EJB

1 EJB概述

1.1 EJB概述

EJB 是 Enterprise Java Bean 的缩写，又称为企业 Bean，是 Sun 公司提出的服务器端组件规范，它描述了在构建组件的时候所需要解决的一系列问题，如可扩展（Scalable）、分布式（distributed）、事务处理（Transcational）、数据存储（Persistent）、安全性（security）等。

1.2 EJB 3.1组件类型及组成

组件类型：

1. 会话 Bean：会话 Bean 主要是对业务逻辑的封装。EJB 3.1 中将会话 Bean 分成有状态会话 Bean、无状态会话 Bean、单例会话 Bean。
2. 消息驱动 Bean（MDB）：消息驱动 Bean 是设计用来专门处理基于消息请求的组件。一个 MDB 类必须实现`MessageListener`接口。当容器检测到 Bean 守候的队列中存在一条消息时，就调用`onMessage()`方法，将消息作为参数传入。

组件的组成：

1. 接口文件：是 EJB 组件模型的一部分，包含了 EJB 提供的对外服务接口，里面提供的方法一般和需要被远程调用的方法一致。一般情况下，要求类文件必须和接口中的定义保持一致性。
2. 类文件：实现基本方法的类，封装了需要实现的业务逻辑、数据逻辑或消息处理逻辑，具有一定的编程规范，代码不需要被客户端得知。
3. 必要的情况下，编写一些配置文件，用于描述 EJB 部署过程中的一些信息。

1.3 EJB运行原理

利用 EJB 编程，通常包含以下几个步骤：

1. 编写接口。
2. 编写实现接口的 EJB 实现类。
3. 部署到服务器中，将 JNDI 名称发布。
4. 编写客户端程序，并将接口拷贝给客户端，通过 JNDI 查找获得 EJB，调用 EJB 的方法。

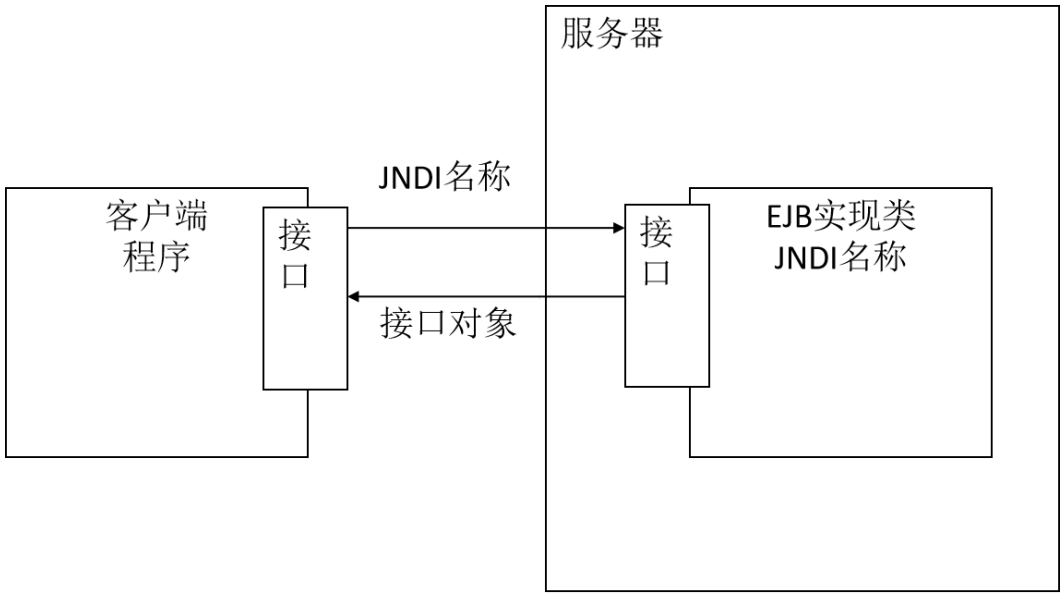

图 1.1　EJB 组件之间的关系

1.4 EJB 3.1新特性

1. 无接口的会话 Bean：EJB 3.0 要求 Bean 至少实现一个接口，而 EJB 3.1 的 Bean 可以不需要接口，一个 Bean 类就是一个 EJB。
2. 单例会话 Bean：EJB 3.1 引入单例会话 Bean 概念主要是为了共享应用数据和支持一致性访问，当一个 Bean 被标记为 Singleton 时，在整个应用层容器可以保证每个客户端共享一个实例。
3. 简化的 EJB 打包机制：EJB 3.1 以前规范始终要求企业 Bean 打包到一个叫做 ejb-jar 的文件模块中，应用程序被强制性要求使用一个 Web 应用程序使用的归档文件（.war），一个企业 Bean 使用的 ejb-jar 文件，还有一个包含其它包的企业归档文件（.ear）。EJB 3.1 中一个重要的改进是可以直接将 EJB 组件打包到 WAR 文件中，不用再独立创建 jar 文件了。

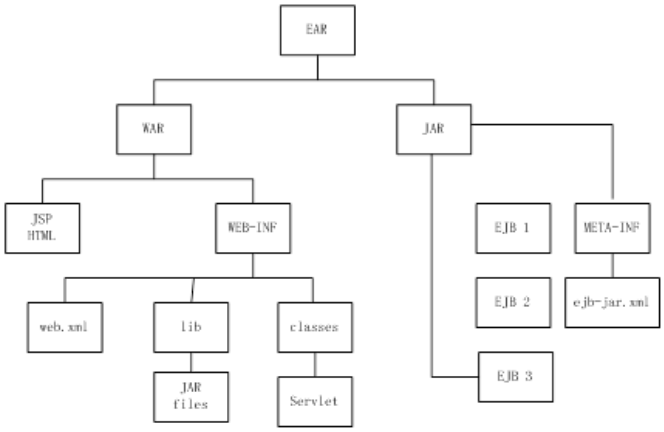

图 1.2　EJB 3.0 打包方式

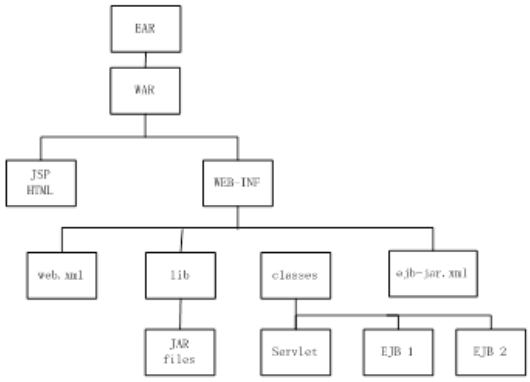

图 1.3　EJB 3.1 打包方式

4. 异步会话 Bean：异步调用可以应用于所有类型的会话 Bean。在 EJB 3.1 之前，在会话 Bean 上的任何函数调用都是同步的。EJB 3.1 规范规定：在容器开始执行某个 Bean 实例的调用之前，异步调用的控制权一定要返回给客户端，因此允许客户端触发并行处理的流程。
5. EJB Lite：EJB 必须按照规范去实现一系列 API，从 EJB 3.1 开始，这组 API 分成了两个：最小配置和完整配置。其中最小配置即 EJB Lite。
   - 会话 Bean 组件
   - 只支持同步调用
   - 容器管理和 Bean 管理事务
   - 声明和编程安全
   - 拦截器
   - 支持部署描述信息（ejb-jar.xml）
6. 统一的全局 JNDI 命名：EJB 3.1 规范定义了全局 JNDI 命名方式，采用统一的方式来获取注册的会话 Bean，因此用户可以使用兼容性的 JNDI 命名了。
   在 JBoss AS7 中，无状态会话 Bean 的 JNDI 命名规则如下：
   `ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-interface>`
   有状态会话 Bean 的 JNDI 命名规则如下：

`ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-interface>?stateful`

其中各个参数的含义如下：

名称	描述	必选
`app-name`	应用程序的名称。如果没有在 application.xml 中指定，则默认的名称就是 EAR 的打包名称。	否
`module-name`	模块的名称。如果没有在 ejb-jar.xml 或 web.xml 中指定， 则默认的名称就是 jar 包或 war 包的名称。	是
`distinct-name`	JBoss AS7 中特有的 EJB 名字，默认为空。	否
`bean-name`	Bean 类的名称。 如果没有使用标注`@Stateless`、`@Stateful`、`@Singleton`或其它布署描述符，则默认的名称就是该 session bean 的类的完全限定名称。	是
`fully-qualified-interface-name`	暴露接口的限定名称。如果是一个无接口的会话 Bean，则它的值为该 Bean 类的完全限定名称。对于有状态会话 Bean，要在末尾添加`?stateful`。	是

2 会话Bean

2.1 会话Bean概述

2.1.1 会话Bean的定义

会话 Bean 是运行在 EJB 容器中的 Java 组件，用于和客户端进行会话。

2.1.2 会话Bean的分类

在 Java EE 6 规范中，根据其是否保存客户的状态，将会话 Bean 分为 3 类：无状态会话 Bean、有状态会话 Bean、单例会话 Bean。

1. 无状态会话 Bean

无状态会话 Bean 不维持和客户端的会话状态。当方法结束的时候，客户端特定的状态不会被保持，因此可以供其他客户端调用。

无状态会话 Bean 采用实例池机制支持大量并发客户端的调用请求，允许 EJB 容器将一个实例分配给任意一个客户端。

即使客户端已经消亡，无状态会话 Bean 的生命周期也不一定结束，它可能依然存在于实例池中，供其他用户调用。

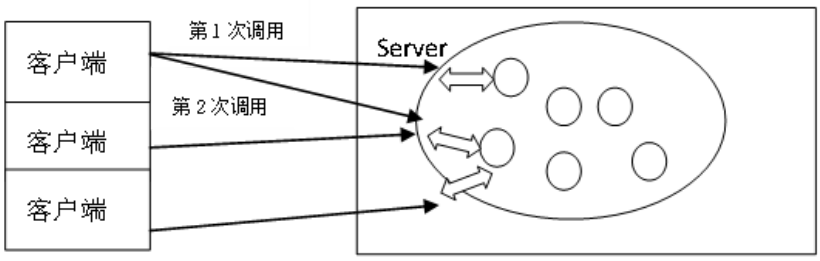

图 2.1　无状态会话 Bean 实例与客户端的对应关系

2. 有状态会话 Bean

有状态会话 Bean 维持与客户端的会话状态，因此实例池中的不同实例是有区别的。

由于每个客户端与特定的一个 Bean 实例建立连接，在客户端的生命周期内，Bean 实例保持了该客户端的信息，一旦客户端灭亡，Bean 实例的生命周期也结束。

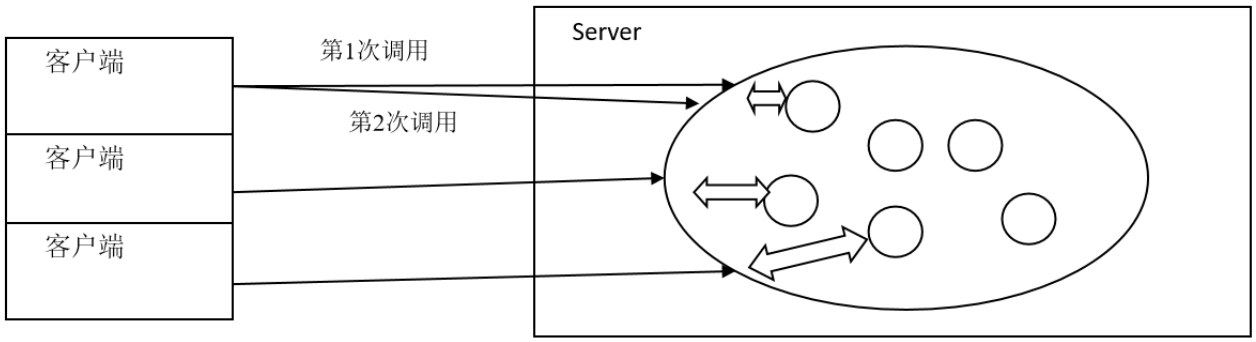

图 2.2　有状态会话 Bean 实例与客户端的对应关系

3. 单例会话 Bean

单例会话 Bean 在每个应用程序中只被实例化一次，在整个应用程序的生命周期中存在。

单例会话 Bean 提供了和无状态会话 Bean 类似的功能，区别是单例会话 Bean 在每个应用程序中只有一个实例，而无状态会话 Bean 会有多个实例存在于实例池中。

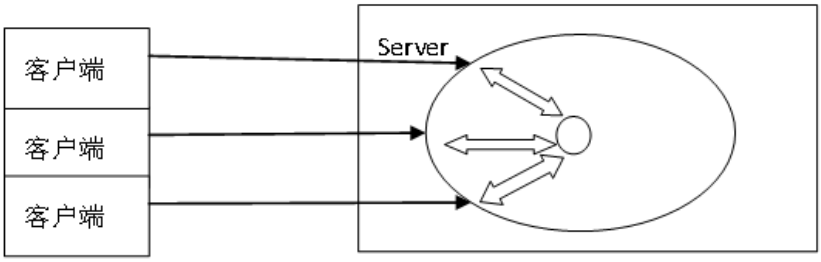

图 2.3　单例会话 Bean 实例与客户端的对应关系

2.1.3 会话Bean的构成

在 EJB 3.1 规范中，会话 Bean 的构成包括：

1. 业务接口
   - 远程接口：不支持远程调用，无法实现计算上的分布性。通常服务于同一个虚拟机上的组件调用。不存在网络传输和 socket 通信，因此调用效率较高。
   - 本地接口：支持分布式计算过程，可以被部署到不同的 Java 虚拟机或不同的计算机上。存在调用数据的打包、网络传输和解包，因此调用效率相对较低。
   - 无接口：只能被本地访问。
2. EJB 实现类

2.2 无状态会话Bean

2.2.1 实现远程接口的无状态会话Bean

实现远程接口的无状态会话 Bean 编程方法：

1. 创建接口，并使用`@Remote`表示该接口是远程业务接口，然后在该接口中加入业务方法声明。由于 EJB 使用 RMI 协议与客户端通信，因此要求方法参数按值传递，并且需要序列化。
2. 创建实现远程接口的 Bean 类，并使用`@Stateless`表示其是无状态会话 Bean，然后为已经声明的业务方法添加实现代码。

例如：

```java
package javaee.ejb.stateless.remote;

import javax.ejb.Remote;

// 创建远程接口
@Remote
public interface HelloBeanRemote
{
    public String sayHello(String name);
}
```

```java
package javaee.ejb.stateless.remote;

import javax.ejb.LocalBean;
import javax.ejb.Stateless;

// 创建实现远程接口Bean类
@Stateless
public class HelloBean implements HelloBeanRemote
{
   public HelloBean() { }

   public String sayHello(String name)
   {
        return "Hello, " + name + "!";
   }
}
```

代码写完后，就可以将会话 Bean 部署到服务器上。

在客户端通过 JNDI 访问会话 Bean 的远程接口的方法有两种：

1. 使用通用 JNDI API

```java
public class StatelessRemoteClient
{
    public static void main(String[] args) throws Exception
    {
        Properties prop = new Properties();
        // 服务器的命名和目录管理地址
        prop.put(Context.PROVIDER_URL, "remote://localhost:4447");
        // 初始化上下文环境工厂
        prop.put(Context.INITIAL_CONTEXT_FACTORY,
                 org.jboss.naming.remote.client.InitialContextFactory.class.getName());
        // 用户验证
        prop.put(Context.SECURITY_PRINCIPAL, System.getProperty("username", "testJNDI"));
        prop.put(Context.SECURITY_CREDENTIALS, System.getProperty("password", "123456"));
        try
        {
            // 获取远程接口
            Context ctx = new InitialContext(prop);
            Object obj = ctx.lookup("SessionEJB/HelloBean!javaee.ejb.stateless.remote.HelloBeanRemote");
            HelloBeanRemote hwr = (HelloBeanRemote) obj;

            // 调用远程接口的方法
            String say = hwr.sayHello("Jilin University");
            System.out.println(say);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

2. 使用 EJB Client API

（1）新建一个普通 Java Project 工程，把 jboss-client.jar 加入到项目。

（2）将接口文件（本例中是 HelloBeanRemote.java）按照原有路径拷贝到本工程中。

（3）创建客户端文件。

（4）在 src 目录下添加 jboss-ejb-client.properties 文件。

例如，对于上面的例子，创建客户端测试文件 StatelessRemoteClient.java 如下所示：

```
public class StatelessRemoteClient
{
    public static void main(String[] args)
    {
        // 在 JBoss 中使用如下方式访问 EJB
        Hashtable<String, String> jndiProperties = new Hashtable<String, String>();
        jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
        try
        {
            Context context = new InitialContext(jndiProperties);

            final String appName = "";
            final String moduleName = "SessionEJB";
            final String distinctName = "";
            Object obj = context.lookup("ejb:" + appName + "/" + moduleName + "/" + distinctName +
                 "/HelloBean!javaee.ejb.stateless.remote.HelloBeanRemote");
            HelloBeanRemote hwr = (HelloBeanRemote) obj;

            String say = hwr.sayHello("Jilin University");
            System.out.println(say);
        }
        catch (NamingException e)
        {
            e.printStackTrace();
        }
    }
}
```

jboss-ejb-client.properties 文件的内容如下所示：

```
endpoint.name=client-endpoint
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=localhost
remote.connection.default.port=4447
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.connection.default.username=testJNDI
remote.connection.default.password=123456
```

2.2.2 实现本地接口的无状态会话Bean

创建实现本地接口的无状态会话 Bean 的方法：

1. 使用`@Local`创建本地接口，并加入业务方法声明。
2. 创建实现本地接口的 Bean 类，使用`@stateless`表示无状态会话 Bean，并加入已声明的业务方法的实现代码。

例如：

```
package javaee.ejb.stateless.local;

import javax.ejb.Local;

// 创建本地接口
@Local
public interface CalculatorBeanLocal
{
    public int add(int x, int y);
}
```

```java
package javaee.ejb.stateless.local;

import javax.ejb.LocalBean;
import javax.ejb.Stateless;

// 创建实现本地接口 Bean 类
@Stateless
public class CalculatorBean implements CalculatorBeanLocal
{
    public CalculatorBean() { }

    public int add(int x, int y)
    {
        System.out.println("\n\t[CalculatorBean]  add() invoked.");
        return x + y;
    }
}
```

实现远程接口的会话 Bean，通过依赖注入引用本地会话 Bean：

```java
package javaee.ejb.stateless.local;

import javax.ejb.*;

@Remote
public interface CallerRemote
{
    public String testMethod();
    public String callEJBOne(int a, int b);
}
```

```java
package javaee.ejb.stateless.local;

import javax.ejb.Stateless;
import javaee.ejb.stateless.local.CalculatorBeanLocal;

@Stateless
public class CallerBean implements CallerRemote
{
    // 注入 EJB
    @EJB
    private CalculatorBeanLocal localbean;

    public String testMethod()
    {
        System.out.println("\n\n\t Bean testMethod() called....");
        return "DONE----returned";
    }

    public String callEJBOne(int a, int b)
    {
        int result=0;
        try
        {
            System.out.println("\n\n\t Bean callEJBOne(a,b) called....");
            result = localbean.add(a, b);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return "DONE----result = " + result;
    }
}
```

客户端通过远程接口访问实现本地接口的无状态会话 Bean：

```
package javaee.ejb.statelessclient;

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

public class StatelessLocalClient
{
    public static void main(String[] args) throws Exception
    {
        String result = "";
        System.out.println("\n\n\t begin ...");
        try
        {
            Hashtable<String, String> jndiProperties = new Hashtable<String, String>();
            jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
            Context context = new InitialContext(jndiProperties);

            final String appName = "";
            final String moduleName = "SessionEJB";
            final String distinctName = "";
            CallerBeanRemote remote = (CallerBeanRemote) context.lookup("ejb:" +
                        appName + "/" + moduleName + "/" + distinctName +
                        "/CallerBean!javaee.ejb.stateless.local.CallerBeanRemote");

            result = remote.callEJBOne(1000, 2000);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        System.out.println("ONE----result = " + result);
    }
}
```

2.2.3 无接口的无状态会话Bean

EJB 3.1 允许会话 Bean 没有实现任何接口，这种会话 Bean 称为无接口会话 Bean，从而用户不用编写独立的业务接口就可以获得相同的企业 Bean 功能。

无接口会话 Bean 与实现本地接口的会话 Bean 具有相同的行为。

例如，下面的代码用无接口会话 Bean 实现 hello 功能：

```
package javaee.ejb.stateless.nointerface;

import javax.ejb.LocalBean;
import javax.ejb.Stateless;

@Stateless
@LocalBean
public class NoInterfaceHelloBean
{
    public NoInterfaceHelloBean() { }

    public String sayHello(String s)
    {
        String message = "hello: " + s;
        return message;
    }
 }
```

在同一项目中使用 Servlet 调用无接口会话 Bean：

```java
public class TestEJBServlet extends HttpServlet
{
    // 注入EJB
    @EJB
    private javaee.ejb.stateless.nointerface.NoInterfaceHelloBean hello;

    protected void doGet(HttpServletRequest arg0, HttpServletResponse arg1)
        throws ServletException, IOException
    {
        this.doPost(arg0, arg1);
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
    {
        try
        {
            String result = hello.sayHello("Jilin University");

            response.setContentType("text/html;charset=UTF-8");
            PrintWriter out = response.getWriter();
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet call nointerfaceEJB</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("NoInterface executed - Result: " + result);
            out.println("</body>");
            out.println("</html>");
        }
        catch (Exception)
        {
            e.printStackTrace();
        }
    }
}
```

2.2.4 无状态会话Bean的生命周期

无状态会话 Bean 的生命周期中存在三种状态，如图 2.4 所示。

图 2.4　无状态会话 Bean 的三种状态

1. 不存在状态：主要针对无状态会话 Bean 组件对象，而不是 EJB 组件本身。在此状态下，无状态会话 Bean 组件对象不存在，但无状态会话 Bean 组件已经被部署到服务容器中。
2. 池状态：在一个无状态会话 Bean 组件被部署到服务器后，EJB 服务器通常会提前创建一定量的无状态会话 Bean 组件对象，并将它们临时缓存在缓冲区中，这种状态的对象称为处于池状态的 EJB 组件对象。
3. 调用状态：对于正在为远程或本地客户提供服务的组件对象，这种状态又称为服务状态。任何一个客户请求被发送到无状态会话 Bean 组件时，EJB 服务器会首先从无状态会话 Bean 组件池中查找特定的无状态会话 Bean 组件对象，并使用这种组件对象为客户请求提供服务，在请求方法完成后，EJB 服务器会将提供服务的组件释放到无状态会话 Bean 组件的对象池中。因此当远程客户连续对同一个无状态会话 Bean 组件进行访问时，很可能由两个完全不同的组件对象提供服务。

在无状态会话 Bean 的整个生命周期中，有两个重要事件：PostConstruct 和 PreDestroy。

1. PostConstruct 事件：在无状态会话 Bean 组件对象创建过程中被触发，表示一个 EJB 组件对象的生成，通常用于对整个无状态会话 Bean 组件对象状态进行初始化。通常 EJB 服务器创建 EJB 组件对象需要经过三个步骤：
   （1）EJB 容器首先调用 EJB 组件的`Class.newInstance()`方法生成一个组件对象；
   （2）然后 EJB 组件服务器会将组件的 XML 配置文件或类似`@Resource`等标注所包含的初始信息，设置给刚生成的 EJB 组件

对象；
（3）最后触发 PostConstruct 事件，以便进一步进行组件自身状态的初始化。

2. PreDestroy 事件：它是 EJB 组件对象被销毁过程中的触发事件，通常用于释放组件对象使用过的资源。在该事件的处理方法结束前，EJB 组件对象仍旧是一个完整对象，同样可以进行各种操作和调用。该事件的处理方法结束后，EJB 组件上的各种引用就被销毁，组件对象进入等待 Java 垃圾收集线程的销毁过程。

在整个 EJB 组件生命周期中，PostConstruct 事件和 PreDestroy 事件均只会被触发一次。

无状态会话 Bean 的事件处理配置方法：

1. 通过 EJB 配置文件进行配置：使用 ejb-jar.xml 文件配置无状态会话 Bean 生命事件。例如：

```
<session>
    <ejb-name>HelloBean</ejb-name>
    <!-- post-construct 标记一定要出现在 pre-destroy 标记之前 -->
    <post-construct>
        <lifecycle-callback-method>initialEJB</lifecycle-callback-method>
    </post-construct>
    <pre-destroy>
        <lifecycle-callback-method>endEJB</lifecycle-callback-method>
    </pre-destroy>
</session>
```

2. 采用标注 `@PostConstruct` 和 `@PreDestroy` 进行配置：将 `@PostConstruct` 或 `@PreDestroy` 写在事件处理方法前面即可。处理无状态会话 Bean 生命事件的方法可以是任何名称，但需要满足下列条件：参数列表必须为空，返回类型为 `void`，且不能抛出异常。例如：

```
@PostConstruct
public void initialEJB()
{
    System.out.println("EJB has been constructed");
}

@PreDestroy
public void endEJB()
{
    System.out.println("EJB will be destroyed");
}
```

2.3 有状态会话Bean

2.3.1 有状态会话Bean的开发

有状态会话 Bean 的开发和无状态会话 Bean 的开发方法基本相同，它同样可以具有远程和本地接口以及无接口的实现方式，唯一差别是有状态会话 Bean 需要维持状态。

实现远程接口的有状态会话 Bean 的例子：

```
1  package javaee.ejb.stateful.remote;
2
3  import javax.ejb.Remote;
4
5  @Remote
6  public interface MulBy2Remote
7  {
8      public int mul();
9      public void cancel();
10 }
```

```java
package javaee.ejb.stateful.remote;

import javax.ejb.LocalBean;
import javax.ejb.Stateful;

@Stateful
public class MulBy2Bean implements MulBy2Remote
{
    int i = 1;

    public MulBy2Bean() { }

    public int mul()
    {
        i = i * 2;
        return i;
    }

    @Remove
    public void cancel()
    {
        i = 0;
        System.out.println("重置i=" + i);
    }

    @PreDestroy
    public void endEJB()
    {
        System.out.println("predestroy is called");
    }
}
```

```java
public class MulBy2Client
{
    public static void main(String[] args)
    {
        Hashtable<String, String> jndiProperties = new Hashtable<String, String>();
        jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
        try
        {
            Context context = new InitialContext(jndiProperties);
            final String appName = "";
            final String moduleName = "SessionEJB";
            final String distinctName = "";

            // 生成第一个实例
            Object obj = context.lookup("ejb:" + appName + "/" + moduleName + "/" + distinctName +
                "/MulBy2Bean!javaee.ejb.stateful.remote.MulBy2Remote?stateful");
            MulBy2Remote mulBy2R1 =(MulBy2Remote)obj;

            // 生成第二个实例
            obj = context.lookup("ejb:" + appName + "/" + moduleName + "/" + distinctName +
                "/MulBy2Bean!javaee.ejb.stateful.remote.MulBy2Remote?stateful");
            MulBy2Remote mulBy2R2 = (MulBy2Remote) obj;

            // 第一个实例调用 2 次乘法操作
            int j1 = mulBy2R1.mul();
            j1 = mulBy2R1.mul();
            System.out.println("the value in Clinet 1:  " + j1);

            // 第二个实例调用 1 次乘法操作
            int j2 = mulBy2R2.mul();
            System.out.println("the value in Clinet 2:  " + j2);
        }
        catch (NamingException e)
        {
            e.printStackTrace();
        }
    }
}
```

2.3.2 有状态会话Bean和无状态会话Bean在代码上的区别

1. 有状态会话 Bean 中的 Bean 类具有类似于普通 Java 类中属性的属性字段（例如`i`），可以通过 Bean 的业务方法对其进行修改。
2. 每个有状态会话 Bean 必须至少定义一个使用`@Remove`注解标记的方法，客户端将使用这些方法来结束与有状态会话 Bean 的会话。调用了一个这样的方法之后，服务器将销毁该 Bean 实例。

2.3.3 有状态会话Bean的生命周期

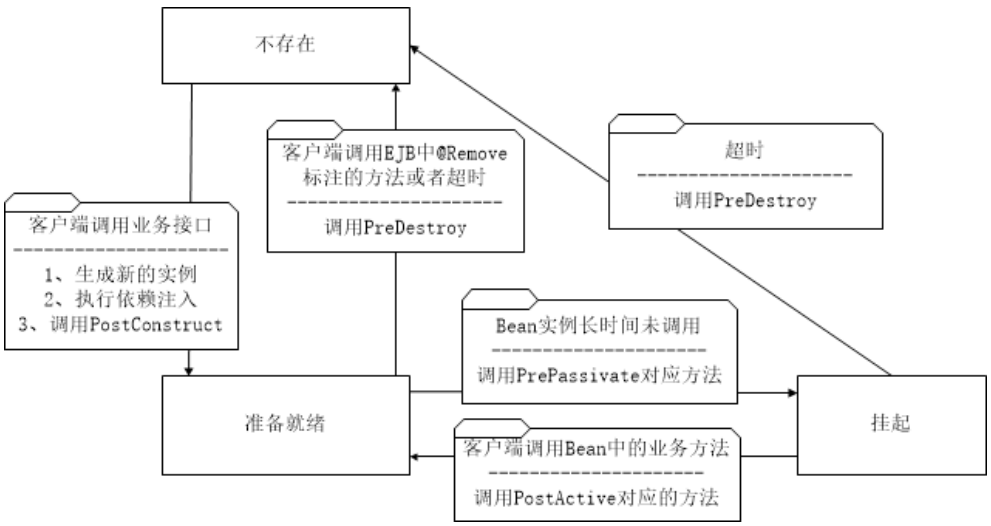

图 2.5　有状态会话 Bean 的生命周期

- 当客户向 EJB 服务器请求一个组件引用时，EJB 组件对象被服务器创建。
- 当容器生成一个实例时，将把它指定给一个客户端，这样每个从这个客户端来的请求都会被传递给同一个实例，因此有状态会话 Bean 的生命周期由客户端决定。
- 在客户端使用组件过程中，EJB 组件对象一直存在；假如客户端长时间没有调用它的 Bean 实例，容器将在 JVM 内存溢出前把实例清除，并持久化这个实例的状态，这一过程称为钝化。而当客户端需要的时候再重新加载进内存，这一过程称为活化。EJB 容器自动管理着 Bean 实例的钝化和活化。
- 当客户端放弃 EJB 组件引用时，对应的 EJB 组件对象会被 EJB 容器销毁。

有状态会话 Bean 生命周期包含 4 个事件：

1. PostConstruct 事件

当用户第一次调用某个有状态会话 Bean 时，EJB 服务器会调用`newInstance()`方法创建一个 EJB 组件对象。后面过程和无状态会话 Bean 类似，EJB 服务器会对该有状态会话 Bean 实例进行初始化设置，并触发 PostConstruct 事件。

由于有状态会话 Bean 实例在构造过程中，系统会调用一个不带任何参数的构造方法，因而要求有状态会话 Bean 组件必须提供一个不带任何参数的默认构造方法，否则组件对象在构造过程中会产生异常。

2. PreDestroy 事件

当有状态会话 Bean 对象在活动状态或者挂起状态时，客户端可以通过调用组件的`@Remove`方法实现组件对象的销毁；或者在活动状态下的组件对象超过其寿命，服务器也会销毁该组件对象，将该组件对象由活动状态转变为不存在状态。

当有状态会话 Bean 实例被销毁前，EJB 服务器触发组件上的 PreDestroy 事件。PreDestroy 是组件整个生命周期中最后执行的行为，在方法中可以对组件用过的资源进行释放。

3. PrePassivate 事件

当一个处于活动状态的有状态会话 Bean 对象长时间不使用时，EJB 服务器通常会将该组件对象切换到休眠状态（钝化）。EJB 组件休眠的本质是将 EJB 组件对象的状态和当前环境状态统一保存起来，然后将对象从内存中删除。

在有状态会话 Bean 对象由活动状态转变到休眠状态时，EJB 服务器会触发 PrePassivate 事件。在该方法中可以对一些休眠前的状态进行保存。

4. PostActive 事件

当有状态会话 Bean 对象由休眠状态切换到活动状态后，会马上触发 EJB 组件上的 PostActive 事件。在该方法中可以对 EJB 组件中的状态进行一些休眠后的恢复工作。

有状态会话 Bean 的四种生命周期事件可以通过标注进行配置。

2.3.4 有状态会话Bean和无状态会话Bean的区别

1. 组件对象进入休眠（缓存状态）的时刻

有状态会话 Bean 进入休眠状态，通常是因为对应的组件对象长时间不被使用，或服务器负载十分重的情况下才会发生；而无状态会话 Bean 在一个方法执行完毕后就会被释放到实例池中。

2. 组件对象在休眠状态（缓存状态）的差别

无状态会话 Bean 进入缓存状态后，其对应的对象在服务器上还存在；而有状态会话 Bean 对象进入休眠状态后，其 EJB 组件对象被销毁，只是其对象状态被保存到了硬盘或数据库中。

3. 组件对象进入休眠（缓存状态）时，EJB 服务器所保留的组件数据

无状态会话 Bean 组件对象进入缓存状态后，其被保留的是组件上下文环境；而有状态会话 Bean 组件对象进入休眠状态时被保存的不但是 EJB 组件的上下文环境，而且还包括 EJB 组件的各种属性状态。

2.4 单例会话Bean

单例会话 Bean 在每个应用程序中只被实现一次，在所有客户端之间共享状态，在整个应用程序的生命周期中存在。

2.4.1 单例会话Bean示例

```
package javaee.ejb.singleton.remote;

import javax.ejb.Remote;

@Remote
public interface SimpleSingletonRemote
{
    public void change01();
    public int getValue();
}
```

```
@Singleton
public class SimpleSingletonBean implements SimpleSingletonRemote
{
    private int i=0;

    public SimpleSingletonBean () { }

    public void change01()
    {
        if (i == 0)
            i = 1;
        else
            i = 0;
    }

    public int getValue()
    {
        return i;
    }
}
```

```java
public class SimpleSingletonClient
{
    public static void main(String[] args)
    {
        Hashtable<String, String> jndiProperties = new Hashtable<String, String>();
        jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
        try
        {
            Context context = new InitialContext(jndiProperties);

            final String appName = "";
            final String moduleName = "Singleton";
            final String distinctName = "";

            // 生成 2 个客户端
            SimpleSingletonRemote single01 = (SimpleSingletonRemote) context.lookup(
                "ejb:" + appName + "/" + moduleName + "/" + distinctName + "/SimpleSingletonBean!" +
                "javaee.ejb.singleton.remote.SimpleSingletonRemote");

            SimpleSingletonRemote single02 = (SimpleSingletonRemote) context.lookup(
                "ejb:" + appName + "/" + moduleName + "/" + distinctName + "/SimpleSingletonBean!" +
                "javaee.ejb.singleton.remote.SimpleSingletonRemote");

            int value1 = single01.getValue();
            int value2 = single02.getValue();
            System.out.println("Singleton01初始值: " + value1);
            System.out.println("Singleton02初始值: " + value2);

            single01.change01();
            value1 = single01.getValue();
            value2 = single02.getValue();
            System.out.println("Singleton01值: "+ value1);
            System.out.println("Singleton02值: "+ value2);

            single02.change01();
            value1 = single01.getValue();
            value2 = single02.getValue();
            System.out.println("Singleton01值: "+ value1);
            System.out.println("Singleton02值: "+ value2);
        }
        catch (NamingException e)
        {
            e.printStackTrace();
        }
    }
}
```

2.4.2 单例会话Bean的并发控制

2.4.2.1 容器管理并发（CMC）

容器管理并发（Container-Managed Concurrency，CMC） 由 EJB 容器控制客户端访问单例会话 Bean 的业务方法，使用`@Lock`标注来指定当客户端调用方法时容器如何管理并发。

`@Lock`的值可以为`READ`或`WRITE`。

- `@Lock(LockType.WRITE)`：这是一个排它锁，对其他客户锁定正在调用的方法，直至这个方法被调用完毕。
- `@Lock(LockType.READ)`：这是共享锁，允许多个客户并发访问或共享。

`@Lock`标注可以在类、方法上标注，也可以同时标注。

如果未指定标注属性，默认为`@Lock(LockType.WRITE)`。

CMC 是默认管理方式，相当于在所有业务方法之上的写锁定。如果不指定并发管理方式，单例会话 Bean 默认使用 CMC。

CMC 的示例如下：

```
package javaee.ejb.singleton;

@ConcurrentcyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
@LocalBean
public class SingletonCMCBean()
{
    private String state;

    @Lock(LockType.READ)
    @AccessTimeout(value = 20, unit = TimeUnit.SECONDS)
    public String getState()
    {
        return state;
    }

    @Lock(LockType.WRITE)
    @AccessTimeout(value = 20, unit = TimeUnit.SECONDS)
    public void setState(String aState)
    {
        state=aState;
        try
        {
            CountDownLatch latch = new CountDownLatch(1);
            latch.await(10, TimeUnit.SECONDS); // 等待 10 秒
        }
        catch(Exception e)
        {
            System.out.println("写锁出现错误");
        }
    }
}
```

CMC 的语法是固定的，只能添加在方法或类前面，因此作用域就是对整个方法进行并发管理。

2.4.2.2 Bean管理并发（BMC）

如果使用 Bean 管理并发（Bean Managed Concurrency，BMC），需要使用`@ConcurrencyManagement(ConcurrencyManagementType.BEAN)`，并且利用 Java 编程语言的同步原语进行同步控制。BMC 中使用`synchronized`同步机制。

BMC 的示例如下所示：

```
package javaee.ejb.singleton;

@ConcurrentcyManagement(ConcurrencyManagementType.BEAN)
@Singleton
public class SingletonBMCBean()
{
    private String state;

    public String getState()
    {
        return state;
    }

    public synchronized void setState1 (String aState)
    {
        state = aState;
    }

    public void setState2 (State state)
    {
        synchronized (this)
        {
            this.state = state;
        }
    }
}
```

2.4.3 单例会话Bean的生命周期

单例会话 Bean 与无状态会话 Bean 在大体上是一样的，只不过单例会话 Bean 在实例池中只有一个实例存在，而无状态会话 Bean 在实例池中存在多个实例。

单例会话 Bean 的生命周期事件与无状态会话 Bean 相同。

2.5 多接口会话Bean

一个会话 Bean 可以实现多个 Remote 型或者 Local 型接口，但不能在同一个接口上既使用`@Remote`，又使用`@Local`。

例如，下面的会话 Bean 实现了 2 个远程接口和 1 个本地接口：

```
package javaee.ejb.mulinterface.remote;

import javax.ejb.Remote;

@Remote
public interface HelloRemote
{
    public String sayHelloFromRemote(String name);
}
```

```
package javaee.ejb.mulinterface.remote;

import javax.ejb.Local;

@Local
public interface HelloLocal
{
    public String sayHelloFromLocal(String name);
}
```

```java
package javaee.ejb.mulinterface.remote;

import javax.ejb.Remote;

@Remote
public interface MulBy2Remote
{
    public int mul();
}
```

```java
public class MulInterfaceBean implements HelloRemote, HelloLocal, MulBy2Remote
{
    private int value = 1;

    // 生成本地接口实例
    @EJB
    javaee.ejb.mulinterface.remote.HelloLocal hello;

    public MulInterfaceBean() {}

    public String sayHelloFromLocal(String name)
    {
        return " Hello " + name + " from local!";
    }

    public String sayHelloFromRemote(String name)
    {
        String result = null;
        result = hello.sayHelloFromLocal(name);
        return result;
    }

    public int mul()
    {
        value = value * 2;
        return value;
    }
}
```

2.6 会话Bean异步调用

默认情况下，通过远程接口、本地接口或无接口视图调用会话 Bean 时采用同步的通信方式：客户端调用一个方法，然后客户端被阻塞，直到被调用方法处理完成返回结果给客户端，客户端才能继续后面的工作。

在 EJB 3.1 之前，异步调用只能通过 JMS 和 MDB 来实现。在 EJB 3.1 规范中，可以在一个会话 Bean 的方法上添加一个`@javax.ejb.Asynchronous`标注来实现异步调用。

异步方法可以返回一个`Future`对象或`void`。`Future`对象容纳了异步操作返回的结果，它提供`get()`方法检索结果值，也可以检查异常或取消一个正在处理中的调用。`AsyncResult`是`Future`接口的具体实现类。

下面的示例使用异步调用实现网络打印：

```java
package javaee.ejb.asyncall;

@Stateless
@LocalBean
public class PrintBean
{
    @Asynchronous
    public void printAndForget()
    {
        System.out.println("*** printAndForget ***");
    }

    @Asynchronous
    public Future<String> printAndCheckLater()
    {
        System.out.println("*** printAndCheckLater ***");
        return new AsyncResult<String>("OK");
    }
}
```

```java
public class PrintServlet extends HttpServlet
{
    @EJB
    javaee.ejb.asyncall.PrintBean printBean;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try
        {
            // 调用 printAndForget() 方法
            printBean.printAndForget();
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet PrintServlet</title>");
            out.println("</head>");
            out.println("<body>");

            // 调用 printAndCheckLater() 方法
            Future<String> futureResult = printBean.printAndCheckLater();
            if (futureResult.isDone() && !futureResult.isCancelled())
            {
                try
                {
                    out.println("<h3>printAndCheckLater executed - Result: " +
                                futureResult.get() + "</h3>");
                }
                catch (ExecutionException e)
                {
                    e.printStackTrace();
                }
            }
            else
            {
                out.println("<h3>printAndCheckLater is not prepared</h3>");
            }
        }
        catch (ExecutionException e)
        {
            e.printStackTrace();
        }
    }
}
```

3 JMS与消息驱动Bean

3.1 JMS的基本概念

JMS（Java Message Service，Java 消息服务）是由 Sun 公司开发的一组接口和相关规范，这些接口和规范定义了 JMS 客户端访问消息系统的方法。

JMS 为 Java 程序提供了一种创建、发送、接收和读取消息的通用方法。JMS 可以为企业软件模块间提供方便、可靠的异步通信机制。

JMS 是 Java 解决方案的消息服务类型，它的通信管道是消息队列。

早期的通信系统通常都有专用的调用接口，如图 3.1 所示。

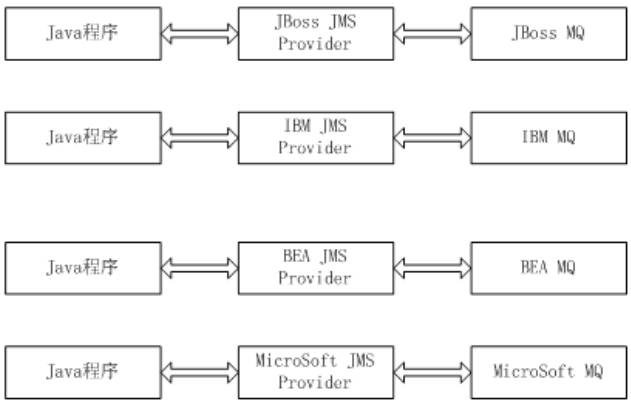

图 3.1　早期 MQ 系统调用结构示意图

Sun 公司为了统一不同消息服务和消息调用接口的标准，推出了 JMS 规范，它规定了 Java 消息队列的设计和调用方法，如图 3.2 所示。

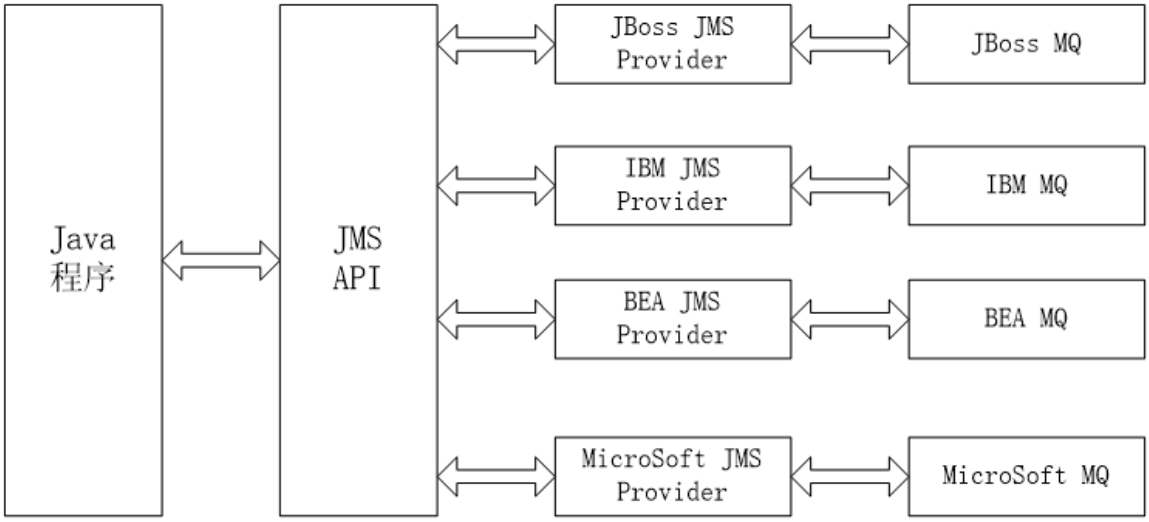

图 3.2　JMS API 示意图

JMS 规范通过标准 JMS API 屏蔽了对不同 JMS Provider 的调用差异，因此针对 JMS API 设计的 Java 程序可以方便地在不同消息服务提供者间移植。

3.2 JMS消息传递模型

JMS 消息系统支持两种消息传递模型：点对点（Point-to-Point，PTP）、发布/订阅（Publish/Subscribe，Pub/Sub）。

PTP 中消息只有一个消费者，一个消息产生后只能被一个消费者消费；Pub/Sub 中消息产生后会被发送给所有的使用者，消息的接收者不唯一。

3.2.1 PTP消息传递模型

PTP 模型是基于队列（Queue）的，它定义了客户端如何向队列发送消息，以及如何从队列接收消息。其结构如图 3.3 所示。

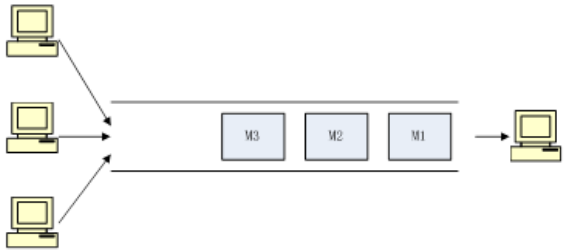

图 3.3　PTP 类型示意图

PTP 模型的特点：

1. PTP 消息队列可以同时有多个发送者，每个发送者可以自由地向当前队列发送消息，被发送的消息按照先进先出的原则依次排列在队列中。
2. 每个消息只能有一个消费者，一旦一个消息被使用后，就将从消息队列中删除，此时其后的消息才能被使用者使用。
3. 消息的使用者可以使用队列中的所有消息，但默认情况下，必须按照消息在队列中的次序依次使用。
4. 当发送者发送了消息后，不管发送者有没有正在运行，都不会影响到接收者接收消息。
5. 接收者在成功接收消息之后须向队列应答成功。
6. 队列可以长久地保存消息直到接收者收到消息。接收者不需要因为担心消息会丢失而时刻和队列保持激活的连接状态，充分体现了异步传输模式的优势。

3.2.2 Pub/Sub消息传递模型

Pub/Sub 模型中定义了如何向一个内容节点发布和订阅消息，这些节点被称作主题（Topic）。即发送者发送消息到主题，订阅者从主题订阅消息。

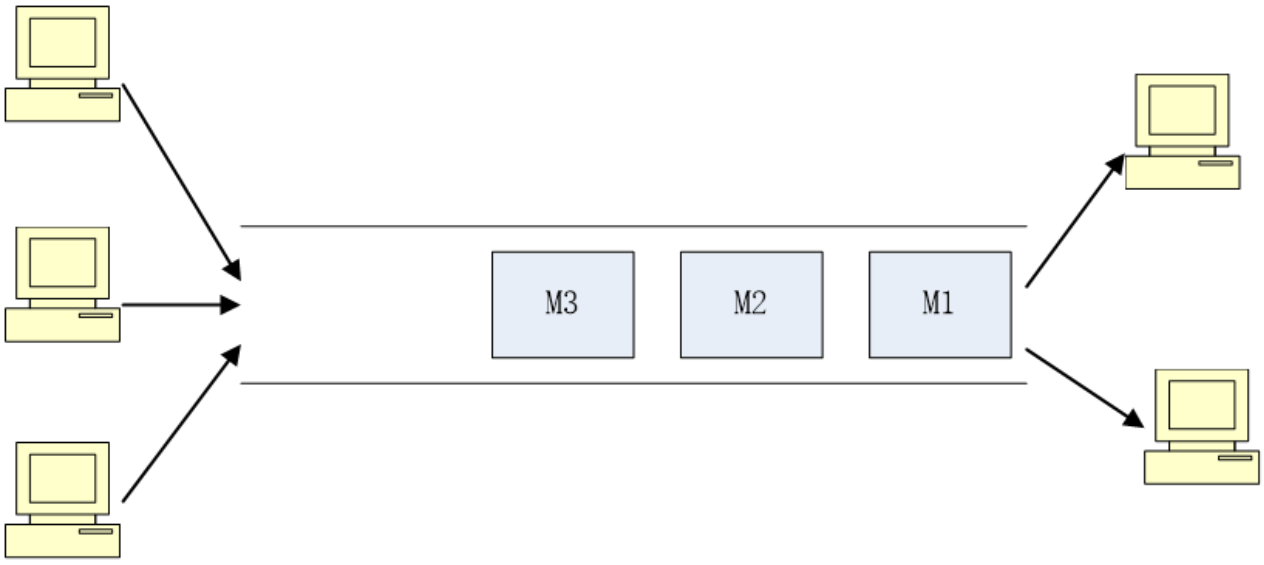

图 3.4　Pub/Sub 类型示意图

Pub/Sub 模型的特点：

1. Pub/Sub 中一个消息可以有多个消费者。默认情况下，如果当前队列没有使用者，则队列中的消息会自动被丢弃。
2. 所有发送者可以自由地向消息队列中发送消息，也遵循先进先出的原则，即最先发送的消息先进入队列，后发送的消息排在队列后面。
3. 针对一个主题的订阅者，它必须创建一个订阅之后，才能消费发布者的消息。

4. 消息订阅分为非持久订阅（non-durable subscription）和持久订阅（durable subscrip-tion）。非持久订阅只有当客户端处于激活状态，也就是和 JMS Provider 保持连接状态才能收到发送到某个主题的消息，而当客户端处于离线状态，这个时间段发到主题的消息将会丢失，永远不会收到。而持久订阅允许订阅者在没有运行时也能收到发布者的消息。

3.3 JBoss MQ配置

在 JBoss 7.1 中，使用 HornetQ 作为默认的 JMS 的提供者。

在 JBoss MQ 中，消息队列和主题的创建可以通过配置文件来完成，JBoss 会在启动时创建配置的消息队列，并启动消息队列服务。

JBoss 默认使用 JBOSS_HOME\standalone\configuration 文件夹下的 standalone.xml 文件作为配置文件。在该文件中找到`hornetq-server`节点，在该节点下有`jms-destinations`配置信息。默认的配置信息如下所示。

```
<jms-destinations>
    <jms-queue name="testQueue">
        <entry name="queue/test"/> <!-- 本地JDNI -->
        <entry name="java:jboss/exported/jms/queue/test"/> <!-- 远程JDNI -->
    </jms-queue>
    <jms-topic name="testTopic">
        <entry name="topic/test"/>
        <entry name="java:jboss/exported/jms/topic/test"/>
    </jms-topic>
</jms-destinations>
```

3.4 JMS消息结构

JMS 消息由以下几部分组成：消息头，属性和消息体。

消息头	描述
JMSDestination	消息的目的地，Topic 或 Queue。
JMSDeliveryMode	消息的发送模式：persistent 或 nonpersistent。 可以通过下面的方式设置：`producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);`
JMSExpiration	表示一个消息的有效期。默认值为 0，表示消息永不过期。 可以通过下面的方式设置：`producer.setTimeToLive(3600000);`（单位为毫秒）
JMSPriority	消息的优先级。0-4为正常的优先级，5-9为高优先级。 可以通过下面方式设置：`producer.setPriority(9);`
JMSMessageID	一个字符串，用来唯一标示一个消息。
JMSTimestamp	当调用`send()`方法的时候，JMSTimestamp 会被自动设置为当前时间。 可以通过下面方式得到这个值：`long timestamp = message.getJMSTimestamp();`
JMSCorrelationID	通常用来关联多个 Message。例如需要回复一个消息，可以把 JMSCorrelationID 设置为所收到的消息的 JMSMessageID。
JMSReplyTo	表示需要回复本消息的目的地。
JMSType	消息类型的识别符。
JMSRedelivered	如果这个值为`true`，表示消息是被重新发送了。

消息类型	消息体
`TextMessage`	`java.lang.String` 对象，如 xml 文件内容
`MapMessage`	名/值对的集合，名是 `String` 对象，值类型可以是 Java 任何基本类型
`BytesMessage`	字节流
`StreamMessage`	Java 中的输入输出流
`ObjectMessage`	Java 中的可序列化对象

3.5 JMS API

3.5.1 JMS API的组成

在 Java EE 6 中，使用的是 JMS API 1.1 版本。它将两种消息传递模型看成相同类型队列，提供一套统一编程接口来处理消息通信。

JMS API 定义了一组基本接口，如图 3.5 所示。

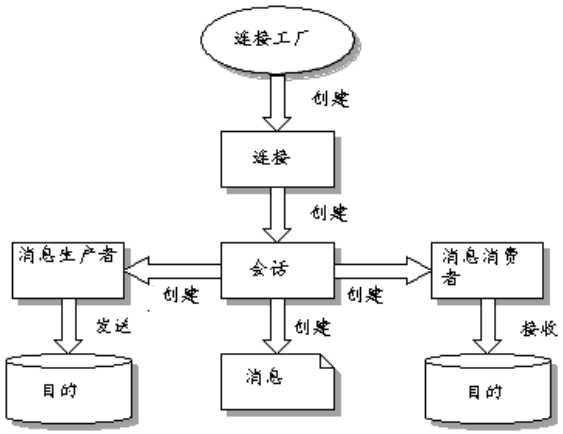

图 3.5　JMS API 的基本接口

1. 连接工厂（`ConnectionFactory`）：为客户端创建一个连接的管理对象，由服务器管理员创建，并绑定到 JNDI 树上。客户端使用 JNDI 检索 `ConnectionFactory`，然后利用它建立 JMS 连接。
2. JMS 连接（`Connection`）：JMS 连接表示一个客户端到消息服务器间的连接通道，所有发送或接收的消息都要经过该通道。
3. JMS 会话（`Session`）：标识客户端和消息队列服务器间的一次会话过程。
4. JMS 消息生产者（`MessageProducer`）：通过 `Session` 向消息队列发送消息，负责将消息对象转变成消息队列能够处理的格式以及消息对象序列化。
5. JMS 消息消费者（`MessageConsumer`）：通过 `Session` 从消息队列中获取消息，封装了对消息格式进行恢复的算法。如果消息的类型是一个复杂对象，则在消息获取的同时还需要对对象进行反序列化操作。

3.5.2 使用JMS API发送消息

编程步骤：

1. 获得一个 JNDI 上下文的引用。
2. 通过 JNDI 查找获得连接工厂。
3. 通过 JNDI 查找目的地（Queue 或 Topic）。
4. 使用连接工厂创建一个连接。
5. 使用连接创建一个会话。

6. 使用会话和目的地创建消息的生产者。
7. 使用会话创建一个需要发送的消息类型的实例。
8. 发送消息。

由于 JBoss 对安全性的限制，为了使用户能够发送消息，必须为用户添加 send 权限。在 standalone.xml 文件中做如下配置：

```
<security-settings>
    <security-setting match="#">
        <!-- 在相应权限中添加用户角色 -->
        <permission type="send" roles="guest testrole"/>
        <permission type="consume" roles="guest testrole"/>
    </security-setting>
</security-settings>
```

示例：向 JMS 消息队列发送消息。

```java
package javaee.jms.producer;

public class JMSProducer
{
    private static final Logger log =
        Logger.getLogger(JMSProducer.class.getName());

    private static final String DEFAULT_MESSAGE = "Welcome to JMS!";
    private static final String DEFAULT_MESSAGE_COUNT = "1";

    private static final String DEFAULT_CONNECTION_FACTORY = "jms/RemoteConnectionFactory";
    private static final String DEFAULT_DESTINATION = "jms/queue/test";
    private static final String DEFAULT_USERNAME = "testJNDI";
    private static final String DEFAULT_PASSWORD = "123456";
    private static final String INITIAL_CONTEXT_FACTORY =
        "org.jboss.naming.remote.client.InitialContextFactory";
    private static final String PROVIDER_URL = "remote://localhost:4447";

    public static void main(String[] args) throws Exception
    {
        Context context = null;
        Connection connection = null;
        try
        {
            // 设置上下文
            System.out.println("设置JNDI访问环境信息也就是设置应用服务器的上下文信息!");
            final Properties env = new Properties();
            env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
            env.put(Context.PROVIDER_URL, PROVIDER_URL); // Context 服务提供者的 URL
            env.put(Context.SECURITY_PRINCIPAL, DEFAULT_USERNAME); // 应用用户的登录名
            env.put(Context.SECURITY_CREDENTIALS, DEFAULT_PASSWORD); // 应用用户的密码

            // 获取 InitialContext 对象
            context = new InitialContext(env);

            // 通过 JNDI 获取连接工厂
            System.out.println("获取连接工厂!");
            ConnectionFactory connectionFactory = (ConnectionFactory) context.lookup(DEFAULT_CONNECTION_FACTORY);

            // 通过 JNDI 获取目的地
            System.out.println("获取目的地!");
            Destination destination = (Destination) context.lookup(DEFAULT_DESTINATION);

            // 使用连接工厂创建一个连接
            connection = connectionFactory.createConnection(DEFAULT_USERNAME, DEFAULT_PASSWORD);

            // 使用连接创建一个会话
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

            // 使用会话和目的地创建消息的生产者
            MessageProducer producer = session.createProducer(destination);
            connection.start();

            // 发送消息
            TextMessage message = null;
            int count = Integer.parseInt(DEFAULT_MESSAGE_COUNT);
            for (int i = 0; i < count; i++)
            {
                message = session.createTextMessage(DEFAULT_MESSAGE);
                producer.send(message);
                System.out.println("message:" + message);
                System.out.println("message:" + DEFAULT_MESSAGE);
            }

            // 等待30秒退出
            CountDownLatch latch = new CountDownLatch(1);
            latch.await(30, TimeUnit.SECONDS);
        }
```

```
        catch (Exception e)
        {
            log.severe(e.getMessage());
            throw e;
        }
        finally
        {
            if (context != null)
            {
                context.close();
            }
            // 关闭连接
            if (connection != null)
            {
                connection.close();
            }
        }
    }
}
```

3.5.3 使用JMS API接收消息

JMS 消息接收分为同步接收和异步接收。

- 同步接收：发送程序向接收程序发送消息后，发送程序会阻塞当前自身的进程，等待接收程序的响应。当发送程序得到接收程序的返回消息之后才会继续向下运行。
  同步接收使用`MessageConsumer.receive()`方法实现，如果消息可用，JMS 服务器将这个消息返回，否则接收者一直等待消息的到来。
- 异步接收：发送方发送消息后不用等待接收方的响应，可以接着处理其他的任务。
  异步接收通过向`MessageConsumer`注册消息监听器对象`javax.jms.MessageListener`，实现队列中消息的异步接收，消息的处理在`onMessage()`方法中实现。

接收消息的编程步骤：

1. 获得一个 JNDI 上下文的引用。
2. 通过 JNDI 查找获得连接工厂。
3. 通过 JNDI 查找目的地（Queue 或 Topic）。
4. 使用连接工厂创建一个连接。
5. 使用连接创建一个会话。
6. 使用会话和目的地创建消息的消费者。
7. 接收消息。

同步接收示例：

```java
public class SyncMesConsumer
{
    private static final Logger log = Logger.getLogger(JMSProducer.class.getName());

    private static final String DEFAULT_CONNECTION_FACTORY = "jms/RemoteConnectionFactory";
    private static final String DEFAULT_DESTINATION = "jms/queue/test";
    private static final String DEFAULT_USERNAME = "testJNDI";
    private static final String DEFAULT_PASSWORD = "123456";
    private static final String INITIAL_CONTEXT_FACTORY =
        "org.jboss.naming.remote.client.InitialContextFactory";
    private static final String PROVIDER_URL = "remote://localhost:4447";

    public static void main(String[] args) throws Exception
    {
        Context context = null;
        Connection connection = null;
        TextMessage msg = null;
        try
        {
            // 设置上下文
            System.out.println("设置JNDI访问环境信息也就是设置应用服务器的上下文信息!");
            final Properties env = new Properties();
            env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
            env.put(Context.PROVIDER_URL, PROVIDER_URL); // Context 服务提供者的 URL
            env.put(Context.SECURITY_PRINCIPAL, DEFAULT_USERNAME); // 应用用户的登录名
            env.put(Context.SECURITY_CREDENTIALS, DEFAULT_PASSWORD); // 应用用户的密码

            // 获取 InitialContext 对象
            context = new InitialContext(env);

            // 通过 JNDI 获取连接工厂
            System.out.println("获取连接工厂!");
            ConnectionFactory connectionFactory = (ConnectionFactory) context.lookup(DEFAULT_CONNECTION_FACTORY);

            // 通过 JNDI 获取目的地
            System.out.println("获取目的地!");
            Destination destination = (Destination) context.lookup(DEFAULT_DESTINATION);

            // 使用连接工厂创建一个连接
            connection = connectionFactory.createConnection(DEFAULT_USERNAME, DEFAULT_PASSWORD);

            // 使用连接创建一个会话
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

            // 使用会话和目的地创建消息的消费者
            MessageConsumer consumer = session.createConsumer(destination);
            connection.start();

            // 接收消息
            CountDownLatch latch = new CountDownLatch(1);
            while (msg == null)
            {
                System.out.println("开始从JBOSS端接收信息-----");

                // 同步接收方法，参数为超时时间，单位为毫秒
                // 如果不填参数或者参数为 0，方法将一直处于封锁状态，直至消息到来
                // 如果超时参数大于 0，则根据指定的超时参数等待一个消息的到来
                // 如果在这个时间内有可用的消息，则返回消息；如果超时后没有可用的消息，则返回 null
                msg = (TextMessage) consumer.receive(5000);

                latch.await(1, TimeUnit.SECONDS);
            }
            System.out.println("接收到的消息的内容: " + msg.getText());
        }
        catch (Exception e)
        {
            log.severe(e.getMessage());
            throw e;
```

```
        }
        finally
        {
            if (context != null)
            {
                context.close();
            }
            // 关闭连接
            if (connection != null)
            {
                connection.close();
            }
        }
    }
}
```

在消息的同步接收模式中，接收消息的调用语句会等待消息到达，只有在获取到队列中的消息或等待超时情况下，方法调用才结束。程序通过`receive()`方法从消息队列中获取消息，如果当前队列中有现成消息，该方法就立即退出，并获取对应消息；如果当前队列没有消息，则该方法就阻塞当前程序，等待消息到达，直到方法等待超时或消息到达，该调用才返回。

异步接收示例：

```
public static class AsynMesListener implements MessageListener
{
    public void onMessage(Message msg)
    {
        TextMessage tm = (TextMessage) msg;
        try
        {
            System.out.println("onMessage, recv text=" + tm.getText());
        }
        catch (Throwable t)
        {
            t.printStackTrace();
        }
    }
}
```

```java
public class AsyncMesConsumer
{
    private static final Logger log = Logger.getLogger(JMSProducer.class.getName());

    private static final String DEFAULT_CONNECTION_FACTORY = "jms/RemoteConnectionFactory";
    private static final String DEFAULT_DESTINATION = "jms/queue/test";
    private static final String DEFAULT_USERNAME = "testJNDI";
    private static final String DEFAULT_PASSWORD = "123456";
    private static final String INITIAL_CONTEXT_FACTORY =
        "org.jboss.naming.remote.client.InitialContextFactory";
    private static final String PROVIDER_URL = "remote://localhost:4447";

    public static void main(String[] args) throws Exception
    {
        Context context = null;
        Connection connection = null;
        TextMessage msg = null;
        try
        {
            // 设置上下文
            System.out.println("设置JNDI访问环境信息也就是设置应用服务器的上下文信息!");
            final Properties env = new Properties();
            env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
            env.put(Context.PROVIDER_URL, PROVIDER_URL); // Context 服务提供者的 URL
            env.put(Context.SECURITY_PRINCIPAL, DEFAULT_USERNAME); // 应用用户的登录名
            env.put(Context.SECURITY_CREDENTIALS, DEFAULT_PASSWORD); // 应用用户的密码

            // 获取 InitialContext 对象
            context = new InitialContext(env);

            // 通过 JNDI 获取连接工厂
            System.out.println("获取连接工厂!");
            ConnectionFactory connectionFactory = (ConnectionFactory) context.lookup(DEFAULT_CONNECTION_FACTORY);

            // 通过 JNDI 获取目的地
            System.out.println("获取目的地!");
            Destination destination = (Destination) context.lookup(DEFAULT_DESTINATION);

            // 使用连接工厂创建一个连接
            connection = connectionFactory.createConnection(DEFAULT_USERNAME, DEFAULT_PASSWORD);

            // 使用连接创建一个会话
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

            // 使用会话和目的地创建消息的消费者
            MessageConsumer consumer = session.createConsumer(destination);

            // 注册监听器
            AsynMesListener l = new AsynMesListener();
            consumer.setMessageListener(l);

            // 启动连接
            connection.start();

            // 接收消息
            CountDownLatch latch = new CountDownLatch(1);
            while (msg == null)
            {
                System.out.println("开始从JBOSS端接收信息-----");
                latch.await(5, TimeUnit.SECONDS);
            }
        }
        catch (Exception e)
        {
            log.severe(e.getMessage());
            throw e;
        }
        finally
```

```
        {
            if (context != null)
            {
                context.close();
            }
            // 关闭连接
            if (connection != null)
            {
                connection.close();
            }
        }
    }
}
```

消息异步接收是以事件方式接收队列中的消息，该方法通常将一个消息接收对象注册到某个队列接收器中，然后程序继续执行其他逻辑。当队列消息到达时，消息接收对象会自动将消息接收下来；如果队列中没有消息，则消息接收逻辑也不会被阻塞。消息的异步接收模式可以有效提高程序执行效率。上述程序中的`AsynMesListener`是消息接收监听器，必须被注册到消息接收对象上，才能对队列中的消息进行异步接收。

3.6 消息驱动Bean

3.6.1 消息驱动Bean的基本概念

消息驱动 Bean（Message-Driven Bean，MDB）是一种通过消息方式为外界提供服务的组件。

JMS 是 MDB 支持的一种最基本消息类型，也是 Java EE 服务器必须提供的消息机制。

MDB 的特点：

1. MDB 组件只是提供一种基于消息的服务请求模式，和具体业务逻辑无关。具体的逻辑通常委托给其他 EJB 组件实现，而 MDB 只是提供一种基于消息的逻辑调用机制。
2. MDB 只从指定消息队列中接收消息，而对消息内容并不知情。当消息到达时，EJB 服务器将指定消息发送到特定的 MDB 组件，消息内容要在 MDB 的`onMessage()`方法中处理。
3. MDB 主要用来处理异步消息，客户端向容器发送一个 JMS 消息之后，不必等待 MDB 处理完毕便可直接返回。消息被发送给由容器管理的 JMS 消息队列，容器在适当的时候通知 MDB 的方法`onMessage()`加以处理。
4. MDB 组件是单线程的。在任一给定时间，一个 MDB 组件对象只处理一个消息，但 EJB 服务器可以为消息队列中的消息提供很多 MDB 组件对象，通过组件对象的并发性，实现消息处理的并发性。

3.6.2 监听PTP模型的MDB开发方法

一个 MDB 需要使用`@MessageDriven`标注，并实现`javax.jms.MessageListener`接口。

监听 PTP 模型的 MDB 示例：

```java
// 设置 MDB 监听的目的类型为 Queue、目的地址为 queue/test
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination", propertyValue = "queue/test")
})
public class PTPMessageBean implements MessageListener
{
    @Resource
    private MessageDrivenContext mdc;

    public PTPMessageBean() {}

    public void onMessage(Message inMessage)
    {
        TextMessage msg = null;
        try
        {
            if (inMessage instanceof TextMessage)
            {
                msg = (TextMessage) inMessage;
                System.out.println("消息驱动Bean:接收到的消息 " + msg.getText());
            }
            else
            {
                System.out.println("消息的类型不正确:  " + inMessage.getClass().getName());
            }
        }
        catch (JMSException e)
        {
            e.printStackTrace();
            mdc.setRollbackOnly();
        }
        catch (Throwable te)
        {
            te.printStackTrace();
        }
    }
}
```

当 EJB 服务器加载 MDB 组件时，首先检查组件所绑定的消息队列是否存在，如果存在，就将 MDB 组件和该队列相关联；如果不存在，就首先创建该队列然后再将其与 MDB 关联。

如果同时存在多个 MDB 组件监听同一个消息队列，处理第一个消息的 MDB 组件是随机选择的，之后各 MDB 组件循环处理后续消息。

PTP 模型的消息是可以永久存储的，消息的发送者和接收者之间没有时间上的依赖性。

3.6.3 监听Pub/Sub模型的MDB开发方法

基于 Topic 队列的 MDB 组件开发方法与基于 Queue 型队列的组件开发方法类似，但是基于 Topic 型队列的 MDB 组件需要配置 `subscriptionDurability` 属性。该属性值可以是 `Durable` 和 `NonDurable`，其中 `NonDurable` 是默认值，表示非持久订阅，即如果某个时刻 EJB 服务器失去了和 JMS 消息服务器的连接，则 MDB 组件会丢掉该时刻所有的消息；反之，对于 `Durable` 型 MDB 组件，JMS 服务会将消息进行保存然后再发给消息接收者，因此消息不会丢失。

基于 `Durable` 型的 MDB 组件必须设置 `clientID` 属性，它是设定 MDB 在消费 Topic 队列时使用的唯一身份标识，必须在队列服务器范围内唯一。而基于 `NonDurable` 类型的 MDB 可以不配置该属性，且开发方法与前面监听点对点的 MDB 开发方法相同。

非持久订阅 MDB 示例：

```java
// 设置消息驱动 Bean 监听的目的类型为 Topic、目的地址为 topic/test
// subscriptionDurability 属性保持默认，为 NonDurable
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "destination", propertyValue = "topic/test")
})
public class NonDurablePubSubMessageBean implements MessageListener
{
    @Resource
    private MessageDrivenContext mdc;

    public void onMessage(Message inMessage)
    {
        TextMessage msg = null;
        try
        {
            if (inMessage instanceof TextMessage)
            {
                msg = (TextMessage) inMessage;
                System.out.println("消息驱动Bean:接收到的消息 " + msg.getText());
            }
            else
            {
                System.out.println("消息的类型不正确:  " + inMessage.getClass().getName());
            }
        }
        catch (JMSException e)
        {
            e.printStackTrace();
            mdc.setRollbackOnly();
        }
        catch (Throwable te)
        {
            te.printStackTrace();
        }
    }
}
```

持久订阅 MDB 示例：

```java
// 设置消息驱动 Bean 监听的目的类型为 Topic、目的地址为 topic/test
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "destination", propertyValue = "topic/test"),
    @ActivationConfigProperty(propertyName = "subscriptionDurability", propertyValue = "Durable"),
    @ActivationConfigProperty(propertyName = "clientID", propertyValue = "consumer")
})
public class DurablePubSubMessageBean implements MessageListener
{
    @Resource
    private MessageDrivenContext mdc;

    public void onMessage(Message inMessage)
    {
        TextMessage msg = null;
        try
        {
            if (inMessage instanceof TextMessage)
            {
                msg = (TextMessage) inMessage;
                System.out.println("消息驱动Bean:接收到的消息 " + msg.getText());
            }
            else
            {
                System.out.println("消息的类型不正确:  " + inMessage.getClass().getName());
            }
        }
        catch (JMSException e)
        {
            e.printStackTrace();
            mdc.setRollbackOnly();
        }
        catch (Throwable te)
        {
            te.printStackTrace();
        }
    }
}
```

3.6.4 消息驱动Bean的生命周期

图 3.6　消息驱动 Bean 的生命周期

1. 不存在状态：主要针对 MDB 组件对象不存在的状态。
2. 池状态：在一个 MDB 组件被部署到服务器后，服务器通常会提前创建一定量的 MDB 组件对象，并将它们临时缓存在缓冲区中，这种状态的对象称为处于池状态的 MDB 组件对象。
3. 调用状态：当有消息到达 MDB 组件所关联的消息队列时，EJB 服务器会接受该消息，并从 MDB 组件池中选取或重新创建一个 MDB 组件对象，调用该组件上的 `onMessage()` 方法处理消息。一个 MDB 组件在特定时间只能处理一个消息，处理完消息的 MDB 组件对象会被释放到组件池中或被销毁。

消息驱动 Bean 的生命周期事件与无状态会话 Bean 相同，也是 PostConstruct 事件和 PreDestroy 事件。