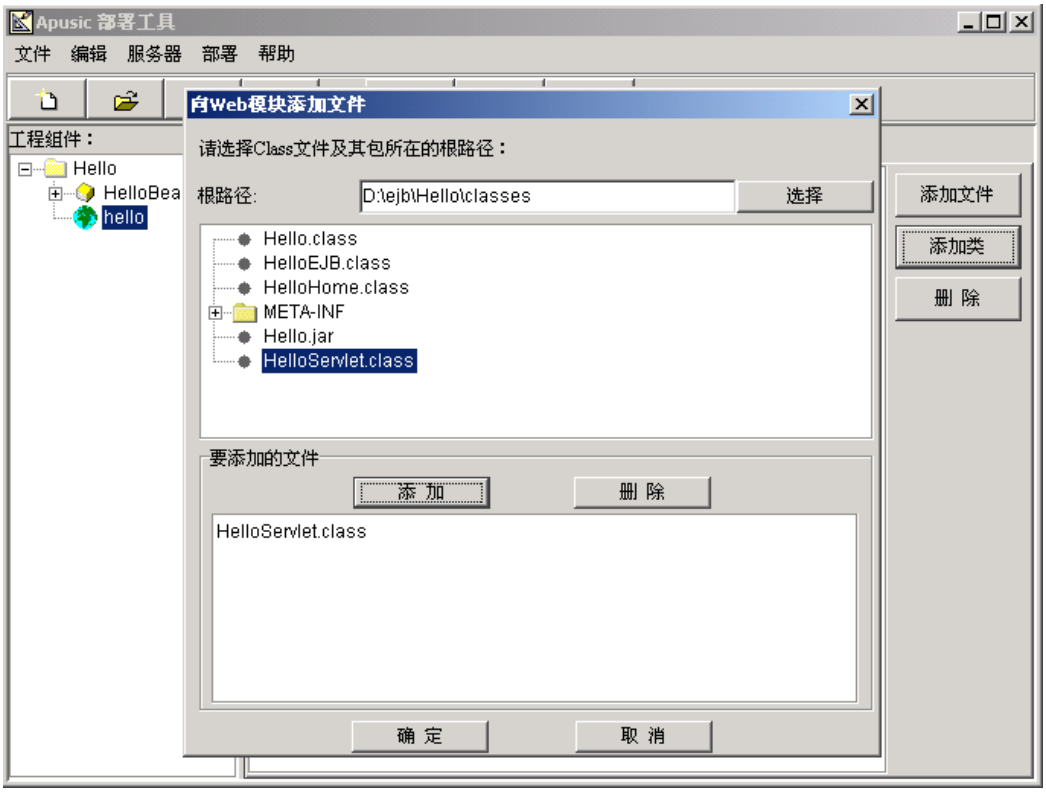


图 1-4

在 hello 配置属性一页中，单击“11.模块中用到的 EJB 的定义”按照图 1-4 进行设置

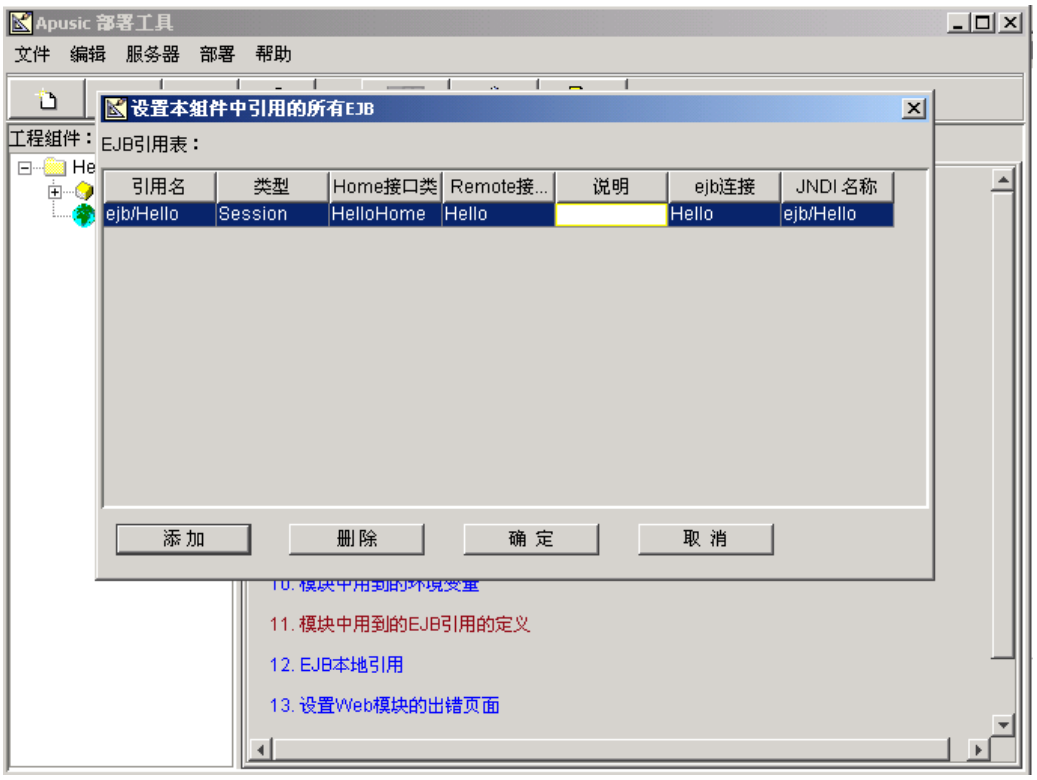


图 1-5

点击部署—>部署到 Apusic 应用服务器完成部署工作。

运行测试程序

打开浏览器，在浏览器中输入：

```
http://localhost:6888/hello/servlet/HelloServlet
localhost—代表 Web Server 的主机地址
6888—应用服务器端口，根据不同的应用服务器，端口号可能不同
hello—我们部署 servlet 时指定的 WWW 根路径值
servlet—ejb 容器执行 servlet 的路径
HelloServlet—测试程序
```

实战 EJB 之二 开发会话 Bean（无状态会话 Bean）

会话 Bean 可以分为有状态会话 Bean(stateful Bean)和无状态会话 Bean(stateless Bean),有状态会话 Bean 可以在客户访问之间保存数据，而无状态会话 Bean 不会在客户访问之间保存数据。两者都实现了 javax.ejb.SessionBean 接口，EJB 容器区通过部署文件 ejb-jar.xml 来判断是否为一个 SessionBean 提供保存状态的服务，另外，在程序实现上，无状态 Bean 不能声明实例变量，每个方法只能操作方法传来的参数。实际上，我们第一节中的第一个 EJB 程序就是一个无状态 Session Bean。

在本节中你将了解到：

- 什么是无状态 Session Bean?
- 无状态 Session Bean 寿命周期
- 编写一个无状态 Session Bean 程序
- 部署到应用服务器
- 开发和部署测试程序
- 运行测试程序

什么是无状态 Session Bean?

无状态 Session Bean 每次调用只对客户提供业务逻辑，但不保存客户端的任何数据状态。但并不意味着 stateless 类型的 Bean 没有状态，而是这些状态被保持在客户端，容器不负责管理。如《再别康桥》中写到的“悄悄的我走了，正如我悄悄的来。挥一挥衣袖，不带走一片云彩”。

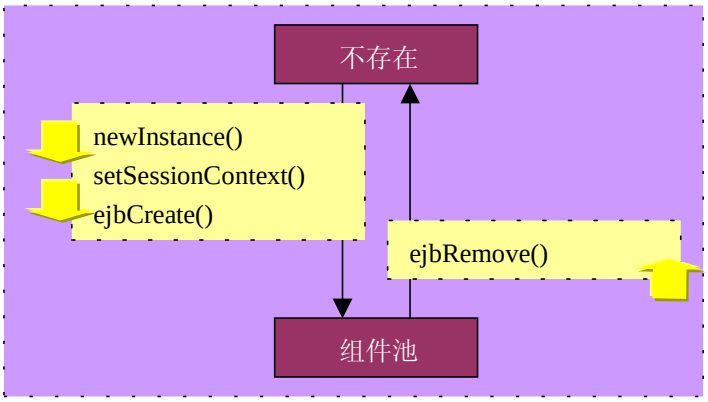
无状态 Session Bean 在 EJB 中是最简单的一种 Bean，如果数据实际上是数据的瞬像，则建议使用无状态会话 Bean。但是无状态会话 Bean 也有自己的问题，本该存储在服务器端（J2EE 服务器）的数据被存储在客户中，每次调用这些数据都要以参数的方式传递给 Bean，如果是一个比较复杂的数据集合，则网络需要传递大量数据，造成更多的负载。在客户端维护状态还要注意安全性问题，如果数据状态非常敏感，则不要使用无状态会话 Bean，这些情况可以使用状态会话 Bean，将用户状态保存到服务器中。

无状态 Session Bean 寿命周期

无状态 Session Bean 寿命周期由容器控制，Bean 的客户并不实际拥有 Bean 的直接引用，当我们部署一个 EJB 时，容器会为这个 Bean 分配几个实例到组件池（component pooling）中，当客户请求一个 Bean 时，J2EE 服务器将一个预先被实例化的 Bean 分配出去，在客户的一次会话里，可以只引用一次 Bean，就可以执行这个 Bean 的多个方法。如果又有客户请求同样一个 Bean，容器检查池中空闲的 Bean(不在方法中或事务中，如果一个客户长时间引用一个 Bean 但执行一个方法后需要等待一段时间再执行另一个方法，则这段时间也是空闲的)，如果全部的实例都已用完则会自动生成一个新的实例放到池中，并分配给请求者。当

负载减少时，池会自动管理 Bean 实例的数量，将多余的实例从池中释放。

无状态 Session Bean 有两种状态：存在或不存在。



当客户端不存在一个无状态 Session Bean 时，通过远程主接口的 create()方法创建一个 Bean，newInstance()负责将 Bean 实例化，EJB 容器调用 Bean 类的 setSessionContext()方法把运行环境对象 SessionContext 传递给 Bean;随后调用 Bean 的 ejbCreate 方法进行必要的初始化和资源分配。在下面这个实战例子中，Bean 的实现类就是 StatelessDateEJB 类。

编写一个无状态的 Session Bean 程序

这个 Session Bean 组件提供一个日期计算器，通过 getDayInRange()方法计算两个日期之间相差的天数,通过 getDayForOlympic()得到距离北京申办 2008 年奥林匹克运动会天数。并且我们为这个 Bean 起名为 **StatelessDate**

设计一个无状态的 Session Bean 至少包括四个步骤：

- 开发主接口
- 开发组件接口
- 开发 Bean 实现类
- 编写部署文件

注意：本节假设你使用的 Windows 操作系统。如果使用其他操作系统，可能影响到存储路径和 JDK 命令，但这与程序代码和部署文件内容无关。

1.开发主接口（StatelessDateHome.java）：

是由 Bean 开发人员编写的一个 Bean 的主接口（interface）程序，负责控制一个 Bean 的生命周期（生成、删除、查找 Bean）。只需要开发人员给出一个主接口类，类方法的实现由容器来完成。
主接口扩展了 javax.ejb.EJBHome 接口，参考 avax.ejb.EJBHome 接口定义如下：

```
package javax.ejb;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface EJBHome extends Remote{
  public abstract EJBMetaData getEJBMetaData() throws RemoteException;
  public abstract HomeHandle getHomeHandle() throws RemoteException;
  public abstract void remove(Object obj) throws RemoteException,RemoveException;
  public abstract void remove(Handle handle) throws RemoteException,RemoveException;
}
```

- 方法 getEJBMetaData()返回 EJBMetaData 接口的引用，取得 Bean 的信息，EJBMetaData 不是远程接口。这个类扩展了 java.io.Serializable，所以可序列化，具有序列化的特性
- 方法 getHomeHandle()返回主对象的句柄，句柄是主接口 StatelessDateHome 的持久性引用，这个类扩展了 java.io.Serializable，所以可序列化，具有序列化的特性,HomeHandle 对象可以传递给另一个 JVM，且不传递安全信息，这样新的应用可以不使用 JNDI 来查找对象既可以获得这个主接口，并来创建和获得 Bean 实例。
- 方法 remove()用来删除一个 Bean 的实例，对于一个会话 Bean，执行 Remove 操作将引用的 Bean 返回到池中，由池来管理其生命周期。

一般情况下，习惯将主接口的命名规则规定为<bean-name>Home，所以我们把这个主接口类起名为 **StatelessDateHome**

大部分逻辑方法已经被 EJBHome 定义，在我们要设计的远程主接口 StatelessDateHome 里，不必再重新定义。值得注意的是，我们需要为这个接口定义一个 create()方法，用来获得一个实例 Bean 的引用，返回的对象类型是组件接口类 StatelessDate。

StatelessDateHome.java 代码：

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface StatelessDateHome extends EJBHome{
  public StatelessDate create() throws RemoteException,CreateException;
}
```

假设我们保存到 D:\ejb\StatelessDate\src\StatelessDateHome .java

2.开发组件接口(StatelessDate.java)：

当远程用户调用主接口类生成方法（create()）时，客户要得到一个组件的远程引用，因此 EJB 容器要求你为这个 Bean 的所有方法提供一个接口类，而类的实现则与远程主接口 StatelessDateHome 一样由容器在部署时自动生成。

组件接口扩展了 avax.ejb.EJBObject 接口，参考 avax.ejb.EJBObject 接口定义如下：

```
package javax.ejb;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface EJBObject extends Remote{
   public abstract EJBHome getEJBHome() throws RemoteException;
   public abstract Handle getHandle() throws RemoteException;
   public abstract Object getPrimaryKey() throws RemoteException;
   public abstract boolean isIdentical(EJBObject ejbobject) throws RemoteException;
   public abstract void remove() throws RemoteException,RemoveException;
}
```

- 方法 getEJBHome()返回远程主接口对象的引用
- 方法 getHandle() 当前组件接口对象的句柄，和远程主接口的句柄 HomeHandle 一样，这个对象是被序列化的,所以可以保存到本地或通过 RMI/IIOP 协议传输给其他 JVM 上的客户使用，而免去 JNDI 查找和调用主接口的 create 方法，只要执行 Handle.getEJBObject()方法即可取得这个 Bean 实例的引用。
- getPrimaryKey()方法一般用于 Entity Bean，如果在 Session Bean 中调用，抛出 java.rmi.RemoteException。
- 方法 isIdentical()用于对当前引用的 Bean 实例和另一 Bean 实例进行比较，因为即便是 Bean 实例相同但有可能不是来自同一个引用，不能使用 equals()方法。
- 方法 remove() 删除当前引用的 Bean 实例，由容器来决定是否真的释放气内存，通常会返换到组件池中。注意删除之后要将对象的引用指向为 null。

一般情况下，习惯将组件接口的命名规则规定为<bean-name>，所以我们把这个组件接口类起名为 **StatelessDate**

大部分逻辑方法已经被 EJBObject 定义，在我们要设计的组件接口 StatelessDate 里，不必再重新定义,只要我们重申组件中有关业务逻辑的接口即可。逻辑方法 getDayInRange()得到两个日期间的天数间隔，如果输入的时间非法或不合适将抛出 InsufficientDateException 异常。逻辑方法 getDayForOlympic()得到距离北京申办奥运会的天数，如果输入的时间非法或不合适将抛出 InsufficientDateException 异常。

StatelessDate.java 代码：

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.util.Date;

public interface StatelessDate extends EJBObject{
 public int getDayInRange(Date lowerLimitDate,Date upperLimitDate)
       throws RemoteException,InsufficientDateException;
 public int getDayForOlympic()
```

```
        throws RemoteException,InsufficientDateException;
}
```

假设我们保存到 D:\ejb\StatelessDate\src\StatelessDate .java

InsufficientDateException.java 代码：

```
public class InsufficientDateException extends java.lang.Exception{
            public InsufficientDateException(){}
}
```

假设我们保存到 D:\ejb\StatelessDate\src\InsufficientDateException.java

3.开发 Bean 实现类(StatelessDateEJB.java)：

这个类包含了业务逻辑的所有详细设计细节。会话 Bean 的实现类实现了(implements)javax.ejb.SessionBean 所定义的接口，首先我们先熟悉一下 SessionBean 的定义：

```
package javax.ejb;
impot java.rmi.RemoteException;

public interface SessionBean extends EnterpriseBean{
   public void setSessionContext(SessionContext ctx) throws EJBException,RemoteException;
   public void ejbRemove() throws EJBException,RemoteException;
   public void ejbActivae()throws EJBException,RemoteException;
   public void ejbPassivate()throws EJBException,RemoteException;
}
```

容器通过这些方法将相关信息通知给 Bean 实例，所有的方法都抛出 RemoveException 方法是为了与 1.0 规范兼容，之后版本编写的 Bean 只需要抛出 EJBException 即可。

setSessionContext()方法将会话的语境放到对象变量中，容器在结束会话 Bean 或自动超时死亡之前将会自动调用 ejbRemove()方法，所以在此可以填入用来释放某些资源的代码。当实例被钝化或被激活时，调用 ejbActivae()和 ejbPassivate()方法，无状态会话 Bean 不会发生这些情况，在下一节将介绍。

一般情况下，习惯将组件实现类的命名规则规定为<bean-name>EJB，所以我们把这个组件类起名为 **StatelessDateEJB**
类 StatelessDateEJB 声明要实现 SessionBean 的定义，所以，对于 StatelessDateEJB 类，我们必须完全实现 SessionBean 的接口定义。

StatelessDateEJB.java 代码：

```
import javax.ejb.*;
```

```
import java.util.Date;

public class StatelessDateEJB implements SessionBean{
  public void ejbCreate(){}
  public void ejbRemove(){}
  public void ejbActivate(){}
  public void ejbPassivate(){}
  public void setSessionContext(SessionContext ctx){}

  //计算两个日期之间相隔的天数
  public int getDayInRange(Date lowerLimitDate,Date upperLimitDate)
     throws InsufficientDateException{

     long upperTime,lowerTime;
     upperTime=upperLimitDate.getTime();
     lowerTime=lowerLimitDate.getTime();
     if(upperTime<lowerTime)
        throw new InsufficientDateException();
     Long result=new Long((upperTime-lowerTime)/(1000*60*60*24));
     return result.intValue();
  }

  //得到距离 2008 年奥运会天数
  public int getDayForOlympic()
     throws InsufficientDateException {

     Date olympic=new Date("2008/01/01");
     Date today=new Date(System.currentTimeMillis());
       return getDayInRange(today,olympic);
  }
}
```

假设我们保存到 D:\ejb\StatelessDate\src\StatelessDateEJB .java

到此为止我们的 Bean 程序 StatelessDate 已经编写完毕了，使用如下命令进行编译：

```
cd bean\StatelessDate\src
mkdir classes
cd src
javac -classpath %CLASSPATH%;../classes -d ../classes InsufficientDateException.java
StatelessDate.java StatelessDateHome.java StatelessDateEJB.java
```

如果顺利你将可以在..\StatelessDate\classes 目录下发现有四个类文件。

4．编写部署文件：

一个完整的 ejb 是由 java 类和一个描述其特性的 ejb-jar.xml 文件组成，部署工具将根据这些文件部署到容器中，并自动生成容器所需的残根类。

按照下面个格式编写一个 ejb-jar.xml 文件，对于 DTD 介绍此处省去。

ejb-jar.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <description>
        This is StatelessDate EJB example
    </description>
    <display-name>StatelessDateBean</display-name>
    <enterprise-beans>
        <session>
            <display-name>StatelessDate</display-name>
            <ejb-name>StatelessDate</ejb-name>
            <home>StatelessDateHome</home>
            <remote>StatelessDate</remote>
            <ejb-class>StatelessDateEJB</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

假设我们保存到 D:\ejb\StatelessDate\classes\META-INF\ejb-jar.xml(注意 META-INF 必须大写)

现在让我们看看当前的目录结构：

- StatelessDate <文件夹>
  - classes<文件夹>
    - META-INF<文件夹>
      - ejb-jar.xml
    - InsufficientDateException.class
    - StatelessDate.class
    - StatelessDateEJB.class
    - StatelessDateHome.class
  - src<文件夹>

- InsufficientDateException.java
- StatelessDate.java
- StatelessDateEJB.java
- StatelessDateHome.java

部署到应用服务器

在部署之前我们需要将这些类文件和 xml 文件做成一个 jar 文件，EJB JAR 文件代表一个可被部署的 JAR 库，在这个库里，包含了服务器代码与 EJB 模块的配置。ejb-jar.xml 文件被放置在 JAR 文件所指定的 META-INF 目录中。我们可以使用如下命令得到 EJB JAR 文件：

```
cd d:\ejb\StatelessDate\classes      (要保证类文件在这个目录下，且有一个 META-INF 子目录存放 ejb-jar.xml 文件)
jar -cvf    StatelessDate.jar *.*
```

确保 statelessDate.jar 文件包括的文件目录格式如下：

- <根>
  - **META-INF**<文件夹>
    - **ejb-jar.xml**
  - InsufficientDateException.class
  - **StatelessDate.class**
  - **StatelessDateEJB.class**
  - **StatelessDateHome.class**

部署工具一般由 Java 应用服务器的制造商提供，在这里我使用了 Apusic 应用服务器，并讲解如何在 Apusic 应用服务器部署这个 StatelessDate 组件。

注意，如果使用其他部署工具，原理是一样的。要使用 Apusic 应用服务器，可以到 www.apusic.com 上下载试用版。

确定你的 Apusic 服务器已经被启动。
打开“部署工具”应用程序，点击文件—>新键工程：
第一步：选择“新建包含一个 EJB 组件打包后的 EJB-jar 模块”选项
第二步：选择一个刚才我们生成的 StatelessDate.jar 文件，
第三步：输入一个工程名，可以随意，这里我们输入 StatelessDate
第四步：输入工程存放的地址，这里我们假设被存放到 D:\ejb\StatelessDate\deploy 目录下
完成四个步骤后，如果没有问题将出现 StatelessDateBean 的部署界面，基本的参数配置已经在我们刚才编写的 ejb-jar.xml 中定义，可以点击部署—>部署到 Apusic 应用服务器完成部署工作。

开发和部署测试程序

SessionBean 组件是没有任何运行界面的，组件的实例被容器所管理，所以我们要测试这个 Bean 组件，需要写一段测试程序。这里，我们写一段小服务程序（Java Servlet）。

关于如何编写 Servlet 我们这里不做介绍。InitialContext 对象用来获取当前 servlet 小应用程序的语境，方法 lookup 从组件池中查找一个 JNDI 对象，并取得一个远程主接口的引用，java:comp/env/ejb/StatelessDate 是我们刚才 StatelessDate 组件的 JNDI 名，请参考 ejb-jar.xml 中的<ejb-name>项。要注意的是，lookup()方法返回的是一个 Object 类型的远程主接口对象的残根，为此需要使用 javax.rmi.PortableRemoteObject 的 narrow()方法来获取一个具体的对象引用，narrow()方法：第一个参数是 lookup()方法返回的对象，第二个参数是要得到的引用类型。我们通过 narrow()方法并经过造型得到了一个 StatelessDateHome 对象的实例引用。调用 create()方法获取一个 StatelessDate 组件接口的实例引用，然后就可以与本地一样去引用这个实例。使用完毕后不要忘记将实例的引用指向一个 Null。

下面是提供的代码：

StatelessDateServlet .java 文件：

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import javax.ejb.*;
import javax.naming.InitialContext;

public class StatelessDateServlet extends HttpServlet{
  public void service(HttpServletRequest req,HttpServletResponse res) throws IOException {
     res.setContentType("text/html");
     PrintWriter out =res.getWriter();
     out.println("<html><head><title>StatelessDateServlet</title></head>");
     out.println("<body><h2>Test Result:<hr>");
        try{
            InitialContext ctx =new InitialContext();
            Object objRef =ctx.lookup("java:comp/env/ejb/StatelessDate");
            StatelessDateHome home=(StatelessDateHome)
            javax.rmi.PortableRemoteObject.narrow(
            objRef,StatelessDateHome.class);

            StatelessDate bean=home.create();
            java.util.Date lowerLimitDate= new java.util.Date(new
String(req.getParameter("lowerLimitDate")));
            java.util.Date upperLimitDate= new java.util.Date(new
String(req.getParameter("upperLimitDate")));
            out.println("the lowerLimitDate:"+lowerLimitDate+"<br>");
```

```
            out.println("the upperLimitDate:"+upperLimitDate+"<br>");
            out.println("the method getDayInRange()
Result:"+bean.getDayInRange(lowerLimitDate,upperLimitDate)+" days");
            out.println("<hr>");
            out.println("the method getDayForOlympic() Result:"+bean.getDayForOlympic()+"
days");
            bean=null;
        }catch(javax.naming.NamingException ne){
            out.println("Naming Exception caught:"+ne);
            ne.printStackTrace(out);
        }catch(javax.ejb.CreateException ce){
            out.println("Create Exception caught:"+ce);
            ce.printStackTrace(out);
        }catch(java.rmi.RemoteException re){
            out.println("Remote Exception caught:"+re);
            re.printStackTrace(out);
        }catch(InsufficientDateException ie){
            out.println("InsufficientDate Exception caught:"+ie);
            ie.printStackTrace(out);
        }
        out.println("</body></html>");
  }
}
```

假设我们将文件保存到 D:\ejb\StatelessDate\src\StatelessDateServlet.java

使用如下命令编译 Servlet

```
cd D:\ejb\StatelessDate
mkdir test
cd test
mkdir WEB-INF
cd WEB-INF
mkdir classes
cd D:\ejb\StatelessDate\src
javac -classpath %CLASSPATH%;../classes/ -d ../test/WEB-INF/classes StatelessDateServlet.java
```

编译成功后将这个 servlet 部署到与 StatelessDate 同一工程中，在部署前需要我们为部署编写一个 web.xml,其中告诉部署工具这个 servlet 需要参考的一些资源和部署描述，这里我们将定义一个 JNDI 参考：

```
  <ejb-ref>
    <description></description>
    <ejb-ref-name>ejb/StatelessDate</ejb-ref-name>
```

```
    <ejb-ref-type>Session</ejb-ref-type>
    <home>StatelessDateHome</home>
    <remote>StatelessDate</remote>
    <ejb-link>StatelessDate</ejb-link>
  </ejb-ref>
```

<ejb-ref-name>告诉部署工具，这个 Web 模块需要参考的资源 JNDI 名称，<ejb-ref-type>被指定参考一个 SessionBean 组件。

web.xml 文件内容如下：

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

<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN' 'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
  <icon>
    <small-icon></small-icon>
    <large-icon></large-icon>
  </icon>
  <display-name>StatelessDate</display-name>
  <description></description>
  <context-param>
    <param-name>jsp.nocompile</param-name>
    <param-value>false</param-value>
  </context-param>
  <context-param>
    <param-name>jsp.usePackages</param-name>
    <param-value>true</param-value>
    <description></description>
  </context-param>
  <ejb-ref>
    <description></description>
    <ejb-ref-name>ejb/StatelessDate</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>StatelessDateHome</home>
    <remote>StatelessDate</remote>
    <ejb-link>StatelessDate</ejb-link>
  </ejb-ref>
</web-app>
```

假设我们将文件保存到 D:\ejb\StatelessDate\test\WEB-INF\web.xml

下面我们要部署这个 Servlet 到 J2EE 服务器。J2EE Web 应用可以包括 Java Servlet 类、JavaServer Page 组件、辅助的 Java 类、HTML 文件、媒体文件等，这些文件被集中在一个 War 文件中。其中 War 结构具有固定的格式，根目录名为 WEB-INF，同一目录下应该有一个 web.xml 文件，用来描述被部署文件的部署信息，Jsp、html 等文件可以放置在这个目录下，同时 WEB-INF 目录下可能存在一个 classes 目录用于存放 Servlet 程序，如果引用了一些外部资源，则可以被放置到 WEB-INF\lib 目录下。使用下面的命令生成这个 Servlet 测试程序的 war 文件：

```
cd D:\ejb\StatelessDate\test\
jar -cvf statelessDate.war *.*
```

确保 statelessDate.war 文件包括的文件目录格式如下：

- WEB-INF<文件夹>
  - **classes**<文件夹>
    - **StatelessDateServlet.class**
  - web.xml

成功编译后，将这个 servlet 一同部署到 StatelessDate 工程中，我们回到“部署工具”，点击编辑 **à** 填加一个 Web 模块,选择我们刚刚编译成的 statelessDate.war 文件
点击部署—>部署到 Apusic 应用服务器完成部署工作。

运行测试程序

打开浏览器，在浏览器中输入：
http://localhost:6888/statelessDate/servlet/StatelessDateServlet?lowerLimitDate=2001/01/01&upperLimitDate=2001/12/31
localhost—Web Server 的主机地址
:6888—应用服务器端口，根据不同的应用服务器，端口号可能不同
/statelessDate—部署 servlet 时指定的 WWW 根路径值
/servlet—ejb 容器执行 servlet 的路径
/StatelessDateServlet—测试程序
?LowerLimitDate—参数 1，起始日期
&upperLimitDate—参数 2，结束日期

如果运行正常应该能够看到下面的结果。

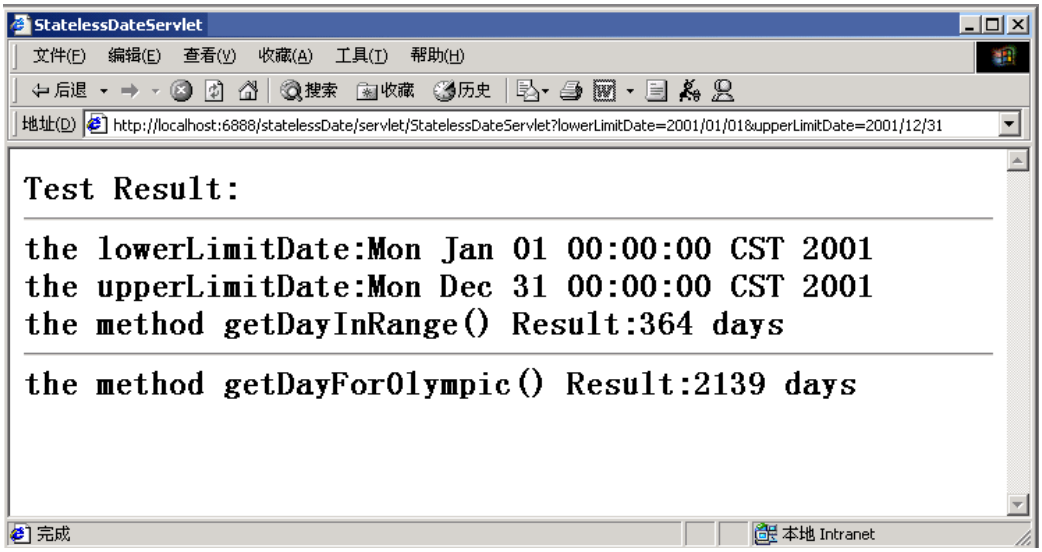


图 2-1

实战 EJB 之三 开发会话 Bean（有状态会话 Bean）

会话 Bean 可以分为有状态会话 Bean(stateful Bean)和无状态会话 Bean(stateless Bean),有状态会话 Bean 可以在客户访问之间保存数据，而无状态会话 Bean 不会在客户访问之间保存数据。两者都实现了 javax.ejb.SessionBean 接口，EJB 容器区通过部署文件 ejb-jar.xml 来判断是否为一个 SessionBean 提供保存状态的服务，另外，在程序实现上，无状态 Bean 不能声明实例变量，每个方法只能操作方法传来的参数，如果需要在引用期间维持一些数据状态，以在其他方法中可以引用，则可以把 Bean 设计成有状态会话 Bean。在第二节中我们用一个 SatatelessDate Bean 例子描述了开发无状态会话 Bean 的过程及特性，下面的一节将介绍关于有状态会话 Bean 的一些特性和寿命周期，并用一个例子来证明这些特性。

在本节中你将了解到：

- 什么是有状态 Session Bean?
- 有状态 Session Bean 寿命周期
- 编写一个有状态 Session Bean 程序
- 部署到应用服务器
- 开发和部署测试程序
- 运行测试程序

什么是有状态 Session Bean?

有状态会话 Bean(Stateful Session Bean)就是在客户引用期间维护 Bean 中的所有实例数据的状态值,这些数据在引用期间可以被其他方法所引用,其他客户不会共享同一个 Session Bean 的实例。Bean 的状态被保存到临时存储体中，因为 Bean 是可以被序列化的，所以同样也可以把一个 Bean 状态保存到文件系统或数据库中。因为在调用方法时需要维护状态（这部分是有开销的），所以只有需要维护客户状态时才使用有状态会话 Bean。典型的会话 Bean 是购物车，当一个客户第一次打开购物车时，系统为他分配一个购物车的会话 Bean，在以后，每当客户选购了商品将改变购物车的商品记录，而这些记录数据将保存到用户会话数据中。

有状态 Session Bean 寿命周期

有状态 Session Bean 寿命周期由容器控制,Bean 的客户并不实际拥有 Bean 的直接引用，当我们部署一个 EJB 时，容器会为这个 Bean 分配几个实例到组件池（component pooling）中，当客户请求一个 Bean 时，J2EE 服务器将一个预先被实例化的 Bean 分配出去，在客户的一次会话里，可以只引用一次 Bean，就可以执行这个 Bean 的多个方法。如果又有客户请求同样一个 Bean，容器检查池中空闲的 Bean(不在方法中或事务中，如果一个客户长时间引用一个 Bean 但执行一个方法后需要等待一段时间再执行另一个方法，则这段时间也是空闲的)，如果全部的实例都已用完则会自动生成一个新的实例放到池中，并分配给请求者。当

负载减少时，池会自动管理 Bean 实例的数量，将多余的实例从池中释放。

有状态会话 Bean 的寿命周期比无状态会话 Bean 更加的复杂，有状态会话 Bean 有四种状态：

- Ø 不存在
- Ø 方法现成
- Ø 事务中方法现成
- Ø 钝化

如图 3-1 所示：

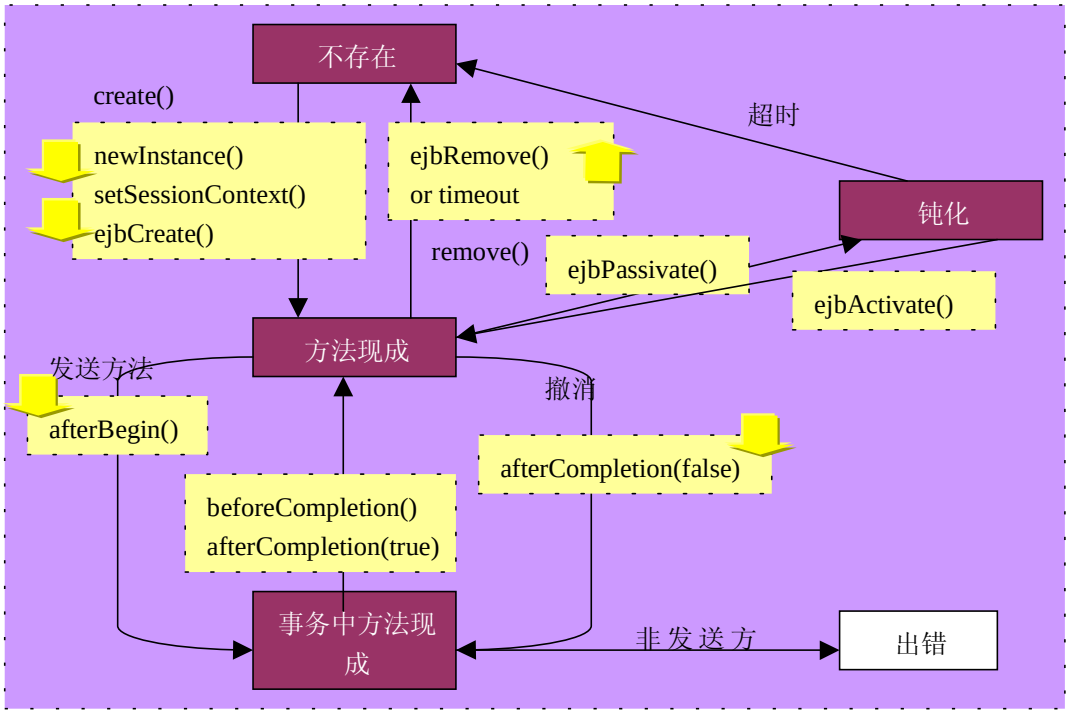


图 3-1

有状态会话 Bean 的初始化状态为不存在，当有客户引用一个 Bean 时，按照顺序调用 newInstance()、setSessionContext()和 ejbCreate()方法，与第一节中讲到的无状态调用顺序相同。当处于方法现成状态时，如果客户调用 remove()方法，则回到不存在状态，并触发 Bean 的 ejbRemove()方法。如果客户长时间不调用 Bean 或服务器准备释放一些内存资源，则容器将这些 Bean 从组件池中钝化，钝化过程容器将调用 Bean 的 ejbPassivate()方法，使程序员有机会在钝化 Bean 时释放分配的资源。当一个客户请求一个被钝化的 Bean 时，容器可以激活 Bean，激活过程容器将调用 ejbActivate()放，使程序员有机会在 Bean 转到方法现成状态时分配 Bean 所需的资源。

Bean 本身可以管理事务（BMT Bean-Managed Transactions），也可以由容器管理事务（CMT Container-Managed Transation）。对于 CMT，容器在方法开始时打开事务，在方法结束时实现事务。Bean 开发人员可以通过 afterBegin()、beforeCompletion()、afterCompletion(Boolean)来获取事务的各个状态，如果 afterCompletion(Boolean)中 Boolean 变量为 true 表示事务完成，为 false 表示事务被撤消。

编写一个有状态 Session Bean 程序

假设这次我们要为一个基金组织编写一个基金帐户的 Bean 组件，这个组件将为基金管理系统提供一个基金帐户的基本功能。为了能够描述清楚有状态会话 Bean 的特性，我们将之简化成提供三个业务逻辑接口：addFunds()方法为一个基金帐户添加基金，removeFunds()方法从基金帐户中取出基金，方法 getBalance()为我们提供一个基金帐户的余额查询。我们为这个 Bean 起名为 **StatefulAccount**

设计一个有状态的 Session Bean 至少包括四个步骤：

- 开发主接口
- 开发组件接口
- 开发 Bean 实现类
- 编写部署文件

注意：本节假设你使用的 Windows 操作系统。如果使用其他操作系统，可能影响到存储路径和 JDK 命令，但这与程序代码和部署文件内容无关。

1.开发主接口（StatefulAccountHome.java）：

是由 Bean 开发人员编写的一个 Bean 的主接口（interface）程序，负责控制 Bean 的生命周期（生成、删除、查找 Bean）。只需要开发人员给出一个主接口类，类方法的实现由容器来完成。

主接口扩展了 javax.ejb.EJBHome 接口，参考 avax.ejb.EJBHome 接口定义如下：

```
package javax.ejb;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface EJBHome extends Remote{
  public abstract EJBMetaData getEJBMetaData() throws RemoteException;
  public abstract HomeHandle getHomeHandle() throws RemoteException;
  public abstract void remove(Object obj) throws RemoteException,RemoveException;
  public abstract void remove(Handle handle) throws RemoteException,RemoveException;
}
```

- 方法 getEJBMetaData()返回 EJBMetaData 接口的引用，取得 Bean 的信息，EJBMetaData 不是远程接口。这个类扩展了 java.io.Serializable，所以可序列化，具有序列化的特性
- 方法 getHomeHandle()返回主对象的句柄，句柄是主接口 StatelessAccountHome 的持久性引用，这个类扩展了 java.io.Serializable，所以可序列化，具有序列化的特性,HomeHandle 对象可以传递给另一个 JVM，且不传递安全信息，这样新的应用可以不使用 JNDI 来查找对象既可以获得这个主接口，并来创建和获得 Bean 实例。
- 方法 remove()用来删除一个 Bean 的实例，对于一个会话 Bean，执行 Remove 操作将引用的 Bean 返回到池中，由池来管理其生命周期。

一般情况下，习惯将主接口的命名规则规定为<bean-name>Home，所以我们把这个主接口类

起名为 **StatefulAccountHome**

大部分逻辑方法已经被 EJBHome 定义，在我们要设计的远程主接口中，不必再重新定义。值得注意的是，我们需要为这个接口定义一个 create()方法，用来获得一个实例 Bean 的引用，返回的对象类型是组件接口类 **StatefulAccount**。与第二节的 StatelessDateHome 类定义基本相同，不同的是 Create()方法需要一个 double 类型的 fund 参数，当客户创建一个 Bean 引用时，我们将通过这个参数初始化基金帐户的余额。Fund 数值的状态将由容器来维护。

StatefulAccountHome.java 代码:

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface StatefulAccountHome extends EJBHome{
 public StatefulAccount create(double fund) throws RemoteException,CreateException;
}
```

假设我们保存到 D:\ejb\StatefulAccount\src\StatefulAccountHome .java

2.开发组件接口(StatefulAccount.java):

当远程用户调用主接口类生成方法（create(double)）时，客户要得到一个组件的远程引用，因此 EJB 容器要求你为这个 Bean 的所有方法提供一个接口类，而类的实现则与远程主接口 StatefulAccountHome 一样由容器在部署时自动生成。

组件接口扩展了 avax.ejb.EJBObject 接口，参考 avax.ejb.EJBObject 接口定义如下：

```
package javax.ejb;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface EJBObject extends Remote{
   public abstract EJBHome getEJBHome() throws RemoteException;
   public abstract Handle getHandle() throws RemoteException;
   public abstract Object getPrimaryKey() throws RemoteException;
   public abstract boolean isIdentical(EJBObject ejbobject) throws RemoteException;
   public abstract void remove() throws RemoteException,RemoveException;
}
```

- 方法 getEJBHome()返回远程主接口对象的引用
- 方法 getHandle() 当前组件接口对象的句柄，和远程主接口的句柄 HomeHandle 一样，这个对象是被序列化的，所以可以保存到本地或通过 RMI/IIOP 协议传输给其他 JVM 上的客户使用，而免去 JNDI 查找和调用主接口的 create 方法，只要执行 Handle.getEJBObject()方法即可取得这个 Bean 实例的引用。
- getPrimaryKey()方法一般用于 Entity Bean，如果在 Session Bean 中调用，抛出 java.rmi.RemoteException。

- 方法 isIdentical()用于对当前引用的 Bean 实例和另一 Bean 实例进行比较，因为即便是 Bean 实例相同但有可能不是来自同一个引用，不能使用 equals()方法。
- 方法 remove() 删除当前引用的 Bean 实例，由容器来决定是否真的释放气内存，通常会返换到组件池中。注意删除之后要将对象的引用指向为 null。

一般情况下，习惯将组件接口的命名规则规定为<bean-name>，所以我们把这个组件接口类起名为 **StatefulAccount**

大部分逻辑方法已经被 EJBObject 定义，在我们要设计的组件接口 StatefulAccount 中，不必再重新定义,只要我们重申组件中有关业务逻辑的接口即可。为了使远程客户能够得到基金帐户的业务方法，我们必须对应 Bean 的实现类申明业务逻辑接口的方法。方法 AddFunds(double)向一个帐户增加基金，removeFunds(double)方法从一个帐户中取出基金，getBalance()方法得到当前基金帐户的余额，注意这些类都被声名成接口方法，不需要我们在此实现。

在方法 AddFunds 和 removeFunds 中，如果参数不符合要求，抛出 InsufficientFundException 异常错误。

StatefulAccount.java 代码:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface StatefulAccount extends EJBObject{
 public void addFunds(double fund)
       throws RemoteException,InsufficientFundException;
 public void removeFunds(double fund)
       throws RemoteException,InsufficientFundException;
 public double getBalance()
       throws RemoteException;
}
```

假设我们保存到 D:\ejb\StatefulAccount\src\StatefulAccount .java

InsufficientFundException.java 代码:

```
public class InsufficientFundException extends java.lang.Exception{
            public InsufficientFundException(){super();}
            public InsufficientFundException(String msg){super(msg);}
}
```

假设我们保存到 D:\ejb\StatefulAccount\src\InsufficientFundException .java

3.开发 Bean 实现类(StatefulAccountEJB.java):

这个类包含了业务逻辑的所有详细设计细节。会话 Bean 的实现类实现了

(implements)javax.ejb.SessionBean 所定义的接口，首先我们先熟悉一下 SessionBean 的定义：

```
package javax.ejb;
impot java.rmi.RemoteException;

public interface SessionBean extends EnterpriseBean{
   public void setSessionContext(SessionContext ctx) throws EJBException,RemoteException;
   public void ejbRemove() throws EJBException,RemoteException;
   public void ejbActivae()throws EJBException,RemoteException;
   public void ejbPassivate()throws EJBException,RemoteException;
}
```

容器通过这些方法将相关信息通知给 Bean 实例，所有的方法都抛出 RemoveException 方法是为了与 1.0 规范兼容，之后版本编写的 Bean 只需要抛出 EJBException 即可。

setSessionContext()方法将会话的语境放到对象变量中，容器在结束会话 Bean 或自动超时死亡之前将会自动调用 ejbRemove()方法，所以在此可以填入用来释放某些资源的代码。当实例被钝化或被激活时，调用 ejbActivae()和 ejbPassivate()方法。在钝化过程之前，容器调用 ejbPassivate()方法，Bean 的开发人员可以在这个方法里释放占用的资源，待 Bean 被激活时，在 ejbPassivate()方法中再分配这些资源。

一般情况下，习惯将组件实现类的命名规则规定为<bean-name>EJB，所以我们把这个组件类起名为 **StatefulAccountEJB**

类 StatefulAccountEJB 声明要实现 SessionBean 的定义，所以，我们必须完全实现 SessionBean 的接口定义。

StatefulAccountEJB.java 代码：

```
import javax.ejb.*;

public class StatefulAccountEJB implements SessionBean{
 public void ejbCreate(double fund)throws CreateException{
      if (fund<0)
          throw new CreateException("Invalid fund");
      this.fundBalance=fund;
 }

 public void ejbRemove(){}
 public void ejbActivate(){}
 public void ejbPassivate(){}
 public void setSessionContext(SessionContext ctx){}

 //实例变量，有状态的会话 bean 将在组件池中维护这个实例的值
 private double fundBalance;
```

```
//向基金帐户增加基金
public void addFunds(double fund)throws InsufficientFundException{
    if (fund<0)
        throw new InsufficientFundException("Invalid fund");
    this.fundBalance+=fund;
}

//从基金帐户撤除部分基金
public void removeFunds(double fund)throws InsufficientFundException{
   if(fund<0)
       throw new InsufficientFundException("Invalid fund");
   if(this.fundBalance<fund)
       throw new InsufficientFundException("the balance less than fund");
   this.fundBalance-=fund;
}

//得到基金帐户的余额
public double getBalance(){
   return this.fundBalance;
}
}
```

假设我们保存到 D:\ejb\StatefulAccount\src\StatefulAccountEJB .java

到此为止我们的 Bean 程序组件已经编写完毕了，使用如下命令进行编译：

```
cd bean\StatefulAccount
mkdir classes
cd src
javac -classpath %CLASSPATH%;../classes -d ../classes InsufficientFundException.java StatefulAccount.java StatefulAccountHome.java StatefulAccountEJB.java
```

如果顺利你将可以在..\StatefulAccount\classes 目录下发现有四个类文件。

4．编写部署文件：

一个完整的 ejb 是由 java 类和一个描述其特性的 ejb-jar.xml 文件组成，部署工具将根据这些文件部署到容器中，并自动生成容器所需的残根类。

按照下面个格式编写一个 ejb-jar.xml 文件，对于 DTD 介绍此处省去。

ejb-jar.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <description>
```

```
            This is StatefulAccount EJB example
        </description>
        <display-name>StatefulAccountBean</display-name>
        <enterprise-beans>
            <session>
                <display-name>StatefulAccount</display-name>
                <ejb-name>StatefulAccount</ejb-name>
                <home>StatefulAccountHome</home>
                <remote>StatefulAccount</remote>
                <ejb-class>StatefulAccountEJB</ejb-class>
                <session-type>Stateful</session-type>
                <transaction-type>Container</transaction-type>
            </session>
        </enterprise-beans>
</ejb-jar>
```

假设我们保存到 D:\ejb\StatefulAccount\classes\META-INF\ejb-jar.xml(注意 META-INF 必须大写)

现在让我们看看当前的目录结构：

- StatefulAccount <文件夹>
  - classes<文件夹>
    - META-INF<文件夹>
      - ejb-jar.xml
    - InsufficienFundException.class
    - StatefulAccount .class
    - StatefulAccount EJB.class
    - StatefulAccount Home.class
  - src<文件夹>
    - InsufficientFundException.java
    - StatefulAccount.java
    - StatefulAccountEJB.java
    - StatefulAccountHome.java

部署到应用服务器

在部署之前我们需要将这些类文件和 xml 文件做成一个 jar 文件，EJB JAR 文件代表一个可

被部署的 JAR 库，在这个库里，包含了服务器代码与 EJB 模块的配置。ejb-jar.xml 文件被放置在 JAR 文件所指定的 META-INF 目录中。我们可以使用如下命令得到 EJB JAR 文件：

```
cd d:\ejb\StatefulAccount\classes      (要保证类文件在这个目录下，且有一个 META-INF 子目录存放 ejb-jar.xml 文件)
jar -cvf    StatefulAccount.jar *.*
```

确保 StatefulAccount.jar 文件包括的文件目录格式如下：

- o　<根>
  - Ø　**META-INF**<文件夹>
    - §　**ejb-jar.xml**
  - Ø　InsufficientFundException.class
  - Ø　**StatefulAccount.class**
  - Ø　**StatefulAccountEJB.class**
  - ø　**StatefulAccountHome.class**

部署工具一般由 Java 应用服务器的制造商提供，在这里我使用了 Apusic 应用服务器，并讲解如何在 Apusic 应用服务器部署这个 StatefulAccount 组件。

注意，如果使用其他部署工具，原理是一样的。要使用 Apusic 应用服务器，可以到 www.apusic.com 上下载试用版。

确定你的 Apusic 服务器已经被启动。
打开“部署工具”应用程序，点击文件—>新键工程：
第一步：选择“新建包含一个 EJB 组件打包后的 EJB-jar 模块”选项
第二步：选择一个刚才我们生成的 StatefulAccount.jar 文件，
第三步：输入一个工程名，可以随意，这里我们输入 StatefulAccount
第四步：输入工程存放的地址，这里我们假设被存放到 D:\ejb\StatefulAccount\deploy 目录下
完成四个步骤后，如果没有问题将出现 StatefulAccountBean 的部署界面，基本的参数配置已经在我们刚才编写的 ejb-jar.xml 中定义，可以点击部署—>部署到 Apusic 应用服务器完成部署工作。

开发和部署测试程序

SessionBean 组件是没有任何运行界面的，组件的实例被容器所管理，所以我们要测试这个 Bean 组件，需要写一段测试程序，这里，我们写一段小服务程序（Java Servlet）。

关于如何编写 Servlet 我们这里不做介绍。InitialContext 对象用来获取当前 servlet 小应用程序的语境，方法 lookup 从组件池中查找一个 JNDI 对象，并取得一个远程主接口的引用。java:comp/env/ejb/StatefulAccount 是我们刚才部署 StatefulAccount 组件的 JNDI 名，请参考 ejb-jar.xml 中的<ejb-name>项。要注意的是，lookup()方法返回的是一个 Object 类型的远程主接口对象的残根，为此需要使用 javax.rmi.PortableRemoteObject 的 narrow()方法来获取一个具体的对象引用，narrow()方法：第一个参数是 lookup()方法返回的对象，第二个参数是

要得到的引用类型。我们通过 narrow()方法并经过造型得到了一个 StatefulAccountHome 对象的实例引用，调用 create()方法获取一个 StatefulAccount 组件接口的实例引用，然后就可以与本地一样去引用这个实例。使用完毕后不要忘记将实例的引用指向一个 Null。
在 create 方法中，我们为基金帐户初始化一个数值，通过引用，把这个值传递给 StatefulAccountEJB 中定义的 fundBalanced 对象，注意这个对象的状态由容器维护。通过 addFund、removeFund 和 getBalance 方法我们可以测试 fundBalanced 状态改变所带来的影响。

下面是提供的代码：

StatefulAccountServlet .java 文件：

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import javax.ejb.*;
import javax.naming.InitialContext;

public class StatefulAccountServlet extends HttpServlet{
  public void service(HttpServletRequest req,HttpServletResponse res) throws IOException {
     res.setContentType("text/html");
     PrintWriter out =res.getWriter();
     out.println("<html><head><title>StatefulAccountServlet</title></head>");
     out.println("<body><h2>Test Result:<hr>");
        try{
            InitialContext ctx =new InitialContext();
            Object objRef =ctx.lookup("java:comp/env/ejb/StatefulAccount");
            StatefulAccountHome home=(StatefulAccountHome)
               javax.rmi.PortableRemoteObject.narrow(
               objRef,StatefulAccountHome.class);

            //得到一个基金帐户对象的引用，并初始化帐户金额为 100000
            StatefulAccount bean=home.create(100000);
            out.println("the account balance:"+bean.getBalance()+"<br>");
            //向基金帐户增加 5000.25
            bean.addFunds(5000.25);
            out.println("method of addFunds(5000.25) Result:"+bean.getBalance()+"<br>");
            //从基金帐户调出 1000.02
            bean.removeFunds(1000.02);
            out.println("method of removeFunds(1000.02) Result:"+bean.getBalance());
            out.println("<hr>");
            out.println("current account balance:"+bean.getBalance());

            bean=null;

        }catch(javax.naming.NamingException ne){
            out.println("Naming Exception caught:"+ne);
```

```
            ne.printStackTrace(out);
        }catch(javax.ejb.CreateException ce){
            out.println("Create Exception caught:"+ce);
            ce.printStackTrace(out);
        }catch(java.rmi.RemoteException re){
            out.println("Remote Exception caught:"+re);
            re.printStackTrace(out);
        }catch(InsufficientFundException ie){
            out.println("InsufficientFund Exception caught:"+ie);
            ie.printStackTrace(out);
        }
        out.println("</body></html>");
  }
}
```

假设我们将文件保存到 D:\ejb\StatefulAccount\src\StatefulAccountServlet.java

使用如下命令编译 Servlet

```
cd D:\ejb\StatefulAccount
mkdir test
cd test
mkdir WEB-INF
cd WEB-INF
mkdir classes
cd D:\ejb\StatefulAccount\src
javac -classpath %CLASSPATH%;../classes/ -d ../test/WEB-INF/classes
StatefulAccountServlet.java
```

编译成功后将这个 servlet 部署到与 StatefulAccount 同一工程中，在部署前需要我们为部署编写一个 web.xml,并制作 Web 模块文件（war 文件），其中告诉部署工具这个 servlet 需要参考的一些资源和部署信息，这里我们将定义一个 JNDI 参考：

```
  <ejb-ref>
    <description></description>
    <ejb-ref-name>ejb/StatefulAccount</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>StatefulAccountHome</home>
    <remote>StatefulAccount</remote>
    <ejb-link>StatefulAccount</ejb-link>
  </ejb-ref>
```

<ejb-ref-name>告诉部署工具，这个 Web 模块需要参考的资源 JNDI 名称，<ejb-ref-type>被指定参考一个 SessionBean 组件。

web.xml 文件内容如下：

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

```
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
  <icon>
    <small-icon></small-icon>
    <large-icon></large-icon>
  </icon>
  <display-name>StatefulAccount</display-name>
  <description></description>
  <context-param>
    <param-name>jsp.nocompile</param-name>
    <param-value>false</param-value>
  </context-param>
  <context-param>
    <param-name>jsp.usePackages</param-name>
    <param-value>true</param-value>
    <description></description>
  </context-param>
  <ejb-ref>
    <description></description>
    <ejb-ref-name>ejb/StatefulAccount</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>StatefulAccountHome</home>
    <remote>StatefulAccount</remote>
    <ejb-link>StatefulAccount</ejb-link>
  </ejb-ref>
</web-app>
```

假设我们将文件保存到 D:\ejb\StatefulAccount\test\WEB-INF\web.xml

下面我们要部署这个 Servlet 到 J2EE 服务器。J2EE Web 应用可以包括 Java Servlet 类、JavaServer Page 组件、辅助的 Java 类、HTML 文件、媒体文件等，这些文件被集中在一个 War 文件中。其中 War 结构具有固定的格式，根目录名为 WEB-INF，同一目录下应该有一个 web.xml 文件，用来描述被部署文件的部署信息，Jsp、html 等文件可以放置在这个目录下，同时 WEB-INF 目录下可能存在一个 classes 目录用于存放 Servlet 程序，如果引用了一些外部资源，则可以被放置到 WEB-INF\lib 目录下。使用下面的命令生成这个 Servlet 测试程序的 war 文件：

```
cd D:\ejb\StatefulAccount\test\
jar -cvf statefulAccount.war *.*
```

确保 statefulAccount.war 文件包括的文件目录格式如下：

- WEB-INF<文件夹>
  - **classes**<文件夹>
    - **StatefulAccountServlet.class**
  - web.xml

成功编译后，将这个 servlet 一同部署到 statefulAccount 工程中，我们回到“部署工具”，点击编辑 **à** 填加一个 Web 模块,选择我们刚刚编译成的 statefulAccount.war 文件
点击部署—>部署到 Apusic 应用服务器完成部署工作。

运行测试程序

打开浏览器，在浏览器中输入：
http://localhost:6888/statefulAccount/servlet/StatefulAccountServlet
localhost—Web Server 的主机地址
:6888—应用服务器端口，根据不同的应用服务器，端口号可能不同
/statelessfulAccount—部署 servlet 时指定的 WWW 根路径值
/servlet—ejb 容器执行 servlet 的路径
/StatefulAccountServlet—测试程序

如果运行正常应该能够看到下面的结果。

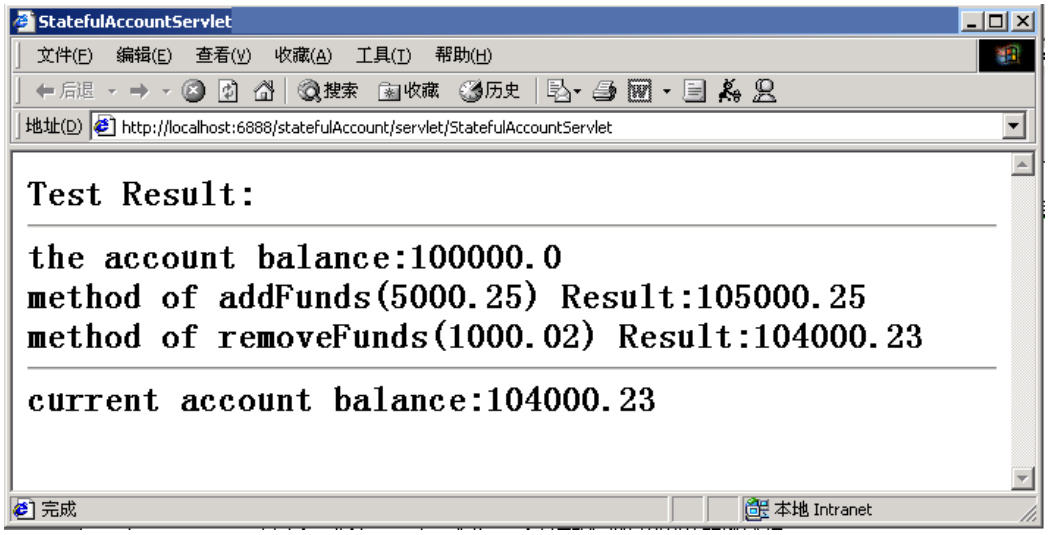


图 3-2

实战 EJB 之四 开发实体 CMP（EJB 1.1 规范）

在前面的几篇文章里我介绍了如何开发会话 Bean，下面将向大家介绍如何开发 Entity Bean，首先充实一些关于 Entity Bean 的基本知识。

实体(entity) bean 用来代表底层的对象，最常用的是用 Entity Bean 映射关系数据库中的记录。在一个 Entity Bean 中，关系型数据库的字段可以被一对一的映射到一个 Entity Bean 中，而表与表之间的关系就可以看成是 Entity Bean 之间的关系。一个 Entity Bean 的实例可能会对应表中一个特定的行记录描述或者对于一个查询结果。比如我们在数据库中设计了一个 BOOK 表，一个相对于 BOOK 表的 Entity Bean 就可以封装表中的部分或全部字段，当客户端获取一个 Book Bean 的实例引用时，就如同我们使用一个 SELECT 语句从数据库中检索了一条特定的关于一本图书的记录，并可以通过对象方法的方式去访问记录的值，当然你也可以使用 remove 方法去删除这条记录，用 setXXX 去改变某个字段的值，新的 EJB 2.0 查询语言（EJB QL,EJB 2.0 query language）使你可以通过 SELECT 的方式直接从组件池中查询 Bean。

由于这种 Bean 对应于数据库中的记录，所以数据库记录的任何改变也应该被同步到我们的组件池中相关的 Bean 中，这个过程被成为持久性（persistencd），这是 Entity Bean 最重要的一个特征。根据持久性的管理者的不同分为：容器管理持久性（CMP,Container-Managed Persistence）和 Bean 管理持久性(BMP,Bean-Managed Persistence)。何谓容器管理者，就是在 Bean 与基础数据库表记录值之间负责同步工作的操作者。

CMP Bean 的持久性由 EJB 容器负责持，Bean 开发者不需要参与操作数据库的代码部分，与数据库的操作在部署 EJB 时由 EJB 部署者描述，由容器实现 SQL 操作和同步工作。BMP Bean 的持久性由 Bean 负责，也就是由 Bean 开发者负责与数据库交互的代码部分。

Entity Bean 支持 EJB 的 1.1 和 2.0 规范，并且不能同时支持两者，我们将按照规范 1.1 和 2.0 分别介绍 BMP 和 CMP 的特性，本节将主要介绍 CMP 在 EJB 1.1 规范定义下的应用。当然上面的这些知识不足使你全部了解 Entity Bean，你应该从相关的书籍或文章阅读有关的介绍。

在本节中你将了解到：

- EJB 1.1 规范中的 CMP
- Entity Bean 的寿命周期
- 编写一个 EJB 1.1 的 CMP 程序
- 部署到应用服务器
- 开发和部署测试程序
- 运行测试程序

EJB 1.1 规范中的 CMP

首先介绍一下容器持久性管理（CMP），然后介绍规范 1.1 中规定的 CMP。

EJB 结构的一个重要优点是 EJB 容器可以自动的为 Entity Bean 提供各种有用的功能，容器持久管理(CMP)可以使 Bean 开发者不用编写一行对数据库操作的代码就可以完成对数据库的基本操作，这样可以简化 Bean 的开发，使我们集中于纯业务逻辑部分，这也是 EJB 的一个目标。以为使用 CMP 方式编写的 Bean 对于数据库的操作是在部署时由部署者映射

到实际的数据库字段的，所以这样就增强程序的移植性，CMP Bean 的不会为某种特定的数据库去设计。如果你还对 CMP 不甚了解，下面可以帮助你迅速解答你一部分疑问：

CMP Bean 如何连接到数据库？

如果你是一个 Bean 的开发者，打消这个念头吧，因为你已经不许要考虑这些问题了！这些工作将在在部署 Bean 时由部署者为 CMP Bean 指定一个数据库连接池的 JNDI 命名。Java 应用服务器提供数据库连接池管理，并可以通过 JNDI 命名来获得一个引用。当我们要改变数据库类型或改变数据库的连接地址时，只需从新配置这个数据库资源即可。

CMP Bean 如何映射一个数据表？

这是一个值得考虑的问题，因为这是你在设计一个具体应用是考虑使用 CMP 还是使用 BMP 的依据之一。在设计一个 CMP Bean 时，Bean 被固定映射一个实体表，表中的每个指定字段被映射成 bean 的一个 public 型类变量，在实际开发中，只需要在 Bean 的实现类中声明这些类变量，映射操作和 SQL 处理被交于部署者和容器自动完成。当然你可以迅速的开发出一个 CMP Bean，但可能会因为复杂的数据逻辑处理而放弃使用 CMP Bean 而采用 BMP Bean，至少在规范 1.1 版本，对 CMP Bean 规范的定义带来束缚还是比较大。

CMP Bean 主键如何理解？

只有 Entity Bean 有主键，Session 调用主键方法将抛出一个异常。Entity Bean 是数据面向数据对象的表示，每个 Bean 的实例代表一行记录，所以就必须有一个主键来标识这个对象，以能够对其进行持久性操作。

CMP Bean 由容器来负责实例的生成、装入、寻找、更新、删除等，所以主键也由容器来控制。对于 CMP Bean，javax.ejb.EJBObject 类已经为我们定义了一个默认的构造方法 public abstract Object getPrimaryKey()，并且不需要我们再为其改造。主键类型一般对应于数据表主关键字类型，比如在表 BOOK 中定义了一个 Varchar2 类型的关键字，那么应该告诉这个 Bean 的 PrimaryKey 的类型应该是一个 java.lang.String 类型的。如果你仔细，会发现默认的 getPrimaryKey()返回的是一个 Object 类，既然我们没有重新改造这个类，容器是如何知道的呢？EJB 没有那么聪明，CMP Bean 的主键是在部署者部署 Bean 时被指定的。比如我在部署 Book 这个 Bean 类时，就会告诉部署工具，我设计的这个类的主键对应数据表中哪个字段，同时其字段类型被映射一个 Java 的类型。在 Bean 的实现类中 ejbCreate()方法里，CMP Bean 返回一个 NULL 类型的值，BMP Bean 返回一个主键类型对象；在 Bean 的远程主接口中，create 方法用来插入一条数据，并根据 ejbCreate()方法返回的值返回一个 Bean 的引用（组件接口）。

规范 1.1 定义了设计一个 CMP Bean 的接口、部署规范和 CMP Bean 的能力：

CMP Bean 和会话 Bean 一样需要设计远程主接口、组件接口和 Bean 的实现类。远程主接口扩展了 javax.ejb.EJBHome 接口，组件接口扩展了 javax.ejb.EJBObject 接口，这两个接口的设计与会话 Bean 设计相似。组件实现类实现了 javax.ejb.EntityBean 接口，并用来实现一些组件的业务逻辑方法。

在一个实际的业务对象中，可能会要求一个 Bean 映射成多个数据表，在规范 1.1 中，可以由两种方法实现：

1. 另一个 Entity Bean
   可以用另一个 Entity Bean 来降低数据表的关系复杂度，比如对于一个定单 Bean，可能包括多条关于购买图书的定单项目的引用，可以设计另一个定单项目 Entity Bean 来解决这种问题。
2. 简单的 Java 类
   可以使用一个简单的 java 数据结构，比如将数据存放到 java.util.LinkedList 类中，

通过序列化把其保存到数据库表的一个字段值中。由于这种方法违背了关系型数据库的设计原则，并且容易在同一个事务总造成数据重影问题，所以不被推荐。

规范 1.1 规定 CMP 的查询方法在部署时被部署者指定， Bean 开发者不需要实现，只需在组件接口中定义即可。

Entity Bean 的寿命周期

前几节介绍了会话 Bean 的寿命周期，与会话 Bean 不同的是，Entity Bean 的寿命将超过创建它的客户端寿命，尽管客户在调用完一个 Entity Bean 释放其资源后，Entity Bean 的实例本身仍然存在于组件池中，与映射的数据库记录保持持久性。图 4-1 画出了 Entity Bean 的状态图：

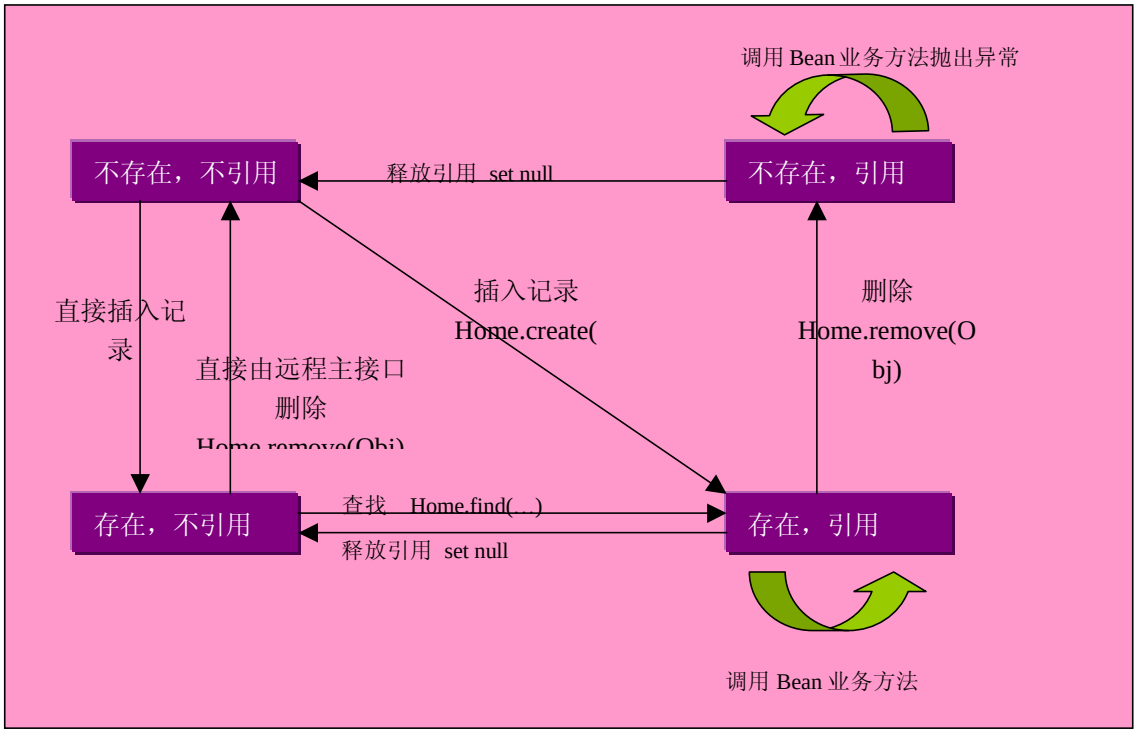


图 4-1

从图 4-1 中看出，Entity Bean 起初状态为“不存在，不引用”。当客户直接向数据库插入数据记录后，新的记录将被映射 Bean 的实例放到组件池中等待引用，并改变状态为“存在，不引用”，此时可以通过主接口的 find 方法查找这些对象，也可以由主接口调用 remove()方法将其删除。当客户通过远程主接口 create()方法创建一个对象引用时，一个 Entity Bean 状态从“不存在，不引用”改变为“存在，引用”，引用的句柄由 create 方法返回。只有 Entity Bean 处于“存在，引用”时，才可以调用组件的业务方法。将组件的引用指向为 NULL 将会释放该客户的引用资源，改变状态为“存在，不引用”，当一个 Entity Bean 的状态被“存在且引用”时，使用主接口的 remove 方法或组件接口的 remove 方法将删除被映射的数据记录，释放 Entity Bean 实例资源，但引用资源仍未释放，所以此时的状态改变为“不存在，引用”，将组件的引用设置成 Null 值，释放组件引用资源后，组件状态恢复到原来的“不存在，不引用”。分析图 4-1 的四个状态，可以得出，当一个 Entity Bean 处于不存在状态时，

其映射的数据库记录也不存在，当数据库记录被其他应用程序或进程直接插入数据后，容器将自动维持其持久性特性，在组件池中为新添的记录创建尚未被引用的实例 Bean,在客户端执行完一个 Entity Bean 的调用后一定要释放引用的资源，既设置引用为 Null。

Entity Bean 的寿命周期图如图 4-2 所示：

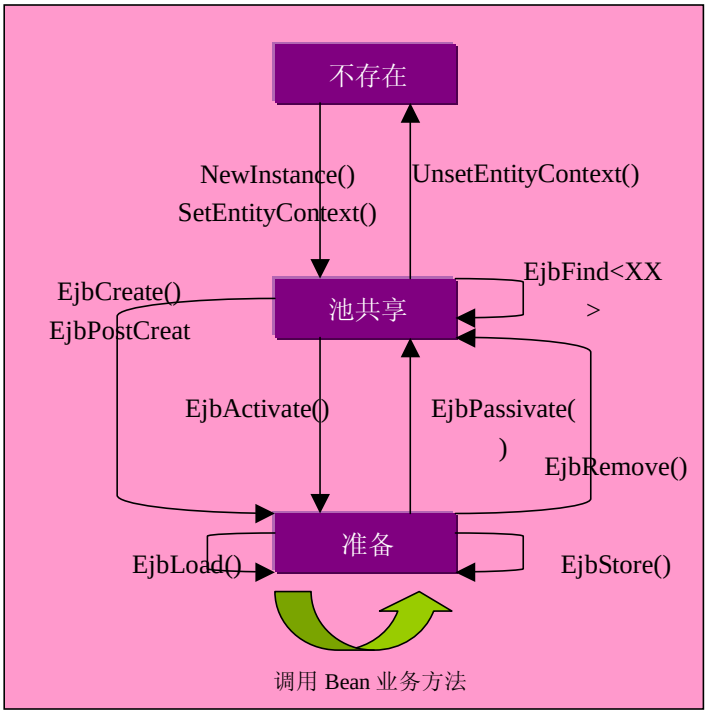


图 4-2

在图 4-2 中显示了 Bean 的实现类在不同阶段所调用的方法，这些方法大部分实现了 javax.ejb.EntityBean 的接口。当一个远程客户调用远程主接口的 create()方法时，容器调用 newInstance()方法创建一个 Bean 实例，然后调用 setEntityContext(..)方法将当前的情境传递给 Bean，进入池共享阶段。如果调用来客户的 create()方法，将调用组件的 ejbCreate()方法和 ejbPostCreate()方法，完全初始化 Bean 状态，并返回这个 Bean 的引用，此时 Bean 进入准备阶段，进入准备阶段的 Bean 业务逻辑方法可以被客户调用，在调用 setXX 或 getXX 等操作时，容器(CMP)或 Bean(BMP)可能会多次调用更新（ejbStore()方法）和提取（ejbLoad()方法）来维护组件的持久性（persistence）。

编写一个 EJB 1.1 的 CMP 程序

我们计划要设计一个关于一个定单系统中描述商品：图书的 CMP Bean，这个 Bean 里包括了图书的编号，书名和定价几个字段，通过 findInPrice()方法，我们可以从数据库中查询符合某个定价范围的图书。为这个 Bean 起名为 **Cmp1Book**，Cmp 表示这个 Bean 是一个 CMP，1 代表使用了 1.1 规范，这样起名完全没有任何根据，只是因为我们接下来的几节还会以 Book 为例测试 Entity Bean 的特性，所以这样起名只是防止出现命名重复。

图 4-3

设计一个 CMP Bean 至少包括四个步骤：

- 开发主接口
- 开发组件接口
- 开发 Bean 实现类
- 编写部署文件

注意：本节假设你使用的 Windows 操作系统。如果使用其他操作系统，可能影响到存储路径和 JDK 命令，但这与程序代码和部署文件内容无关。

1.开发主接口（Cmp1BookHome.java）：

开发主接口与开发 Session Bean 主接口相似，同样需要扩展 javax.ejb.EJBHome 接口，关于 javax.ejb.EJBHome 接口的定义和介绍请参考有关 Session Bean 开发的章节，此处不在详述。

一般情况下，习惯将主接口的命名规则规定为<bean-name>Home，所以我们把这个主接口类起名为 **Cmp1BookHome**

大部分逻辑方法已经被 EJBHome 定义，在我们要设计的远程主接口中，不必再重新定义。值得注意的是，我们需要为这个接口定义一个 create()方法，用来创建一个实例 Bean 的引用，返回的对象类型是组件接口类 **Cmp1Book**。参数 bookid、bookname、bookprice 将在初始化一个 Bean 时被引用，并根据此值为数据库插入一条记录。记住 session bean 的 create()方法是创建并取得一个 Bean 的引用，而 Entity Bean 则是向数据库插入一条记录，并返回这条记录的映射对象。此外，我们还必须声明 findByPrimaryKey()方法，这是在设计 Session Bean 所不需要的，findByPrimaryKey()方法的入口参数类型是组件的关键字类型，通过给定的关键字值，可以从组件池中查询相关的组件实例，并返回对这个组件实例的引用。findInPrice（）方法声明两个 double 型的参数，用来指定一个书的定价范围，按照范围值从组件容器中检索 Bean 的实例对象，将符合条件的 Bean 放到一个 Collection 结构中。在 CMP 中，类似 find<xxx>的 SQL 查询方法的实现由部署工具完成，不需要 Bean 开发者参与，所以我们再此声明这两个方法接口即可。

Cmp1BookHome.java 代码：

```
import java.util.Collection;
import java.rmi.RemoteException;
import javax.ejb.*;
//EJB CMP 1.1 实战例子
public interface Cmp1BookHome extends EJBHome{
    public Cmp1Book create(String bookid,String bookname,double bookprice)
        throws RemoteException,CreateException;
```

```
    //按主键[bookid 字段]查找对象
    public Cmp1Book findByPrimaryKey(String bookid)
      throws FinderException,RemoteException;
    //查找定价符合范围内的图书，将结果放到 Collection 中
    public Collection findInPrice(double lowerLimitPrice,double upperLimitPrice)
      throws FinderException,RemoteException;
}
```

假设我们保存到 D:\ejb\Cmp1Book\src\Cmp1BookHome.java

2.开发组件接口(Cmp1Book.java)：

开发组件接口与开发 Session Bean 组件接口相似，同样需要扩展 javax.ejb.EJBObject 接口，关于 javax.ejb.EJBObject 接口的定义和介绍请参考有关 Session Bean 开发的章节，此处不在详述。

一般情况下，习惯将组件接口的命名规则规定为<bean-name>，所以我们把这个组件接口类起名为 **Cmp1Book**

组件接口类声明的接口方法必须在 BEAN 实现类中实现。大家注意到接口方法中没有声明对 BOOKID 的操作是因为 BOOKID 作为组件的主键有其默认的操作方法，使用 GETPRIMARYKEY（）方法可以获取组件的主键值，注意返回的是一个 OBJECT 类型，但是在你的客户端程序中可以通过上溯造型成合适的类型。

Cmp1Book.java 代码：

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
//EJB CMP 1.1 实战例子
public interface Cmp1Book extends EJBObject{
    public void setBookName(String bookname) throws RemoteException;
    public void setBookPrice(double bookprice) throws RemoteException;
    public String getBookName() throws RemoteException;
    public double getBookPrice() throws RemoteException;
}
```

假设我们保存到 D:\ejb\Cmp1Book \src\Cmp1Book .java

3.开发 Bean 实现类(Cmp1BookEJB.java)：

这个类包含了业务逻辑的所有详细设计细节。Entity Bean 的实现类实现了(implements)javax.ejb.EntityBean 所定义的接口，首先我们先熟悉一下 **EntityBean** 的定义：

```
package javax.ejb;
impot java.rmi.RemoteException;

public interface EntityBean extends EnterpriseBean{
   public abstract void ejbActivate() throws EJBException,RemoteException;
   public abstract void ejbLoad() throws EJBException,RemoteException;
```

```
    public abstract void ejbPassivate() throws EJBException,RemoteException;
    public abstract void ejbRemove() throws RemoveException,EJBException,RemoteException;
    public abstract void ejbStore() throws EJBException,RemoteException;
    public abstract void setEntityContext(EntityContext entitycontext)
        throws EJBException,RemoteException;
    public abstract void unsetEntityContext() throws EJBException,RemoteException;
}
```

EjbActivate()方法和 ejbPassivate()方法在 Bean 激活或钝化时被调用，ejbLoad()方法从数据库中读取数据记录，ejbStore()方法提交当前数据状态到记录。EjbRemove()方法释放实例对象并删除相关映射的数据记录。SetEntityContext()方法可以使当前 Bean 实例访问 Bean 情境，unSetEntityContext()方法释放特定的情境资源。

Entity Bean 激活时的调用顺序：ejbActivate() à ejbLoad()

Entity Bean 钝化时的调用顺序：ejbStore() à ejbPassivate()

一般情况下，习惯将组件实现类的命名规则规定为<bean-name>EJB，所以我们把这个组件类起名为 **Cmp1BookEJB**

类 Cmp1BookEJB 声明要实现 EntityBean 的定义，所以，我们必须完全实现 EntityBean 的接口定义。除此之外还必须实现 ejbCreate()方法和 ejbPostCreate()方法，ejbCreate()方法必须匹配主接口 Cmp1BookHome 对 create()方法的声明，而 ejbPostCreate()方法只需实现主接口 create()方法的声明即可。

对于 CMP 的 ejbCreate()方法的声明返回类型为主关键字类型，但是由于是容器来实现，所以只需在此方法中关联相关的映射字段，然后返回 NULL 即可。必须在类中定义与数据库表相关联的映射字段，并声明成 Public 的类变量，因为这些类变量将被容器所引用。

这个类没有书写过多的业务逻辑，基本的存储逻辑已被容器所实现。

Cmp1BookEJB.java 代码：

```
import java.util.*;
import javax.ejb.*;

//EJB CMP 1.1 实战例子
public class Cmp1BookEJB implements EntityBean{
    //映射 bookid 字段
    public String bookid;
    //映射 bookname 字段
    public String bookname;
    //映射 bookprice 字段
    public double bookprice;

    public void setBookName(String bookname){
       this.bookname=bookname;
    }
```

```
    public void setBookPrice(double bookprice){
       this.bookprice=bookprice;
    }

    public String getBookName(){
       return this.bookname;
    }

    public double getBookPrice(){
       return this.bookprice;
    }

    public String ejbCreate(String bookid,String bookname,double bookprice)
       throws CreateException{

       if(bookid==null)
          throw new CreateException("The bookid is required");
       this.bookid=bookid;
       this.bookname=bookname;
       this.bookprice=bookprice;

       return null;
    }

    public void ejbPostCreate(String bookid,String bookname,double bookprice){}
    public void ejbLoad(){}
    public void ejbStore(){}
    public void ejbRemove(){}
    public void unsetEntityContext(){}
    public void setEntityContext(EntityContext context){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
}
```

假设我们保存到 D:\ejb\Cmp1Book\src\Cmp1BookEJB .java

到此为止我们的 Bean 程序组件已经编写完毕了，使用如下命令进行编译：

```
cd bean\Cmp1Book
mkdir classes
cd src
javac -classpath %CLASSPATH%;../classes -d ../classes    *.java
```

如果顺利你将可以在..\Cmp1Book\classes 目录下发现有三个类文件。

4．编写部署文件：

一个完整的 ejb 是由 java 类和一个描述其特性的 ejb-jar.xml 文件组成，部署工具将根据这些文件部署到容器中，并自动生成容器所需的残根类。

按照下面个格式编写一个 ejb-jar.xml 文件，对于 DTD 介绍此处省去。

ejb-jar.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <description>
        This is CMP 1.1 Book EJB example
    </description>
    <display-name>Cmp1BookBean</display-name>
    <enterprise-beans>
        <entity>
            <display-name>Cmp1Book</display-name>
            <ejb-name>Cmp1Book</ejb-name>
            <home>Cmp1BookHome</home>
            <remote>Cmp1Book</remote>
            <ejb-class>Cmp1BookEJB</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.String</prim-key-class>
            <reentrant>False</reentrant>

            <cmp-field>
               <description>no description</description>
               <field-name>bookid</field-name>
            </CMP-FIELD>
            <cmp-field>
               <description>no description</description>
               <field-name>bookname</field-name>
            </cmp-field>
            <cmp-field>
               <description>no description</description>
               <field-name>bookprice</field-name>
            </cmp-field>
            <primkey-field>bookid</primkey-field>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
```

```
        <container-transaction>
          <method>
            <ejb-name>Cmp1Book</ejb-name>
            <method-name>*</method-name>
          </method>
          <trans-attribute>NotSupported</trans-attribute>
        </container-transaction>
      </assembly-descriptor>
</ejb-jar>
```

假设我们保存到 D:\ejb\Cmp1Book\classes\META-INF\ejb-jar.xml(注意 META-INF 必须大写)

现在让我们看看当前的目录结构：

- Cmp1Book <文件夹>
  - classes<文件夹>
    - META-INF<文件夹>
      - ejb-jar.xml
    - Cmp1Book .class
    - Cmp1Book EJB.class
    - Cmp1BookHome.class
  - src<文件夹>
    - Cmp1Book.java
    - Cmp1BookEJB.java
    - Cmp1BookHome.java

部署到应用服务器

在部署之前我们需要将这些类文件和 xml 文件做成一个 jar 文件，EJB JAR 文件代表一个可被部署的 JAR 库，在这个库里，包含了服务器代码与 EJB 模块的配置。ejb-jar.xml 文件被放置在 JAR 文件所指定的 META-INF 目录中。我们可以使用如下命令得到 EJB JAR 文件：

```
cd d:\ejb\Cmp1Book\classes      (要保证类文件在这个目录下，且有一个 META-INF 子目录存放 ejb-jar.xml 文件)
jar -cvf    cmp1Book.jar *.*
```

确保 cmp1Book.jar 文件包括的文件目录格式如下：

- <根>
  - META-INF<文件夹>
    - ejb-jar.xml
  - Bmp1Book.class
  - Bmp1BookEJB.class
  - Bmp1BookHome.class

部署工具一般由 Java 应用服务器的制造商提供，在这里我使用了 Apusic 应用服务器，并讲解如何在 Apusic 应用服务器部署这个组件。

注意，如果使用其他部署工具，原理是一样的。要使用 Apusic 应用服务器，可以到 www.apusic.com 上下载试用版。

确定你的 Apusic 服务器已经被启动。
打开“部署工具”应用程序，点击文件—>新键工程：
第一步：选择“新建包含一个 EJB 组件打包后的 EJB-jar 模块”选项
第二步：选择一个刚才我们生成的 cmp1Book.jar 文件，
第三步：输入一个工程名，可以随意，这里我们输入 cmp1Book
第四步：输入工程存放的地址，这里我们假设被存放到 D:\ejb\cmp1Book\deploy 目录下
完成四个步骤后，如果没有问题将出现 cmp1BookBean 的部署界面，基本的参数配置已经在我们刚才编写的 ejb-jar.xml 中定义，但是比部署 SessionBean 稍微复杂的是，需要提供一些 EntityBean 特性的配置：

选择 cmp1Book 的配置页，点击“实体 EJB 的持续性管理”，点击“部署”按钮，进入如下画面：

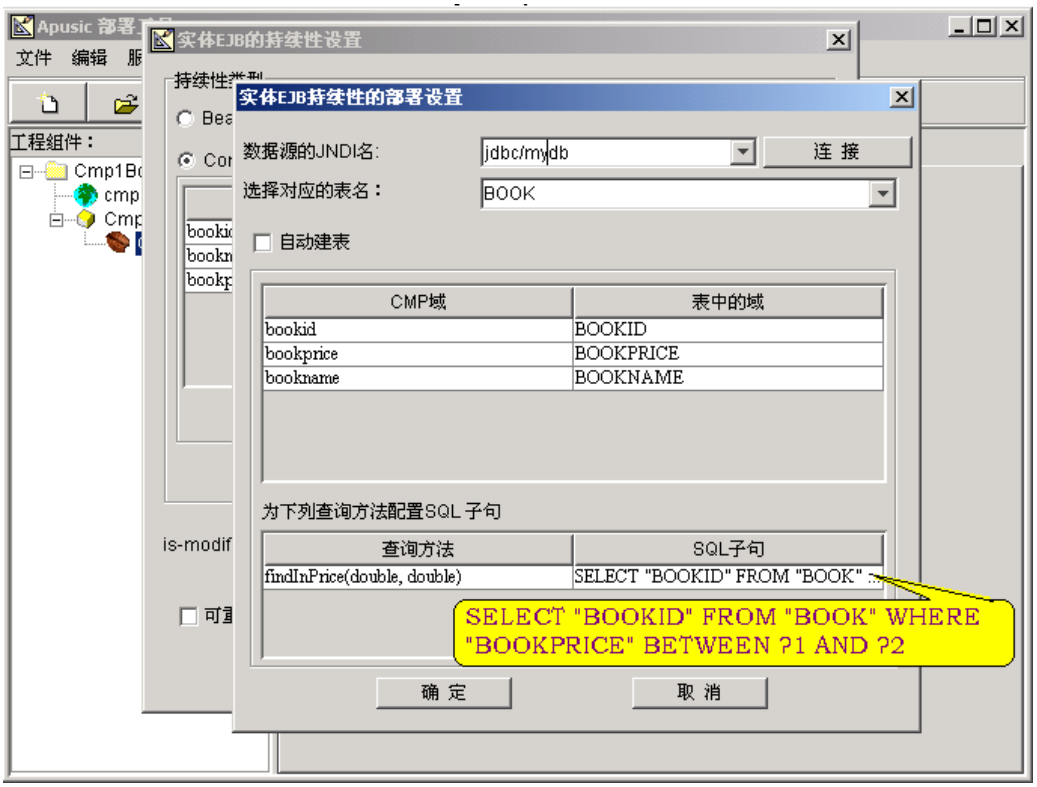


图 4-4

数据源的 JNDI 名：

必须由部署者给出当前组件的可用数据源 JNDI 名，一般每个应用服务器都会提供配置数据库连接池的功能。为了保证 Cmp1Book 组件最终能够在测试环境下运行，我介绍一下在 Apusic 服务器中配置数据库连接池的方法，其他应用服务器基本类似。编辑 config 目录下的 datasources.xml 文件，在<datasources>下添加下面的代码：

```
<datasource name="OracleTest"
    jndi-name="jdbc/mydb"
    driver-class="oracle.jdbc.driver.OracleDriver"
    url="jdbc:oracle:thin:@127.0.0.1:1521:Test"
    min-spare-connections="5"
    max-spare-connections="30"
    idle-timeout="300"
>
    <property name="user" value="jackliu"/>
    <property name="password" value="go"/>
</datasource>
```

当然，可以根据你的实际需要把 oracle 数据库换为其他的数据员，这些改动不会影响到 Bean。配置完毕后，重新启动应用服务器。

选择对应的表名：

假定这个 CMP Bean 的数据表名为 BOOK，添入这个值。

自动建表：

如果选定此项，在部署时，将由部署工具自动为你在数据库中创建所需的 BOOK 表结构。我不提倡这种懒的办法，因为自动创建的表结构对字段属性并没有优化，比如长度，类型，并且不会创建一些可以提高、优化数据库查询效率的各类索引，所以如果你是一个有经验的数据库开发人员不要养成这种习惯，当然如果你对关系型数据库不熟悉或这个逻辑对表结构关系要求不高，使用此项可以降低你的开发难度。假设我们手工在数据库中创建一个表结构如下：

```
CREATE TABLE BOOK(BOOKID VARCHAR2(5) NOT NULL,BOOKNAME
VARCHAR2(64),BOOKPRICE NUMBER(12,2) ,PRIMARY KEY (BOOKID));
```

配置 SQL 子句：

我们已经知道在 CMP 中，所有的查询方法将在部署时被部署工具实现，对于 findByPrimaryKey()方法，由于只有一个 Where 关键字段=xx 的 WHERE 子句，所以，不需要部署者特别的指定，但是在我们的设计的远程主接口 Cmp1BookHome 中还定义了一个 findInPrice(double lowerLimitPrice,double upperLimitPrice)方法，需要部署者为这个方法指定一个 SQL 语句，填写下面的 SQL：

```
SELECT "BOOKID" FROM "BOOK" WHERE "BOOKPRICE" BETWEEN ?1 AND ?2
```

其中?1 表示接收 lowerLimitPrice 参数的值，并代入这个 SQL 语句
?2 表示接收 upperLimitPrice 参数的值，并代入这个 SQL 语句

上述步骤完成后就可以点击部署—>部署到 Apusic 应用服务器完成部署工作。

开发和部署测试程序

EntityBean 组件是没有任何运行界面的，组件的实例被容器所管理，所以我们要测试这个 Bean 组件，需要写一段测试程序。这里，我们写一段小服务程序（Java Servlet）。

关于如何编写 Servlet 我们这里不做介绍。使用 InitialContext 对象用来获取当前 servlet 小应用程序的语境，方法 lookup 从组件池中查找一个 JNDI 对象，并取得一个远程主接口的引用。java:comp/env/ejb/Cmp1Book 是我们刚才部署 Cmp1Book 组件的 JNDI 名，请参考 ejb-jar.xml 中的<ejb-name>项。要注意的是，lookup()方法返回的是一个 Object 类型的远程主接口对象的残根，为此需要使用 javax.rmi.PortableRemoteObject 的 narrow()方法来获取一个具体的对象引用，narrow()方法：第一个参数是 lookup()方法返回的对象，第二个参数是要得到的引用类型。我们通过 narrow()方法并经过造型得到了一个 Cmp1BookHome 对象的实例引用，调用 create()方法获取一个 Cmp1Book 组件接口的实例引用，然后就可以与本地一样去引用这个实例。使用完毕后不要忘记将实例的引用指向一个 Null。

下面是提供的代码：

Cmp1BookServlet .java 文件：

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
import javax.ejb.*;
import javax.naming.InitialContext;
import java.util.Collection;
import java.util.Iterator;

public class Cmp1BookServlet extends HttpServlet{
  public void service(HttpServletRequest req,HttpServletResponse res) throws IOException {
     res.setContentType("text/html");
     PrintWriter out =res.getWriter();
     out.println("<html><head><title>Cmp1BookServlet</title></head>");
     out.println("<body><h2>Test Result:<hr>");
        try{
            InitialContext ctx =new InitialContext();
            Object objRef =ctx.lookup("java:comp/env/ejb/Cmp1Book");
            Cmp1BookHome home=(Cmp1BookHome)
               javax.rmi.PortableRemoteObject.narrow(
               objRef,Cmp1BookHome.class);

            //插入书籍信息记录，并返回这个书籍对象的引用
            //-----------------------------------------------
            Cmp1Book bean;
            bean=home.create("BK001","Java in a NutShell",79.00);
            //输出 bean 的信息
            printBookInf(out,bean);

            bean=home.create("BK002","Graphic Java 2 Mastering the JFC",108.08);
            //输出 bean 的信息
            printBookInf(out,bean);

            bean=home.create("BK003","Thinking in JAVA",60.00);
            //输出 bean 的信息
            printBookInf(out,bean);

            bean=home.create("BK004","Building Java Enterprise System with J2EE",105.00);
            //输出 bean 的信息
            printBookInf(out,bean);
            //-----------------------------------------------


            //查找一个书籍编号为 BK003 的 Bean,注意由远程主接口 Cmp1BookHome 执行这
个方法
            //-----------------------------------------------
            out.println("<hr>the method of findByPrimaryKey('BK003'):<br>");
            bean=home.findByPrimaryKey("BK003");
```

```
        //输出 bean 的信息
        printBookInf(out,bean);
        //如果找到，修改单价为 100
        if(bean!=null){
           out.println("<br>the method of setBookPrice(100):<br>");
           bean.setBookPrice(100);
           //输出 bean 的信息
           printBookInf(out,bean);
        }
        //-------------------------------------------------


        //查找价格在 100 到 200 元之间的书籍
        //-------------------------------------------------
        out.println("<hr>the method of findInPrice(100,200):<br>");
        //-------------------------------------------------
        Collection bookCollection =home.findInPrice(100.0d,200.0d);
        Iterator it=bookCollection.iterator();
           while(it.hasNext()){
              Object objRef2=it.next();
              bean=(Cmp1Book)javax.rmi.PortableRemoteObject.narrow(
                     objRef2,Cmp1Book.class);
              //输出 bean 的信息
              printBookInf(out,bean);
              //删除数据记录
              bean.remove();
           }
        bean=null;

     }catch(javax.naming.NamingException ne){
        out.println("Naming Exception caught:"+ne);
        ne.printStackTrace(out);
     }catch(javax.ejb.CreateException ce){
        out.println("Create Exception caught:"+ce);
        ce.printStackTrace(out);
     }catch(java.rmi.RemoteException re){
        out.println("Remote Exception caught:"+re);
        re.printStackTrace(out);
     }catch (javax.ejb.FinderException fe){
        out.println("Finder Exception caught:"+fe);
        fe.printStackTrace(out);
      }catch (javax.ejb.RemoveException me){
        out.println("Remove Exception caught:"+me);
        me.printStackTrace(out);
```

```
        }
        out.println("</body></html>");
  }

  private void printBookInf(PrintWriter out,Cmp1Book book)
      throws java.rmi.RemoteException{
      if (book==null)
         return;

      out.println("<li>BookId:"+(String)book.getPrimaryKey()+"</li>");
      out.println("<li>BookName:"+book.getBookName()+"</li>");
      out.println("<li>BookPrice:"+book.getBookPrice()+"</li>");
      out.println("<br>");
  }
}
```

假设我们将文件保存到 D:\ejb\Cmp1Book\src\Cmp1BookServlet.java

使用如下命令编译 Servlet

```
cd D:\ejb\Cmp1Book
mkdir test
cd test
mkdir WEB-INF
cd WEB-INF
mkdir classes
cd D:\ejb\Cmp1Book\src
javac -classpath %CLASSPATH%;../classes/ -d ../test/WEB-INF/classes Cmp1BookServlet.java
```

编译成功后将这个 servlet 部署到与 Cmp1Book 同一工程中，在部署前需要我们编写一个 web.xml,并制作成一个 Web 模块文件（war 文件）

web.xml 文件内容如下：

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

<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
   <icon>
      <small-icon></small-icon>
      <large-icon></large-icon>
   </icon>
   <display-name>Cmp1BookServlet</display-name>
   <description></description>
   <context-param>
```

```
    <param-name>jsp.nocompile</param-name>
    <param-value>false</param-value>
  </context-param>
  <CONTEXT-PARAM>
    <param-name>jsp.usePackages</param-name>
    <param-value>true</param-value>
    <description></description>
  </context-param>
  <ejb-ref>
    <description></description>
    <ejb-ref-name>ejb/Cmp1Book</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>Cmp1BookHome</home>
    <remote>Cmp1Book</remote>
    <ejb-link>Cmp1Book</ejb-link>
  </ejb-ref>
</web-app>
```

假设我们将文件保存到 D:\ejb\Cmp1Book\test\WEB-INF\web.xml

J2EE Web 应用可以包括 Java Servlet 类、JavaServer Page 组件、辅助的 Java 类、HTML 文件、媒体文件等，这些文件被集中在一个 War 文件中。其中 War 结构具有固定的格式，根目录名为 WEB-INF，同一目录下应该有一个 web.xml 文件，用来描述被部署文件的部署信息，Jsp、html 等文件可以放置在这个目录下，同时 WEB-INF 目录下可能存在一个 classes 目录用于存放 Servlet 程序，如果引用了一些外部资源，则可以被放置到 WEB-INF\lib 目录下。使用下面的命令生成这个 Servlet 测试程序的 war 文件：

```
cd D:\ejb\Cmp1Book\test\
jar -cvf cmp1Book.war *.*
```

确保 cmp1Book.war 文件包括的文件目录格式如下：

- WEB-INF<文件夹>
  - **classes**<文件夹>
    - cmp1Book**Servlet.class**
  - web.xml

成功编译后，将这个 servlet 一同部署到 cmp1Book 工程中，我们回到“部署工具”，点击编辑 **à** 填加一个 Web 模块,选择我们刚刚编译成的 cmp1Book.war 文件
点击部署—>部署到 Apusic 应用服务器完成部署工作。

运行测试程序

打开浏览器，在浏览器中输入：
http://localhost:6888/cmp1Book/servlet/Cmp1BookServlet
localhost—Web Server 的主机地址
:6888—应用服务器端口，根据不同的应用服务器，端口号可能不同
/cmp1Book—部署 servlet 时指定的 WWW 根路径值
/servlet—ejb 容器执行 servlet 的路径
/Cmp1BookServlet—测试程序

如果运行正常应该能够看到下面的结果。

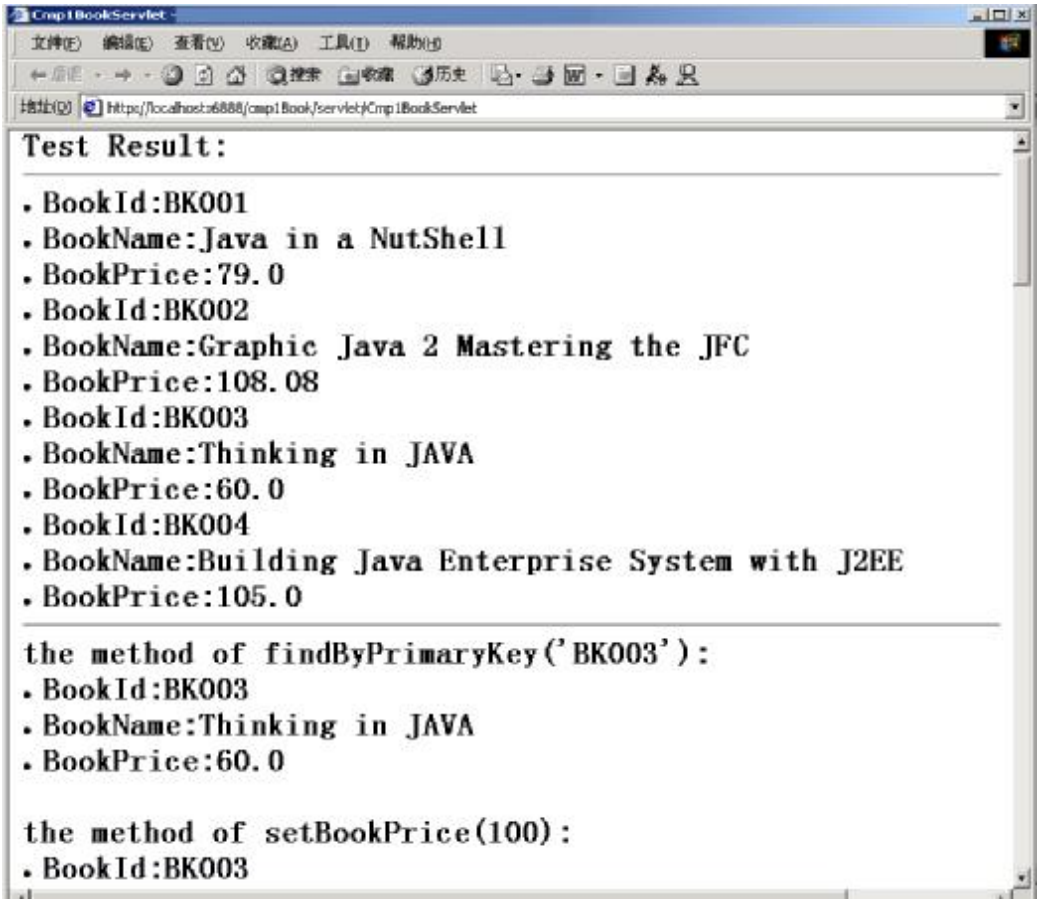


图 4-5

实战 EJB 之五 开发实体 BMP（EJB 1.1 规范）

前一节介绍了 EntityBean 的有关介绍，并通过开发、部署实体 CMP 的例子介绍 EJB1.1 规范的 CMP 的有关特性，在这一节中你将了解如下内容：

- EJB 1.1 规范中的 BMP
- 编写一个 EJB 1.1 的 BMP 程序
- 部署到应用服务器
- 开发和部署测试程序
- 运行测试程序

EJB 1.1 规范中的 BMP

根据规范中定义的 EJB 事务持久性（persistence）的特性被分为容器管理持久性（CMP）和 Bean 管理持久性(BMP)。虽然使用容器管理持久性给编程带来极大的方便，但是将事务持久性交于容器来控制降低了 Bean 的开发能力；BMP 的 Bean 具有灵活的业务处理能力和更灵活的持久性控制能力，常用来映射一些复杂的数据视图或很难用 CMP 实现的复杂逻辑处理。

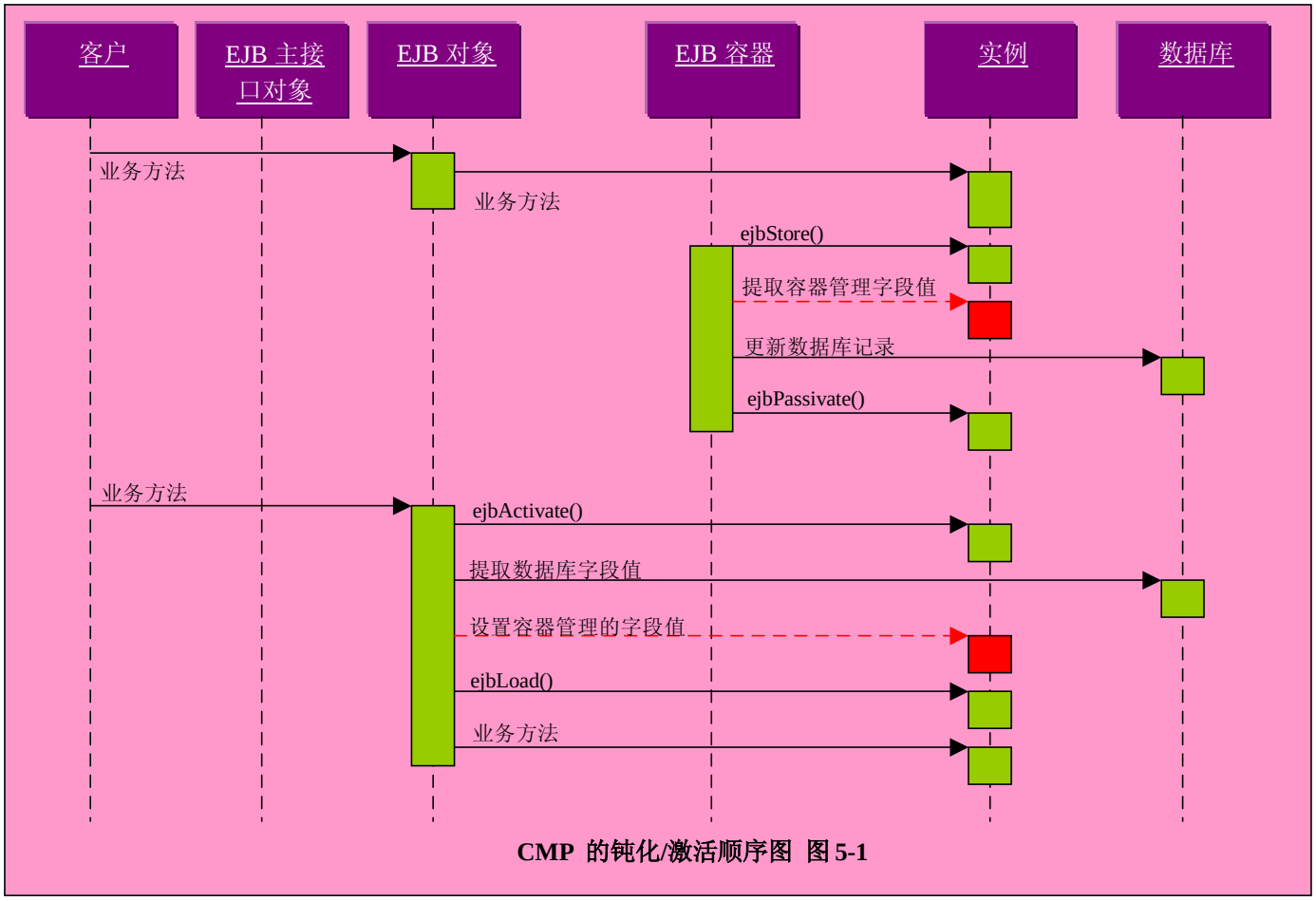
BMP 的寿命周期和 CMP 的寿命周期管理机制是相同的，不同的是 BMP 的事务持久性管理机制交于 Bean 开发者，所以，在创建、更新、删除等数据库操作时，两种类型的 Bean 的顺序图是不一样的。为了说明这一点，可以从 CMP 和 BMP 在钝化/激活顺序图中分析，当然 Bean 的创建、查找、删除也是不同的：

通过图 5-1 和 5-2 的比较，我们很容易会发现：

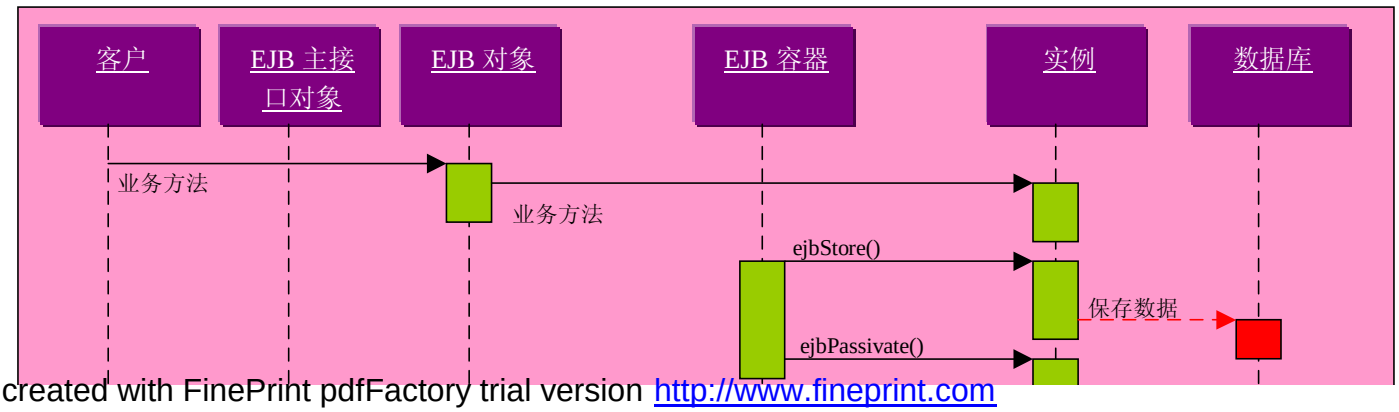
CMP：当一个 Bean 实例被客户引用，并执行一个业务方法后，容器会自动读取 Bean 的实例字段（还记得我们在上一节在实现一个 CMP 时，为 Bean 定义了映射到数据库字段的 Public 型类字段吗），然后，通过容器与数据库发生关系，保存改变的数据，执行完毕后 Bean 被钝化，并调用 ejbPassviate()方法通知 Bean。当客户过一端时间又调用这个 Bean 的某业务方法时，被钝化的 Bean 又重新的激活，但是并不是马上执行这个业务方法，而是由 EJB 对象首先调用 ejbActivate()方法通知 Bean，Bean 实例要激活，然后从数据库中提取数据，

并自动将数据值映射到 Bean 实例，然后调用 ejbLoad()方法，实例被再一次初始化，最后才开始执行要执行的业务方法，红色箭头和红色时间块做了明显的表示。

BMP：当一个 Bean 实例被客户引用，并执行一个业务方法后，容器会执行 Bean 的 ejbStore()方法，并由这个方法把数据保存到数据库中（下面的例子你将会发现，我们不再为 Bean 定义全局类变量，而是定义一些私有类变量），执行完毕后 Bean 被钝化，并调用 ejbPassviate()方法通知 Bean。当客户过一端时间又调用这个 Bean 的某业务方法时，被钝化的 Bean 又重新的激活，但是并不是马上执行这个业务方法，而是由 EJB 对象首先调用 ejbActivate()方法通知 Bean，Bean 实例要激活，然后调用 Bean 的 ejbLoad()方法，这个方法负责从数据库中提取数据，Bean 实例被初始化，最后才开始执行要执行的业务方法。



CMP 的钝化/激活顺序图 图 5-1



BMP Bean 要求所有的数据库操作都要由 Bean 实例完成，这些方法基本上包括：

setXXX();

因为 BMP 不在为容器声明 public 类型的由容器来管理的映射字段，所以 setXX 方法需要开发者实现

getXXX();

取得 Bean 字段值

ejbCreate();

在 CMP 中，由容器实现，并返回一个 NULL 值，在 BMP 中必须由开发者自己实现，返回创建记录的主键值

ejbLoad();

在 CMP 中，由容器实现，在 BMP 中必须由开发者自己实现，以实现组件非持久性状态缓存持久性信息

ejbStore();

在 CMP 中，由容器实现，在 BMP 中必须由开发者自己实现，将信息从组件的非持久性状态转到持久性状态

ejbRemove();

在 CMP 中，由容器实现，在 BMP 中必须由开发者自己实现

unsetEntityContext();

在情境要求被释放时，释放在 setEntityContext()中缓存的情境资源和取得的资源

setEntityContext();

设置情境资源，初始化数据库连接对象

ejbActivate();

在 CMP 中，由容器实现，在 BMP 中必须由开发者通过情境参数设置主键值

ejbPassivate();

在 CMP 中，由容器实现，在 BMP 中必须由开发者取消 Bean 与数据库记录的持久性工作，进入钝化状态

ejbFindByPrimaryKey();

在 CMP 中，由容器实现，在 BMP 中必须由开发者自己实现

ejbFindXXX();

在 CMP 中，由容器实现，在 BMP 中必须由开发者自己实现

……

总体来看，在规范 1.1 中，CMP 和 BMP 各有千秋，从机制上没有实质的差异，对于客户端的引用是不会察觉到两者的使用差异。不过是一个善于开发，灵活性小，且增加了部署工作（字段映射、编写 SQL 处理语句）；另一个不善于开发，灵活性大，部署工作较少（没有了字段影射等麻烦，但却增加了配置外部引用资源[因为 Bean 会通过一个 JNDI 来查找数据库连接池]，移植性较 CMP 差）。

关于 BMP 的寿命周期请参看上一节介绍的 EntityBean 寿命周期

编写一个 EJB 1.1 的 BMP 程序

上一节编写了一个 CMP 的例子，同样我们可以试者将它改写成一个 BMP，假设功能需求不变化（功能介绍参看第四节的相关章节），为这个 BMP 起名为 **Bmp1Book**

设计一个 BMP Bean 与 CMP 同样至少包括四个步骤：

- 开发主接口
- 开发组件接口
- 开发 Bean 实现类
- 编写部署文件

1.开发主接口（Bmp1BookHome.java）：

主接口的设计与 CMP 的主接口设计一样，参照上一节主接口的设计，改动之处用黑体加粗显示。

Bmp1BookHome.java 代码：

```
import java.util.Collection;
import java.rmi.RemoteException;
import javax.ejb.*;
//EJB BMP 1.1 实战例子
public interface Bmp1BookHome extends EJBHome{
    public Bmp1Book create(String bookid,String bookname,double bookprice)
       throws RemoteException,CreateException;
    //按主键[bookid 字段]查找对象
    public Bmp1Book findByPrimaryKey(String bookid)
       throws FinderException,RemoteException;
    //查找定价符合范围内的图书，将结果放到 Collection 中
    public Collection findInPrice(double lowerLimitPrice,double upperLimitPrice)
       throws FinderException,RemoteException;
}
```

假设我们保存到 D:\ejb\Bmp1Book\src\Bmp1BookHome.java

2.开发组件接口(Bmp1Book.java)：

组件接口的设计与 CMP 的组件接口设计一样，参照上一节组件接口的设计，改动之处

用黑体加粗显示。

Bmp1Book.java 代码：

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
//EJB BMP 1.1 实战例子
public interface Bmp1Book extends EJBObject{
    public void setBookName(String bookname) throws RemoteException;
    public void setBookPrice(double bookprice) throws RemoteException;
    public String getBookName() throws RemoteException;
    public double getBookPrice() throws RemoteException;
}
```

假设我们保存到 D:\ejb\Bmp1Book \src\Bmp1Book .java

3.开发 Bean 实现类(Bmp1BookEJB.java)：

最大的改动是 Bean 的实现类，这个类里将包括更多 SQL 实现的细节代码。首先要引用更多的开发包：java.sql.*;javax.sql.*;javax.naming.*;

Bean 不在声明全局的类变量，类变量的映射改较给 Bean 来管理。另外，还需要声明一个 EntityContext 情境变量，我们将通过这个变量的 getPrimaryKey()方法得到保存在情境中的主关键字值，以便在 Bean 在激活时重新初始化 Bean 数据。因为要对数据库直接操作，所以我们要定义一个 DataSource 对象，在 Bean 初始化时从连接池中取得一个有效数数据库对象。定义的 Connect 对象将在获取一个数据库连接时被引用。dbjndi 存放了一个获得数据库资源的 JNDI 名。改造后的 Bmp1BookEJB 如下：

Bmp1BookEJB.java 代码：

```
import java.util.*;
import javax.ejb.*;
//引入 sql 处理包
import java.sql.*;
import javax.sql.*;
import javax.naming.*;



//EJB BMP 1.1 实战例子
public class Bmp1BookEJB implements EntityBean{
    //保存 bookid 字段值
    PRIVATE STRING BOOKID;
    //保存 bookname 字段值
    private String bookname;
    //保存 bookprice 字段值
    private double bookprice;


    private EntityContext ctx;
    private DataSource ds;
```

```
PRIVATE STRING DBJNDI="JAVA:COMP/ENV/JDBC/OADB";
private Connection con;



public void setBookName(String bookname){
   this.bookname=bookname;
}

public void setBookPrice(double bookprice){
   this.bookprice=bookprice;
}

public String getBookName(){
   return this.bookname;
}

public double getBookPrice(){
   return this.bookprice;
}

//BMP 需要 Bean 提供数据的插入
public String ejbCreate(String bookid,String bookname,double bookprice)
   throws CreateException{


   if(bookid==null)
      throw new CreateException("The bookid is required");

   try{
      String sql="INSERT INTO BOOK VALUES(?,?,?)";
      con=ds.getConnection();
      PreparedStatement stmt =con.prepareStatement(sql);
      stmt.setString(1,bookid);
      stmt.setString(2,bookname);
      stmt.setDouble(3,bookprice);
      stmt.executeUpdate();
      stmt.close();
   }catch (SQLException se){
      throw new EJBException(se);
   }finally{
      try{
         if(con!=null)
            con.close();
```

```
        }catch(SQLException se){}
      }

      this.bookid=bookid;
      this.bookname=bookname;
      this.bookprice=bookprice;
      //由 Bean 负责事务持久性，Bean 负责返回主键值
      return bookid;
    }

    public void ejbPostCreate(String bookid,String bookname,double bookprice){}

    //根据 bookid 值提取数据
    public void ejbLoad(){
      try{
        String sql="SELECT BOOKID，BOOKNAME，BOOKPRICE FROM BOOK WHERE
BOOKID=?";
        con=ds.getConnection();
        PreparedStatement stmt =con.prepareStatement(sql);
        stmt.setString(1,this.bookid);
        ResultSet rset=stmt.executeQuery();
        if(rset.next()){
          this.bookname=rset.getString("BOOKNAME");
          this.bookprice=rset.getDouble("BOOKPRICE");
          stmt.close();
        }else{
          stmt.close();
          throw new NoSuchEntityException("BOOK ID:"+this.bookid);
        }

      }catch (SQLException se){
        throw new EJBException(se);
      }finally{
        try{
          if(con!=null)
            con.close();
        }catch(SQLException se){}
      }

    }

    //保存被关联的数据记录
    public void ejbStore(){
      try{
```

```
        String sql="UPDATE BOOK SET BOOKNAME=?,BOOKPRICE=? WHERE
BOOKID=?";
        con=ds.getConnection();
        PreparedStatement stmt =con.prepareStatement(sql);
        stmt.setString(1,this.bookname);
        stmt.setDouble(2,this.bookprice);
        stmt.setString(3,this.bookid);
        if(stmt.executeUpdate()!=1){
          stmt.close();
          throw new EJBException("occount a error on saved");
        }
        stmt.close();
      }catch (SQLException se){
        throw new EJBException(se);
      }finally{
        try{
          if(con!=null)
            con.close();
        }catch(SQLException se){}
      }

    }

    //删除关联的记录
    public void ejbRemove(){
      try{
        String sql="DELETE FROM BOOK    WHERE BOOKID=?";
        con=ds.getConnection();
        PreparedStatement stmt =con.prepareStatement(sql);
        stmt.setString(1,this.bookid);
        if(stmt.executeUpdate()!=1)
          throw new EJBException("occount a error on remove");
        stmt.close();
      }catch (SQLException se){
        throw new EJBException(se);
      }finally{
        try{
          if(con!=null)
            con.close();
        }catch(SQLException se){}
      }
    }
```

```
public void unsetEntityContext(){
   this.ctx=null;
}
//初始化数据库连接，初始化情境参数
public void setEntityContext(EntityContext context){
   this.ctx=context;

   try{
      InitialContext initial =new InitialContext();
      ds=(DataSource)initial.lookup(this.dbjndi);
   }catch(NamingException ne){
      throw new EJBException(ne);
   }
}

//在 Bean 激活时，从情境参数中获取 Bean 的主键值，然后会自动调用 ejbLoad()
public void ejbActivate(){
   this.bookid=(String)ctx.getPrimaryKey();
}

//解除当前 Bean 实例与数据库记录的关系
public void ejbPassivate(){
   this.bookid=null;
}

//根据主键查找对象
public String ejbFindByPrimaryKey(String primarykey)
   throws FinderException{

   try{
      String sql="SELECT BOOKID FROM BOOK WHERE BOOKID=?";
      con=ds.getConnection();
      PreparedStatement stmt =con.prepareStatement(sql);
      stmt.setString(1,primarykey);
      ResultSet rset=stmt.executeQuery();
      if(!rset.next()){
         stmt.close();
         throw new ObjectNotFoundException();
      }
      stmt.close();
      //查到数据库中存在此条记录
      return primarykey;
   }catch (SQLException se){
      throw new EJBException(se);
```

```
        }finally{
            try{
                if(con!=null)
                    con.close();
            }catch(SQLException se){}
        }
    }

    //查找书单定价在指定范围内的 Bean 的集合
    public Collection ejbFindInPrice(double lowerLimitPrice,double upperLimitPrice)
        throws FinderException{

        try{
            String sql="SELECT BOOKID FROM BOOK WHERE BOOKPRICE BETWEEN ?
AND ?";
            System.out.println(sql);
            con=ds.getConnection();
            PreparedStatement stmt =con.prepareStatement(sql);
            stmt.setDouble(1,lowerLimitPrice);
            stmt.setDouble(2,upperLimitPrice);
            ResultSet rset=stmt.executeQuery();

            ArrayList booklist=new ArrayList();
            while(rset.next())
                booklist.add(rset.getString("BOOKID"));

            stmt.close();

            return booklist;
        }catch (SQLException se){
            throw new EJBException(se);
        }finally{
            try{
                if(con!=null)
                    con.close();
            }catch(SQLException se){}
        }
    }
}
```

假设我们保存到 D:\ejb\Bmp1Book\src\Bmp1BookEJB .java

到此为止我们的 Bean 程序组件已经改写完毕了，使用如下命令进行编译：

```
cd bean\Bmp1Book
mkdir classes
cd src
javac -classpath %CLASSPATH%;../classes -d ../classes    *.java
```

如果顺利你将可以在..\Bmp1Book\classes 目录下发现有三个类文件。

4．编写部署文件：

ejb-jar.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <description>
        This is BMP 1.1 Book EJB example
    </description>
    <display-name>Bmp1BookBean</display-name>
    <enterprise-beans>
        <entity>
            <display-name>Bmp1Book</display-name>
            <ejb-name>Bmp1Book</ejb-name>
            <home>Bmp1BookHome</home>
            <remote>Bmp1Book</remote>
            <ejb-class>Bmp1BookEJB</ejb-class>
            <persistence-type>Bean</persistence-type>
            <PRIM-KEY-CLASS>JAVA.LANG.STRING</PRIM-KEY-CLASS>
            <reentrant>False</reentrant>
            <resource-ref>
                <res-ref-name>jdbc/oadb</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
            </resource-ref>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
      <container-transaction>
        <method>
          <ejb-name>Bmp1Book</ejb-name>
          <method-name>*</method-name>
        </method>
        <trans-attribute>NotSupported</trans-attribute>
      </container-transaction>
```

```
    </assembly-descriptor>
</ejb-jar>
```

假设我们保存到 D:\ejb\Bmp1Book\classes\META-INF\ejb-jar.xml(注意 META-INF 必须大写)

现在让我们看看当前的目录结构：

- Bmp1Book <文件夹>
  - classes<文件夹>
    - META-INF<文件夹>
      - ejb-jar.xml
    - Bmp1Book .class
    - Bmp1Book EJB.class
    - Bmp1BookHome.class
  - src<文件夹>
    - Bmp1Book.java
    - Bmp1BookEJB.java
    - Bmp1BookHome.java

部署到应用服务器

在部署之前我们需要将这些类文件和 xml 文件做成一个 jar 文件，EJB JAR 文件代表一个可被部署的 JAR 库，在这个库里，包含了服务器代码与 EJB 模块的配置。ejb-jar.xml 文件被放置在 JAR 文件所指定的 META-INF 目录中。我们可以使用如下命令得到 EJB JAR 文件：

```
cd d:\ejb\Bmp1Book\classes      (要保证类文件在这个目录下，且有一个 META-INF 子目录存放 ejb-jar.xml 文件)
jar -cvf    bmp1Book.jar *.*
```

确保 bmp1Book.jar 文件包括的文件目录格式如下：

- <根>
  - **META-INF**<文件夹>
    - **ejb-jar.xml**
  - InsufficientFundException.class
  - **StatefulAccount.class**
  - **StatefulAccountEJB.class**

- StatefulAccountHome.class

部署工具一般由 Java 应用服务器的制造商提供，在这里我使用了 Apusic 应用服务器，并讲解如何在 Apusic 应用服务器部署这个组件。

注意，如果使用其他部署工具，原理是一样的。要使用 Apusic 应用服务器，可以到 www.apusic.com 上下载试用版。

确定你的 Apusic 服务器已经被启动。
打开“部署工具”应用程序，点击文件—>新键工程：
第一步：选择“新建包含一个 EJB 组件打包后的 EJB-jar 模块”选项
第二步：选择一个刚才我们生成的 bmp1Book.jar 文件，
第三步：输入一个工程名，可以随意，这里我们输入 bmp1Book
第四步：输入工程存放的地址，这里我们假设被存放到 D:\ejb\bmp1Book\deploy 目录下
完成四个步骤后，如果没有问题将出现 bmp1BookBean 的部署界面，基本的参数配置已经在我们刚才编写的 ejb-jar.xml 中定义，虽然我们在部署时免去了映射字段、编写 SQL 操作语句的要求，但是需要提供一些 BMP 特性的配置：

选择 bmp1Book 的配置页，点击“4.资源引用”，画面上应该出现了我们在 ejb-jar.xml 文件中设置的数据库引用，我们设置一下共项范围和 JNDI 名，设置后的画面如下图 5-3：

图 5-3

上述步骤完成后就可以点击部署—>部署到 Apusic 应用服务器完成部署工作。

开发和部署测试程序

对于客户端，引用 BMP 实例的方式和引用 CMP 实例的方式是一样的，所以我们不需要改

动上一节的 servlet 程序，只需做少许改动：

Bmp1BookServlet .java 文件：

```
….
public class Bmp1BookServlet extends HttpServlet{
   ……
}
```

假设我们将文件保存到 D:\ejb\Bmp1Book\src\Bmp1BookServlet.java

使用如下命令编译 Servlet

```
cd D:\ejb\Bmp1Book
mkdir test
cd test
mkdir WEB-INF
cd WEB-INF
mkdir classes
cd D:\ejb\Bmp1Book\src
javac -classpath %CLASSPATH%;../classes/ -d ../test/WEB-INF/classes Bmp1BookServlet.java
```

编译成功后将这个 servlet 部署到与 Bmp1Book 同一工程中，在部署前需要我们编写一个 web.xml,并制作成一个 Web 模块文件（war 文件）

web.xml 文件内容如下：

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

<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN' 'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
   <icon>
      <small-icon></small-icon>
      <large-icon></large-icon>
   </icon>
   <display-name>Bmp1BookServlet</display-name>
   <description></description>
   <context-param>
      <param-name>jsp.nocompile</param-name>
      <PARAM-VALUE>FALSE</PARAM-VALUE>
   </context-param>
   <context-param>
      <param-name>jsp.usePackages</param-name>
      <param-value>true</param-value>
      <description></description>
   </context-param>
```

```
  <ejb-ref>
    <description></description>
    <ejb-ref-name>ejb/Bmp1Book</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>Bmp1BookHome</home>
    <remote>Bmp1Book</remote>
    <ejb-link>Bmp1Book</ejb-link>
  </ejb-ref>
</web-app>
```

假设我们将文件保存到 D:\ejb\Bmp1Book\test\WEB-INF\web.xml

J2EE Web 应用可以包括 Java Servlet 类、JavaServer Page 组件、辅助的 Java 类、HTML 文件、媒体文件等，这些文件被集中在一个 War 文件中。其中 War 结构具有固定的格式，根目录名为 WEB-INF，同一目录下应该有一个 web.xml 文件，用来描述被部署文件的部署信息，Jsp、html 等文件可以放置在这个目录下，同时 WEB-INF 目录下可能存在一个 classes 目录用于存放 Servlet 程序，如果引用了一些外部资源，则可以被放置到 WEB-INF\lib 目录下。使用下面的命令生成这个 Servlet 测试程序的 war 文件：

```
cd D:\ejb\Bmp1Book\test\
jar -cvf bmp1Book.war *.*
```

确保 bmp1Book.war 文件包括的文件目录格式如下：

- WEB-INF<文件夹>
  - **classes**<文件夹>
    - bmp1Book**Servlet.class**
  - web.xml

成功编译后，将这个 servlet 一同部署到 bmp1Book 工程中，我们回到“部署工具”，点击编辑 **à** 填加一个 Web 模块,选择我们刚刚编译成的 bmp1Book.war 文件
点击部署—>部署到 Apusic 应用服务器完成部署工作。

运行测试程序

打开浏览器，在浏览器中输入：
http://localhost:6888/bmp1Book/servlet/Bmp1BookServlet
localhost—Web Server 的主机地址
:6888—应用服务器端口，根据不同的应用服务器，端口号可能不同
/bmp1Book—部署 servlet 时指定的 WWW 根路径值
/servlet—ejb 容器执行 servlet 的路径
/Bmp1BookServlet—测试程序

如果运行正常，运行的结果应该和上一节 CMP 的例子相同

实战 EJB 之六 开发 EJB 2.0 的 CMP(本地接口例程)

实战 EJB 之七 开发 EJB 2.0 的 CMP(EJB QL)

实战 EJB 之八 开发 EJB 2.0 的 JMS