Apusic 金蝶天燕

生成时间：2026-05-19



用户手册

金蝶Apusic应用服务器V9

版权所有 © 深圳市金蝶天燕云计算股份有限公司2026。保留所有权利。

版权声明

本文档所涉及的软件著作权、版权等知识产权已依法进行了注册，由金蝶天燕云计算股份有限公司合法拥有。受《中华人民共和国著作权法》《计算机软件保护条例》《知识产权保护条例》和相关国际版权条约、法律、法规以及其它知识产权法律和条约的保护。未经授权许可，不得非法使用。

免责声明

本文档包含的版权信息由金蝶天燕云计算股份有限公司合法拥有，受法律的保护，金蝶天燕云计算股份有限公司对本文档可能涉及到的非金蝶天燕云计算股份有限公司的信息不承担任何责任。在法律允许的范围内，您可以查阅并仅能够在《中华人民共和国著作权法》规定的合法范围内复制和打印本文档。任何单位和个人未经金蝶天燕云计算股份有限公司书面授权许可，不得使用、修改、再发布本文档的任何部分和内容，否则将被视为侵权，金蝶天燕云计算股份有限公司有依法追究其责任的权利。

本文档如有更新，不另行通知。对本文档中的问题您可向金蝶天燕云计算股份有限公司告知或查询。未经本公司明确授予的任何权利均予保留。

商标声明

Apusic是深圳市金蝶天燕云计算股份有限公司向中华人民共和国国家商标局申请注册的注册商标，注册商标专用权由金蝶天燕合法拥有，受法律保护。未经金蝶天燕的书面许可，任何单位及个人不得以任何方式或理由对该商标的任何部分进行使用、复制、修改、传播、抄录或与其它产品捆绑使用销售。凡侵犯金蝶天燕商标权的，金蝶天燕将依法追究其法律责任。本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

目录

- 1 部分I 发行说明
  - 1.1 概述
    - 1.1.1 摘要
    - 1.1.2 关于本发行说明
    - 1.1.3 发行说明修订历史记录
  - 1.2 应用服务器V9.0功能修订说明
    - 1.2.1 摘要
    - 1.2.2 新特性概述
    - 1.2.3 新增及增强功能
    - 1.2.4 已解决问题（包含Bug修复）
- 2 部分II 技术概览
  - 2.1 概述
  - 2.2 体系结构
    - 2.2.1 Apusic服务器、Apusic域和Apusic实例
    - 2.2.2 Apusic体系结构组成
  - 2.3 Apusic Core内核
  - 2.4 容器
    - 2.4.1 Web Container
      - 2.4.1.1 JSP/Servlet容器
      - 2.4.1.2 JSF引擎
      - 2.4.1.3 Portlet容器
      - 2.4.1.4 虚拟主机
      - 2.4.1.5 Http Connector
      - 2.4.1.6 WebDav
    - 2.4.2 EJB Container
    - 2.4.3 Client Container
    - 2.4.4 容器Session管理
      - 2.4.4.1 Http Session
        - 2.4.4.1.1 In-memory Cache
        - 2.4.4.1.2 存储
        - 2.4.4.1.3 Session Failover
        - 2.4.4.1.4 Session Stick
        - 2.4.4.1.5 Session Timeout
      - 2.4.4.2 Stateful Session Bean
  - 2.5 基础服务
    - 2.5.1 JCA Services

- 2.5.2 Naming
- 2.5.3 JTA
- 2.5.4 死锁检测
- 2.5.5 并发控制
- 2.5.6 JDBC
- 2.5.7 JDBC 语句缓存
- 2.5.8 连接池
- 2.5.9 JMS
- 2.5.10 JavaMail
- 2.5.11 JMX
- 2.6 Web Services
  - 2.6.1 Web Services引擎
    - 2.6.1.1 JAX-WS 规范
    - 2.6.1.2 对规范的支持
    - 2.6.1.3 金蝶Apusic应用服务器的消息安全机制
    - 2.6.1.4 UDDI
- 2.7 安全框架
  - 2.7.1 JAVA授权和鉴定服务（JAAS）
  - 2.7.2 Java EE 容器的授权合约（JACC）
  - 2.7.3 Apusic Security Extensions
  - 2.7.4 Apusic SSO
  - 2.7.5 身份鉴定
  - 2.7.6 授权
  - 2.7.7 加密
  - 2.7.8 安全数据存储
- 2.8 管理与监控
  - 2.8.1 管理接口
  - 2.8.2 管理工具
    - 2.8.2.1 Apusic管理控制台
    - 2.8.2.2 Web管理控制台
    - 2.8.2.3 Shell Script
- 2.9 集群服务
  - 2.9.1 Web集群
  - 2.9.2 JNDI集群
  - 2.9.3 EJB集群
  - 2.9.4 JMS集群
    - 2.9.4.1 消息路由
    - 2.9.4.2 消息队列

- 2.10 技术汇总
- 2.11 Apusic Platform
  - 2.11.1 Apusic Studio
  - 2.11.2 金蝶Apusic消息中间件
- 3 部分III 快速开始
  - 3.1 概述
    - 3.1.1 金蝶Apusic应用服务器的基本介绍
    - 3.1.2 相关资源
    - 3.1.3 默认路径和文件名
  - 3.2 基本功能快速入门
    - 3.2.1 Apusic应用服务器管理工具
    - 3.2.2 安装
      - 3.2.2.1 Linux下安装
      - 3.2.2.2 Windows下安装
    - 3.2.3 卸载
      - 3.2.3.1 Linux下卸载
      - 3.2.3.2 Windows下卸载
    - 3.2.4 启动
      - 3.2.4.1 在Linux上启动服务器
      - 3.2.4.2 在Windows上启动
    - 3.2.5 停止
      - 3.2.5.1 在Linux上停止服务器
      - 3.2.5.2 在Windows上停止服务器
  - 3.3 部署应用程序
    - 3.3.1 摘要
    - 3.3.2 应用打包和部署简介
    - 3.3.3 部署应用程序
      - 3.3.3.1 目录结构
      - 3.3.3.2 准备工作
      - 3.3.3.3 自动部署
      - 3.3.3.4 使用appctl工具部署
      - 3.3.3.5 通过管理控制台部署
      - 3.3.3.6 通过Apusic Studio部署
  - 3.4 小结
- 4 部分IV 安装指南
  - 4.1 前言
  - 4.2 系统配置要求
  - 4.3 安装

- 4.3.1 Linux下安装
- 4.3.2 Windows下安装
- 4.4 卸载
- 4.4.1 Linux下卸载
- 4.4.2 Windows下卸载
- 4.5 许可证授权
- 5 启动与停止
- 5.1 启动
- 5.1.1 在Linux上启动服务器
- 5.1.2 在Windows上启动
- 5.2 停止
- 5.2.1 在Linux上停止服务器
- 5.2.2 在Windows上停止服务器
- 6 部分V 管理指南
- 6.1 摘要
- 6.2 启动和停止服务器
- 6.2.1 命令行启动或停止服务器
- 6.2.1.1 使用startapusic启动脚本
- 6.2.2 开始菜单(限于Windows)
- 6.2.3 Windows系统服务(限于Windows)
- 6.2.4 使用firststep工具启动服务器
- 6.3 配置Classloader
- 6.3.1 Classloader的基本概念
- 6.3.1.1 类装载的方式
- 6.3.1.2 一个基本的Classloader的层次结构
- 6.3.1.3 基本的Classloader
- 6.3.1.4 自定义Classloader
- 6.3.1.5 Caller Classloader和线程上下文Classloader
- 6.3.1.6 JVM中类的唯一性
- 6.3.2 Apusic的Classloader体系
- 6.3.2.1 JavaEE应用对Classloader的要求
- 6.3.2.2 Apusic的Classloader和它们的层次结构
- 6.3.2.3 ServletClassLoader的特性
- 6.3.2.3.1 类的动态加载
- 6.3.2.3.2 ServletClassLoader的多层结构
- 6.3.2.3.3 Session中对象的类动态装载
- 6.3.2.3.4 类装载的Web优先策略
- 6.3.2.4 类装载查看服务

- 6.4 日志服务
  - 6.4.1 理解日志服务子系统
    - 6.4.1.1 日志管理器(LogManager)
    - 6.4.1.2 日志级别
    - 6.4.1.3 日志记录器对象(Logger)
    - 6.4.1.4 日志输出(Handler)
  - 6.4.2 管理与配置
    - 6.4.2.1 编辑logging.xml文件
    - 6.4.2.2 记录客户端IP
- 6.5 基于JMX的可植入服务配置
  - 6.5.1 理解JMX
  - 6.5.2 可植入服务命名
    - 6.5.2.1 域名(domain name)
    - 6.5.2.2 标识属性(key attribute)
    - 6.5.2.3 命名的字符串表示
  - 6.5.3 配置和管理可植入服务
- 6.6 线程池服务配置
  - 6.6.1 默认的公共线程池配置
  - 6.6.2 多路复用线程池配置
  - 6.6.3 WebServer线程池配置
  - 6.6.4 自动调优线程池配置
- 6.7 管理多路复用
  - 6.7.1 理解多路复用
  - 6.7.2 管理与配置
    - 6.7.2.1 apusic.conf文件中相关配置属性说明
    - 6.7.2.2 SSL配置
      - 6.7.2.2.1 TLS/SSL简介
      - 6.7.2.2.2 建立自己的认证授权机构CA
      - 6.7.2.2.3 生成服务器端证书
      - 6.7.2.2.4 颁发并发布个人证书
      - 6.7.2.2.5 配置服务器允许双向认证
      - 6.7.2.2.6 测试双向认证
    - 6.7.2.3 国密配置
      - 6.7.2.3.1 配置
      - 6.7.2.3.2 启动
      - 6.7.2.3.3 访问
- 6.8 管理Apusic Web服务器
  - 6.8.1 管理与配置

- 6.8.2 配置Web应用
  - 6.8.2.1 部署Web应用
  - 6.8.2.2 配置web上下文根路径
- 6.8.3 配置虚拟主机
- 6.8.4 WebDAV配置
- 6.9 管理Session服务
- 6.9.1 管理与配置
- 6.9.2 配置Session序列化
- 6.9.3 配置Session存储
- 6.9.4 管理Session失效时间
- 6.10 使用Apache作为Web代理
- 6.10.1 HTTP Proxy
- 6.10.2 AJP Proxy
- 6.11 使用Microsoft IIS作为Web代理
- 6.11.1 理解ACP(Apusic Connector Protocol)
- 6.11.2 安装
- 6.11.3 管理与配置
- 6.12 管理JDBC服务
- 6.12.1 理解数据库连接池
- 6.12.2 数据库连接池管理与配置
  - 6.12.2.1 编辑datasources.xml文件
    - 6.12.2.1.1 datasource元素
    - 6.12.2.1.2 property
    - 6.12.2.1.3 remote-acl
    - 6.12.2.1.4 结果集缓存
    - 6.12.2.1.5 语句缓存
    - 6.12.2.1.6 连接失效检测
- 6.12.3 活动连接监控
- 6.12.4 JDBC事件监控
- 6.13 消息服务配置
- 6.13.1 理解Apusic消息服务
- 6.13.2 消息路由与存储
  - 6.13.2.1 配置消息存储目录
  - 6.13.2.2 配置消息路由
    - 6.13.2.2.1 理解消息路由
    - 6.13.2.2.2 配置和管理路由
- 6.13.3 管理消息服务
  - 6.13.3.1 配置连接工厂

- 6.13.3.2 配置消息目的地(Destination)
  - 6.13.3.2.1 配置消息队列
  - 6.13.3.2.2 配置消息主题
- 6.13.3.3 安全相关配置
  - 6.13.3.3.1 安全角色
  - 6.13.3.3.2 消息目的地(Destination)访问许可
  - 6.13.3.3.3 范例
- 6.14 管理事务服务
  - 6.14.1 配置事务服务
- 6.15 管理命名服务
  - 6.15.1 配置命名服务
- 6.16 管理应用服务器域
  - 6.16.1 应用服务器域的概念
  - 6.16.2 应用服务器域的管理
- 6.17 应用服务器安全提供程序
  - 6.17.1 安全框架介绍
  - 6.17.2 内置安全提供程序
    - 6.17.2.1 文件存储配置
    - 6.17.2.2 JDBC存储配置
    - 6.17.2.3 LDAP存储配置
  - 6.17.3 自定义安全提供程序
    - 6.17.3.1 实体信息的存储与读取
    - 6.17.3.2 实现身份验证提供程序
    - 6.17.3.3 实现授权提供程序
    - 6.17.3.4 配置映射
- 6.18 管理集群
  - 6.18.1 集群的基本概念及相关术语
    - 6.18.1.1 可扩展性（Scalability）
    - 6.18.1.2 可用性（Availability）
    - 6.18.1.3 可靠性（Reliability）
    - 6.18.1.4 失效转移（Fail-Over）
    - 6.18.1.5 负载均衡（Load Balance）
    - 6.18.1.6 会话粘滞（Session Stick）
    - 6.18.1.7 幂等性（Idempotent）
  - 6.18.2 J2EE集群
    - 6.18.2.1 负载均衡
    - 6.18.2.2 失效转移
  - 6.18.3 Apusic集群

- 6.18.3.1 Web集群
  - 6.18.3.1.1 Apusic负载均衡
  - 6.18.3.1.2 Apusic Http Server负载均衡
  - 6.18.3.1.3 第三方负载均衡
  - 6.18.3.1.4 Apusic Session复制粒度
  - 6.18.3.1.5 Web集群方案
  - 6.18.3.1.6 Session分布式存储
- 6.18.3.2 JNDI集群
  - 6.18.3.2.1 本地JNDI和JNDI集群
  - 6.18.3.2.2 JNDI 负载均衡和JNDI失效恢复
  - 6.18.3.2.3 JNDI集群配置
- 6.18.3.3 EJB集群
  - 6.18.3.3.1 EJB负载均衡和EJB高可用性
  - 6.18.3.3.2 EJB集群配置
  - 6.18.3.3.3 EJB集群实践
- 6.18.3.4 JMS集群
  - 6.18.3.4.1 消息路由
- 6.18.3.5 集群队列
  - 6.18.3.5.1 配置集群队列
  - 6.18.3.5.2 使用集群队列
- 6.19 管理sip服务
  - 6.19.1 SIP概述
- 6.20 SIP的体系结构
  - 6.20.1 SIP协议结构
    - 6.20.1.1 消息体与SDP
    - 6.20.1.2 SIP与Java
  - 6.20.2 配置Apusic SIP服务
- 6.21 配置和管理工具
  - 6.21.1 appctl工具
  - 6.21.2 apclient工具
  - 6.21.3 firststep工具
  - 6.21.4 dep工具
  - 6.21.5 adminmain工具
    - 6.21.5.1 选项
    - 6.21.5.2 用法
    - 6.21.5.3 关于服务的说明
  - 6.21.6 JSPC工具
- 6.22 使用Jconsole监控

- 6.22.1 用Jconsole监控和操作Apusic应用服务器
- 7 附录 A. 附录
- 8 A.2. 附录：缺省的apusic.conf配置文件
- 9 A.3. 附录：缺省的datasources.xml配置文件
- 10 A.4. 附录：缺省的jms.xml配置文件
- 11 A.5. 附录：缺省的logging.xml配置文件
- 12 A.6. 附录：缺省的server.xml配置文件
- 13 A.7. 附录：缺省的auth.cfg配置文件
- 14 A.8. 附录：缺省的clientauth.cfg配置文件
- 15 A.9. 附录：缺省的acp_isapi.properties配置文件
- 16 部分VI 开发指南
  - 16.1 Web开发手册
    - 16.1.1 Apusic JSP 开发
      - 16.1.1.1 JSP概述
        - 16.1.1.1.1 什么是JSP
        - 16.1.1.1.2 JSP如何处理请求
      - 16.1.1.2 Apusic JSP变动及其升级
        - 16.1.1.2.1 从JSP2.0到JSP2.1的主要变动
        - 16.1.1.2.2 从JSP1.2到JSP2.0的主要变动
      - 16.1.1.3 Apusic JSP开发指南
        - 16.1.1.3.1 JSP页面的生命周期
        - 16.1.1.3.2 初始化和结束一个JSP页面
        - 16.1.1.3.3 创建静态内容
        - 16.1.1.3.4 创建动态内容
        - 16.1.1.3.5 表达式语言
        - 16.1.1.3.6 使用JavaBeans
        - 16.1.1.3.7 在JSP页面中包含其他内容
        - 16.1.1.3.8 传递控制到其他Web组件
        - 16.1.1.3.9 在JSP页面中包含Applet
        - 16.1.1.3.10 使用Apusic JSP预编译
        - 16.1.1.3.11 启用JSP热加载
    - 16.1.2 Apusic Http Servlets 开发
      - 16.1.2.1 Http Servlets 概述
        - 16.1.2.1.1 什么是Servlet
        - 16.1.2.1.2 使用Servlet的目的
        - 16.1.2.1.3 Servlet开发概述
        - 16.1.2.1.4 Servlet与JavaEE
        - 16.1.2.1.5 Servlet API参考

- 16.1.2.2 Apusic Http Servlets变动及其升级
  - 16.1.2.2.1 从Servlet2.4到Servlet2.5的主要变动
  - 16.1.2.2.2 从Servlet2.3到Servlet2.4的变动
- 16.1.2.3 Apusic Http Servlets编程初步
  - 16.1.2.3.1 Servlet生命周期
  - 16.1.2.3.2 编写一个基本HTTP Servlet的步骤
  - 16.1.2.3.3 高级特征
- 16.1.2.4 Apusic Http Servlets编程进阶
  - 16.1.2.4.1 初始化Servlet
  - 16.1.2.4.2 使用注解
  - 16.1.2.4.3 编写Service方法
  - 16.1.2.4.4 处理多线程情况
  - 16.1.2.4.5 维护客户端状态
  - 16.1.2.4.6 访问Servlet环境(Servlet Context)
  - 16.1.2.4.7 使用Apusic Services
  - 16.1.2.4.8 调用其他Web资源
  - 16.1.2.4.9 结束一个Servlet
- 16.1.2.5 Apusic Http Servlets高级开发
  - 16.1.2.5.1 Servlet Filtering
  - 16.1.2.5.2 Application Events
- 16.1.3 JSP扩展标记
  - 16.1.3.1 JSP 扩展标记介绍
  - 16.1.3.2 使用扩展标记
    - 16.1.3.2.1 声明扩展标记
    - 16.1.3.2.2 扩展标记的类型
  - 16.1.3.3 定义扩展标记
    - 16.1.3.3.1 标记库描述符
    - 16.1.3.3.2 标记处理类（Tag Handlers ）
    - 16.1.3.3.3 简单标记
    - 16.1.3.3.4 带属性的标记
    - 16.1.3.3.5 带标记体的标记
    - 16.1.3.3.6 定义脚本变量的标记
    - 16.1.3.3.7 协作标记
  - 16.1.3.4 标记处理类如何被调用
- 16.1.4 Apusic JSF
  - 16.1.4.1 什么是JavaServer Faces?
  - 16.1.4.2 JSF页面的生命周期
    - 16.1.4.2.1 重建视图阶段（Restore View Phase）

- 16.1.4.2.2 应用视图值阶段（Apply Request Values Phase）
- 16.1.4.2.3 处理校验阶段（Process Validations Phase）
- 16.1.4.2.4 更新模型值阶段（Update Model Values Phase）
- 16.1.4.2.5 调用应用阶段（Invoke Application Phase）
- 16.1.4.2.6 显示应答阶段（Render Response Phase）
- 16.1.4.3 UI组件模型
- 16.1.4.3.1 UI组件类
- 16.1.4.3.2 组件显示模型
- 16.1.4.3.3 转换模型
- 16.1.4.3.4 事件和监听器模型
- 16.1.4.3.5 校验模型
- 16.1.4.4 导航（Navigation）模型
- 16.1.4.5 Backing Bean管理
- 16.1.4.6 更多参考：
- 16.2 EJB开发手册
- 16.2.1 简介
- 16.2.1.1 Enterprise JavaBean简介
- 16.2.1.1.1 作为组件模型的EJB
- 16.2.1.1.2 EJB组件模型
- 16.2.1.1.3 EJB的类型
- 16.2.1.2 Session Bean
- 16.2.1.2.1 什么是Session Bean
- 16.2.1.2.2 Session Bean的生存时间
- 16.2.1.2.3 会话状态
- 16.2.1.3 Entity Bean
- 16.2.1.3.1 什么是Entity Bean
- 16.2.1.3.2 Entity Bean的特征
- 16.2.1.4 Message-driven Bean
- 16.2.1.4.1 Message-driven Bean 的产生原因
- 16.2.1.4.2 Message-driven Bean 作为一般的JMS 使用者(consumer)
- 16.2.1.4.3 Message-driven Bean 与其他Enterprise Bean
- 16.2.1.5 使用接口定义客户访问
- 16.2.1.5.1 远程访问
- 16.2.1.5.2 本地访问
- 16.2.1.5.3 本地接口与容器管理关系
- 16.2.1.5.4 方法参数和返回值
- 16.2.1.5.5 远程还是本地
- 16.2.1.5.6 执行效率与组件接口类型

- 16.2.1.6 Enterprise Bean的内容
- 16.2.1.7 EJB组件模型的灵活性
- 16.2.1.8 何时使用EJB组件
- 16.2.2 会话Bean
  - 16.2.2.1 会话Bean
    - 16.2.2.1.1 容器与会话Bean
    - 16.2.2.1.2 会话Bean的会话状态
  - 16.2.2.2 组件模型单元
    - 16.2.2.2.1 Home接口
    - 16.2.2.2.2 业务（组件）接口
    - 16.2.2.2.3 组件类
  - 16.2.2.3 生存周期
    - 16.2.2.3.1 有状态Session Bean的生存周期
    - 16.2.2.3.2 无状态Session Bean的生存周期
  - 16.2.2.4 Message-driven Bean
  - 16.2.2.5 Message-driven Bean
    - 16.2.2.5.1 Message-driven Bean与EJB容器、客户端、消息系统
  - 16.2.2.6 组件模型单元
    - 16.2.2.6.1 组件类
    - 16.2.2.6.2 必须遵守的规则
  - 16.2.2.7 生存周期
- 16.2.3 Entity Bean
  - 16.2.3.1 概览
    - 16.2.3.1.1 数据的对象视图
    - 16.2.3.1.2 组件结构
    - 16.2.3.1.3 Primary Key
  - 16.2.3.2 客户端
    - 16.2.3.2.1 JNDI
    - 16.2.3.2.2 远程客户
    - 16.2.3.2.3 本地客户
  - 16.2.3.3 容器与Entity Bean
    - 16.2.3.3.1 通过JNDI定位远程Home接口
    - 16.2.3.3.2 通过JNDI定位本地Home接口
  - 16.2.3.4 组件模型
    - 16.2.3.4.1 Home接口
    - 16.2.3.4.2 Primary Key
    - 16.2.3.4.3 组件接口
    - 16.2.3.4.4 组件类

- 16.2.3.5 生存周期
- 16.2.4 Bean管理持久性的Entity Bean
  - 16.2.4.1 概述
  - 16.2.4.2 Bean管理持久性的Entity Bean
    - 16.2.4.2.1 打开与释放资源
    - 16.2.4.2.2 实例的创建
    - 16.2.4.2.3 实例状态与持久存储
    - 16.2.4.2.4 实例的清除
  - 16.2.4.3 必须遵守的规则与范例
    - 16.2.4.3.1 组件类
    - 16.2.4.3.2 Home接口
    - 16.2.4.3.3 组件接口
  - 16.2.4.4 异常处理
    - 16.2.4.4.1 系统级异常
    - 16.2.4.4.2 应用级异常
    - 16.2.4.4.3 组件异常类型
- 16.2.5 容器管理持久性的Entity Bean
  - 16.2.5.1 概述
  - 16.2.5.2 容器管理持久性的Entity Bean
    - 16.2.5.2.1 CMP模型
    - 16.2.5.2.2 例子说明
    - 16.2.5.2.3 抽象持久性模式（Abstract persistence schema）
    - 16.2.5.2.4 Primary Key
    - 16.2.5.2.5 实例的清除
    - 16.2.5.2.6 Finder方法
    - 16.2.5.2.7 Select方法
    - 16.2.5.2.8 实例的生存周期与开发中的约定
  - 16.2.5.3 必须遵守的规则
    - 16.2.5.3.1 类与接口
    - 16.2.5.3.2 组件类
    - 16.2.5.3.3 辅助类
    - 16.2.5.3.4 ejbCreate方法
    - 16.2.5.3.5 ejbPostCreate方法
- 16.2.6 EJB QL
  - 16.2.6.1 概述
  - 16.2.6.2 定义
    - 16.2.6.2.1 抽象持久类型与查询范围
    - 16.2.6.2.2 命名

- 16.2.6.2.3 范例
- 16.2.6.2.4 返回值类型
- 16.2.6.2.5 FROM子句与定位声明
- 16.2.6.2.6 SELECT子句
- 16.2.6.2.7 ORDER BY字句
- 16.2.6.2.8 NULL值
- 16.2.6.2.9 相等语义
- 16.2.6.2.10 查询语句的限制
- 16.2.6.3 范例
- 16.2.6.3.1 简单查询
- 16.2.6.3.2 使用关系的查询
- 16.2.6.3.3 使用输入参数的查询
- 16.2.6.3.4 定义select方法的查询
- 16.2.6.3.5 EJB QL与SQL
- 16.2.6.4 EJB QL BNF
- 16.2.7 Java Persistence API
- 16.2.7.1 概览
- 16.2.7.2 实体
- 16.2.7.2.1 什么是实体
- 16.2.7.2.2 实体类的一些需求
- 16.2.7.2.3 持久类的持久值域与持久属性
- 16.2.7.2.4 持久值域
- 16.2.7.2.5 持久属性
- 16.2.7.2.6 实体的主键
- 16.2.7.2.7 主键类
- 16.2.7.3 实体的关系
- 16.2.7.3.1 实体关系的多重性
- 16.2.7.3.2 实体关系的方向
- 16.2.7.3.3 实体的继承关系
- 16.2.7.3.4 继承结构映射策略
- 16.2.7.4 管理实体
- 16.2.7.4.1 实体管理器Entity Manager
- 16.2.7.4.2 管理实体实例的生存周期
- 16.2.7.4.3 创建查询
- 16.2.7.4.4 查询中的命名参数
- 16.2.7.4.5 查询中的顺序参数
- 16.2.7.5 持久单元
- 16.2.7.5.1 persistence.xml文件

- 16.2.8 Java持久查询语言
  - 16.2.8.1 摘要
  - 16.2.8.2 新特性
    - 16.2.8.2.1 批量更新与删除
    - 16.2.8.2.2 连接操作
    - 16.2.8.2.3 GROUP BY和HAVING子句
    - 16.2.8.2.4 投影
    - 16.2.8.2.5 SELECT子句中的构造表达式
    - 16.2.8.2.6 子查询
    - 16.2.8.2.7 查询中的命名参数
  - 16.2.8.3 Java持久查询语言BNF
- 16.2.9 EJB的环境
  - 16.2.9.1 概述
  - 16.2.9.2 环境项（environment entry）
    - 16.2.9.2.1 访问环境项
    - 16.2.9.2.2 设置环境项
    - 16.2.9.2.3 EJB引用
  - 16.2.9.3 资源管理器的连接创建器引用（resource manager connection factory reference）
    - 16.2.9.3.1 使用资源管理器的连接创建器引用
    - 16.2.9.3.2 声明连接创建器引用
    - 16.2.9.3.3 引用的映射
    - 16.2.9.3.4 访问引用
    - 16.2.9.3.5 标准的连接创建器类型
  - 16.2.9.4 资源环境引用（resource environment references）
    - 16.2.9.4.1 声明资源环境引用
    - 16.2.9.4.2 访问资源环境引用
  - 16.2.9.5 javax.transaction.UserTransaction接口
- 16.2.10 EJB的事务
  - 16.2.10.1 概述
  - 16.2.10.2 客户端事务与未指明的事务上下文
  - 16.2.10.3 EJB的事务划分
    - 16.2.10.3.1 Bean管理事务（程序型的事务划分）
    - 16.2.10.3.2 容器管理事务（声明型的事务划分）
  - 16.2.10.4 使用Bean管理事务与容器管理事务
- 16.2.11 EJB的安全管理
  - 16.2.11.1 安全模型
    - 16.2.11.1.1 安全角色（Security Role）与方法权限
    - 16.2.11.1.2 安全主体（Security Principal）

- 16.2.11.1.3 安全角色与安全主体的映射
- 16.2.11.2 使用安全模型
- 16.2.11.2.1 声明安全角色
- 16.2.11.2.2 定义方法许可
- 16.2.11.2.3 定义角色映射
- 16.2.11.2.4 调用中的身份传播
- 16.2.11.2.5 关于安全角色引用
- 16.2.11.2.6 getCallerPrincipal
- 16.2.11.2.7 isCallerInRole
- 16.2.11.2.8 声明代码中使用的安全角色引用
- 16.2.12 EJB Timer Service
- 16.2.12.1 概述
- 16.2.12.2 组件模型单元
- 16.2.12.2.1 接口描述
- 16.2.12.2.2 TimerService接口
- 16.2.12.2.3 TimedObject接口
- 16.2.12.2.4 TimerHandle接口
- 16.2.12.2.5 Timer接口
- 16.2.12.2.6 功能实现框架
- 16.2.12.3 使用EJB Timer Service
- 16.2.12.3.1 使用范围
- 16.2.12.3.2 创建Timer
- 16.2.12.3.3 定义Timer需要执行的内容
- 16.2.12.3.4 取消和保存Timer
- 16.2.12.3.5 获得Timer的信息
- 16.2.12.3.6 Timer和交易
- 16.3 Web Service开发手册
- 16.3.1 概述
- 16.3.1.1 Web Services与SOA
- 16.3.1.2 Web服务类型
- 16.3.1.2.1 RPC方式的Web服务
- 16.3.1.2.2 基于SOAP的Web服务
- 16.3.1.2.3 REST风格Web服务
- 16.3.1.2.4 选择适用的Web服务类型
- 16.3.2 在Apusic AS中开发WS
- 16.3.2.1 通用构件
- 16.3.2.1.1 WSDL
- 16.3.2.1.2 SEI

- 16.3.2.1.3 webservices.xml
- 16.3.2.2 JAX-WS POJO
- 16.3.2.3 Stateless EJB
- 16.3.3 开发WS客户端
- 16.3.3.1 静态代理
- 16.3.3.2 直接操作XML消息动态调用
- 16.3.3.3 JavaEE托管环境中的WS客户端注入
- 16.3.4 Apusic AS的SPI支持
- 16.4 客户端开发手册
- 16.4.1 客户端简介
- 16.4.1.1 客户端类型
- 16.4.1.1.1 Web客户端
- 16.4.1.1.2 应用客户端
- 16.4.1.1.3 CORBA客户端
- 16.4.1.1.4 Web Services客户端
- 16.4.2 使用应用客户端容器
- 16.4.2.1 应用客户端容器简介
- 16.4.2.2 开发应用客户端
- 16.4.2.2.1 定位Home接口
- 16.4.2.2.2 创建EJB实例
- 16.4.2.2.3 调用业务方法
- 16.4.2.3 打包应用客户端
- 16.4.2.4 运行应用客户端
- 16.4.2.5 不使用应用客户端容器
- 16.4.2.6 关于范例
- 16.4.3 CORBA客户端开发
- 16.4.3.1 开发CORBA客户端
- 16.5 金蝶Apusic应用服务器系统服务
- 16.5.1 数据库连接池开发
- 16.5.1.1 概述
- 16.5.1.2 使用JDBC连接池
- 16.5.1.2.1 服务器端JDBC应用
- 16.5.1.2.2 客户端JDBC应用
- 16.5.1.2.3 范例
- 16.5.2 消息服务开发
- 16.5.2.1 理解JMS
- 16.5.2.1.1 消息中间件
- 16.5.2.1.2 JMS接口描述

- 16.5.2.1.3 JMS和其它企业级JAVA API的关系
- 16.5.2.1.4 JMS消息模型
- 16.5.2.1.5 消息的同步异步接收
- 16.5.2.1.6 PTP模型
- 16.5.2.1.7 PUB/SUB模型
- 16.5.2.2 使用JMS
- 16.5.2.2.1 开发JMS Client的步骤
- 16.5.2.2.2 PTP模型应用
- 16.5.2.2.3 PUB/SUB模型应用
- 16.5.2.2.4 权限设置
- 16.5.3 使用JAAS用户认证
- 16.5.3.1 JAAS简介
- 16.5.3.1.1 一个简单的例子
- 16.5.3.1.2 JAAS核心类和接口
- 16.5.3.1.3 配置LoginModules
- 16.5.3.1.4 编写Policy文件
- 16.5.3.1.5 运行范例程序
- 16.5.3.2 JAAS和Apusic
- 16.5.3.3 Apusic JAAS开发
- 16.5.4 使用Apusic事务服务
- 16.5.4.1 事务服务简介
- 16.5.4.2 配置Apusic事务服务
- 16.5.4.3 分布式事务恢复
- 16.5.4.3.1 需配置的参数
- 16.5.4.3.2 使用方法
- 16.5.4.4 使用事务服务
- 16.5.4.4.1 在EJB中使用事务服务
- 16.5.4.4.2 在客户端中使用事务服务
- 17 部分VII 应用程序部署指南
- 17.1 打包和部署Java EE应用
- 17.1.1 摘要
- 17.1.2 打包和部署简介
- 17.1.3 打包和部署Web模块
- 17.1.3.1 Web应用的目录结构
- 17.1.3.2 Web模块的部署描述文件
- 17.1.3.3 Web模块的两种形式
- 17.1.3.4 部署Web模块
- 17.1.3.5 自动部署

- 17.1.4 打包和部署EJB模块
  - 17.1.4.1 EJB-JAR文件结构
  - 17.1.4.2 EJB模块的部署描述文件
  - 17.1.4.3 EJB模块的两种形式
  - 17.1.4.4 部署EJB模块
  - 17.1.4.5 自动部署
- 17.1.5 打包和部署Java EE应用
  - 17.1.5.1 Java EE应用的目录结构
  - 17.1.5.2 Java EE应用的部署描述文件
  - 17.1.5.3 Java EE应用的两种形式
  - 17.1.5.4 部署Java EE应用
  - 17.1.5.5 自动部署
- 17.1.6 Apusic部署描述文件
  - 17.1.6.1 apusic-application.xml中的元素
    - 17.1.6.1.1 apusic-application元素
    - 17.1.6.1.2 module元素
    - 17.1.6.1.3 ejb元素
    - 17.1.6.1.4 entity元素
    - 17.1.6.1.5 jndi-name元素
    - 17.1.6.1.6 local-jndi-name元素
    - 17.1.6.1.7 cmp元素
    - 17.1.6.1.8 jdbc元素
    - 17.1.6.1.9 datasource-name元素
    - 17.1.6.1.10 table-name元素
    - 17.1.6.1.11 auto-create-table元素
    - 17.1.6.1.12 auto-generate-key元素
    - 17.1.6.1.13 field-mapping元素
    - 17.1.6.1.14 field-name元素
    - 17.1.6.1.15 column-name元素
    - 17.1.6.1.16 load-unit元素
    - 17.1.6.1.17 unit-name元素
    - 17.1.6.1.18 cmp-field元素
    - 17.1.6.1.19 cmr-field元素
    - 17.1.6.1.20 finder元素
    - 17.1.6.1.21 description元素
    - 17.1.6.1.22 method元素
    - 17.1.6.1.23 method-intf元素
    - 17.1.6.1.24 method-name元素

- 17.1.6.1.25 method-params元素
- 17.1.6.1.26 method-param元素
- 17.1.6.1.27 sql-clause元素
- 17.1.6.1.28 is-modified-method-name元素
- 17.1.6.1.29 ejb-ref元素
- 17.1.6.1.30 ejb-ref-name元素
- 17.1.6.1.31 ejb-local-ref元素
- 17.1.6.1.32 resource-ref元素
- 17.1.6.1.33 res-ref-name元素
- 17.1.6.1.34 resource-env-ref元素
- 17.1.6.1.35 resource-env-ref-name元素
- 17.1.6.1.36 session元素
- 17.1.6.1.37 message-driven元素
- 17.1.6.1.38 message-driven-destination元素
- 17.1.6.1.39 connection-factory-name元素
- 17.1.6.1.40 destination-name元素
- 17.1.6.1.41 auto-create-destination元素
- 17.1.6.1.42 client-id元素
- 17.1.6.1.43 durable-subscription-name元素
- 17.1.6.1.44 relationship-mapping元素
- 17.1.6.1.45 ejb-relation-name元素
- 17.1.6.1.46 source-role元素
- 17.1.6.1.47 sink-role元素
- 17.1.6.1.48 cmp-resource元素
- 17.1.6.1.49 java元素
- 17.1.6.1.50 web元素
- 17.1.6.1.51 context-root元素
- 17.1.6.1.52 mail-session元素
- 17.1.6.1.53 property元素
- 17.1.6.1.54 security-role元素
- 17.1.6.1.55 role-name元素
- 17.1.6.1.56 principal元素
- 17.1.6.1.57 group元素

- 17.1.7 application_5_0.xsd
- 17.1.8 web-app_2_4.xsd
- 17.1.9 ejb-jar_3_0.xsd
- 17.1.10 application-client_5_0.xsd
- 17.1.11 apusic-application_5_0.dtd

- 18 部分VIII 管理控制台使用指南
  - 18.1 进入Apusic管理监控台
    - 18.1.1 进入到登录页面
    - 18.1.2 登录到管理监控首页
  - 18.2 应用管理
    - 18.2.1 部署J2EE应用
    - 18.2.2 管理J2EE应用
    - 18.2.3 监控J2EE应用
    - 18.2.4 类库管理
    - 18.2.5 AAR管理
  - 18.3 数据源管理
    - 18.3.1 新增数据源
    - 18.3.2 配置和测试数据源
    - 18.3.3 停止和启动数据源
  - 18.4 JNDI树
    - 18.4.1 查看JNDI树
  - 18.5 日志监控
    - 18.5.1 日志监控
    - 18.5.2 下载日志
  - 18.6 服务面板配置
    - 18.6.1 HTTP服务配置
    - 18.6.2 系统服务配置
    - 18.6.3 高级服务配置
    - 18.6.4 线程池服务
  - 18.7 JMS消息服务配置
    - 18.7.1 管理JMS消息服务
    - 18.7.2 配置JMS服务
  - 18.8 系统配置
    - 18.8.1 系统参数配置
    - 18.8.2 日志配置
    - 18.8.3 用户组
    - 18.8.4 安全域参数配置
    - 18.8.5 生命周期监听
    - 18.8.6 系统备份
    - 18.8.7 JNDI资源
  - 18.9 监控
    - 18.9.1 javaEE组件监控
    - 18.9.2 数据源监控

- 18.9.3 应用会话监控
- 18.9.4 JTA事务监控
- 18.9.5 HTTP链接监控
- 18.9.6 JVM内存监控
- 18.9.7 线程监控
- 18.9.8 线程DUMP
- 18.9.9 类加载监控
- 18.10 三员分立
- 18.11 角色管理
- 18.11.1 新建角色
- 18.11.2 编辑角色
- 18.11.3 角色授权
- 18.11.4 删除角色
- 18.12 用户管理
- 18.12.1 新建用户
- 18.12.2 编辑用户信息
- 18.12.3 删除用户信息
- 18.13 密码策略
- 18.14 重置密码
- 18.15 审计日志配置
- 18.16 日志审计
- 18.16.1 操作日志
- 18.16.2 审计日志
- 18.16.3 自动备份设置
- 18.17 管控台转为普通应用
- 19 部分IX Apusic Http Server使用指南
- 19.1 Apusic Http Server启动和停止
- 19.2 使用Apusic Http Server作为web代理
- 19.3 使用Apusic Http Server作为负载均衡前置机
- 19.3.1 配置Apusic Http Server
- 19.3.2 配置Apusic应用服务器
- 19.3.2.1 配置高可用性
- 19.3.2.2 配置Session粘滞
- 20 部分X Node Manager使用指南
- 20.1 概述
- 20.2 NodeManager启动与停止
- 20.3 NodeManager目录结构
- 20.3.1 目录结构

- 20.4 主要的配置说明
  - 20.4.1 Node Manager系统服务的配置文件
  - 20.4.2 日志输出配置文件
  - 20.4.3 NodeManger插件配置文件
- 21 部分XI 动态集群管理
  - 21.1 概述
    - 21.1.1 前言
      - 21.1.1.1 系统网络拓扑图
      - 21.1.1.2 系统架构图
  - 21.2 节点管理器
    - 21.2.1 NodeManager启动与停止
    - 21.2.2 NodeManager目录结构及配置文件详细说明
  - 21.3 管理控制中心
    - 21.3.1 概述
    - 21.3.2 安装与配置
      - 21.3.2.1 部署“管理控制中心”应用包
      - 21.3.2.2 “管理控制中心”个性化配置
        - 21.3.2.2.1 部署机器为多网卡，设置指定与NodeManger交互的端口
        - 21.3.2.2.2 数据库配置
        - 21.3.2.2.3 数据库创建脚本
    - 21.3.3 动态集群管理监控台
      - 21.3.3.1 进入到登录页面
      - 21.3.3.2 登录到动态集群管理监控首页
      - 21.3.3.3 进入动态集群管理
      - 21.3.3.4 资源池
        - 21.3.3.4.1 资源管理
        - 21.3.3.4.2 资源统计
      - 21.3.3.5 运行环境
        - 21.3.3.5.1 策略管理
        - 21.3.3.5.2 数据源管理
        - 21.3.3.5.3 集群管理
      - 21.3.3.6 应用管理
        - 21.3.3.6.1 部署应用
        - 21.3.3.6.2 应用列表
        - 21.3.3.6.3 应用操作
      - 21.3.3.7 定时任务管理
        - 21.3.3.7.1 新增任务
        - 21.3.3.7.2 删除任务

- 21.3.3.7.3 查看详细
- 21.3.3.7.4 查看执行日志
- 21.3.3.8 补丁管理
- 21.3.3.8.1 新增
- 21.3.3.8.2 升级
- 21.3.3.8.3 查看
- 21.3.3.8.4 删除
- 21.3.3.9 系统管理
- 21.3.3.9.1 系统日志
- 21.3.3.9.2 策略调度
- 21.3.3.10 证书管理
- 21.3.3.10.1 基本概念
- 21.3.3.10.2 使用自签名证书配置AAS SSL
- 21.3.3.10.3 单向认证
- 21.3.3.10.4 使用CA签发证书配置 AAS SSL
- 22 部分 XII. 第三方监控工具
- 22.1 Zabbix
- 22.1.1 安装
- 22.1.2 配置
- 22.1.3 Zabbix 监控
- 23 部分XIII 许可证授权
- 23.1 特征码获取
- 23.2 提示解读
- 24 部分XIV 常用参数说明及常见问题解决
- 24.1 Apusic应用服务器常用参数说明
- 24.1.1 VM.options配置
- 24.1.2 web.xml全局配置参数
- 24.1.3 JVM参数
- 24.2 Apusic应用服务器常见问题及解决
- 24.2.1 解决安全软件检测后出现的问题
- 24.2.2 如何使用AAS进行应用预编译
- 24.2.3 编码问题
- 24.2.3.1 基础知识
- 24.2.3.2 常见编码问题解析
- 24.2.3.3 Apusic的解决方案
- 24.2.4 多个应用共享session
- 24.2.5 Spring的nativeJdbcExtractor问题
- 24.2.6 管理应用运行在独立端口配置的问题

- 24.2.7 AIX系统下ORBSingleton问题
- 24.2.8 应用服务器启动慢的问题
- 24.2.9 配置使用openJPA
- 24.2.10 GemFile与Apusic集成
  - 24.2.10.1 安装配置GemFile
  - 24.2.10.2 配置apusic应用服务器
  - 24.2.10.3 启动apusic应用服务器
- 24.2.11 应用程序设置停止时销毁应用创建的线程池、定时器
- 24.2.12 开启IPV6
- 24.2.13 同一域下创建多个监听器，使应用可以部署到不同的端口上
- 24.2.14 配置CORS（跨域资源共享）

1 部分I 发行说明

1.1 概述

1.1.1 摘要

金蝶Apusic应用服务器V9.0是一款标准、安全、高效、集成并且具有丰富功能的企业级应用服务器（Enterprise Application Server），它用于实现基于SOA的企业应用和服务，为企业应用和服务提供坚不可摧的基础架构支撑。金蝶Apusic应用服务器V9.0在8.0的基础上进行了大量扩充和优化，在产品功能、性能、安全性、扩展性及兼容性等方面得到显著提升。金蝶Apusic应用服务器V9.0大大简化了创建和管理JavaEE应用的任务，并为之提供了可伸缩、高性能、高可用的运行环境。

1.1.2 关于本发行说明

本发行说明包含金蝶Apusic应用服务器V9.0发行时的重要信息，主要包括新增加的功能、改进或增强的功能、已处理和解决的问题等。开始使用金蝶Apusic应用服务器V9.0之前，请先阅读本文档。

1.1.3 发行说明修订历史记录

发行说明修订历史记录如下表所示：

修订日期	说明
2014年1月	金蝶Apusic应用服务器V9.0产品的发行版，增强管控特性及优化配置，提供对新规范的支持，比如EJB3.1、Servlet3.0。
2014年6月	金蝶Apusic应用服务器V9.0产品的发行版，修复兼容性问题，优化了服务器性能表现，并增加了对Hyper-v的支持。
2014年12月	金蝶Apusic应用服务器V9.0产品的发行版，修复兼容性问题，增加websocket支持，增加多数据源支持，支持应用多版本部署，增加配置备份等功能，并提高了服务器的安全防护。
2015年9月	金蝶Apusic应用服务器V9.0产品的发行版，修复兼容性问题，对Web引擎、数据库连接池、Session管理模块等功能进行了重构和性能的提升，同时对龙芯操作系统上的性能进行了优化。
2016年9月	金蝶Apusic应用服务器V9.0产品的发行版，对web引擎和数据源管理等模块进行了优化、增加对SIP协议支持、增加自动调优线程池等模块。在管理上增加了证书管理、多版本部署以及补丁管理等功能，有效提升管理效率。
2016年11月	金蝶Apusic应用服务器V9.0产品的发行版，增加了控制台上应用的类库管理，axis应用的AAR管理，JNDI资源管理3个功能。对服务器的密码规则进行了改变，不再是使用默认密码，而是第一次启动时设置密码。
2020年9月	金蝶Apusic应用服务器V9.0产品的发行版，对web引擎、日志模块、管理监控等模块进行了优化，支持更高的安全性，支持JDK9及JDK11的运行环境等
2023年12月	更新授权码获取方式

1.2 应用服务器V9.0功能修订说明

1.2.1 摘要

金蝶Apusic应用服务器V9.0在8.0基础上，对产品功能、性能、安全性、兼容性等方面进行了扩充和优化，具备更高的效率和更好的用户体验。本发行说明所述内容是自9.0以来，金蝶Apusic应用服务器的产品功能更新和Bug修复情况。

1.2.2 新特性概述

1. 增加了一个新的数据源的链接池实现，能够支持更大的并发用户访问。默认使用新的链接池，如果需要使用旧的数据源线池实现，则需要配置jvm参数apusic.connector.useOldCP=true。

2. 增加在NIO的模式下支持sendfile的功能，提高对静态文件输出的性能，配置说明如下：在apusic.conf中Muxer服务增加属性: `<ATTRIBUTE NAME="SendFile" VALUE="False"/>`

表示多路复用端口访问的静态文件启用sendfile支持。该值默认为False，表示不开启该功能，如果设置为true则开启该功能；开启sendfile后，在应用的web.xml中增加参数sendfileSize,配置文件大小大于该值的情况下文件通过sendFile发送，默认为49152字节(48K)，配置例子如下：

```
<context-param>
    <description>file size >= sendfileSize may trigger sendfile,default is
48kb </description>
    <param-name>sendfileSize</param-name>
    <param-value>49152</param-value>
</context-param>
```

3. 提供包共享的方式，解决多个应用共享jar包的问题。

一种方式是设置每一个应用的web.xml，只影响到设置的应用，如下：

```
<context-param>
    <param-name>com.apusic.webapp.sharelib</param-name>
    <param-value>D:/aa/ojdbc6_g.jar,D:/aa/jtds-1.2.2.jar</param-value>
</context-param>
```

一种方式是在vm.options设置，影响域下所有的应用，如下：

```
com.apusic.webapp.sharelib=D:/aa/ojdbc6_g.jar,D:/aa/jtds-1.2.2.jar
```

4. 应用可以配置自定义ClassLoader，自定义的ClassLoader必须是ServletClassLoader的子类，在web.xml进行配置，类似：

```
<context-param>
    <param-name>apusic.SERVLET_CLASSLOADER_CLASS</param-name>
    <param-
```

```
value>org.springframework.instrument.apusic.ApuiscInstrumentableClassLoader</pa
value>
</context-param>
```

5. 增加通用的用户定义jndi对象的绑定，可以在全局进行使用。目前绑定类型的构造和注入属性都必须是基本数据类型，在config/jndi_resource.xml文件进行配置。
6. 实现JAVAEE7下的WebSocket功能。
7. 实现多重数据源功能，支持负载均衡策略和失效转移。
8. 实现应用版本管理，支持Web应用不中断更新，支持应用不同版本的部署。
9. 实现配置备份功能。
10. 提供更强大的安全防护，处理了安全问题，例如跨站点脚本攻击，请求定置，SSL DOC等。
11. 提供程序调用脚本以及第三方开发API。

通过脚本和api，第三方客户可以实现对应用服务器的基础管理。

12. 新增基于JDK线程池开发的应用服务器线程池，简化线程调度和统一信息，提高线程执行效率。配置方法在vm.options中增加：

```
com.apusic.ThreadPoolImpl=com.apusic.util.JDKThreadPoolImp
```

13. 新增基于web的轻量级数据源，在不使用两阶段事务的场景下，性能更高。

在apusic.conf中增加SimpleDSService服务，如下：

```
<SERVICE CLASS="com.apusic.jdbc.simple.SimpleDSService">
</SERVICE>
```

在域的config/simpledatasource.xml文件中配置需要使用的数据源。

14. 增加对虚拟化环境Hyper-v的支持。
15. 自动调优线程池。

根据用户的并发数，动态调整线程池的配置，能够有效利用资源，最大化应用系统的吞吐量。

要使用自动调优线程池，需要调整线程池的实现程序，把需要使用自动调优功能的线程池实现类修改为com.apusic.util.SelfTuneThreadPoolService。其中服务属性MinSpareThreads和MaxThreads用于控制自动调优的线程数量区间，其他参数和普通线程池参数作用类似。

```
<SERVICE CLASS="com.apusic.jdbc.simple.SimpleDSService">
</SERVICE>
<SERVICE CLASS="com.apusic.util.SelfTuneThreadPoolService"
NAME="apusic:service=ThreadPool,name=HTTPHandler">
    <ATTRIBUTE NAME="MinSpareThreads" VALUE="5" />
    <ATTRIBUTE NAME="MaxSpareThreads" VALUE="30" />
    <ATTRIBUTE NAME="MaxThreads" VALUE="400" />
    <ATTRIBUTE NAME="MaxQueueSize" VALUE="5000" />
    <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
</SERVICE>
```

16. SIP协议支持。

SIP（Session Initiation Protocol，会话初始协议）是由IETF制定的多媒体通信协议。它是一个基于文本的应用层控制协议，定义了如何在通信设备之间相互连接和信息交换，用于创建、修改和释放一个或多个参与者的会话。

该版本完全兼容SIP Servlets1.1规范（JSR289），要在Apusic应用服务器中使用SIP，需要配置相应的服务（配置在应用部署服务（J2EEDeployer）之前）。

首先配置SipService，主要包括路由文件和Sip协议栈文件的配置,如下所示：

```
<SERVICE CLASS="com.apusic.web.sip.SipService">
    <ATTRIBUTE NAME="DarConfigurationFileLocation" VALUE="config/sip/sip-
dar.properties" />
    <ATTRIBUTE NAME="SipStackPropertiesFileLocation" VALUE="config/sip/sip-
stack.properties" />
</SERVICE>
```

其次是配置SipConnectorService，如下：

```
<SERVICE CLASS="com.apusic.web.sip.SipConnectorService"
NAME="apusic:service=SipConnector,name=udp">
    <ATTRIBUTE NAME="Port" VALUE="5080" />
    <ATTRIBUTE NAME="SignalingTransport" VALUE="udp" />
</SERVICE>
<SERVICE CLASS="com.apusic.web.sip.SipConnectorService"
NAME="apusic:service=SipConnector,name=tcp">
    <ATTRIBUTE NAME="Port" VALUE="5080" />
    <ATTRIBUTE NAME="SignalingTransport" VALUE="tcp" />
</SERVICE>
```

SipConnectorService主要是配置监听的端口和使用的协议。SipConnectorService可以配置多个，除了上面的udp和tcp，还可以配置ws和tls，而且它们的端口必须唯一，更多的说明请参考用户手册。

17. 内置对国密算法的支持，支持客户端的方式访问基于国密算法的HTTPS协议

内置对国密算法SM2、SM3以及SM4支持，并可用于客户端方式访问基于国密算法的HTTPS协议。

18. 内置提供多个便利的工具

在安装目录的tools目录下，内置了多个便利的工具和文档。

1. LDAP服务器:文件ldap_server.zip，提供了基于web的ldap服务应用，将该应用部署后，即可以提供ldap的服务；
2. 与Hibernate框架整合：文件integerate_with_hibernate.zip是与hibernate框架整合的相关内容及说明；
3. 与Spring框架整合：文件integerate_with_spring.zip是与Spring框架整合的相关内容及说明；
4. 开发插件：目录dev-plugins下提供了基于eclipse及myeclipse开发插件；
5. 管控API：文件aas-monitor-api.zip提供了与AAS进行交互的管控API和说明；
6. 脚本管理工具：目录中提供了数据源配置、应用发布等功能的脚本管理工具。

19. 多种序列化方式的支持

在用户并发数很多的时候，会产生很多的Session，如果都存在于内存中，会占用太多的内存空间，需要把Session存储的信息保存到外部的存储中，而存储的关键是需要把Session存储的对象进行序列化。Session对象序列化的性能，对应用的性能有很大的影响。

该版本提供了4种序列化的方式，分别是jdk、fst、hessian、kryo，用户可以根据实际场景选择最优的方式进行，默认是jdk方式。各个方式的对比如下：

- fst 序列化：
  1. 序列化相关的类必须实现Serializable/Externalizable接口
  2. 兼容jdk序列化
  3. 速度比jdk默认序列化快2-10倍
- hessian序列化：
  1. 序列化相关的类可以不实现Serializable/Externalizable接口
  2. 速度较慢，比jdk方式快
- kryo序列化：
  1. 序列化的类需要有无参构造函数
  2. 速度快,生成的字节小

序列化速度： kryo > fst > hessian > jdk可以通过vm.options文件中配置属性apusic.http.session.serializer值指定序列化方式。

20. 增加使用Redis存储 session的功能，配置说明：

在apusic.conf中修改SessionStoreService服务实现，如：

```
<SERVICE CLASS="com.apusic.web.session.RedisSessionStoreService">
    <ATTRIBUTE NAME="ConfigPath" VALUE="config/redis_session.properties" />
    <ATTRIBUTE NAME="ConnectionMode" VALUE="single" />
</SERVICE>
```

1.2.3 新增及增强功能

- 接入的IO类型配置参数com.apusic.net.bio=false 现已废弃不用，改用新的方式：apusic.web.io.type=nio，取值为 bio,nio，方便后续的方式扩展。
- 修改完善相关jar包的包名，避免和应用引用的包冲突。
- 日志队列存储改为异步保持，避免阻塞，默认队列长度为2048，可以通过启动脚本增加jvm参数修改队列长度，如-Dapusic.log.blocking.queueSize=10240。
- AAS属性参数去掉apusic.disable.webservice属性，增加apusic.webservice.enabled属性，该属性的默认值为false。
- apusic.log.blocking.queue属性的默认值为false。
- 增加AAS属性参数com.apusic.errorPage.hideServerInfo，设置为true时，在现实错误页面信息时，不包含应用服务的版本等信息，默认值为false。
- 原来在Muxer中配置的EnableGZip属性被废除了，新的配置在WebService服务中，如：

```
<ATTRIBUTE NAME="EnableGZip" VALUE="true" />
<ATTRIBUTE NAME="CompressableMimeTypes" VALUE="text/html,text/xml,text/css"
/>
```

EnableGZip:可选配置，默认值为"false", 合法取值有三种："true", "false",或"2048"。最后一种形式是一个数值，表示一个最小长度的字节数，只有当输出内容大于该长度时，才执行压缩。如果配置为"true"，则最小长度默认值为2048个字节。

CompressableMimeTypes:可选配置，默认值为"text/html,text/xml,text/plain,text/css"，其取值为逗号分隔的字符串，表示这些类型的内容才会被压缩。

- 在应用中配置了url-pattern为/的Servlet为应用默认的Servlet，请求资源匹配没有成功的时候，则使用该servlet进行处理。
- 更新bcprov的实现，支持JDK1.5到JDK1.7，同时避免出现安全漏洞。
- 默认用户密码策略修改，主要修改为在应用服务器启动时设置密码，并增加了密码复杂度的检查，不符合复杂度的密码不能设置。

- 如果线程池maxSpareThreads未设置而maxThreads已设置，则把maxSpareThreadsSet设为与maxThreads相等。
- 添加对HttpSessionIdListener的支持。
- 根据规范将rtexprvalue默认值设置为true。
- 增加对类分析的异常捕捉，因为可能有一些第三方的类引用其他的第三方类，客户没有直接使用到，应用没有包含进来，会出现类找不到，导致应用启动失败。
- 添加应用初始化参数com.apusic.cdi.disabled是否启用CDI的参数设置。
- 增加vmoption参数apusic.nio.write.buffer.size，默认为0，用于提高低并发时候应用服务器的性能。
- 修改了只有admin用户进行jmx链接的功能，现在属于administrators组的用户都可以使用进行jmx连接。如果都不是admin和administrators组的用户，则不能进行连接。
- 增加memcache客户端超时控制参数以及链接失效恢复检查的时间间隔。配置方法：在CacheSessionStoreService服务中配置OpTimeout、HealSessionInterval、ConnectTimeout属性，单位为秒。
- 修改注入方法查找，兼容用户在不规范的setter方法上注入的问题，比如setxyz()中属性第一个字母x没有大写的问题。
- 更改多个应用进行session共享时，必须配置apusic-application.xml中context-root的问题，现在修改为：该配置不是必须，如果没有配置，则使用server.xml配置的name属性。
- 连接池泄露检测和强制关闭实现。

增强了JDBCTracerService的功能，可以对应用中的连接使用情况进行跟踪，超过时间没有释放则会进行提示；同时也提供一种强制关闭的方式，避免连接耗尽，影响应用的正常运行。

```
<SERVICE CLASS="com.apusic.jdbc.trace.JDBCTracerService">
    <ATTRIBUTE NAME="TraceAllow" VALUE="False" />
    <ATTRIBUTE NAME="StackTraceAllow" VALUE="False" />
    <ATTRIBUTE NAME="checkAbandoned" VALUE="False" />
    <ATTRIBUTE NAME="removeAbandoned" VALUE="False" />
    <ATTRIBUTE NAME="checkAbandonedTimeout" VALUE="300" />
</SERVICE>
```

新增参数说明：

checkAbandoned：是否开启检测连接池泄露，值设置为true则表示开启，其默认值为false，而检测的间隔时间和checkAbandonedTimeout属性的时间一致；

checkAbandonedTimeout：连接使用时间超过该值则认为该连接为泄露，单位为秒，默认值为300；

removeAbandoned 是否强制关闭泄露的连接。如果设置为true，则会自动关闭检测为超时的数据库连接；如果设置为false，则会在后台打印警告信息，默认值为false；

- 修改对HTTP连接是否为keepalive的逻辑。若连接是基于http1.1协议，则只要头属性connection没有指定为close,则表示keepalive；若是http1.0及以下版本，则必须指定为keep-alive,才表示开启keepalive；
- 将Muxer服务的属性lookAheadTimeout默认值修改为与timeout属性一致，且默认值修改为30秒；
- 应用类加载时缓存类文件与jar文件的修改时间，优化类加载的性能；
- jsessionId的cookie默认设置为httpOnly
- 增加非http协议对应用服务器的访问控制，如通过jconsole连接的控制。如果开启控制，则需要设置允许访问的ip列表。在Muxer服务中增加如下的属性配置：

```
<ATTRIBUTE NAME="EnableMuxAllow" VALUE="true" />
<ATTRIBUTE NAME="MuxAllowIPs" VALUE="192.168.106.205" />
```

EnableMuxAllow：表示是否开启这个功能，true表示开启，默认值为false；

MuxAllowIPs：设置允许访问的ip地址列表，多个ip以逗号分割，默认本机可以访问。

- 增加在数据源连接释放时自动提交或回滚的处理，避免一些应用没有正常释放连接导致的连接溢出问题。当数据源配置属性resetAutoCommit为true，同时配置了vm.options文件中参数jdbc.tx.auto.complete.decision的值（值可以为commit或rollback），则会按配置的值执行该处理；
- 除了可以通过管理控制台修改用户密码方式外，增加通过脚本修改用户密码的功能，详细参考脚本securityadmin的使用说明；
- 在启动HTTPS协议时，默认过滤不安全的套件。如果需要修改或添加更多的过滤套件，可以通过修改vm.options文件中参数com.apusic.ciphersuite.exclude的值；
- 日志打印时增加线程号信息的打印；
- 修改事务完成时释放逻辑连接的逻辑，先断开逻辑连接与物理连接，当逻辑连接被再次使用时，逻辑连接再自动关联物理连接；
- 在基于HTTPS访问应用时，默认情况下所有的cookie都是secure的。可以通过配置vm.options文件中参数com.apusic.cookie.noSecurity的值改变这个默认的行为，该参数设置为true，则表示cookie不要设置secure；

在开启情况下，可以通过配置vm.options文件中参数com.apusic.cookie.securityExclude指定哪些名称的cookie不需要设置secure；

- 可以通过统一的参数设置所有的cookie属性httpOnly值true，可以通过配置vm.options文件中参数com.apusic.cookie.httpOnly，如设置为true，则所有的cookie都会是httpOnly；
- 可以配置禁用HTTP协议访问功能，修改Muxer服务，增加如下属性配置：

```
<ATTRIBUTE NAME="DisableHttp" VALUE="True" />
```

- 在动态集群管理系统中，增加了证书管理、多版本部署、定时任务管理以及补丁管理等功能；
- 去掉自带的并发包的实现代码，统一使用JDK默认并发包代码；
- 增强了SessionService服务对于内存中Session数目超过MaxSessionsInCache时的管理。

  删掉了服务原来的ExpireOldestUnusedSessionOnMax参数，增加了SessionCreationPolicyOnMax及BackgroundPolicyOfOverrideOnMax参数。

  SessionCreationPolicyOnMax属性说明：

  表示当内存中的活动session数目大于MaxSessionsInCache所设定的值时，应用服务器在创建新session时采取何种策略，可选的值为：Override、NotAllow、ExpireOld及SwapOutOld，默认值为SwapOutOld。各取值含义说明如下：

  - Override:忽略最大限制，继续创建新Session，由后台线程负责清除多余的session，清除多余session的策略，由辅助参数BackgroundPolicyOfOverrideOnMax指定；
  - NotAllow:不允许超过最大限制，如超过，直接抛异常；
  - ExpireOld:如超过最大限制，先废弃掉多余的旧session，再创建新的；
  - SwapOutOld:如超过最大限制，先把内存中多余的旧session交换到持久存储SessionStore中，再创建新的。

BackgroundPolicyOfOverrideOnMax属性说明：

该属性为SessionCreationPolicyOnMax的辅助属性，当SessionCreationPolicyOnMax的取值为Override时，需要设置该属性。其可选的取值为：ExpireOld及SwapOutOld，默认值为SwapOutOld，取值的含义同SessionCreationPolicyOnMax中的解释。

1.2.4 已解决问题（包含Bug修复）

- 修改contenttype实现，兼容text/html;application/xhtml+xml;charset=UTF-8这样的写法。
- 修改ThreadLocalCleaner 实现，避免在JDK1.6时清除线程缓存数据出现锁的问题。
- 修复部分登陆时在从request获取principal出现空指针的问题。
- 修改使用Apache作为前置机的时候，访问资源多了一个/的问题。这个问题在授权认证时导致和单独访问AAS的结果不一致。
- 修改访问的URL存在分号时的处理，主要和tomcat保持一致，如index.jsp;a=b,index.jsp;a=/b等场景。
- 完善PersistenceUnit没有完善的方法。
- 修复注解扫描到的mdb加入destination的数组中，否则在apusic-applications.xml配置映射时检查通不过。
- 修复在webmodule获取persitenceUnit失败时，在app再进行查找一下。
- 修改el实现，兼容对这样的el表达式：${ob1.equals("test")?ob2:ob3}的解析。
- 改变启动文件内存设置位置，避免部分启动方式时不生效的问题。
- 修复在jdk1.8之后java.util.logging.LogManager中的成员manager修饰符为private static final的问题。

- 兼容有些应用的web.xml 版本是2.5，而在配置的时候使用了3.0的标记，如异步标记等。
- 解决部署tuxedo的适配器时，出现找不到EJB Remote接口weblogic.wtc.jatmi.TuxedoService的错误。
- 修复一个应用发生错误后退出AAS的问题。
- 修复在web.xml中配置resource-env-ref cdi beanManager时找不到对象的问题。
- 禁止不安全的TLS协议密码重新协商功能，解决HTTP SSL DOS攻击。
- 修复在Acunetix Web Vulnerability Scanner扫描下出现Host header attack漏洞的问题，主要出现于使用认证登陆，因为发送过来的host头是不受信任的，重定向到登陆页面的response.sendRedirect()方法会使用恶意的host头信息，通过aas参数com.apusic.defaultServerName可以设置默认受信任的host。
- 管理控制台的更新：增加用户修改密码等操作的友好提示，增加rar jca资源的监控功能。
- 修复使用spring框架时，访问不存在的资源可能会出现循环查找，导致线程堆栈不足的问题。
- 修复JSP文件更新为旧文件后需要重启2次AAS才能生效的问题。
- 修复在修改卸载应用脚本失败和创建目录错误的问题。
- 修改keepalive策略在keepaliveTimout时间内selecor管辖的任何连接完全不做任何读写才清理连接，只要某个用户时不时的访问一下服务器，就会导致其他应该被清除的连接也得不到被清除的机会，现在改正之，每个连接开始idle时针对这个连接单独计时，超过了keepalive的时间这个连接就会被清除。
- 修复访问JSP页面时如果URL包含jsessionid时显示源码的问题。
- 修复修改JSP页面后需要重启两次应用服务器才能看到修改效果的问题。
- 修复客户直接调用sendError(int,string)方法导致跨站点脚本错误的问题。
- 修复未设置机器名称时启动应用服务器导致获取机器名出现空指针的问题。
- 修复事务超时时抛出反映超时的异常的问题，比如：在执行statement.executeUpdate方法时抛出java.sql.SQLTimeoutException。如果超时，AAS会自动回滚事务，但不应该重复执行回滚动作。
- 修复在contentType为;charset=utf8这样的格式下抛移除的问题，兼容tomcat。
- 修复filter复制时没有对支持异步的参数复制的问题。
- 修复跨站点脚本攻击和跨站点脚本编制的问题。
- 修复session定置的问题。
- 修复gson版本冲突导致的数据显示问题。
- 修复在多版本运行时，设置session的cookie-name不起作用的问题。
- 修复在少数环境下会出现注销失败的问题。
- 修复jacc版本导致的不需要限制的方法进行验证。

- 兼容在jdk1.8之后java.util.logging.LogManager中的成员manager修饰符为private static final的问题。
- 修复在nio接入下的连接计数方法可能存在重复减1的问题，这个问题会导致链接数计算不正确。
- 修复在nio接入下keepalive不生效的问题。
- 修改apusic.conf中WebService服务属性MaxKeepAliveConnections可以设置为-1，表示keepalive的链接数不受限制。
- 修复在一些使用spring框架的应用下，欢迎页面访问不了的问题。
- 修复jndi名称不能以/开头的问题，这个问题可能会导致使用以/开头的Jndi名称的开源框架出现问题。
- 修复jvm启动一个agent初始化logManager，导致AAS控制台不打日志的问题。
- 修复在jdk7_51以上版本启动应用服务器时，出现corba异常的问题。
- 修复在一些情况下，使用jms会出现cpu过高的问题。
- 修复错误匹配"/xxx/main_W.jsp/xxxx.upm?aaaa=1"之类URL的问题，在这个URL中，以前版本会匹配到jsp Servlet来进行处理，而不是应用配置的upm匹配的servlet处理。
- 修复EL找不到类中的静态方法问题。
- 修复在使用cache进行session存储时，应用session共享可能出现误删除session的问题。
- 修复Forwward路径有"("时出现java.util.regex.PatternSyntaxException: Unclosed group near index错误的问题。
- 修复在解析JSP document 及Tag files 时，没有初始化FunctionMapper的问题，会导致如SpringIntegration这样的应用部署时出现错误。
- 修复在某些情况下，会出现"java.lang.IllegalStateException: Session no longer valid"的问题。
- 修复在某些情况下，会在WebModule.mergeAdditively中出现java.lang.NullPointerException问题。
- 修复在部分龙芯环境下启动，会出现RuntimeOperationsException: Exception occurred trying to register the MBean 异常问题。
- 修复在部分龙芯环境下启动，会出现java.lang.IllegalStateException: The reader is not positioned at attributes 异常问题。
- 修复在bin目录下执行config命令新建域失败的问题。
- 修复在英语环境下，使用config命令创建域时，在选择模板类型没有序号的问题。
- 修复用户在应用中调用了LogManager.readConfiguration(InputStream)方法后日志无法打印的问题。
- 修复在应用重新加载时关闭了其他缓存包，导致其他应用访问时出现打不开jar包等错误。
- 解决在管理控制台修改数据库连接池参数时报空指针错误；
- 解决websocket在一些特殊场景下出现循环调用的问题;
- 修复访问管理管理控制台时，会随机出现ArrayIndexOutOfBoundsException异常问题；

- Muxer服务中的WaitingClientTimeout属性值单位为秒，而线程池把其作为毫秒进行使用，会导致在线程池满时，请求很快就被抛弃；
- 修复com.apusic.elite.fnLength参数的笔误，该参数用于指定功能函数的最大字符长度，在一些三元表达式解析时需要使用，默认值为30；
- 解决在安全测试https中，出现使用弱安全套件的问题；
- 修复EL解析对象为JSON字符时和其他服务器不一致的问题;
- 修复当频繁请求较大的文件时，可能会导致内存溢出和不断触发垃圾回收的问题；
- 修复Session删除和同步时出现的锁同步问题，避免在大并发时出现性能下降；
- 修复数据库连接池当连接数等于最大值时，设置等待时间大于0可能导致陷入无限等待问题；
- 修复集群包中带有日志代码的问题；
- 修复应用的名称带有反斜杠时，sessionstore创建文件失败问题；
- 修复应用创建动态代理类后，重新加载时却出现异常的问题；
- 修复防攻击过滤器在处理特殊字符时出错的问题;
- 修复访问应用根目录后显示欢迎页面时，没有经过过滤器处理的问题；
- 修改Session序列化内容各个属性的分割方式，避免内容中包含分割字符导致异常；
- 修复在bio模式下，重复减连接数，导致计数不正确的问题；
- 解决在形成URL时，会把空格编码成%20，然后在把URL转换成File时，没有相应decode，造成含有%20的文件实际上指向另一个文件的问题。
- 解决jsp:attribute属性omit的值不能是el表达式的问题

2 部分II 技术概览

2.1 概述

2006年5月，新一代Java企业级平台开发规范JavaEE5正式通过了JCP（Java Community Process）的批准。Java EE 5规范是Java平台在企业级应用上的一次重大的升级，犹如Java EE 5提倡的口号：Do more with less work！Java EE 5的推出，标志着Java EE 平台开始朝着轻量级，快速的方向发展。

JavaEE5为我们带来了以下的新特性：

- 简化的配置

这标志着Java EE 开发更加简便，快捷。

- 全新的，简化的EJB模型

新的EJB 3规范，使得在保持了EJB原有强大功能的同时，EJB的开发也变得更加简单。

- 新的WEB组件开发模型

新加入的JSF规范，使得开发表现丰富的WEB层更为简单快捷。

- 更完善的Web Service支持

新的规范使得开发Web Service程序更加简单，对标准的支持也更为完善。

金蝶Apusic应用服务器不仅允许客户使用所有JavaEE5,JavaEE6规范的新技术特性，更提供了一组金蝶Apusic应用服务器特有的优势技术，包括：

- 完善的集群支持

金蝶Apusic应用服务器提供了完善的集群解决方案，包括对Web、JNDI、EJB、JMS进行集群。Apusic集群提供了创新性的设计和实现，采用了Client Session Cache(客户端会话缓存)等独创技术，以保障企业应用的高可用性及水平可扩展性。

- Apusic Domain支持

金蝶Apusic应用服务器提供的Apusic域概念，使得应用服务器的管理和部署更加简便和灵活。

- 对第三方Http服务器的集成

金蝶Apusic应用服务器提供了一个自己的Http服务器实现，但也允许用户集成使用其它的Http服务器，以获取增加的功能。例如集成Apache Http Server和IIS。

- WebDav

金蝶Apusic应用服务器提供了符合RFC2518的WebDav实现，这使得对大型Web站点的内容管理工作变得更为简单。

- 可加密混淆的JSPC工具

使用JSPC工具发布应用时，只需发布编译好的JSP class，而不需要提供JSP源程序文件，从而避免暴露JSP源代码。JSPC也省略了应用运行时对JSP的编译过程，从而节省了首次编译时间。此外，Apusic JSPC工具还可以与金蝶中间件提供的Java编译混淆器"Jocky"联合使用，使得JSPC编译出来的class文件不可反编译，从而有效保护客户知识产权。

- 开发工具的良好支持

Apusic Studio集成开发环境基于Eclipse平台技术，为金蝶Apusic应用服务器提供了良好的支持。使用Apusic Studio，可快速的构建基于Java EE 5的应用，并能将应用方便的部署到金蝶Apusic应用服务器中。

- 支持多个Selector的多路复用服务

金蝶Apusic应用服务器在NIO的方式下，能够使用多个Selector来服务用户的请求，从而在多CPU的环境下提升性能。

- 可扩展的安全框架

金蝶Apusic应用服务器提供了一个灵活的、可扩展的安全框架。将Java EE 的安全认证与LDAP，数据库或其它身份验证服务器集成，提供认证和授权服务。并与Apusic SSO紧密集成。

- 高度安全性

国外厂商在诸如安全算法、加密位数等核心技术上是有严格的海外出口限制的，这对我国的信息化安全是一种严重的挑战。金蝶Apusic应用服务器完整实现了Java安全框架，包括Java认证和授权服务（JAAS）、Java 安全套接字扩展（JSSE） 和Java 加密扩展（Java Cryptography Extension，JCE），加密位数不再受任何限制，从而更好的保护我国企业、政府的信息安全。

- 远程监控和管理工具

金蝶Apusic应用服务器提供了一组远程监控和管理工具，可以监控应用服务器内存、线程、类装载等状态。

- 可靠、高效的JMS实现

金蝶Apusic应用服务器提供了可靠、高效的JMS实现，构建于非常成熟的，经过实践验证的Apusic的MQ中间件之上，并提供了标准之外的增强功能。

金蝶Apusic应用服务器为复杂应用提供了一个简便、快速的开发和运行平台，对于分布式的企业级应用，提供了易扩展、可伸缩和高安全性等特性。本文将介绍金蝶Apusic应用服务器的体系结构及其提供的服务和功能，从而展示金蝶Apusic应用服务器对开发大型应用系统的支持能力。

2.2 体系结构

金蝶Apusic应用服务器是一个标准兼容的应用服务器，构建于特有的RTTC微内核技术之上。下图展示了金蝶Apusic应用服务器的体系结构。

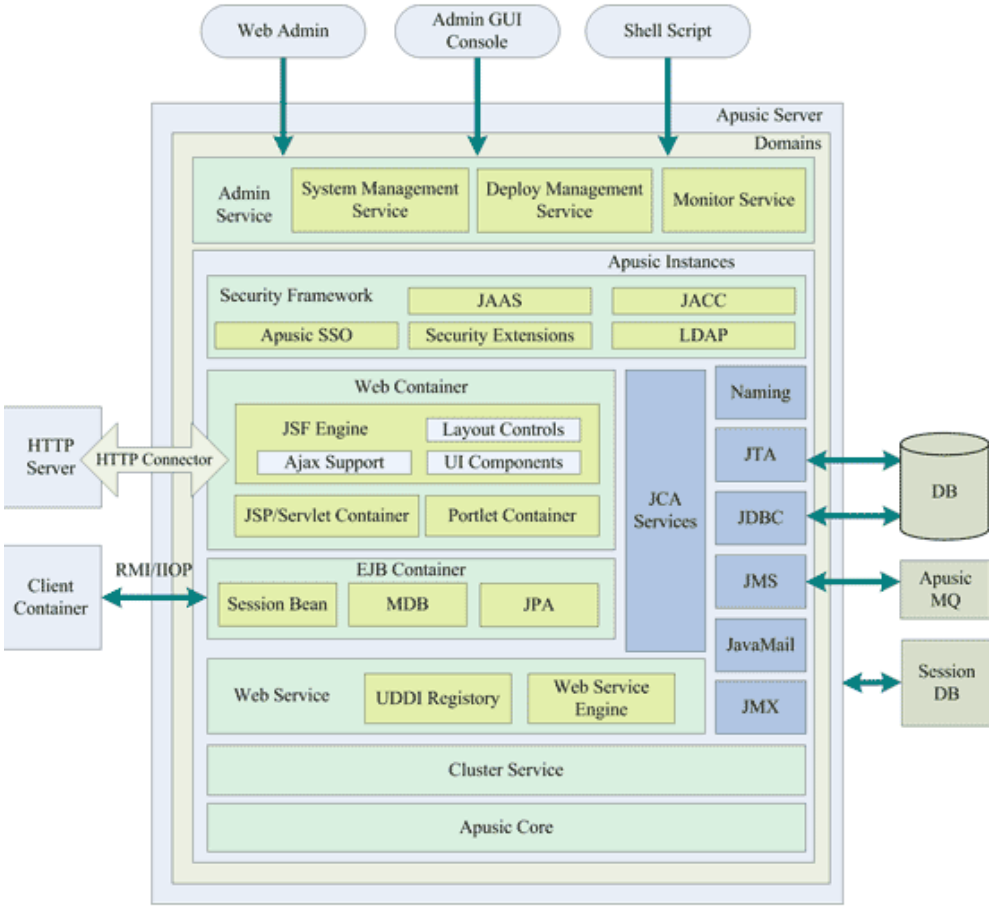


图 金蝶Apusic应用服务器的体系结构图

2.2.1 Apusic服务器、Apusic域和Apusic实例

要理解金蝶Apusic应用服务器如何进行部署和管理，需要理解Apusic服务器、Apusic域和Apusic实例的概念。

- Apusic服务器（Apusic Server）

Apusic Server是应用服务器的物理部署单元。直观的来说，就是金蝶Apusic应用服务器在一台用户机器上的一个物理安装。

- Apusic域（Apusic Domain）

Apusic域是一个逻辑的管理配置单元。一个Apusic域包含着一组配置，这些配置可以被多个Apusic实例（Apusic Instance）共享。每个Apusic域都有自己的一个或多个Apusic实例。Apusic域可以将自己的Apusic实例委托给其它Apusic域管理，也可以接受其它Apusic域委托，管理委托者的Apusic实例。

- Apusic实例（Apusic Instance）

Apusic实例是一组运行的进程，为用户提供应用服务器的各种服务。Apusic实例受到Apusic域的管理并使用它的配置。

通过Apusic域，部署在不同物理位置上的Apusic实例，可以共享一套公用配置。同时，可以在单一节点上，通过一个Apusic域统一管理多个Apusic实例。这使得金蝶Apusic应用服务器的部署和管理变得简单和灵活。

2.2.2 Apusic体系结构组成

金蝶Apusic应用服务器的体系结构分成7个部分：

- Apusic Core内核
- 容器
- 基础服务
- Web Services
- 安全框架
- 管理和监控
- 集群服务

接下来的章节将对这7部分内容分别进行介绍。

2.3 Apusic Core内核

在金蝶Apusic应用服务器中，各项基础服务以可插拔的方式加入并提供服务，而Apusic Core则为这些基础服务解决资源、线程、事务、通讯等关键领域的问题。

- 资源

Apusic Core对以下资源进行管理：JDBC Resources、Mail、JCA Resources、JMS、URL、Resource Environment。

- 线程

Apusic Core提供高效稳定的线程池服务，对线程进行管理，维护线程的生命周期，处理并发请求。

- 事务

Apusic Core的事务管理器可进行全局事务管理。可管理的事务资源包括三种，数据库连接池、消息连接和符合Java EE™连接器架构（Java EE Connector Architecture，JCA）的资源。

- 通讯

Apusic Core负责处理Apusic应用服务器与客户端的通讯，支持HTTP、HTTPS、RMI / IIOP等通信协议。

2.4 容器

Java EE 容器为Java EE 应用提供了运行的基础环境。

金蝶Apusic应用服务器包括三种类型的容器，Web Container、EJB Container和Client Container。

2.4.1 Web Container

金蝶Apusic应用服务器提供了一个全功能的Web容器，用于处理客户端发出的静态和动态WEB内容请求。Web容器首先接收客户端发来的Http请求，对于静态的Http内容请求，由Http静态内容引擎负责处理。对于JSP/Servlet和其它类型的动态内容请求，转发给JSP/Servlet容器进行处理。

2.4.1.1 JSP/Servlet容器

JSP/Servlet容器负责解析JSP页面，以及执行和管理Servlet组件。金蝶Apusic应用服务器兼容标准，实现了对Servlet 2.5和JSP 2.1的支持。

2.4.1.2 JSF引擎

Apusic JSF引擎包括以下的特性：

- 独立的JSF引擎

不依赖于应用服务器，只需要将单独的jar包置于应用服务器类路径中即可使用。

- 容器级别的AJAX支持

在设计时，充分考虑对AJAX的支持，无需任何配置即可实现AJAX效果，开发扩展AJAX组件也更加容易。

- 简化的ManagedBean管理

使用JDK1.5的annotation，在类上标注@ManagedBean即可将一个POJO定义为ManagedBean，省去了维护faces-config.xml的烦恼。

- 扩展的导航机制

扩展了标准的JSF导航机制，除了允许使用导航配置规则中的view-id进行导航外，Apusic JSF也允许直接使用页面地址导航。

- 增强的布局和模板组件

Apusic JSF提供的布局和模板组件，提供了强大的页面布局管理能力。

- 扩展的富客户端组件

Apusic JSF提供了一组扩展的富客户端组件，包括DateField，TabBox，Menu，Tree，DataGrid等。

- 统一的资源和皮肤管理

Apusic JSF提供了统一的资源和皮肤管理机制，具有良好的扩展性。使用者可以根据应用需求制作自己的界面皮肤，将制作好的皮肤打包成jar放在应用中即可。

Apusic JSF的开源版本被称作OperaMasks JSF，关于OperaMasks JSF的更多信息，请访问www.operamasks.org网站。

2.4.1.3 Portlet容器

Portlet是一种Web组件，为Portal页面服务。当一个Portal页面被访问时，通常会引发多个Portlet被调用。这些Portlet执行后生成的标记段组合在一起，被嵌入到Portal页面中。

金蝶Apusic应用服务器对Web Container进行了扩展，提供了实现了JSR-168规范的Portlet容器。

2.4.1.4 虚拟主机

虚拟主机是指能在单机上模拟多个主机的服务能力。在Apusic服务器中，我们能够将指定的某些J2EE应用与虚拟主机关联起来。当用户对虚拟主机发出的请求，实际上是对该J2EE应用的请求，同时该虚拟主机的资源无法通过其他的方式进行访问。从而有效实现了在共享硬件与软件资源的情况下，模拟多个主机服务用户请求的效果。

2.4.1.5 Http Connector

虽然金蝶Apusic应用服务器自己可以处理静态Http内容请求，但在很多情况下，用户会选择使用其它的Http Server来处理静态Http内容请求。金蝶Apusic应用服务器提供了一组Http Connector，可以很容易的将其它的Http Server产品，如Apache Http Server和IIS

等，集成到金蝶Apusic应用服务器应用服务器中。

2.4.1.6 WebDav

WebDav（Web-based Distributed Authoring and Versioning）是一组Http协议的扩展，允许你从远程访问和修改Web站点上的内容。金蝶Apusic应用服务器提供了自己的WebDav实现，这使得对大型Web站点的内容管理工作变得更为简单。

2.4.2 EJB Container

EJB Container为EJB提供部署和管理需要的所有运行时服务。Apusic 应用服务器支持符合EJB1.X、EJB2.x和EJB3.0规范的EJB，包括Session Bean、Entity Bean、Message Driven Bean，以及EJB3.0中的JPA。EJB Container为这些EJB提供对象池、多线程、分布式、安全控制、事务支持和生命周期管理等底层服务，管理JPA、CMP的数据存储和提取。

2.4.3 Client Container

Client Container是由一组Java类和XML部署描述符组成，它同客户端应用一起运行在客户端的Java虚拟机中，管理应用客户端组件的执行。像其他Java EE 应用组件一样，应用客户端的执行依赖于客户端容器提供的系统服务。客户端容器和金蝶Apusic应用服务器通讯使用RMI/IIOP。和其他服务器端的Java EE 容器相比，客户端容器可以说是相对简单的容器。

客户端容器提供的系统服务有：

- 创建客户端运行环境，负责和应用服务器进行通讯。
- 提供JNDI包装，使客户端能够使用java:comp/env名字空间。
- 认证客户端。应用客户端容器自动完成JAAS用户认证。

2.4.4 容器Session管理

2.4.4.1 Http Session

绝大部分web应用都使用session来存储用户相关的会话信息，金蝶Apusic应用服务器为Session提供了一系列管理手段。

2.4.4.1.1 IN-MEMORY CACHE

金蝶Apusic应用服务器将Session缓存在内存中，以保证Session访问的效率。

2.4.4.1.2 存储

当Session数量超过默认缓存大小，金蝶Apusic应用服务器会将内存中的Session持久化到存储介质中，并根据Session的活跃性对存储中的Session和缓存中的Session进行交换。

管理员可配置Session缓存池的大小。金蝶Apusic应用服务器支持的Session持久化包括文件系统、RDBMS、BerkerlyDB，管理员可通过管理工具切换Session的存储方式。

如果Session中的数据非常重要，即使服务器失效，这些数据也不能丢失，那么建议采用数据库来持久化Session。

2.4.4.1.3 SESSION FAILOVER

在集群环境中，当一个Apusic实例（Apusic Instance）失效时，它原来服务的所有用户的请求，将会由集群内的另一个Apusic实例响应。这个过程对用户来说要求是透明的，这意味着用户的登陆信息以及其它会话数据在此情况下不能丢失。由于web应用通常用HTTP Session来保存会话信息，因此，Apusic提供内存复制、Session迁移、数据库存储等多种技术来保证HTTP Session的失败恢复。

内存复制： Apusic集群中的任意Apusic实例可通过内部的复制技术，在网络中复制集群内另一个Apusic实例的Session信息，从而在该实例内存中形成另一个Apusic实例的Session备份。当某apusic实例失效时，另一个Apusic实例会接收此实例原来服务的请求，整个过程对用户是透明的，用户感觉不到原先对他进行服务的Apusic实例已经失效。

Session迁移： Session迁移是指在集群中使用Apusic负载均衡器来转发请求时，所采用的Session Failover技术。负载均衡器负责将失效的Apusic实例中的Session迁移到另一个Apusic实例中。

数据库存储： 如果集群中的Apusic实例不使用In-memory Cache缓存Session，而使用数据库作为Session的存储中心。这时，由于Session不在Apusic实例中存储，因此，Apusic实例失效不会影响Session的使用。

2.4.4.1.4 SESSION STICK

Apusic集群中的负载均衡器，比如Apache或Apusic Load Balancer，在转发用户请求到集群内的Apusic实例中时，总是将一个用户的请求固定的转发到第一次响应他的Apusic实例中，这就是Apusic集群的Session Stick。这可以避免频繁的Session迁移，减小网络和服务器的负担。

2.4.4.1.5 SESSION TIMEOUT

金蝶Apusic应用服务器可以对部署在其上的Web应用的Session超时时间进行全局的设置，也可以对每个应用的Session超时时间进行单独的设置。

2.4.4.2 Stateful Session Bean

在单个Apusic实例时，Stateful Session Bean的Session状态保存在服务器端的Stateful Session Bean对象实例中。而在Apusic集群中，采用创新的CSC（Client Session Cache）技术保存Stateful Session Bean的状态。CSC直接将Session状态保存在客户端，当服务器失效时将Session状态转移到可用的服务器上。Stateful Session Bean的特点决定了CSC技术是有效的，根据EJB规范，一个Stateful Session Bean仅限于单个客户使用，不存在共享的情况，因此只需要在客户和服务器之间共享Session状态，而不需要在服务器之间共享Session状态。CSC避免了Session状态发生改变时，在集群节点之间频繁的Session复制，提升了集群的性能。

2.5 基础服务

2.5.1 JCA Services

Java EE连接器架构是为了和传统的应用系统进行连接定义了一套标准的架构。它定义了一套可升级的，安全的，能基于事务的机制，以便Java EE平台可以和传统的应用系统进行整合。Java EE连接器架构还定义了公用客户端接口，使其它平台的应用能访问Java EE平台。对于不同的应用，金蝶Apusic应用服务器需要其提供资源适配器，金蝶Apusic应用服务器能将符合规范的资源适配器插入应用服务器，以达到应用服务器访问异构系统的目的。

金蝶Apusic应用服务器目前提供了 JCA1.5的完整实现。

2.5.2 Naming

由于JavaEE应用的关键特性是分布式的，其提供的服务可能分布在任何一个机器或网络，应用的开发统一通过JNDI（Java Naming and Directory Interface）来获得服务，开发者在开发时可以不用关心服务的物理位置。

Apusic 完全支持使用JNDI1.2，并且开发完成后，部署者(Deployer)可以在部署工具中配置应用所需要的环境、资源、服务，使应用获得所需的分布式服务，完成最终运行环境的建立。

2.5.3 JTA

JavaEE中对于事务的处理是分布式的，金蝶Apusic应用服务器完全提供对XA事务的支持。为了简化开发者对事务的处理，金蝶Apusic应用服务器在EJB容器中提供容器管理事务（Container Managed Transaction），应用开发者使用EJB管理数据即自动获得了事务的保证，由于EJB容器对数据库的管理是分布式的，开发者由此自动获得了对应用开发较为困难的分布事务管理能力，这一切都由应用服务器完成。

为保证应用数据和业务逻辑的完整性，除了提供完全符合规范和强壮的事务管理器之外，金蝶Apusic应用服务器还提供了强壮而灵活的事务构架。JTA（Java事务API）对数据的完整性起到关键作用，Apusic支持的事务包括EJB,JMS,JCA,JDBC等。支持分布式事务，两阶段提交.

2.5.4 死锁检测

金蝶Apusic应用服务器能够根据资源等待图自动检测出死锁状态，当事务发生死锁时将其中一个事务回滚，以释放事务所占用的资源，使其他事务能够继续执行。

2.5.5 并发控制

金蝶Apusic应用服务器提供了灵活和强壮的EJB并发控制机制，单独分离出了并发控制部分，可以使用插件的形式同时提供多种并发控制协议。ejb存在大量并发访问的时候，出于事务的一致性和处理性能的考虑，需要进行控制，AAS应用服务器默认支持乐观锁和悲观锁两种方式。悲观锁时，假定一个线程修改数据时，其他线程修改数据的可能性很大，所以在整个数据处理过程中，数据都处于锁定状态，直到提交了所做的更改，再释放锁。使用悲观锁时事务的一致性高，但是它的性能较低。乐观锁时，假定在一个线程修改数据时，其他线程修改数据的可能性不大，所以在准备提交数据的时候才对数据进行锁定，由于锁定的时间短，所以具有比悲观锁更好的并发性。

除了悲观锁和乐观锁，AAS应用服务器还允许通过插件的形式提供锁策略。通过配置后就可以作为应用服务器的锁策略进行并发的控制。

2.5.6 JDBC

Apusic应用服务器模型中的数据层提供企业应用对关系型数据库和传统企业应用数据的高效而可靠的访问。使得企业应用可以通过JDBC实现对关系型数据库安全、可靠而且高效的访问。目前Apusic应用服务器提供了对最新的JDBC4.0 版本的支持。任意合法的Java EE编程模型都可以从Apusic应用服务器数据库连接池中获得对关系型数据库的连接。金蝶Apusic应用服务器提供了很多功能或特性，使对数据层访问的开发更为简单，同时达到更为高效和安全的目标。

JDBC 结果集缓存

对于Session Bean、JSP、Servlets为主的Java EE应用中，频繁访问数据库而缺少对应的数据缓存等机制，往往会成为企业应用的瓶颈。因此，金蝶Apusic应用服务器提供了JDBC结果集缓存技术，通过将数据库返回的结果集保存在内存中，可以大幅提高应用系统的性能，同时，使用结果集缓存对应用开发者是完全透明的，保证了应用的可移植特性。

2.5.7 JDBC 语句缓存

企业应用频繁对数据库的访问，往往会成为企业应用的瓶颈。因此，金蝶Apusic应用服务器还提供了JDBC语句缓存技术，同时使用预编译的查询语句，提高应用程序访问数据库的效率。

2.5.8 连接池

对于复杂的企业应用而言，对数据库连接池的可靠、效率，成本都有着极高的要求，Apusic应用服务器提供了对连接资源的优化，使有限的数据库连接资源得到最大程度的利用。

2.5.9 JMS

JAVA消息服务(JMS)定义了Java中访问消息中间件的接口。消息中间件提供数据的异步传输，通过消息中间件，一些原本互相孤立的业务组件可以组合成一个可靠的、灵活的系统。消息中间件分为两类：Point-to-Point(PTP)和Publish-Subscribe(Pub/Sub)。

PTP是点对点传输消息，建立在消息队列的基础上，每个客户端对应一个消息队列，客户端发送消息到对方的消息队列中，从自己的消息队列读取消息。

Pub/Sub是将消息定位到某个层次结构栏目的节点上，Pub/Sub通常是匿名的并能够动态发布消息，Pub/Sub必须保证某个节点的所有发布者(Publisher)发布的信息准确无误地发送到这个节点的所有消息订阅者(Subscriber)。

金蝶Apusic应用服务器支持以上两种模型，并完全实现了JMS1.1版本。

2.5.10 JavaMail

JavaMail是一个用于阅读，编写和发送电子消息的API，可以用来建立基于标准的电子邮件客户机，它支持各种因特网邮件协议，包括SMTP,POP,IMAP,MIME,NNTP,S/MIME及其他协议。

Apusic服务器支持所有邮件协议，并完全支持JavaMail 1.4版本。

2.5.11 JMX

JMX（Java Management Extensions，即Java管理扩展）是一个为应用程序、设备、系统等植入管理功能的框架。JMX可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活的开发无缝集成的系统、网络和服务管理应用。基于JMX架构的金蝶Apusic应用服务器具备高度可管理性。而服务器基于JMX的体系架构，也带来管理的便利性与可靠性。我们可以通过JMX管理获取服务器的运行现状，并能够通过微内核所提供的诊断数据了解服务器的健康水平。

2.6 Web Services

Web Services是一种自包含、模块化的应用，且能够在网络上发布、定位和调用。Web Services是一种发展中的用于实现服务驱动架构(SOA)的技术，其注意力主要集中于如何使应用功能模块能够通过标准的互联网协议进行互用，而与运行平台、编程语言等无关。

2.6.1 Web Services引擎

金蝶Apusic应用服务器提供了符合相关规范的Web Services服务支持。Apusic对Web Service的支持包括：

2.6.1.1 JAX-WS 规范

J2EE 1.4使用JAX-RPC作为Web Services的API规范，JavaEE5引入了Java API for XML-Based Web Services (JAX-WS)规范。这个规范是对JAX-RPC扩展，由一系列相关规范整合而成，包括JAX-WS 2.1、 JAXB 2.0、 SAAJ 1.3等等，用于取代原有的JAX-RPC规范。Apusic Web Services引擎完全支持JAX-WS规范，同时向下兼容JAX-RPC规范。

遵循JAX-WS规范，金蝶Apusic应用服务器的Web Services引擎具有以下特性：

- UDDI （Universal Discovery Description and Integration）

UDDI提供了在互联网上进行全球范围的业务注册并相互发现的开放式注册框架。

- JAXB 2.0

JAXB 2.0定义了Java与XML数据绑定规范，并提供了Java与XML间双向的自定义数据绑定规则。它提供了一种简便的方法，通过使用Java对象来处理XML文档。

- SOAP 1.2

SOAP定义了在HTTP协议上通过XML实现跨系统信息交换的轻量级协议。

- SAAJ（SOAP with Attachments API for Java）

SAAJ为Java平台提供了在互联网上发送XML文档的标准方法。

- 支持元数据注解 （Metadata annotation）

JAX-WS定义了Java语言注解(JSR 175)在Web Services中的应用，并支持Java平台Web Services注解(JSR 181)，使客户端与服务器端应用开发更为方便简洁。

- 支持WS-I Basic Profile 1.1

WS-I BP是由Web Services可互用性组织(Web Services Interoperability Organization)整理的一系列非其私有的规范集合，并对这些规范进行了阐明、修正、扩展等，用于提高Web Services的可互用性。

- 异步操作

加入了对客户端异步操作的支持。

- 加强了对非HTTP传输机制的支持

JAX-WS增强了XML消息与底层传输机制的分离，简化了非HTTP传输的使用。

- 基于消息的会话管理

JAX-RPC 1.1的会话（Session）管理依赖于HTTP协议。JAX-WS 2.0加入了对基于消息的会话管理的支持。

2.6.1.2 对规范的支持

金蝶Apusic应用服务器遵循Web Services相关规范（JSR109），允许建立基于Java EE 组件架构的Web Services服务架构，提供了一个可在多应用服务器间移植和互动的客户端与服务器端编程模型。这个模型提供了可扩展的安全环境，并为Java EE 开发人员所熟悉。

在客户端，允许服务客户通过JNDI查找来获取服务接口(Service Interface)或服务端点接口(Service Endpoint Interface)。服务客户还可使用WebServiceRef注解来注入服务或服务端点。金蝶Apusic应用服务器扮演中间代理的角色，保证JNDI查找在web容器、EJB容器以及客户端容器中顺利进行。

在服务器端，允许服务提供方使用无状态会话Bean来作为服务实现Bean（Service Implementation Bean），在实现服务逻辑过程中，可查找或请求容器注入上下文实例，访问容器所提供的各种资源与服务。金蝶Apusic应用服务器自动管理服务实现Bean的生存周期与实例池。

金蝶Apusic应用服务器还为所部署的Web Services提供基于相关规范的安全环境。包括Java EE 身份验证与授权模型。

2.6.1.3 金蝶Apusic应用服务器的消息安全机制

金蝶Apusic应用服务器采用OASIS的Web Services安全标准(WS-Security)，这是一套为可交互的Web Services安全性而订的国际标准，由加盟OASIS的主要网络服务技术提供商共同制定。

Web Services安全标准是一套使用XML加密技术与XML数字签名技术的消息安全机制，保证在SOAP协议上的消息安全性。消息安全机制将安全信息插入到消息中，伴随消息一起传输到发送目的地。消息安全机制独立于传输层安全机制，因此能保证在传输完成后，消息仍然是受保护的。

金蝶Apusic应用服务器遵循Web Services安全标准，支持多种安全标识(Token)，包括X.509认证，SAML断言，Username/Password标识等，来验证与加密SOAP上的Web Services消息。

2.6.1.4 UDDI

UDDI提供了一个基于XML，可以让全球范围的业务注册并相互发现的开放式注册框架，其最终目的是允许不同公司的业务系统能在全球互联网上相互发现，共享描述信息并协同工作，简化在线事务处理。金蝶Apusic应用服务器支持全部UDDI V3 API以及部分UDDI V1与UDDI V2 API。同时，金蝶Apusic应用服务器中整合了符合UDDI V3标准的UDDI注册服务。

2.7 安全框架

对于企业应用而言，应用服务器所能提供的安全方面的功能，是保证企业应用数据完整、逻辑完整和减少被入侵可能的重要特性。针对多层的分布式企业应用的安全要求，Apusic 应用服务器提供了可靠高效的安全构架。同时，为应用程序的加密需求提供了一个安全、强壮、高效的JCE(Java Cryptography Extension)提供者。

同时，金蝶Apusic应用服务器以一种可靠的安全框架实现对资源(Web资源，EJB资源，数据源资源)进行有效保护，金蝶Apusic应用服务器的安全框架提供了灵活，易于扩展的机制，并允许用户实现自己的安全认证服务。

2.7.1 JAVA授权和鉴定服务（JAAS）

JAVA授权和鉴定服务（JAAS）是基于用户的访问控制，即根据谁在运行代码来进行授权。JAAS认证被实现为可插入的方式，允许应用程序与具体认证技术保持独立，新增或者更新认证方法并不需要更改应用程序本身。

JAAS目前已经整合进了Java 2 SDK 1.4及以上版本，作为标准的用户认证与授权模型。金蝶Apusic应用服务器也提供了JAAS的实现。

2.7.2 Java EE 容器的授权合约（JACC）

Java EE 容器的授权合约（JACC），定义了J2EE容器和外部授权系统之间的合约，定义了外部授权提供者如何与J2EE 进行交互，以达到使用其它授权系统的目的。例如，J2EE和网络管理系统的结合等。

金蝶Apusic应用服务器提供了JACC1.0的完整实现。

2.7.3 Apusic Security Extensions

Apusic Security Extensions即金蝶Apusic应用服务器的安全扩展特性，Apusic服务器提供了一套方便、灵活的安装机制能够用于处理金蝶Apusic应用服务器本身的安全，同时也提供了与第三方安全验证产品整合的能力，如：LDAP、数据库系统、Sun Access Manager等，共同完成身份验证和授权工作。同时，金蝶Apusic应用服务器所提供的安全扩展功能，允许开发自己的安全提供程序，来完成与自己的特有系统进行安全整合。

2.7.4 Apusic SSO

SSO全称Single Sign On，即单点登录，是目前比较流行的企业业务整合的解决方案之一。SSO的定义是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。Apusic SSO提供了单点登录的解决方案，市场上所提供的单点登录产品有Yale CAS，OpenSSO，Sun Access Manager等，金蝶Apusic应用服务器能够有效地与这些产品整合。

2.7.5 身份鉴定

金蝶Apusic应用服务器提供了面向用户、调用过程和客户端调用等方式的身份鉴定，通过使用用户名和密码、证书等等方式，将合法的应用用户和调用程序与非法的入侵者区分开来。

2.7.6 授权

授权是使企业应用的完整性和安全性得以保证的重要因素，金蝶Apusic应用服务器对企业应用的授权策略提供了易于配置和修改的方法与工具，同时提供了可靠安全的授权机制。

2.7.7 加密

对于企业应用中的敏感数据的保存和交换，金蝶Apusic应用服务器提供了高效可靠的加密机制。Apusic安全服务提供了数字签名、消息摘要、消息鉴别码、RSA 非对称加密、序列密码加密、分组密钥密码加密、密钥生成、密钥交换等常用的算法。

2.7.8 安全数据存储

对于系统中的关键数据，如消息、用户信息等等，金蝶Apusic应用服务器提供了安全的内部保护存储机制。

2.8 管理与监控

金蝶Apusic应用服务器管理模型基于JMX框架并遵循JSR160规范。使用JMX框架能通过Java语言封装软硬件资源并将其暴露在分布式环境下。JMX同时还提供了映射框架，以便将已存在的管理协议如SNMP集成到自身的管理体系结构中。

金蝶Apusic应用服务器提供管理服务，它作为管理工具与服务器MBean之间的接口，为管理工具提供必要的管理功能。

2.8.1 管理接口

金蝶Apusic应用服务器提供管理服务作为管理接口，分为以下几部分：

- 系统管理服务

提供对金蝶Apusic应用服务器大部分的管理功能，如启动与停止应用服务器、创建数据源及设置优化选项、配置多路复用端口号、设置事务缺省超时时间、调整HTTP线程池最大线程数等。

- 部署管理服务

提供应用部署管理功能，如部署与卸载应用、启动与停止应用等。

- 监控服务

提供监控金蝶Apusic应用服务器运行时状态的功能，如观察某个数据源的连接创建数、某个Servlet的执行时间、HTTP线程池的当前工作线程数等。

2.8.2 管理工具

金蝶Apusic应用服务器提供以下管理工具：

2.8.2.1 Apusic管理控制台

Admin管理控制台（Admin GUI Console）是基于Eclipse技术的管理工具，通过它可以方便的对金蝶Apusic应用服务器以及Apusic MQ服务器进行配置与监控操作。

Admin管理控制台提供性能监控功能，使用它可以通过曲线视图的方式观察最近自定义时间段内金蝶Apusic应用服务器组件及服务的运行状况，如观察最近一段时间内某个Servlet的服务时间；同时用户可以通过表格的方式查看当前监控项的一些统计信息，如Servlet服务的最大时间、最小时间、平均时间等等。它还能够在一个视图中监控多个项，如在一个视图中监控两台金蝶Apusic应用服务器中HTTP线程池的工作线程个数，方便用户对多个监控项进行比较查看。

2.8.2.2 Web管理控制台

Web管理控制台是基于Web浏览器的管理工具。用户通过Web浏览器就能够对金蝶Apusic应用服务器进行配置与管理操作。Web浏览器支持目前比较流行的IE及Firefox，其中IE支持5.2及以上版本，Firefox支持1.5及以上版本。

Web管理控制台的界面风格与Apusic管理控制台类似，实现的配置功能也完全相同，但它不提供监控功能。如果没有安装Apusic管理控制台，可以通过浏览器完成大部分相同的配置操作。

2.8.2.3 Shell Script

为方便用户进行自动化操作，金蝶Apusic应用服务器提供了通过Shell命令方式对金蝶Apusic应用服务器进行配置与管理操作，这些命令存在于<APUSIC_HOME>/bin以及<DOMAIN_HOME>/bin下。

金蝶Apusic应用服务器还提供了一些Ant task，如对Web模块进行JSP的预编译、对应用进行管理操作等。

2.9 集群服务

金蝶Apusic应用服务器对集群服务提供了优异的支持，通过Apusic集群服务，企业应用能够获得高可用性及水平可扩展性。

我们知道在集群中需要在各结点之间同步一些状态信息，每当一个结点的状态发生变化时，都需要通过多播等方式向其他结点传递状态信息，随着集群内部结点的增多，内存复制将会非常频繁，从而造成广播风暴，耗费大量的网络带宽，对性能造成很大影响。Apusic集群针对不同的场景提供了各具特色的创新性设计与实现。例如：Apusic独创的Client Session Cache(客户端会话缓存)技术。针对Stateful SessionBean，Apusic所采取的技术是客户端缓存，直接将状态信息保存在客户端，当服务器失效时将状态转移到可用服务器，这将有效降低广播风暴出现的机率，从而大大提升集群效果。Apusic自带的负载均衡器也采用类似的技术，将状态信息保存在负载均衡器中，达到同样的效果。除此之外，Apusic还提供了集群节点配对复制技术，以解决在使用第三方web负载均衡器时的状态复制问题，此时，每个节点只把自己的状态复制到集群中的一个节点上，以此避免网络风暴。。

Apusic作为专业的、成熟的应用服务器，它提供了整套的、灵活的、有效的集群方案，是金融、电信、电力等行业的基于规范技术的高性能、高可用、可扩展系统的保障。

Apusic集群主要包括Web集群、JNDI集群、EJB集群、JMS集群。

2.9.1 Web集群

一般来说，Web集群试图解决两个问题：客户请求的负载均衡和Session的高可用。

Apusic Web集群为这两个问题提供了灵活、全面的解决方案。客户请求的负载均衡是指客户的请求依赖特定算法被合理地分配给多台Web Server来处理。Session的高可用性是指当某台Web Server失效，这台Web Server服务的客户的请求会被透明地转发给其它有效Web Server，而会话状态（Session）依然可用。我们把集群范围内具有高可用性的Session称为集群Session。

Apusic Session集群采取Instant Replication，即某节点的Session操作是即时传播（同步）到集群中的其它节点的；一些应用服务器厂商的集群Session同步采取非即时的方式，这会降低Session可用性。还有一个比较常见的场景是新节点加入工作中的Apusic集群时，新节点的Session会自动与集群同步。

2.9.2 JNDI集群

JNDI作为JavaEE的基础技术，JNDI集群往往是其它上层集群技术的必要条件。JNDI集群是Apusic服务器提供的一个服务，一旦启用这个服务，集群的各节点（指Apusic服务器）都拥有一个集群JNDI树，它有别于本地JNDI树，事实上，在JNDI集群节点中，这两棵树是同时存在的。

Apusic JNDI同时具有JNDI负载均衡和失效恢复能力。Apusic JNDI负载均衡和失效恢复能力，可以通过指定JNDI服务的集群节点来实现，也可以通过启动discovery服务，自动发现能够提供JNDI服务的集群节点。

2.9.3 EJB集群

Apusic EJB集群主要面向两个问题：负载均衡和高可用性。值得注意的是，EJB集群一般都是针对EJB远程调用而言，而非本地调用（包括集群节点内部使用远程接口，因为对于这种情况，Apusic会自动优化成本地调用）。

EJB负载均衡指EJB调用通过特定算法分配到多台应用服务器的行为。这里的EJB调用指对EJB Bean接口的调用。然而，Apusic EJB集群还支持Home接口的负载均衡。目前，Apusic EJB集群支持的负载均衡算法有：RANDOM，ROUND_ROBIN，WEIGHTED，STICKY。

EJB高可用性指在某集群节点失效的时候，对它的EJB Home接口的调用或者Bean接口的调用会透明地转移到其它有效节点。

2.9.4 JMS集群

JMS是JavaEE相当重要的一部分，它为开发异步、可靠、高性能、灵活的系统提供支持。目前广泛应用于EAI、EDI等领域。Apusic JMS集群主要包括消息路由和集群队列功能。

2.9.4.1 消息路由

消息路由一般用来解决在发送消息的客户端和消息目的地不能直接连通的情况。依赖智能路由算法，Apusic会在JMS网络中选择一个或多个消息路由器（Router）组成一条到达消息目的地的最优通路，负责转发消息。

2.9.4.2 消息队列

Apusic MQ支持消息队列集群（Message Queue Clustering）。所谓消息队列集群是指在多个Apusic MQ之间建立路由连接，从而组成一个消息路由网络。在网络中任何一个节点上定义的集群队列都将被全部节点所共享，通过任何一个节点向一个集群队列发送/接收消息都是等效的，对客户来说无法察觉是否正在使用集群，集群中网络拓扑结构发生变化对客户也没有任何影响。

当路由节点与其他节点建立连接时，通过发布自身定义的集群队列，从而使集群队列获得共享，无须系统管理员干预，大大减少了路由网络的维护量。使用集群还可以提高系统的可用性，路由网络中单个节点发生故障停机或部分网络无法连通时并不影响集群队列的使用，同时在发送消息时根据各节点的负荷情况对负载进行分配，从而使节点处理能力和网络带宽被充分利用。

2.10 技术汇总

下表列出了金蝶Apusic应用服务器9.0相关联的技术与规范

Java平台
Java Platform, Standard Edition 6
Java Development Kit 1.6
JavaEE规范

Java Platform, Enterprise Edition 6
EJB 3.1
Servlet 3.0
JSP 2.2
Interceptors 1.1
EL 2.2
JMS 1.1
JTA 1.1
JavaMail 1.4
JAX-RS 1.1
JAXB 2.2
JAXR 1.0
CDI for Java EE 1.0
Java EE Management 1.1
Java EE Deployment 1.2
JACC 1.4
JSP Debugging 1.0
JSTL 1.2
JSF 2.0
Java Persistence 2.0
Bean Validation 1.0
Dependency Injection for Java 1.0
JASPIC 1.0
Managed Beans1.0
Security
JAAS 1.0
JACC 1.0
Apusic SSO
Apusic Security Extensions

JSF support
Apusic Ajax Support
Apusic Layout Controls
Apusic UI Components
Portal support
Apusic Portlet Container
Web Services
JAX-WS 2.1
JAX-RPC 1.1
Web services for Java EE 1.2
WS-I Basic Profile 1.2
SAAJ 1.3
UDDI V2 and V3 API support
Private UDDI V3 registry
SOAP 1.2
JAXB 2.0
WSDL 1.1 and 2.0
OASIS Web Services Security:SOAP Message Security 1.1
OASIS Username Token Profile 1.1
OASIS X.509 Token Profile 1.1
Clustering
Failover Support
Load Balance
Administration and Deployment
Apusic Web-based administration center
Apusic GUI-based administration console (Eclipse-based)
Apusic Shell script administration support
Java EE Management (JSR 077)
Java EE Application Deployment (JSR 088)

2.11 Apusic Platform

由金蝶中间件公司开发的金蝶Apusic应用服务器，金蝶Apusic消息中间件及金蝶Apusic Studio集成开发环境等，组成了企业级中间件平台 — Apusic Platform。顺应快速开发企业应用的发展趋势，并努力推动Java EE在国内的推广。

2.11.1 Apusic Studio

金蝶Apusic Studio建立在Eclipse平台之上，它通过良好的工具支持以及与Apusic应用服务器的紧密集成，大幅降低JavaEE应用开发的学习曲线，提升应用开发效率；通过优秀的框架支持及基础组件，成功地解决了应用软件在开发、实施和维护过程中的质量、周期、成本、风险等方面的问题。

2.11.2 金蝶Apusic消息中间件

金蝶Apusic MQ消息中间件是金蝶中间件公司历经多年开发出的产品，采用Java语言开发,支持所有主流的操作系统和硬件平台,同时提供多种语言的调用接口，提供消息的安全可靠传输，并能够根据网络负载情况选择最佳路由。它的最大特点是能够与应用服务器紧密集成。

金蝶Apusic MQ消息中间件实现了消息中间件的所有基本功能，同时还包含许多增强特性，是能高度满足客户需要的产品，可以解决大容量、快速、要求严格的消息传递需求。

3 部分III 快速开始

3.1 概述

本快速入门指南提供了启动/停止服务器、部署应用程序以及设置集群、配置负载平衡的基本过程。

本前言包含对金蝶Apusic应用服务器的基本介绍、相关资源集的信息及其约定。

3.1.1 金蝶Apusic应用服务器的基本介绍

金蝶Apusic应用服务器为复杂应用提供了一个简便、快速的开发和运行平台，对于分布式的企业级应用，提供了易扩展、可伸缩和高安全性等特性。下面将介绍Apusic应用服务器的体系结构及其提供的服务和功能，从而展示Apusic应用服务器对开发大型应用系统的支持能力。

3.1.2 相关资源

针对不同的操作系统，金蝶Apusic应用服务器提供不同的安装包。

Apusic web站点提供有关以下附加资源的信息：

金蝶中间件官方网站http://www.apusic.com

3.1.3 默认路径和文件名

下表介绍了在本书中使用的默认路径和文件名。

变量	说明	默认值
JAVA_HOME	Java环境变量	
APUSIC_HOME	Apusic服务器的安装目录名	Linux安装（超级用户）：/opt/apusic-V9.0 Linux安装（非超级用户）：用户的主目录/apusic-V9.0 Windows的所有安装：系统驱动器：apusic-V9.0
DOMAIN_HOME	域主目录	APUSIC_HOME/domains/[domainname]

3.2 基本功能快速入门

欢迎使用快速入门指南。本指南提供了一组样例过程，您可以通过这些过程来开始使用金蝶Apusic应用服务器。

在执行本文档中的过程之前，您必须已经安装了金蝶Apusic应用服务器软件。下面将介绍金蝶Apusic应用服务器的安装、启动与停止。

3.2.1 Apusic应用服务器管理工具

为了使管理员能够管理在多个主机上运行的服务器实例和集群，Apusic应用服务器提供了以下工具：

- 管理控制台（基于浏览器的Web程序以及基于GUI的桌面应用程序）
- 命令行工具，如startapusic实用程序。有关Apusic应用服务器中提供的命令行工具的完整列表，如下表所示。

- 程序性Java Management Extensions(JMX™)API

工具名称	目录	说明
startapusic	DOMAIN_HOME/bin	启动Apusic应用服务器
stopapusic	DOMAIN_HOME/bin	停止Apusic应用服务器
startas	APUSIC_HOME/bin	启动应用服务器工具，通常需要带域名
stopas	APUSIC_HOME/bin	停止应用服务器工具，通常需要带域名
firststep	APUSIC_HOME/bin	启动应用服务器、信息中心与管理工具
appctl	APUSIC_HOME/bin	用于部署、卸载、重启动、输出应用列表，同时，appctl也可用于远程部署
apclient	APUSIC_HOME/bin	运行应用客户端程序，apclient会启动应用客户端容器，使应用客户端运行在客户端容器中，为客户端提供运行时环境
dep	APUSIC_HOME/bin	打开Apusic应用服务器部署工具
AdminMain	APUSIC_HOME/bin	通过命令行管理工具可以完成通过图形管理工具的管理和配置功能，可对服务器中的服务进行配置，并可察看各配置项的帮助信息

3.2.2 安装

3.2.2.1 Linux下安装

如若安装包为.zip包，放置指定目录，执行命令unzip AAS-V10.zip，解压安装包完成安装。

获取图形化安装包 ，放置指定目录，执行授权命令，chmod 755 AAS-V10.bin；再执行AAS-V10.bin，按照安装指引完成安装。

3.2.2.2 Windows下安装

如若安装包为.zip包，放置指定目录，直接解压安装包AAS-V10.zip 完成安装

如若安装包为.exe，双击执行AAS-V10.exe，按照安装指引安装程序

三元分立版本安装：

如需要安装三元分立版本，解压AAS-V10.zip，确保APUSIC_HOME/lib目录下有admin.war，即可完成安装。如若原本安装包下只有webtool.war，需要将webtool.war删除，把admin.war放置该目录下。

3.2.3 卸载

3.2.3.1 Linux下卸载

如若安装方式为直接解压的zip包，可进入安装目录，直接删除安装目录即可卸载。

如若安装方式为图形化安装，可点击双击"卸载AAS"

3.2.3.2 Windows下卸载

如若安装方式为直接解压的zip包，可进入安装目录，直接删除安装目录即可卸载。

如若安装方式为图形化安装，可点击开始->程序->金蝶Apusic应用服务器->卸载Apusic应用服务器

3.2.4 启动

介绍在Windows/Linux平台上启动应用服务器。

首次启动应用服务器时需要设置管理员admin的密码；并且首次启动服务器，需使用前台启动方式，不能使用后台启动（因为后台启动的话不能输入密码，会一直报密码为空的错误）。

3.2.4.1 在Linux上启动服务器

在Linux平台上启动服务器有三种方式：

1.使用startapusic脚本

与Windows平台上使用startapusic脚本启动apusic服务器类似。进入DOMAIN_HOME/bin/目录，输入startapusic命令并执行即可。

注：终端的退出将导致apusic服务的退出。

2.后台启动

进入DOMAIN_HOME/bin/目录，执行 startapusic &

注：后台启动方式下，终端的退出不会导致apusic服务的退出。

3.系统服务方式

取决于不同的Linux，一般在/etc/rc.local脚本中添加apusic后台启动命令：

```
java -Xms128m -Xmx512m -XX:MaxPermSize=128m -
Dcom.apusic.domain.home=usr/apusic-/domains/mydomain -
classpath %CP%
com.apusic.server.Main -root usr/apusic-
```

注：假定usr/apusic-为服务器的安装目录，变量%CP%的值包含前述启动服务器必需的JAR文件

3.2.4.2 在Windows上启动

在Windows平台上启动服务器有五种方式：

1.使用startapusic脚本

在命令行提示符下，进入DOMAIN_HOME/bin/目录，输入startapusic.cmd命令并执行即可。startapusic提供了多种启动参数，包括：

-d[ebug]：以JDPA形式启动apusic，此时可使用支持JDPA的调试工具（例如Apusic Studio）对部署在服务器上的应用进行调试

-d[ebug]s：以JDPA形式启动apusic，并且一开始是suspend

-p[roduct]：以生产环境启动apusic，此时，JVM加上一些调整参数，譬如-server -Xms -Xmx

-v[erbose]：此时并不启动apusic，而只打印license信息

-va：此时并不启动apusic，而只是把当前系统参数以及license信息显示出来

-m：如果安装了AAS-APM性能监控模块,则可以通过此参数启动并加载Apusic性能监控平台。

2.程序组方式

选择：开始->程序->金蝶Apusic应用服务器->启动Apusic应用服务器

注：开始菜单中的启动服务器以及apusic管理控制台（基于浏览器）的快捷方式面向的是mydomain下的服务器实例。

3.系统服务方式

Apusic服务器执行DOMAIN_HOME/bin/目录下的apusicsvc.exe命令

```
apusicsvc.exe -install -name myserver
```

可将Apusic应用服务器作为Windows平台的系统服务进行管理，服务名称为myserver，如果未指定name，缺省使用ApusicServer。

选择：开始->设置->控制面板->管理工具->服务

在打开的服务面板窗口中，选择myserver项进行配置（myserver为安装服务时指定的服务名称，缺省为ApusicServer），如果服务器已

启动，则可以选择停止myserver服务，反之，则可以选择启动。

同时，可以指定是否禁用myserver服务，或者设置启动类型为手动或者自动。

3.2.5 停止

介绍在Windows/Linux平台上停止服务器。

3.2.5.1 在Linux上停止服务器

在Linux平台上停止服务器有三种方式：

1.使用stopapusic脚本

与Windows平台上用stopapusic脚本停止服务器类似，进入DOMAIN_HOME/bin/目录，输入stopapusic命令并执行即可。

Ctrl + c与Windows平台上用Ctrl + c停止服务器类似，Linux终端方式启动下，用Ctrl + c停止

2.杀进程

通过以下命令查找运行中的进程号：

```
ps -uax|grep java
```

然后用以下命令停止运行中的apusic服务

```
Kill -9 查到的进程号
```

3.2.5.2 在Windows上停止服务器

在Windows平台上停止服务器有三种方式：

1.使用stopapusic脚本

在命令行提示符下，进入DOMAIN_HOME/bin/目录，输入stopapusic.cmd命令并执行即可，例如：

```
stopapusic.cmd admin admin iiop://localhost:6888
```

2.Ctrl + c

Windows下以命令行或脚本方式启动时，用Ctrl + c停止

3.杀进程

打开Windows任务管理器，找到相关进程(通常是java.exe)，选中并结束进程。

3.3 部署应用程序

3.3.1 摘要

根据Java EE平台规范，Java EE应用包含Web模块与EJB模块等。本节介绍Web模块、EJB模块和Java EE应用如何部署到金蝶Apusic应用服务器上。

3.3.2 应用打包和部署简介

应用打包是将应用中的各个分离组件打包到一个单元中，使它可以部署到符合Java EE规范的应用服务器上。包可以是单独的模块，如Web模块war或EJB模块jar，也可以是完整的Java EE应用，包不一定以压缩形式存在，也可以展开的目录形式存在。

Java EE模块是一个或多个属于同一种容器类型（如Web容器，EJB容器）的Java EE组件的集合，并带有这种容器的部署描述文件。其中一个为Java EE标准的部署描述文件（如：application.xml），另一个是专有的部署描述文件（如：apusic-application.xml）。Java EE模块类型有：

- WAR文件：WAR文件由下列项目组成：servlets、JSP、JSP标记库、辅助类、静态页面、Java bean和部署描述文件（web.xml和可选的通过AOM Studio部署pusic-application.xml或apusic-web.xml）。
- EJB JAR文件：EJB JAR文件是标准的EJB打包格式。JAR文件中包含home接口、远程接口、本地接口、用户的实现类、辅助类和部署描述文件（ejb-jar.xml，单独部署EJB JAR时还要包含apusic-application.xml或apusic-ejb-jar.xml）。
- 应用客户端JAR文件：应用客户端支持标准的Java EE客户端规范，它的部署描述文件是application-client.xml。
- 资源RAR文件

部署描述文件中的信息为声明式的，因此不需要更改源文件就能够更改这些信息。EJB JAR和Web模块可以单独部署，这时，它们应该包含apusic-application.xml或apusic-web.xml、apusic-ejb-jar.xml。

JavaEE应用由一个或多个模块组成，使用Java应用档案文件格式把所有模块打包成扩展名为ear的文件，然后部署到Apusic应用服务器上。

	注意

	• Web应用WAR作为模块单独部署时，可能需要包含金蝶Apusic应用服务器的配置文件：apusic-application.xml，放在目录META-INF下或apusic-web.xml，放在目录WEB-INF下 • EJB JAR作为模块单独部署时，必须包含Apusic的配置文件：apusic-application.xml，放在目录META-INF下或apusic-ejb-jar.xml，放在目录META-INF下。 • EJB JAR和Web应用WAR作为模块打包在JavaEE应用EAR中时，配置文件apusic-application.xml应该放在EAR文件的META-INF目录中，或者各个模块独立配置部署描述符（即使用apusic-web.xml、apusic-ejb-jar.xml，描述参考上述两条）

3.3.3 部署应用程序

3.3.3.1 目录结构

- Web应用的目录结构

Web应用（Web Applications）是最小的、可部署的一组可重用的Web资源。Web应用被打包和部署成Web ARchive(WAR) 文件，是一个带有 .war 后缀的JAR文件。

Web应用使用层次结构存放Web资源，在开发阶段表现为文件系统的目录结构。如下图所示：

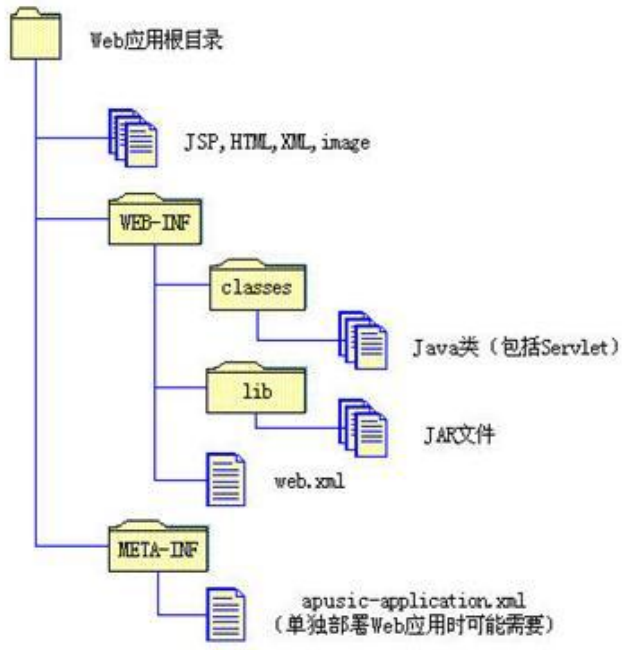


图 Web应用的目录结构

- EJB-JAR文件结构

EJB-JAR文件是一个由开发者提供的、包含一个或多个EJB组件的应用可装配单元文件，是一个一般的JAR文件。一般一个EJB-JAR文件代表一个EJB模块。当一个或多个EJB模块被装配到一个JavaEE应用，并由装配者通过部署描述提供了装配信息和具体运行环境中资源信息之后，即可部署到应用服务器。Apusic应用服务器支持EJB-JAR的单独部署，这时，EJB-JAR中就要包含装配信息。EJB-JAR文件的结构如下图：

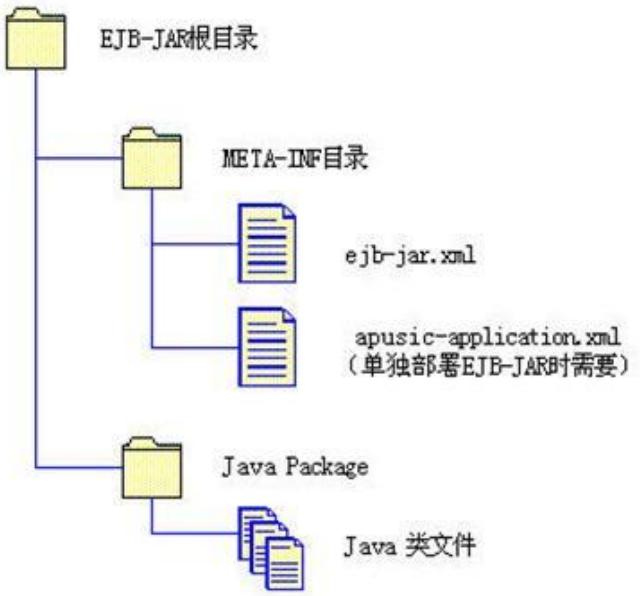


图 EJB-JAR文件结构

- JavaEE应用的目录结构

JavaEE应用由一个或多个JavaEE组件和部署描述文件组成。在金蝶Apusic应用服务器上，实际的Java EE应用被打包成为一个后缀名为".ear"的EAR文件。实际的EAR文件中的目录结构如下图：

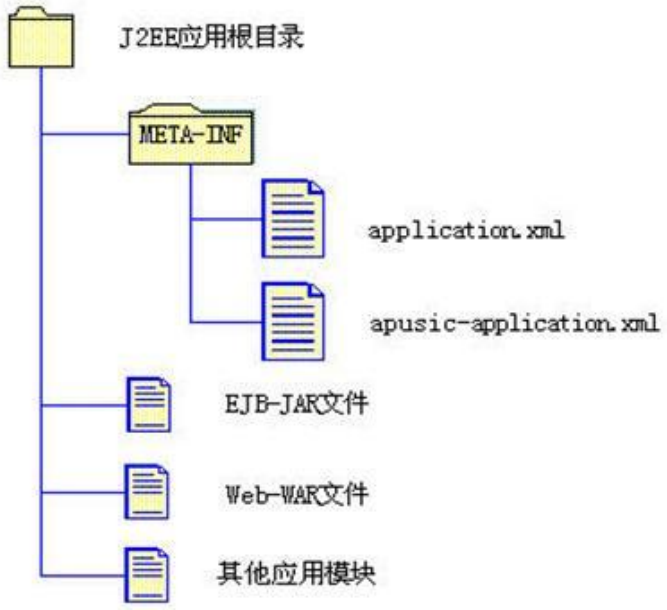


图 JavaEE应用的目录结构

3.3.3.2 准备工作

- 部署Web模块

  部署Web应用首先要配置web.xml文件。开发人员需要在部署描述文件web.xml中配置：ServletContext初始化参数、Session配置、Servlet/JSP定义、Servlet/JSP映射、MIME类型映射、欢迎文件列表、错误页面、安全。

Web模块单独部署时可能还需要在部署描述文件apusic-application.xml（或apusic-web.xml）中配置Apusic相关的信息，如context-root、EJB引用、资源引用等。由于每个web模块都必须指定一个唯一的context-root，因此有三种方法指定一个独立web模块的context-root：

- 在部署时指定base-context，请参考第 4.18.1 节 “appctl工具”
- 在WAR中包含一个apusic-application.xml（或apusic-web.xml）文件指定context-root；
- 如果以上两种方法都没有采用，则使用appctl命令中指定的应用名作为context-root。如果使用了自动部署，WAR文件名去掉.war后缀后作为应用名。

- 部署EJB模块

部署描述文件ejb-jar.xml为EJB模块提供如下两方面的信息：

- 结构信息

ejb-jar.xml文件为模块中的EJB组件提供的结构信息并声明组件的外部依存关系，对于组件的开发者而言，结构信息是必须提供的，而且一般结构信息在装配和部署时不能更改，以免破坏组件的功能。

- 装配信息

装配信息提供将组件装配到一起，形成一个更大的可装配单元的信息，装配信息对于组件提供者而言是可选的，并且在装配和部署时可以通过修改装配信息改变装配后的应用的行为。

ejb-jar.xml文件必须符合ejb-jar_3_0.xsd文档类型声明，并在文件中指定正确的xsd文件：

```
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
version="3.0"
```

EJB模块单独部署时还需要在部署描述文件apusic-application.xml（或apusic-ejb-jar.xml）中配置Apusic相关的信息，如EJB引用、资源引用、CMP2.0的域-数据库列映射、CMP2.0关系映射等。

- 部署JavaEE应用

Java EE标准的部署描述文件application.xml列出了应用包含的所有Java EE模块。

部署描述文件apusic-application.xml配置应用包含模块的Apusic相关的信息，如context-root、EJB引用、资源引用、CMP2.0的域-数据库列映射、CMP2.0关系映射等。

3.3.3.3 自动部署

- 部署Web模块

将打包的Web-WAR模块拷贝到APUSIC_HOME/domains/<DOMAIN_NAME>/applications目录，Web应用会被自动部署。其中<DOMAIN_NAME>为正在运行的Apusic应用服务器实例的域名。

- 部署EJB模块

将打包的EJB-WAR模块拷贝到APUSIC_HOME/domains/<DOMAIN_NAME>/applications目录，EJB模块会被自动部署。

- 部署Java EE应用

将打包的Java EE应用EAR文件拷贝到APUSIC_HOME/domains/<DOMAIN_NAME>/applications目录，Java EE应用会被自动部署。

3.3.3.4 使用appctl工具部署

在命令行提示符下，进入DOMAIN_HOME/bin/目录。

- 部署Web模块

在金蝶Apusic应用服务器上，支持两种形式的Web应用，一种是标准的压缩文件形式，另一种则可以是目录形式的Web应用。

使用Apusic提供的appctl工具部署时。Apusic支持目录形式和打包形式的Web模块，例如：

```
appctl install web_app_name MyWeb.war
```

如果Web模块是目录的形式则Web应用无须安装，仅在server.xml中添加一个条目。如果Web模块是打包的形式则将应用自动解压缩到APUSIC_HOME/domains/<DOMAIN_NAME>/deploy目录下的应用子目录中(其中<DOMAIN_NAME>为当前正在运行的Apusic领域名)，如果该档案文件发生变化，在重启应用时将对Web模块重新自动解包。在应用子目录中每个模块都建有一个临时文件夹，用于存放运行时生成的临时文件。

- 部署EJB模块

在金蝶Apusic应用服务器上，支持两种形式的EJB-JAR，一种是标准的JAR文件形式，另一种则可以是目录形式的EJB-JAR。通常，在组件开发阶段，采用目录形式的EJB-JAR，可以减少使用JAR工具打包的工作，并可方便的对模块进行修改。

而在实际的装配和部署阶段，往往采用易于管理的标准JAR文件格式。

当使用标准的JAR文件形式打包EJB-JAR模块时，只需按照前面描述的EJB-JAR的结构，使用jar工具打包即可。

使用Apusic提供的appctl工具部署EJB模块。Apusic支持目录形式和打包形式的EJB模块，例如：

```
appctl install ejb_app_name MyEJB.jar
```

如果EJB模块是目录的形式则无须安装，仅在server.xml中添加一个条目。如果EJB模块是打包的形式则将应用自动解压缩到APUSIC_HOME/domains/<DOMAIN_NAME>/deploy目录下的应用子目录件（其中<DOMAIN_NAME>为正在运行的Apusic领域名），如果该档案文件发生变化，在重启应用时将对EJB模块重新自动解包。在应用子目录中每个模块都建有一个临时文件夹，用于存放运行时生成的临时文件。

- 部署JavaEE应用

在金蝶Apusic应用服务器上，支持两种形式的Java EE应用，一种是标准的JAR文件形式，另一种则可以是目录形式。通常，在Java EE应用开发阶段，采用目录形式，可以减少使用JAR工具打包的工作，并可方便的对模块进行修改。

而在实际的装配和部署阶段，往往采用易于管理的标准JAR文件格式。

当使用标准的JAR文件形式打包Java EE应用时，只需按照前面描述的JavaEE应用目录结构，使用jar工具打包即可。

使用Apusic提供的appctl工具部署JavaEE应用。Apusic支持目录形式和打包形式的JavaEE应用，例如：

```
appctl install jee_app_name MyApp.ear
```

如果应用是目录的形式则应用无须安装，仅在server.xml中添加一个条目。如果应用块是打包的形式则将应用自动解压缩到APUSIC_HOME/domains/<DOMAIN_NAME>/deploy目录下的应用子目录中（其中<DOMAIN_NAME>为正在运行的Apusic领域名），如果该档案文件发生变化，在重启应用时将对JavaEE应用重新自动解包。

3.3.3.5 通过管理控制台部署

在访问远程管理控制台之前，必须先启动金蝶Apusic应用服务器。远程管理控制台是基于web方式的管理界面。

按照以下步骤启动远程管理控制台：

1. 在浏览器窗口打开以下URL：

```
http://hostname:6888/admin
```

其中：localhost指安装金蝶Apusic应用服务器软件的本机，如若需要远程访问，需要在vm.options文件中写入com.apusic.webtool.allowHosts=172.24.1.1，多个地址用英文逗号分隔；支持通配符`*`，如`172.*`；添加后启动金蝶Apusic应用服务器软件。如果是三元分立版，需要在vm.options文件中写入com.apusic.admin.allowHosts=172.24.1.1。浏览器再输入安装金蝶Apusic应用服务器软件的计算机的IP地址或主机名；6888是金蝶Apusic应用服务器默认的服务端口。

在Windows平台，可用如下方式打开远程管理控制台：

选择：开始->程序->金蝶Apusic->Apusic应用服务器->Apusic管理控制台（基于浏览器）

2. 进入远程控制登录界面时，将会提示输入登录的用户名和密码，输入Apusic初始登录用户名"admin"和登录密码（首次启动时设置的密码），作为提示输入登录的用户名和密码即可进入远程管理界面。



图 使用管理控制台方式部署应用

	注意
	Apusic初始登录系统管理账号用户名是"admin"，登录密码为首次启动时设置的密码。

3.3.3.6 通过Apusic Studio部署

通过Apusic Studio部署工程时先要设置金蝶Apusic应用服务器：

选择：工程->右键属性（Properties）->Apusic工程属性，如下图所示，设置相应的Apusic应用服务器、目标域、JavaEE版本、EAR扩展文件夹。



图 设置金蝶Apusic应用服务器

在开发透视图下，部署工程：

选择：工程->右键（部署到服务器）->部署，如下图所示。

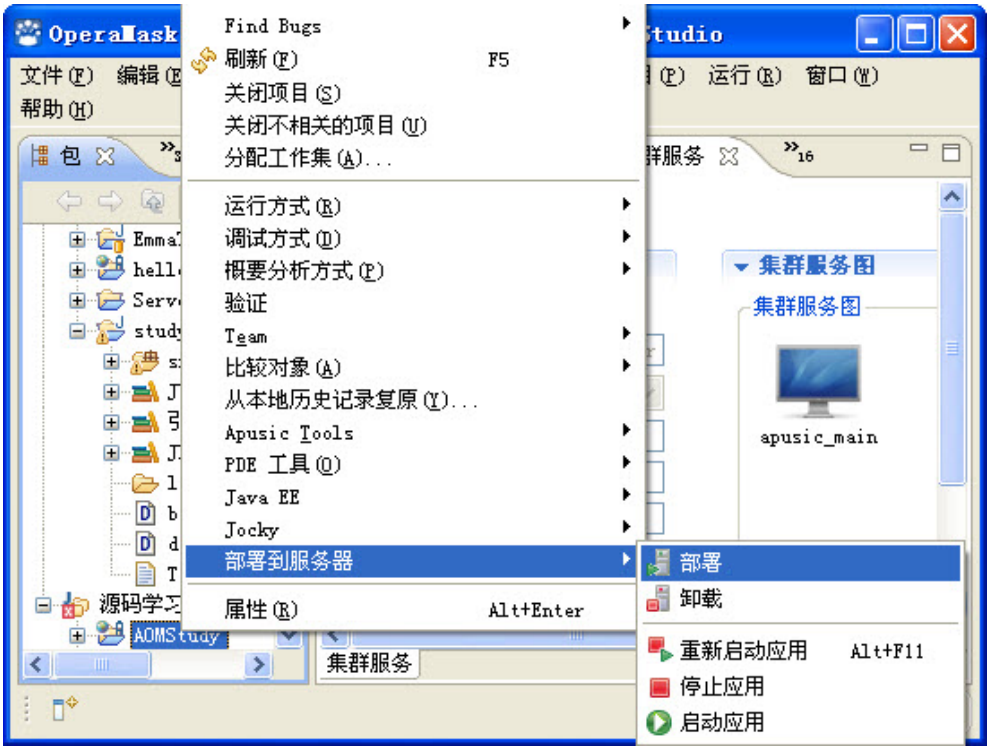


图 使用Apusic Studio部署应用

部署成功后在APUSIC_HOME/domains/[domainname]/deploy和APUSIC_HOME/domains/[domainname]/deploy/.extends目录下生成有关工程的相关文件。

3.4 小结

祝贺您！您现在已经完成了Apusic应用服务器的快速入门。

在此部分中，您安装、配置和启动、停止了Apusic应用服务器，而且还配置了集群和应用程序使其具有高可用性。有关Apusic Server的其他信息，请参见以下其它文档，如：《Apusic应用服务器参考手册》、《发行说明》、《安装指南》、《管理指南》、《开发指南》等。

4 部分IV 安装指南

4.1 前言

本文档描述在Windows及Linux,Unix系统安装金蝶Apusic应用服务器的注意事项，包含如何进行金蝶Apusic应用服务器的注册。

4.2 系统配置要求

安装金蝶Apusic应用服务器的最低系统要求见以下内容：

Windows:

组件	要求
操作系统	Windows NT 4.0 with Service Pack 6a， Windows 2000 with Service Pack 3， Windows XP，Windows 2003
Java环境	Java™ Platform, Standar Edition (JavaSE™) Development Kit 6.0 (jdk1.6)或以上版本
CPU	PentiumIII 500MHz或以上
物理内存	512MB或以上
硬盘	可用空间500MB或以上
浏览器	Microsoft Internet Explorer 6.0或以上，FireFox 1.5或以上，用于支持Apusic应用服务器基于Web的管理客户端

Solaris:

组件	要求
操作系统	Sun SPARC Solaris version 8或以上版本
Java环境	Java™ Platform, Standar Edition (JavaSE™) Development Kit 6.0 (jdk1.6)或以上版本
物理内存	512MB或以上
硬盘	可用空间500MB或以上

Linux:

组件	要求
操作系统	LINUX Red Hat 6.0或以上 (及其它Kernel 2.2.5或以上linux版本)
Java环境	Java™ Platform, Standar Edition (JavaSE™) Development Kit 6.0 (jdk1.6)或以上版本
物理内存	512MB或以上
硬盘	可用空间500MB或以上

AIX:

组件	要求
操作系统	IBM AIX 4.2.1 或以上版本
Java环境	需要AIX的JRE环境(IBM jdk for AIX 1.6或以上)
物理内存	512MB或以上
硬盘	可用空间500MB或以上

HP-UX :

组件	要求
操作系统	HP-UX 11.0或以上版本
Java环境	需要HP-UX的JRE环境
物理内存	512MB或以上
硬盘	可用空间500MB或以上

安装金蝶Apusic应用服务器的推荐配置见下表(Windows和Linux平台):

组件	要求
CPU	Intel Pentium-compatible 1GHz或以上
物理内存	512MB或以上

4.3 安装

4.3.1 Linux下安装

如若安装包为.zip包，放置指定目录，执行命令unzip AAS-V10.zip，解压安装包完成安装。

获取图形化安装包 ，放置指定目录，执行授权命令，chmod 755 AAS-V10.bin；再执行AAS-V10.bin，按照安装指引完成安装

4.3.2 Windows下安装

如若安装包为.zip包，放置指定目录，直接解压安装包AAS-V10.zip 完成安装

如若安装包为.exe，双击执行AAS-V10.exe，按照安装指引安装程序

三元分立版本安装:

如需要安装三元分立版本，解压AAS-V10.zip，确保APUSIC_HOME/lib目录下有admin.war，即可完成安装。如若原本安装包下只有webtool.war，需要将webtool.war删除，把admin.war放置该目录下。

4.4 卸载

4.4.1 Linux下卸载

如若安装方式为直接解压的zip包，可进入安装目录，直接删除安装目录即可卸载。

如若安装方式为图形化安装，可点击双击“卸载AAS”。

4.4.2 Windows下卸载

如若安装方式为直接解压的zip包，可进入安装目录，直接删除安装目录即可卸载。

如若安装方式为图形化安装，可点击开始->程序->金蝶Apusic应用服务器->卸载Apusic应用服务器。

4.5 许可证授权

AAS需要有对应的许可证才能正常使用，通常情况下，金蝶天燕会根据用户购买的产品版本配套对应的许可证，如果在使用过程中出现许可证过期或无效等问题，建议优先联系对接的天燕服务人员，重新申请对应许可证。

许可证位置为${APUSIC_HOME}/license.xml。

5 启动与停止

5.1 启动

介绍在Windows/Linux平台上启动应用服务器。

首次启动应用服务器时需要设置管理员admin的密码；并且首次启动服务器，需使用前台启动方式，不能使用后台启动（因为后台启动的话不能输入密码，会一直报密码为空的错误）。

5.1.1 在Linux上启动服务器

在Linux平台上启动服务器有三种方式：

1.使用startapusic脚本

与Windows平台上使用startapusic脚本启动apusic服务器类似。进入DOMAIN_HOME/bin/目录，输入startapusic命令并执行即可。

注：终端的退出将导致apusic服务的退出。

2.后台启动

进入DOMAIN_HOME/bin/目录，执行 startapusic &

注：后台启动方式下，终端的退出不会导致apusic服务的退出。

3.系统服务方式

取决于不同的Linux，一般在/etc/rc.local脚本中添加apusic后台启动命令：

```
java -Xms128m -Xmx512m -XX:MaxPermSize=128m -
Dcom.apusic.domain.home=usr/apusic-/domains/mydomain -
classpath %CP%
com.apusic.server.Main -root usr/apusic-
```

注：假定usr/apusic-为服务器的安装目录，变量%CP%的值包含前述启动服务器必需的JAR文件

5.1.2 在Windows上启动

在Windows平台上启动服务器有五种方式：

1.使用startapusic脚本

在命令行提示符下，进入DOMAIN_HOME/bin/目录，输入startapusic.cmd命令并执行即可。startapusic提供了多种启动参数，包括：

-d[ebug]：以JDPA形式启动apusic，此时可使用支持JDPA的调试工具（例如Apusic Studio）对部署在服务器上的应用进行调试

-d[ebug]s：以JDPA形式启动apusic，并且一开始是suspend

-p[roduct]：以生产环境启动apusic，此时，JVM加上一些调整参数，譬如-server -Xms -Xmx

-v[erbose]：此时并不启动apusic，而只打印license信息

-va：此时并不启动apusic，而只是把当前系统参数以及license信息显示出来

-m：如果安装了AAS-APM性能监控模块,则可以通过此参数启动并加载Apusic性能监控平台。

2.程序组方式

选择：开始->程序->金蝶Apusic应用服务器->启动Apusic应用服务器

注：开始菜单中的启动服务器以及apusic管理控制台（基于浏览器）的快捷方式面向的是mydomain下的服务器实例。

3.系统服务方式

Apusic服务器执行DOMAIN_HOME/bin/目录下的apusicsvc.exe命令

```
apusicsvc.exe -install -name myserver
```

可将Apusic应用服务器作为Windows平台的系统服务进行管理，服务名称为myserver，如果未指定name，缺省使用ApusicServer。

选择：开始->设置->控制面板->管理工具->服务

在打开的服务面板窗口中，选择myserver项进行配置（myserver为安装服务时指定的服务名称，缺省为ApusicServer），如果服务器已

启动，则可以选择停止myserver服务，反之，则可以选择启动。

同时，可以指定是否禁用myserver服务，或者设置启动类型为手动或者自动。

5.2 停止

介绍在Windows/Linux平台上停止服务器。

5.2.1 在Linux上停止服务器

在Linux平台上停止服务器有三种方式：

1.使用stopapusic脚本

与Windows平台上用stopapusic脚本停止服务器类似，进入DOMAIN_HOME/bin/目录，输入stopapusic命令并执行即可。

Ctrl + c与Windows平台上用Ctrl + c停止服务器类似，Linux终端方式启动下，用Ctrl + c停止

2.杀进程

通过以下命令查找运行中的进程号：

```
ps -uax|grep java
```

然后用以下命令停止运行中的apusic服务

```
Kill -9 查到的进程号
```

5.2.2 在Windows上停止服务器

在Windows平台上停止服务器有三种方式：

1.使用stopapusic脚本

在命令行提示符下，进入DOMAIN_HOME/bin/目录，输入stopapusic.cmd命令并执行即可，例如：

```
stopapusic.cmd admin admin iiop://localhost:6888
```

2.Ctrl + c

Windows下以命令行或脚本方式启动时，用Ctrl + c停止

3.杀进程

打开Windows任务管理器，找到相关进程(通常是java.exe)，选中并结束进程。

6 部分V 管理指南

6.1 摘要

Apusic 应用服务器是符合相关规范的应用服务器，提供了诸如负载均衡，数据库连接池，事务处理服务等等互相交错的功能。管理和配置Apusic应用服务器一般通过Apusic管理控制台来进行，Apusic提供了基于GUI和基于Web界面两种形式的管理控制台，关于如何使用Apusic管理控制台对Apusic 应用服务器进行配置，可参考管理工具的参考手册。本文档将描述如何通过配置文件更加深入地对Apusic应用服务器进行管理和配置。

本文档假定用户熟悉基本的XML知识。对于JavaEE5规范中的事务、命名、消息和JDBC等服务和规范有基本了解。

6.2 启动和停止服务器

启动和停止服务器有以下几种方式：

命令行启动或停止服务器

在命令行提示符下，如同启动其他Java程序，Apusic应用服务器可手工键入命令启动，也可使用预先编写的脚本启动，避免每次输入相同的命令。

开始菜单(限于Windows)

Windows系统服务(限于Windows)

使用firststep工具启动服务器

6.2.1 命令行启动或停止服务器

Apusic应用服务器与其他的Java应用程序一样，可以通过java命令启动，只是需要带有一些特定的参数。

在命令行使用Java命令启动Apusic应用服务器的参数设置如下：

- 指定Java堆内存的最大和最小值

Java的虚拟机是基于堆栈的，缺省Apusic服务器启动时的堆内存最大和最小值都是64兆。这两个值可以在实际使用时根据实际情况指定，推荐使用的值如下：

```
java -Xms128m -Xmx512m
```

- 设置环境变量

启动服务器时，需要使用-classpath设置启动Apusic服务器需要的Jar文件，包含位于Apusic应用服务器的安装目录下classes、sp、lib、common目录中的所有jar文件，如：apusic.jar、operamasks-impl.jar、mejb.jar、javaee.jar、elite.jar及其它。其中：

apusic.jar：Apusic应用服务器的核心类；

operamasks-impl.jar：Apusic贡献的Web开源框架；

mejb.jar：实现JSR-77规范的管理EJB；

javaee.jar：规范定义的javaee标准API

elite.jar：Apusic贡献的开源动态语言实现

- 指定服务器的目录

如当前目录并非Apusic应用服务器的安装目录，需要启动服务器，则必须使用-root参数指定服务器的根目录。

- 指定域主目录

指定你需要启动的域主目录，可通过设置jvm参数指定，如-Dcom.apusic.domain.home=/usr/apusic/domains/mydomain

下面是使用java命令启动apusic应用服务器的示例

Linux(假定/usr/apusic为服务器的安装目录，变量$CP的值包含前述启动服务器必需的JAR文件):

```
java -Xms128m -Xmx512m -XX:MaxPermSize=128m -
Dcom.apusic.domain.home=/usr/apusic/domains/mydomain -classpath $CP
com.apusic.server.Main -root /usr/apusic/
```

Windows(假定c:\apusic为服务器的安装目录，变量%CP%的值包含前述启动服务器必需的JAR文件):

```
java -Xms128m -Xmx512m -XX:MaxPermSize=128m -
Dcom.apusic.domain.home=c:\\apusic\\domains\\mydomain -classpath %CP%
com.apusic.server.Main -root c:\\apusic
```

如需要停止以命令行或脚本方式启动的服务器，只需在命令提示符界面输入Ctrl+C即可。

6.2.1.1 使用startapusic启动脚本

在Apusic服务器域主目录下的bin子目录中，提供了预设的启动脚本startapusic，可直接使用此脚本启动apusic应用服务器。

startapusic提供了多种启动参数，包括：

- -d[ebug]:以JDPA形式启动apusic，此时可使用支持JDPA的调试工具（例如Apusic Studio）对部署在服务器上的应用进行调试
- -d[ebug]s：以JDPA形式启动apusic，并且一开始是suspend
- -p[roduct]：以生产环境启动apusic，此时，JVM加上一些调整参数，譬如-server –Xms –Xmx等等
- -v[erbose]：此时并不启动apusic，而只是把license信息显示出来
- -va:此时并不启动apusic，而只是把当前系统参数以及license信息显示出来

6.2.2 开始菜单(限于Windows)

如果Apusic应用服务器通过安装程序安装在Windows平台上，则可以通过开始菜单中的快捷方式启动。

选择：开始->程序->Apusic应用服务器 V8.0->启动服务器

即可。

	**注意**
	开始菜单中的启动服务器以及Apusic管理控制台（基于浏览器）的快捷方式面向的是mydomain下的服务器实例

6.2.3 Windows系统服务(限于Windows)

如Apusic应用服务器通过执行了服务器目录中/bin目录下的apusicsvc.exe如下：

```
apusicsvc.exe -install -name myserver
```

则可将Apusic应用服务器作为Windows平台的系统服务进行管理，服务名称为myserver，如果未指定name，缺省使用ApusicServer。

选择：开始->设置->控制面板->管理工具->服务

在打开的服务面板窗口中，选择myserver项进行配置（myserver为安装服务时指定的服务名称，缺省为ApusicServer），如果服务器已启动，则可以选择停止ApusicServer服务，反之，则可以选择启动。

同时，可以指定是否禁用ApusicServer服务，或者设置启动类型为手动或者自动。

若需卸载服务，可使用命令行

```
apusicsvc.exe -uninstall myserver
```

卸载的服务名称为myserver，如果未指定，缺省使用ApusicServer

6.2.4 使用firststep工具启动服务器

伴随Apusic应用服务器V8.0发布的工具中包括一个firststep工具。可以通过它来启动应用服务器。详情请参考"firststep工具"

	**注意**
	firststep工具面向的是samples域下的服务器实例

6.3 配置Classloader

JavaEE 规范定义了一个打包机制的框架，用来把JavaEE应用的各个部分组织在一起。不同的应用服务器厂商可以自由的设计自己的类装载层次来装载应用中的类和资源。因此开发者必须非常清楚类和资源应该放置在什么位置对于JavaEE应用才是可用的。理解Apusic应用服务器的类装载体系结构能够帮助JavaEE应用的开发者设计高效和可移植应用打包结构。本章先介绍类装载的基本概念，然后讨论了Apusic应用服务器的类装载层次是如何设计的。

6.3.1 Classloader的基本概念

Classloader 在运行期会以父/子的层次结构存在，每个Classloader的实例都持有其父Classloader的引用，而父Classloader并不持有子Classloader的引用，从而形成一条单向链，当一个类装载请求被提交到某个Classloader时，其默认的类装载过程如下：

- 检查这个类有没有被装载过，如果已经装载过，则直接返回；
- 调用父Classloader去装载类，如果装载成功直接返回；
- 调用自身的装载类的方法，如果装载成功直接返回；
- 上述所有步骤都没有成功装载到类，抛出ClassNotFoundException；

每一层次的Classloader都重复上述动作。

简单说，当Classloader链上的某一Classloader收到类装载请求时，会按顺序向上询问其所有父节点，直至最顶端（BootstrapClassLoader），任何一个节点成功受理了此请求，则返回，如果所有父节点都不能受理，这时候才由被请求的Classloader自身来装载这个类，如果仍然不能装载，则抛出异常。

6.3.1.1 类装载的方式

类装载的方式主要有两种：显式的和隐式的。

- 显式类装载

  发生在使用以下方法调用进行装载类的时候：

  - ClassLoader.loadClass()（使用指定的Classloader进行装载）
  - Class.forName()（使用当前类的Caller Classloader进行装载）

当调用上述方法的时候，指定的Class（以类名为参数）由Classloader装入。这两个方法的行为有轻微的区别，Class.forName()在类装载完成后，会对类进行初始化，而ClassLoader.loadClass()只负责装载类。

- 隐式类装载

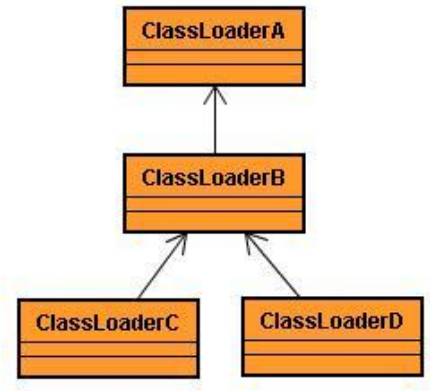
发生在由于引用、实例化或继承导致需要装载类的时候。隐式类装载是在幕后启动的，JVM会解析必要的引用并装载类。

类的装载通常组合了显式和隐式两种方式。例如，Classloader可能先显式地装载一个类，然后再隐式地装载它引用的其它类。

- 类装载发生的时间

从类装载方式的描述中我们可以看到，只有在显式的调用方法或者实例化、引用、继承一个类时，类才真正被装载。由此，我们可以知道，import并不会导致类装载，以及，在一个类实例化之前，调用它的静态方法，会导致这个类和它的父类、实现的接口和相关的静态成员的类会被装载，而它的成员变量的类却不会被装载。

6.3.1.2 一个基本的Classloader的层次结构



上图显示了一个基本的Classloader的层次结构。在给定层次上的Classloader不能引用任何层次低于它的Classloader，另外，它的子Classloader装载的类对于其是不可见的。在上图中，如果Foo.class是由ClassLoaderB装载的，并且Foo.class依赖于Bar.class，那么Bar.class必须由ClassLoaderA或B装载。如果Bar.class只是对ClassLoaderC和D可见，那么将会发生ClassNotFoundException或者NoClassDefFoundError异常。

如果Bar.class分别对于两个平级的Classloader可见（例如C和D），但对于它们的父Classloader不可见，那么当类装载请求发送到这两个Classloader时，每一个Classloader会装载自己版本的类。ClassLoaderC装载的Bar.class的实例将不兼容于ClassLoaderD装载的Bar.class的实例。如果对Classloader的层次结构不了解，试图使用由ClassLoaderC装载的类去造型一个ClassLoaderD装载的Bar.class的实例，则会发生造型失败（ClassCastException）。

6.3.1.3 基本的Classloader

最基本的Classloader是Bootstrap Classloader和System Classloader（也有人称之为AppClassLoader），只要写过java程序，都会用到这两个Classloader。

- Bootstrap Classloader

这个Classloader装载Java虚拟机提供的基本运行时刻类（$JAVA_HOME/jre/lib），还包括放置在系统扩展目录（$JAVA_HOME/jre/lib/ext）内的JAR文件中的类。这个Classloader是java程序最顶层的Classloader，只有它没有父Classloader。如果你将一个自己写的类或第三方jar包放进$JAVA_HOME/jre/lib/ext目录中，那么它将被Bootstrap Classloader装载。

- System Classloader

System Classloader通常负责装载系统环境变量CLASSPATH中设置的类。由System Classloader装载的类对于Apusic服务器内部的类和部署在Apusic服务器上的J2EE应用（通常打包成ear）都是可见的。%APUSIC_HOME%/lib目录下的jar文件是Apusic应用服务器的核心类，一般把这些jar文件都加在系统CLASSPATH中。另外，一些公用类也可以加在系统CLASSPATH中，如JDBC驱动程序等。

6.3.1.4 自定义Classloader

在编写应用代码的时候，常常有需要动态加载类和资源，比如显式的调用classLoader.loadClass("ClassName")，虽然直接使用ClassLoader.getSystemClassLoader()，可以得到SystemlassLoader来完成这项任务。但是，由于System Classloader是JVM创建的Classloader，它的职责有限，只适合于普通的java应用程序，在很多复杂场景中不能满足需求，比如在应用服务器中。这时候就需要自行实现一个Classloader的子类，实现特定的行为。Apusic应用服务器中就定义了若干个特有的Classloader，负责装载部署在Apusic中的JavaEE应用中的类，这里并不试图去描述如何实现一个自定义的Classloader，但本章第二部分将详细描述Apusic自定义的Classloader的行为。

6.3.1.5 Caller Classloader和线程上下文Classloader

动态加载资源时，往往有三种Classloader可选择：System Classloader、Caller Classloader、当前线程的上下文Classloader。System Classloader前面已经描述过了，下面我们看看什么是Caller Classloader、当前线程的上下文Classloader。

- Caller Classloader

Caller Classloader指的是当前所在的类装载时使用的Classloader，它可能是System Classloader，也可能是一个自定义的Classloader，这里，我们都称之为Caller Classloader。我们可以通过getClass().getClassLoader()来得到Caller Classloader。例如，存在A类，是被AClassLoader所加载，A.class.getClassLoader()为AClassLoader的实例，它就是A.class的Caller Classloader。

如果在A类中使用new关键字，或者Class.forName(String className)和Class.getResource(String resourceName)方法，那么这时也是使用Caller Classloader来装载类和资源。比如在A类中初始化B类：

```
/**
* A.java
*/
...
public void foo() {
  B b = new B();
  b.setName("b");
}
```

那么，B类由当前Classloader，也就是AClassloader装载。同样的，修改上述的foo方法，其实现改为：

```
Class clazz = Class.forName("foo.B");
```

最终获取到的clazz，也是由AClassLoader所装载。

那么，如何使用指定的Classloader去完成类和资源的装载呢？或者说，当需要去实例化一个Caller Classloader和它的父Classloader都不能装载的类时，怎么办呢？

一个很典型的例子是JAXP，当使用xerces的SAX实现时，我们首先需要通过rt.jar中的javax.xml.parsers.SAXParserFactory.getInstance()得到xercesImpl.jar中的org.apache.xerces.jaxp.SAXParserFactoryImpl的实例。由于JAXP的框架接口的class位于JAVA_HOME/lib/rt.jar中，由Bootstrap Classloader装载，处于Classloader层次结构中的最顶层，而xercesImpl.jar由低层的Classloader装载，也就是说SAXParserFactoryImpl是在SAXParserFactory中实例化的，如前所述，使用SAXParserFactory的Caller Classloader(这里是Bootstrap Classloader)是完成不了这个任务的。

这时，我们就需要了解一下线程上下文Classloader了。

- 线程上下文Classloader

每个线程都有一个关联的上下文Classloader。如果使用new Thread()方式生成新的线程，新线程将继承其父线程的上下文Classloader。如果程序对线程上下文Classloader没有任何改动的话，程序中所有的线程将都使用System Classloader作为上下文Classloader。

当使用Thread.currentThread().setContextClassLoader(classloader)时，线程上下文Classloader就变成了指定的Classloader了。此时，在本线程的任意一处地方，调用Thread.currentThread().getContextClassLoader()，都可以得到前面设置的Classloader。

回到JAXP的例子，假设xercesImpl.jar只有AClassLoader能装载，现在A.class内部要使用JAXP，但是A.class却不是由AClassLoader或者它的子Classloader装载的，那么在A.class中，应该这样写才能正确得到xercesImpl的实现：

```
AClassLoader aClassLoader = new AClassLoader(parent);
Thread.currentThread().setContextClassLoader(aClassLoader);
SAXParserFactory factory = SAXParserFactory.getInstance();
...
```

JAXP这时就可以通过线程上下文Classloader装载xercesImpl的实现类了，当然，还有一个前提是在配制文件或启动参数中指定了使用xerces作为JAXP的实现。下面是JAXP中的代码片断：

```
ClassLoader cl = Thread.currentThread().getContextClassLoader();
...
Class providerClass = cl.loadClass(className);
...
```

6.3.1.6 JVM中类的唯一性

JVM 为每一个Classloader维护一个唯一标识。在一个JVM里（对应一个Java进程），可以由不同的Classloader装载多个同名的类（指包名和类名都完全相同，下同），为了唯一地标识被不同Classloader装载的类，JVM会在被装载的类名前加上装载该类的Classloader的标识。

6.3.2 Apusic的Classloader体系

在上一节，我们了解了基本的Classloader层次结构模型，知道了Bootstrap Classloader、System Classloader的职责，还知道可以通过自定义Classloader来完成特定的装载任务，除此之外，我们还了解了什么是Caller Classloader和线程上下文Classloader。下面，我们就可以根据这些基本的Classloader概念去看看Apusic Classloader体系是长什么样的了。

6.3.2.1 JavaEE应用对Classloader的要求

Apusic 应用服务器本身运行需要的类都在CLASSPATH中，由System Classloader加载。在上一节中，我们提到Apusic应用服务器中定义了若干个专有的Classloader，负责装载部署在Apusic中的JavaEE应用中的类和资源。Apusic为何要额外的去自定义Classloader呢？把应用需要的类和资源都放在CLASSPATH 中，System Classloader不也可以加载这些类吗？要回答这些问题，我们先考虑一下下面两个简单的需求：

- 不同的应用中，可能有同名的资源文件或类，它们在各自应用中有不同的行为或语义。
- 应用发生变化的时候，例如改了Jsp或者JavaBean，在不重启服务器甚至不重启应用的情况下，需要立即看到修改的效果。

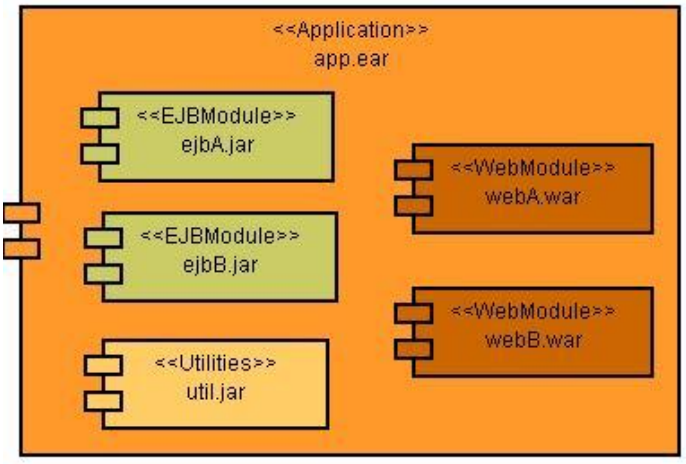
我们前面提到过在一个JVM中一个类的唯一标识，当不能改变类的包名和类名的情况下，除非Classloader的实例发生变化，才有可能实现对一个类的再次加载。显然，在只有System Classloader的情况下，无法满足上面两个简单的需求。这是因为在运行期，我们无法重新创建System Classloader的实例，也没办法让它装载一个已经装载过的类

对于第一个需求，我们可以对不同的应用中的类和资源进行隔离加载，这就需要为每个应用使用不同的Classloader实例；对于第二个需求，当Jsp或JavaBean发生变化时，我们需要把原来装载Jsp的Classloader销毁掉，创建一个新的Classloader实例，并让它去装载修

改后的类，因此，要专门定义一个Classloader 去负责装载Jsp、JavaBean，使得在重新创建Classloader时，受影响的范围尽可能的小。

6.3.2.2 Apusic的Classloader和它们的层次结构

Apusic为装载JavaEE应用中的类定义了EJBClassLoader和ServletClassLoader这两个主要的Classloader。假设一个JavaEE应用的结构如下：



- EJBClassLoader

每个JavaEE应用都有一个EJBClassLoader，用于装载EJB module和公共类。上图中的ejbjarA.jar、ejbjarB.jar、util.jar以及app.ear我们可以看成是一个jar文件，也可以看成是一个目录，它们里边的类和文件都由同一个EJBClassLoader实例装载，因此，同一个JavaEE应用中的EJB module和公共类是相互可见的。

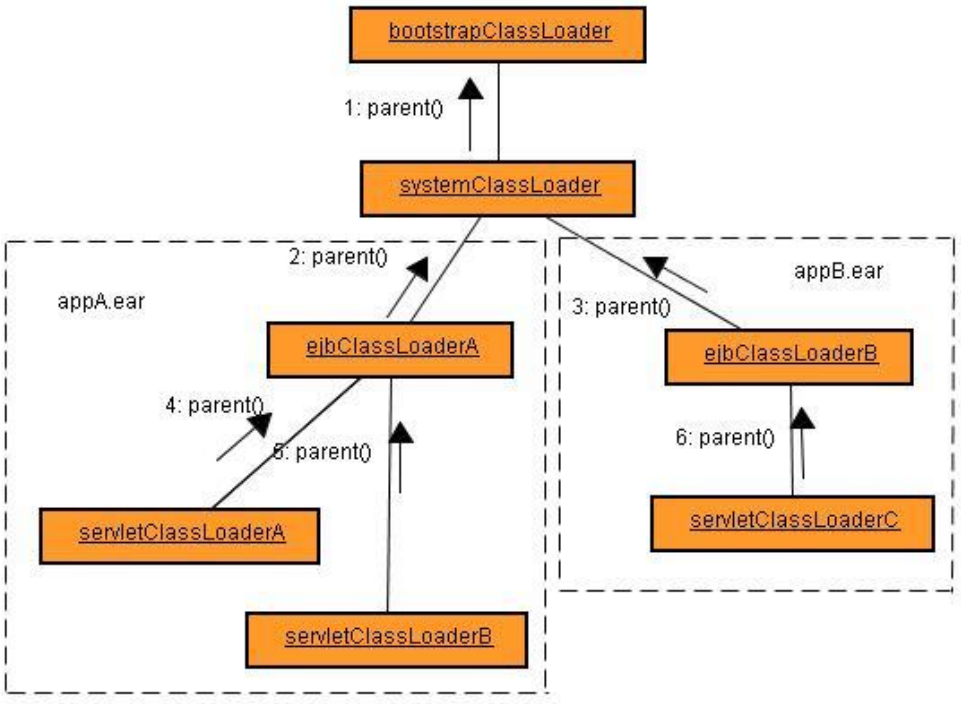
不同的应用，其EJBClassLoader实例也不同，且每个EJBClassLoader实例间是平级关系，所以不同应用中的类是相互不可见的。

- ServletClassLoader

在Apusic应用服务器中，每个Web module都有一个ServletClassLoader，用于装载Web module中的类和资源文件。所以，每个JavaEE应用中都可能有一个或多个ServletClassLoader，例如上图表示的JavaEE应用就有两个ServletClassLoader，它们是平级关系，所以Web module中的类相互不可见。对于ServletClassLoader，还有一些特殊的行为，将在下一节介绍。

- 层次结构

通过以上的介绍，我们可以知道，Apusic应用服务器启动后，假设其中部署了两个应用，分别是appA.ear和appB.ear，那么其Classloader层次结构可表现为：



其中，我们可以看到，EJBClassLoader是ServletClassLoader的父，由ejbClassLoaderA装载的类和文件，对于servletClassLoaderA和servletClassLoaderB装载的类都是可见的。也就是说，同一个应用中的任意Web Module的类（即位于WEB-INF/classes、WEB-INF/lib中的类），都可以使用ejb jar或util jar中的类。

但对于上图中ejbClassLoaderA装载的类，servletClassLoaderC是看不见的，它们属于不同的应用。

6.3.2.3 ServletClassLoader的特性

在前面几节，我们提到过Apusic对于Web module中的类，包括jsp（最终被应用服务器解析成servlet并编译成Java类）、WEB-INF/classes和WEB-INF/lib 里边的class和资源文件，专门定义一个ServletClassLoader进行加载是为了满足类似开发期中类的动态加载、不同Module间类的隔离等的需要。Apusic应用服务器在Classloader体系中做了充分的考虑以降低Web应用开发的复杂性及提升应用服务器的易用性。下面将介绍Apusic的ServletClassLoader的行为特性：

6.3.2.3.1 类的动态加载

在Apusic检测到jsp或WEB-INF/classes目录下的类的更新后，会重新加载修改过的类。对于用户来说，不需要做任何事情，在修改完后马上调用该类就可以看到刚刚做的更新。

考虑到运行期和开发期的要求不同，运行期类和资源文件不会频繁更新，因此，在运行期，不需要频繁检测类文件是否已经更新，可通过配置apusic.conf 中的ServletReloadCheckInterval属性值来修改检测时间。当值小于"0"时，不检测。此值默认是3，即每3秒中检测一次。

6.3.2.3.2 SERVLETCLASSLOADER的多层结构

ServletClassLoader是一层壳，根据配置的不同策略，委托给不同的Classloader执行装载任务。Servlet Classloader的装载行为有两种策略，可通过配置进行指定，配置有两种方式：

- 在web.xml中增加Context Parameter

```
<context-param>
    <param-name>com.apusic.web.ServletClassLoaderDelegate</param-name>
    <param-value>Separated</param-value>
</context-param>
```

这样的配置有效范围只有当前应用。如果修改的是$DOMAIN_HOME/config/web.xml下的配置，则适用所有应用。

- 通过VM参数指定

```
-Dcom.apusic.web.ServletClassLoaderDelegate=Separated
```

这种系统属性配置，所有的应用都生效。

ServletClassLoader的两种装载策略分别通过com.apusic.web.ServletClassLoaderDelegate的两个值来指定：

- Composite

默认值，表示ServletClassLoader的行为委托给了两层Classloader，一层叫CompositeClassLoader，它的父Classloader是EJBClassLoader，它负责WEB-INF/lib和WEB-INF/classes目录下的类和资源的装载，其中，如果在WEB-INF/lib和WEB-INF/classes下有同名的类或资源，WEB-INF/classes下的类将被优先装载；另一层是JSPClassLoader，它的父Classloader是CompositeClassLoader，它负责装载解析编译后的JSP。

- Separated

表示ServletClassLoader的行为委托给了三层Classloader，跟上一种策略不同的是WEB-INF/lib下类和WEB-INF /classes目录下的类和资源由不同的Classloader装载，前者叫StaticClassLoader，它的父是EJBClassLoader；后者我们称为ReloadableClassLoader，它的父是StaticClassLoader，子是JSPClassLoader。根据前面对Classloader父子关系的描述，我们可以知道，WEB-INF/lib下的类看不见WEB-INF /classes下的类，而WEB-INF/classes下的类可以看见WEB-INF/lib下的类。考虑到客户应用中，资源文件一般放在WEB-INF/classes目录中，因此，如果在WEB-INF/lib和WEB-INF/classes下有同名的资源文件，仍然是WEB-INF /classes下的资源优先装载。

如果客户应用系统中，WEB-INF/lib下的类会引用WEB-INF/classes下的类或资源，或者认为WEB-INF/classes下的类应该优先于WEB-INF/lib下的类装载，我们建议使用Composite，即默认的策略。

如果考虑到在开发期WEB-INF/lib下的类或文件不会频繁更新，为了避免检测范围太大而导致的检测时间过长，不扫描WEB-INF/lib下的更新（即此目录下的类只被装载一次，如果有更新，则需要重启应用才能生效），或者认为WEB-INF/lib下的类应该优先于WEB-INF/classes 下的类装载时，可采用Separated策略。

6.3.2.3.3 SESSION中对象的类动态装载

如果session中保存的对象实例的类发生了更改，且类的签名未发生变化，那么对象实例的类型信息将被标识为新装载的类，从Session中取出对象后，它的行为按更新后的类执行。但如果类的签名发生了变化，那么此session中的对象实例将被丢弃。

6.3.2.3.4 类装载的WEB优先策略

在默认情况下，ServletClassLoader遵循大多数Classloader的装载行为，如"Classloader的基本概念"一节描述的那样，会按顺序向上询问其所有父节点装载，如果父没装载到，才会由自身进行加载。这种默认的Java类装载机制有时也会碰到麻烦，比如WEB-INF/classes中有某个类，在系统Classpath中有这个类的另一个版本，Classloader默认的装载行为决定了系统Classpath中的类会被优

先加载。如果我们期望WEB-INF/classes中的类要优先加载，Apusic的Servlet Classloader提供了机会，可以通过配置系统属性或者在web.xml中增加Context Parameter来达到此目的：

- 在web.xml中增加Context Parameter

```
<context-param>
    <param-name>apusic.prefer.war.classes</param-name>
    <param-value>true</param-value>
</context-param>
```

这样的配置有效范围只有当前应用。如果修改的是$DOMAIN_HOME/config/web.xml下的配置，则适用所有应用。

- 通过VM参数指定

```
-Dapusic.prefer.war.classes=true
```

这种系统属性配置，所有的应用都生效。

6.3.2.4 类装载查看服务

Apusic应用服务器提供了类装载查看服务，通过此服务，可以查找指定的类是由哪一层的Classloader装载的，类文件路径等信息，从而可以协助排查一些跟类装载相关的问题。类装载查看服务的相关配置段如下：

```
...
<SERVICE CLASS="com.apusic.util.ClassLoaderViewer">
</SERVICE>

...
```

可以通过Admin Console上提供的类加载器来访问类装载查看服务，如何使用类加载器请参考Admin Console文档。

6.4 日志服务

日志服务是服务器系统中的一个主要工具。通过日志服务，可以对服务器系统中出现的问题、失效以及服务器的执行效率进行跟踪。Apusic应用服务器提供了日志服务以提供对服务器本身的日志进行管理，同时，对于运行于服务器中的应用程序，也提供了一个有效且易用的日志工具。应用组件可以使用JDK标准的logging API输出日志，并且输出结果保存在Apusic的日志文件中。

6.4.1 理解日志服务子系统

Apusic应用服务器日志子系统由以下三个部分组成：

- 接受日志信息输入的日志记录器(Logger)对象；
- 输出日志纪录（LogRecord）到指定目标(如控制台，日志文件)的日志输出(Handler)对象；

- 对日志记录器(Logger)和日志输出(Handler)进行管理的日志管理器(LogManager)对象；

日志子系统进行日志处理的过程概述如下:

1. 日志子系统初始化，初始化一个全局的日志管理器(LogManager)对象，日志管理器对象(LogManager)从域主目录中的config目录下的logging.xml 文件中读取初始化配置。生成全局匿名日志记录器(Global Anonymous Logger)对象，根据配置文件中对日志记录器(Logger)的定义生成相应的日志记录器(Logger)对象； 根据配置文件中对日志输出(Handler)的定义生成相应的全局日志输出(Handler)对象；

	**注意**
	缺省初始化配置中包扩一个匿名的全局日志记录器(Logger)，一个全局的控制台日志输出对象(ConsoleHandler)和一个全局的文件日志输出对象(FileHandler)

2. 其它子系统向特定的日志记录器(Logger)对象输出日志信息；客户应用向应用指定的日志记录器(Logger)对象输出日志信息，如未指定，则输出到全局匿名日志记录器(Global Anonymous Logger)对象；
3. 日志记录器(Logger)对象将为接受到的每个日志信息分配一个描述它的日志记录对象(LogRecord)，然后将此日志记录对象发送给所有的全局日志输出对象(Handler)；
4. 日志输出(Handler)对象将日志记录(LogRecord)格式化输出到控制台，文件，发送到网络日志服务或发送到操作系统日志；

Apusic应用服务器日志子系统的重要组成部分有：

日志管理器(LogManager)

日志级别

日志记录器(Logger)

日志输出(Handler)

6.4.1.1 日志管理器(LogManager)

在启动时，Apusic应用服务器初始化一个全局的日志管理器(LogManager)对象，此全局日志管理器(LogManager)对象用于维护日志记录器(Logger)和日志服务所共享的状态，并且在整个服务器的运行过程中是唯一且固定的：

- 管理日志记录器(Logger)对象的分级命名空间，所有已命名的日志记录器(Logger)对象都存储于此命名空间之中；
- 管理全局日志输出(Handler)对象列表。

日志管理器从位于Apusic应用服务器安装目录中config目录下的配置文件logging.xml读取其初始化配置。

logging.xml配置文件中可使用handler标记装载某个日志输出类文件并注册一个全局日志输出(Handler)对象。每个handler标记可以有零个或多个属性(property)声明，这些属性将通过JavaBean的内省(introspection)机制在初始化时对此Handler对象进行设置。

在日志管理器(LogManager)对日志子系统进行配置时，必须保证所有的日志输出(Handler)类位于系统的类路径中。

对于在logging.xml中声明的日志记录器(Logger)对象，可以按照句点分隔的命名方式进行命名，日志管理器(LogManager)将把这些对象按照其命名组织到一个分级的命名空间中，例如，名字为a.b.c的日志记录器（Logger)对象从属于名字为a.b的日志记录器(Logger)对象，而名字为a.c的日志记录器(Logger)对象与名字为a.b的日志记录器(Logger)对象是同级的关系。

在Apusic应用服务器中，包含多个除了日志子系统之外的其他子系统，其他子系统在日志子系统中注册自己的日志记录器(Logger)分级命名空间，子系统和子系统中的模块向命名空间中相应的日志记录器 （Logger)输出日志信息。这些由应用服务器注册并使用的日志记录器被称为系统日志记录器(Logger)，对应于应用系统注册并使用的应用日志记录器(Logger)。

6.4.1.2 日志级别

Apusic日志系统支持六个日志级别，设置日志级别的目的是用来控制日志的输出。

日志记录器(Logger)对象和日志输出(Handler)对象需要被指定所接受日志请求的日志级别，通过被指定的的级别对输入的日志请求进行过滤，对属于指定级别和高于指定级别的日志请求进行处理，而低于指定级别的日志请求将被忽略。

按照由低到高的顺序，各级别排列如下表：

级别	描述
DEBUG	调试级别日志信息
INFO	报告类信息
NOTICE	常规但重要的信息
WARNING	警告类信息
ERROR	错误类信息
FATAL	重大情况信息(最高级别)

如为日志记录器(Logger)指定了某个级别，如NOTICE，则对于NOTICE, WARNING,ERROR或FATAL级别的日志请求将会被处理，而INFO和DEBUG级别的请求将被日志记录器(Logger)忽略。

另外，可以将日志级别设置为OFF以关闭对日志请求的处理，也可将日志级别设置为ALL输出所有的日志信息。

6.4.1.3 日志记录器对象(Logger)

对于特定系统或应用组件，可使用日志记录器对象进行日志信息的记录。日志记录器通常需要命名，且通常使用句点分隔的分级命名方式。另外，创建不从属于日志记录器(Logger)命名空间的匿名日志记录器(Logger)也是允许的。

对于输入到特定日志记录器对象(Logger)的日志信息，如果此日志记录器(Logger)对象已登记了相应的日志输出（Handler)对象，则日志信息将被输出到被登记的所有日志输出对象，再由相应的日志输出(Handler)对象输出到相应的目标，如控制台，日志文件等等。

6.4.1.4 日志输出(Handler)

日志输出(Handler)从日志记录器(Logger)接受日志记录(LogRecord)并输出到指定的目标，如日志文件，控制台，操作系统日志等等。

通常每个日志输出对象将伴随一个过滤器(Filter)对象和一个格式化(Formatter)对象，过滤器对象用于决定是否接受日志纪录输入的逻辑，格式化对象用于将日志记录(LogRecord) 转化为字符串形式的日志信息。缺省状态下采用com.apusic.logging.SimpleFormatter类作为缺省的Formatter，并且缺省状态下不使用过滤器。

Apusic应用服务器日志子系统提供五个日志输出类:

- 控制台日志输出(ConsoleHandler)

本输出将把日志信息输出到System.err对象，缺省状态下字符集采用平台缺省字符集

- 日志文件输出 (FileHandler)

本输出将把日志信息输出到指定的文件或者一个循环的文件集合

对于循环的文件集合，当对某个文件的输出达到了了指定限制，此文件将被关闭，新的接收日志输入的文件将被打开，旧的日志文件将会在基本日志文件名后被连续地加上如：1,2,3等数字标识。

缺省状态下日志文件输出采用了缓冲机制，但在每个日志记录完成之后即被刷新。

缺省状态下，com.apusic.logging.SimpleFormatter类被用于缺省格式化对象。

- 端口日志输出(SocketHandler)

日志记录被输出到指定的网络端口的输出流，缺省状态下使用com.apusic.logging.XMLFormatter作为格式化对象。

缺省状态下对端口的输出采用了缓冲机制，但每个日志记录完成之后即被刷新。

- 按日期文件日志输出(DateFileHandler)

类名：com.apusic.logging.DateFileHandler

本输出将把日志信息输出到指定的文件，文件名称包含日期信息，一天默认产生一个文件。也可以通过修改配置limit，控制文件的大小，此时必须设置strict属性为false(默认值为true)。

- WEB控制台日志输出(WebConsoleHandler)

类名：com.apusic.tools.admin.WebConsoleHandler

本输出将把日志信息输出到WEB控制台。如果要实现在WEB管理控制台对日志进行实时的监控，需要开启这个日志输出，反之不需要。

6.4.2 管理与配置

6.4.2.1 编辑logging.xml文件

日志子系统的配置文件logging.xml在域主目录下的config目录中，此配置文件中定义了一个全局的控制台日志输出对象，一个全局的文件日志输出对象，还有一个匿名的全局日志记录器对象。

本配置文件的文档类型定义(DTD)，定义了如何编写合法的logging.xml，文档类型定义(DTD)文件和参考可参见DTD reference。

	**注意**
	缺省的日志配置文件logging.xml中定义了一个全局匿名的日志记录器(Logger)对象，此对象的缺省日志级别为INFO，如果其他的日志记录器未指定日志级别，则采用上述的全局匿名日志记录器(Logger)的日志级别对其进行初始化设置。

日志文件输出可配置属性概述:

属性	描述	缺省值
level	日志输出(Handler)对象的日志级别	ALL
filter	指定使用的过滤器(Filter)类名称	空
formatter	指定使用的格式化(Formatter)类名称	com.apusic.logging.SimpleFormatter
encoding	指定使用的输出字符集	平台缺省字符集
limit	指定对每个日志文件可输出的最大近似大小(单位为字节)。如值为零，则文件可接受的输入没有限制	0
count	指定循环文件集合的大小	1
pattern	指定日志文件名的产生模式，（见下文）	%h/logs/apusic.log

一个名字模式(pattern)可包含由以下标记组成的字串：

"/" 文件路径分隔符

"%t" 系统临时目录

"%h" 系统属性中"com.apusic.home"或"user.home"的值

"%g" 日志文件组的标识数字，按"0,1,2,..."序列产生

"%%" %字符

如：

```
%h/logs/apusic%g.log
```

加如%g 部分未指定并且日志输出文件的数量已大于count 值，新的日志文件将会在文件名后加上"."和文件标识数字。如一个模式为"%t/java%g.log"并且日志文件数量被设置为2 的设置在window95 平台将会输出到c:\temp\java0.log 和c:\temp\java1.log 文件。

端口日志输出可配置属性概述：

属性	描述	缺省值
level	日志输出(Handler)对象的日志级别	ALL
filter	指定使用的过滤器(Filter)类名称	空
formatter	指定使用的格式化(Formatter)类名称	com.apusic.logging.XMLFormatter
encoding	指定使用的输出字符集	平台缺省字符集
host	指定需要连接的目标主机名称	(无缺省值)
port	指定需要使用目标TCP端口	无缺省值)

6.4.2.2 记录客户端IP

在Apusic输出的log中，如果是由客户端请求引发的异常，会显示客户端请求的IP。要使用这个功能，需要在启动应用服务器时加上Apusic特有的JVM参数-Dapusic.log.clientIP=true，这个参数默认情况下为false。

```
java -Dapusic.log.clientIP=true ...
```

6.5 基于JMX的可植入服务配置

6.5.1 理解JMX

JMX™规范(Java™ Management extension)定义了Java语言中的应用管理和网络管理方面的架构、设计模式和一套API。 JMX规范提供了Java开发者使用Java语言，创建巧妙的Java代理(agents)，实现分布式的管理中间件和管理器，并且将他们平滑整合到已有系统的方法。

按照JMX架构，JMX服务组件分为三层：

- 工具层(instrumentation level)

JMX可管理资源工具(instruments)组件。JMX可管理资源组件可以是应用系统、某种服务的实现、硬件设备等等，是使用Java开发或者至少提供一个Java语言的包装器（wrapper），并且符合工具层组件的规则以便可由JMX服务进行管理。一般，这一类给定资源的工具以一个或多个MBean的方式提供，详细信息可参考Sun的JMX规范。

- 代理层(agent level)

代理层直接控制资源并提供远程的管理应用程序访问此资源的能力。通常情况下，代理层的组件与其控制的资源处于相同位置。

- 分布式服务层(distributed services level)

分布式服务层提供应用程序通过代理(agent)透明地与被管理的资源交互的功能，并且暴露代理(agent)及其MBean的管理视图、提供分布式管理和安全保证等功能。

例如一个Java语言实现的某种服务，可以通过提供一个按照JMX规范实现的MBean，通过将此MBean注册到服务管理器后，应用程序即可通过服务管理器和代理（组件）取得对此服务的控制。

Apusic 应用服务器提供了对JMX规范中服务层的实现和支持，因此，对于Apusic应用服务器中的服务可以通过提供MBean的方式植入其它非Apusic应用服务器提供的服务，如消息服务，事务服务等等，同时，运行于应用服务器之上的应用程序亦可通过Apusic应用服务器提供的管理系统实现对应用的管理。

6.5.2 可植入服务命名

一般，植入到应用服务器的服务对应于一个或多个MBean，为区分每个注册到JMX管理服务的MBean，每个MBean都拥有一个标识自身的名字。管理服务通过此名字来区分执行管理操作时操作的目标MBean。根据JMX规范，这类名字由两个部分组成：

- 域名部分（domain name)
- 一个或多个无序的标识属性(key attribute)

6.5.2.1 域名(domain name)

域名是一个区分大小写的字符串，它提供了一个全局的管理方案或代理(agent)内的命名空间。域名的命名方式是独立于管理服务的，可以由除冒号、逗号、等于号、星号或问号(即：:,=*?）之外的其他字母组成。

6.5.2.2 标识属性(key attribute)

标识属性用于在指定域中为MBean分配独有的名称标识。一个标识属性是一个名值对，即由属性名和属性值组成。

6.5.2.3 命名的字符串表示

通常，命名的字符串表示如下：

```
[domainName]:property=value[,property=value]*
```

例如，Apusic应用服务器配置文件apusic.conf中MBean组件MUXEndpoint的命名如下：

```
Endpoint:type=mux,service=httpd
```

表示此MBean属于Endpoint域，有两个标识属性，一个名为type、值为mux的属性和一个名为service值为httpd的属性，标识属性间使用逗号分隔。

6.5.3 配置和管理可植入服务

Apusic应用服务器的管理系统通过读取服务器域主目录中config子目录下的配置文件apusic.conf加载各项服务，通过在此文件中增加使用SERVICE标记申明的元素即可声明可植入服务。

一个SERVICE元素表示一个MBean。SERVICE元素通过一个CLASS属性来声明此MBean的完整的类名和包名，并通过一个名为NAME的属性声明此MBean的名字；并且通过一个或多个子ATTRIBUTE标记来指定此MBean的属性值。属性值的声明是跟声明顺序无关的。

以下是声明一个名为com.apusic.samples.jmx.SimpleStandar的MBean示例：

```
...
<SERVICE CLASS="SimpleStandard" NAME="Test:name=testMbean">
    <ATTRIBUTE NAME="num" VALUE="1" />
</SERVICE>

...
```

管理服务在初始化MBean时，将"1"赋值给名为num的域。

	**注意**
	通过SERVICE标记的子标记ATTRIBUTE声明MBean的初始属性值时，对应的MBean的域的类型必须是JMX规范中规定的通用数据类型。详细内容可参考。

6.6 线程池服务配置

线程池服务为com.apusic.util.ThreadPoolService。用到线程池的服务有Muxer,WebService,JMSServer。如果没有为某个特定服务配置线程池，则使用默认的公共线程池，即名为apusic:service=ThreadPool,name=default的线程池。

6.6.1 默认的公共线程池配置

```
...
<SERVICE CLASS="com.apusic.util.ThreadPoolService"
NAME="apusic:service=ThreadPool,name=default">
    <ATTRIBUTE NAME="MinSpareThreads" VALUE="5" />
    <ATTRIBUTE NAME="MaxSpareThreads" VALUE="30" />
    <ATTRIBUTE NAME="MaxThreads" VALUE="-1" />
    <ATTRIBUTE NAME="MaxQueueSize" VALUE="500" />
    <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
</SERVICE>

...
```

属性	描述	值类型	缺省值
MinSpareThreads	此属性决定线程池中的最小备用线程数	整型	5
MaxSpareThreads	此属性决定线程池中的最大备用线程数	整型	30
MaxThreads	此属性决定线程池所能提供的最大线程数，-1表示不限数量	整型	-1
MaxQueueSize	此属性决定等待获取线程队列的最大长度	整型	500
IdleTimeout	等待超时时间。当线程池中的某个线程等待被使用的实际时间超过此属性数值时，线程池自动关闭此线程	整型，单位为秒	300

6.6.2 多路复用线程池配置

```
...
<SERVICE CLASS="com.apusic.util.ThreadPoolService"
NAME="apusic:service=ThreadPool,name=MuxHandler">
    <ATTRIBUTE NAME="MinSpareThreads" VALUE="0" />
    <ATTRIBUTE NAME="MaxSpareThreads" VALUE="10" />
    <ATTRIBUTE NAME="MaxThreads" VALUE="10" />
    <ATTRIBUTE NAME="MaxQueueSize" VALUE="500" />
```

```
    <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
</SERVICE> ...
```

属性	描述	值类型	缺省值
MinSpareThreads	此属性决定线程池中的最小备用线程数	整型	0
MaxSpareThreads	此属性决定线程池中的最大备用线程数	整型	10
MaxThreads	此属性决定线程池所能提供的最大线程数，-1表示不限数量	整型	10
MaxQueueSize	此属性决定等待获取线程队列的最大长度	整型	500
IdleTimeout	等待超时时间，当线程池中的某个线程等待被使用的实际时间超过此属性数值时，线程池自动关闭此线程	整型，单位为秒	300

6.6.3 WebServer线程池配置

```
...
<SERVICE CLASS="com.apusic.util.ThreadPoolService"
NAME="apusic:service=ThreadPool,name=HTTPHandler">
    <ATTRIBUTE NAME="MinSpareThreads" VALUE="5" />
    <ATTRIBUTE NAME="MaxSpareThreads" VALUE="30" />
    <ATTRIBUTE NAME="MaxThreads" VALUE="30" />
    <ATTRIBUTE NAME="MaxQueueSize" VALUE="500" />
    <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
</SERVICE>

...
```

属性	描述	值类型	缺省值
MinSpareThreads	此属性决定线程池中的最小备用线程数	整型	5
MaxSpareThreads	此属性决定线程池中的最大备用线程数	整型	30
MaxThreads	此属性决定线程池所能提供的最大线程数，-1表示不限数量	整型	30
MaxQueueSize	此属性决定等待获取线程队列的最大长度	整型	500
IdleTimeout	等待超时时间，当线程池中的某个线程等待被使用的实际时间超过此属性数值时，线程池自动关闭此线程	整型，单位为秒	300

6.6.4 自动调优线程池配置

```
...
<SERVICE CLASS="com.apusic.util.SelfTuneThreadPoolService"
NAME="apusic:service=ThreadPool,name=HTTPHandler">
    <ATTRIBUTE NAME="MinSpareThreads" VALUE="5" />
    <ATTRIBUTE NAME="MaxSpareThreads" VALUE="30" />
    <ATTRIBUTE NAME="MaxThreads" VALUE="400" />
    <ATTRIBUTE NAME="MaxQueueSize" VALUE="500" />
    <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
</SERVICE>

...
```

属性	描述	值类型	缺省值
MinSpareThreads	此属性决定自动调优线程池中的最小备用线程数	整型	5
MaxSpareThreads	此属性决定线程池中的最大备用线程数	整型	30
MaxThreads	此属性决定自动调优线程池所能提供的最大线程数，-1表示不限数量	整型	30
MaxQueueSize	此属性决定等待获取线程队列的最大长度	整型	500
IdleTimeout	等待超时时间，当线程池中的某个线程等待被使用的实际时间超过此属性数值时，线程池自动关闭此线程	整型，单位为秒	300

在apusic.conf中将想要使用自动调优功能的线程池实现类修改为com.apusic.util.SelfTuneThreadPoolService。MinSpareThreads和MaxThreads用于控制自动调优的区间，其他参数和以前一样。

6.7 管理多路复用

Apusic 应用服务器提供了多种规范中定义的服务，如JMS服务，JNDI服务等等，同时，提供了Apusic特有的Web服务器。一般来讲，每种服务都是相对独立的，应用程序通常通过基于TCP/IP协议的连接对服务进行访问，这样，在对服务器进行管理时，往往需要对每个服务使用的端口号等资源进行设置，在某些情况下，带来了服务器管理的不便。

因此，Apusic应用服务器采用了多路复用技术，以此提高服务器管理的效率。

6.7.1 理解多路复用

Internet的规范中指定了一些协议使用的端口号，通常，这些端口中的大多数都广为人知，但是就应用服务器本身，许多服务都需要指定所使用的端口号，而这些服务所应使用的端口号在现行的Internet相关的规范或标准中没有指定。

通常情况下，应用服务器提供者自行规定这些服务所使用的端口号，加大了应用服务器管理的复杂性。

基于IETF的TCPMUX(RFC1078)，Apusic应用服务器指定了一个多路复用端口（缺省为6888），在此端口上提供多种服务，客户应用只需向多路复用端口发出请求，服务器通过分析客户请求，将其转发给相关的服务或是拒绝此请求。

这样，在管理应用服务器时，对于在多路复用端口上提供的多种服务，只指定一个多路复用端口即可进行管理。

目前，Apusic应用服务器的Web服务、JMS服务、JNDI服务，在一个端口上进行复用；另外，可配置一个安全端口，此端口与上一端口功能一样，只是使用进行SSL通讯。

6.7.2 管理与配置

6.7.2.1 apusic.conf文件中相关配置属性说明

多路复用的配置段在配置文件apusic.conf中，此文件在域主目录下的config目录下。

缺省的在apusic.conf配置文件中的多路复用配置段如下：

```
...
<SERVICE CLASS="com.apusic.net.Muxer">
    <ATTRIBUTE NAME="Port" VALUE="6888" />
    <ATTRIBUTE NAME="Backlog" VALUE="50" />
    <ATTRIBUTE NAME="Timeout" VALUE="300" />
    <ATTRIBUTE NAME="MaxWaitingClients" VALUE="500" />
    <ATTRIBUTE NAME="WaitingClientTimeout" VALUE="5" />
    <ATTRIBUTE NAME="SSLEnabled" VALUE="True" />
    <ATTRIBUTE NAME="SecurePort" VALUE="6889" />
    <ATTRIBUTE NAME="KeyStore" VALUE="config/sslserver" />
    <ATTRIBUTE NAME="KeyPassword" VALUE="keypass" />
</SERVICE>

...
```

目前，提供的多路复用端口的可配置属性概述如下：

属性	描述	值类型	缺省值
Port	多路复用端口号；HTTP、JNDI、JMS服务[注]	0~65535间未被占用的端口号	6888
Backlog	对于未处理的连接请求，可保持在此端口的输入队列中的最大数量	整型	50
Timeout	连接请求的等待时间	整型，单位为秒	300
MaxWaitingClients	表示当服务器过忙而无法及时响应请求时对请求排队的队列长度(针对SSL服务的配置，正常情况下不需要)	整型	500
WaitingClientTimeout	表示当队列已满时经过多长时间将请求抛弃(针对SSL服务的配置，正常情况下不需要)	整型，单位为秒	5

SSLEnabled	是否启用SSL通讯。	“True”或“False”	True
SecurePort	SSL通讯所使用的端口	0~65535间未被占用的端口号	6889
KeyStore	保存服务器密钥和证书的密钥库文件	相对于域的文件路径	config/sslserver
KeyPassword	KeyStore密钥库的管理密码和密钥密码	管理密码和密钥密码必须相同	Keypass
ServerCertificateKey	服务器密钥文件，当不使用KeyStore文件方式时，使用此属性	相对于域的文件路径	无
ServerCertificateChain	对应服务器密钥文件的服务器证书链，以“；”为分隔符，首先是服务器证书，然后是颁发服务器证书的CA证书， 如果还有上层 CA，则依次加上。	相对于域的文件路径	无
NeedClientAuth	是否需要客户端认证	“True”或“False”或“want”	False
MutualAuthPort	双向认证端口	0~65535间未被占用的端口号	6887
TrustStore	信任库文件，它是保存客户端CA证书的密钥库文件	相对于域的文件路径	无
TrustStorePassword	信任库的密码，用于校验文件的完整性，可不填	字符串类型	无
TrustStoreType	信任库类型	字符串类型	JKS
TrustCertificates	客户端CA证书，可填多个，以“；”为分隔符，当不使用TrustStore文件方式时，使用此属性	相对于域的文件路径	无
TrustStorePassword	可信密钥库密码	字符串类型	无
TrustManagerFactoryAlgorithm	可信密钥库算法	字符串类型	无
CRLReloadCheckInterval	CRL检查间隔	整型，单位为秒	33
LookAheadTimeout	预读取超时时间	整型，单位为秒	3
KeyStoreType	密钥库类型	字符串类型	无
KeyManagerFactoryAlgorithm	密钥库算法	字符串类型	无
SSLProtocol	SSL协议类型，默认为TLSv1	字符串类型	TLSv1

Address	多路复用地址	字符串类型	无
TCPNoDelay	对套接口设置TCP_NODELAY禁止Nagle算法,提高数据的实时响应性	“True”或“False”	False
EnableMuxAllow	如果设置为true，表示开启这个功能，默认为false。开启后非http协议能够访问aas,ip地址将在MuxAllowIPs属性设置.	“True”或“False”	False
MuxAllowIPs	设置可以访问非http协议的ip地址，如jconsole访问，默认本机都可以连接，多个ip地址用逗号分隔.如设置EnableMuxAllow为 true后，如果MuxAllowIPs不设置，则默认只有本机能够使用jconsole访问aas，使用jndi获取数据源等功能；如果其他机器访问， 则会被拒绝。如果确认其他机器需要访问，则需要配置ip地址列表.	ip地址	Localhost
DisableHttp	配置是否禁用HTTP协议访问功能	“True”或“False”	False

6.7.2.2 SSL配置

Apusic应用服务器完全支持SSL协议最新版SSL3.0和TLS1.0协议。本文将详细讲述如何在Apusic应用服务器上配置使用SSL双向认证。

6.7.2.2.1 TLS/SSL简介

为了在不安全的网络上安全保密的传输关键信息，Netscape公司开发了SSL协议，后来IETF（Internet Engineering Task Force)把它标准化了，并且取名为TLS，目前TLS的版本为1.0，TLS1.0的完整版本请参考rfc2246（www.ietf.org）。

基于TLS协议的通信双方的应用数据是经过加密后传输的，应用数据的加密采用了对称密钥加密方式，通信双方通过TLS握手协议来获得对称密钥。为了不让攻击者偷听、篡改或者伪造消息，通信的双方需要互相认证，来确认对方确实是他所声称的主体。TLS握手协议通过互相发送证书来认证对方，一般来说只需要单向认证，即客户端能确认服务器便可。但是对于对安全性要求很高的应用往往需要双向认证，以获得更高的安全性。下面详细讲述建立自己的认证机构，并且利用它来颁发服务器证书和客户端个人证书，然后配置服务器来使用双向认证。

6.7.2.2.2 建立自己的认证授权机构CA

用户可以向可信的第三方认证机构（CA）申请证书，也可以自己做CA，由自己来颁发证书，在本文中将讲述如何自己做证书颁发机构。本文所使用的CA软件为Openssl。Openssl用来产生CA证书、证书签名并生成浏览器可导入的PKCS#12格式个人证书。你可以在Openssl的官方网站http://www.openssl.org下载最新版的Openssl，本文使用的是 Openssl 0.9.7d。

从Openssl的官方网站下载并安装了Openssl之后，设置系统环境变量Path指向Openssl的bin目录，下面开始创建CA根证书：

建立工作目录：

```
mkdir ca
cd ca
```

生成ca私钥：

```
openssl dsaparam -out dsaparam 1024
openssl gendsa -out dsakey dsaparam
```

生成ca待签名证书：

```
openssl req -new -out ca-req.csr -key dsakey
```

用CA私钥进行自签名，得到自签名的CA根证书：

```
openssl x509 -req -in ca-req.csr -out ca-cert.cer -signkey dsakey -days 365
```

6.7.2.2.3 生成服务器端证书

服务器端证书用来向客户端证明服务器的身份，也就是说在SSL协议握手的时候，服务器发给客户端的证书。生成服务器证书时用到了JDK的密钥管理工具Keytool，本文采用的jdk是sun jdk1.5.0。可以在Sun公司的网站http://java.sun.com下载jdk1.5.0。

建立工作目录：

```
cd ..
mkdir server
cd server
```

生成服务器私钥对及自签名证书，并且保存在密钥库mykeystore中：

```
keytool -genkey -alias myserver -keyalg RSA -keysize 1024 -keypass keypass -
storepass keypass -dname "cn=localhost, ou=dev,o=apusic,l=Shenzhen,
st=guangdong, c=CN" -keystore mykeystore
```

生成服务器待签名证书：

```
keytool -certreq -alias myserver -sigalg SHA1withRSA -file server.csr -
keypass keypass -storepass keypass -keystore mykeystore
```

请求CA签名服务器待签名证书，得到经CA签名的服务器证书：

```
openssl x509 -req -in server.csr -out server-cert.cer -CA ..\\ca\\ca-cert.cer
-CAkey ..\\ca\\dsakey -days 365 -set_serial 02
```

把CA根证书导入密钥库 mykeystore：

```
keytool -import -alias caroot -file ..\\ca\\ca-cert.cer -noprompt keypass
keypass -storepass keypass -keystore mykeystore
```

把经过CA签名的服务器证书导入密钥库mykeystore：

```
keytool -import -alias myserver -file server-cert.cer -noprompt keypass
keypass -storepass keypass -keystore mykeystore
```

把CA根证书导入信任库truststore：

```
keytool -import -alias caroot -file ..\\ca\\ca-cert.cer -noprompt keypass
keypass -storepass keypass -keystore truststore
```

6.7.2.2.4 颁发并发布个人证书

个人证书用来向服务器证明个人的身份，也就是说在SSL协议握手的时候，客户端发给服务器端的证书。同时个人证书中包含个人信息如用户名等，如果需要，这个用户名将作为登录服务器的用户名。

建立工作目录：

```
cd ..
mkdir client
cd client
```

生成客户端私钥：

```
openssl genrsa -out clientkey 1024
```

生成客户端待签名证书：

```
openssl req -new -out client.csr -key clientkey
```

请求CA签名客户端待签名证书，得到经CA签名的客户端证书：

```
openssl x509 -req -in client.csr -out client.cer -CA ..\\ca\\ca-cert.cer -
CAkey ..\\ca\\dsakey -days 365 -set_serial 02
```

生成客户端的个人证书client.p12：

```
openssl pkcs12 -export -clcerts -in client.cer -inkey clientkey out
client.p12
```

CA根证书导入客户端：

在这里CA的根证书用来在SSL握手时验证服务器发给客户端浏览器的证书。如果没有此证书，浏览器将无法自动验证服务器证书，因此浏览器将弹出确认信息，让用户来确认是否信任服务器证书。在客户端的IE中使用"工具 -> Internet选项 -> 内容 -> 证书 ->导入"把我们生成的CA根证书ca\ca-cert.cer导入，使其成为用户信任的CA。

客户端个人证书导入客户端：

在客户端的IE中使用"工具 -> Internet选项 -> 内容 -> 证书 ->导入"把我们生成的CA根证书client.p12导入，使其成为用户信任的CA。

6.7.2.2.5 配置服务器允许双向认证

Apusic应用服务器默认配置下不支持双向认证，要支持SSL双向认证，需要对配置作如下修改：

修改Muxer服务：

如果采用密钥库文件的方式，示例配置如下：

```
<SERVICE CLASS="com.apusic.net.Muxer">
    <ATTRIBUTE NAME="Port" VALUE="6888" />
    <ATTRIBUTE NAME="Backlog" VALUE="50" />
    <ATTRIBUTE NAME="Timeout" VALUE="300" />
    <ATTRIBUTE NAME="MaxWaitingClients" VALUE="500" />
    <ATTRIBUTE NAME="WaitingClientTimeout" VALUE="5" />
    <ATTRIBUTE NAME="SSLEnabled" VALUE="True" />
    <ATTRIBUTE NAME="SecurePort" VALUE="6889" />
    <ATTRIBUTE NAME="MutualAuthPort" VALUE="6887" />
    <ATTRIBUTE NAME="NeedClientAuth" VALUE="True" />
    <ATTRIBUTE NAME="KeyStore" VALUE="config/keystore" />
    <ATTRIBUTE NAME="KeyPassword" VALUE="keypass" />
    <ATTRIBUTE NAME="TrustStore" VALUE="config/truststore" />
    <ATTRIBUTE NAME="TrustStorePassword" VALUE="keypass" />
    <ATTRIBUTE NAME="TrustStoreType" VALUE="JKS" />
</SERVICE>
```

如果采用证书文件的方式，示例配置如下：

```
<SERVICE CLASS="com.apusic.net.Muxer">
    <ATTRIBUTE NAME="Port" VALUE="6888" />
```

```
    <ATTRIBUTE NAME="Backlog" VALUE="50" />
    <ATTRIBUTE NAME="Timeout" VALUE="300" />
    <ATTRIBUTE NAME="MaxWaitingClients" VALUE="500" />
    <ATTRIBUTE NAME="WaitingClientTimeout" VALUE="5" />
    <ATTRIBUTE NAME="SSLEnabled" VALUE="True" />
    <ATTRIBUTE NAME="SecurePort" VALUE="6889" />
    <ATTRIBUTE NAME="MutualAuthPort" VALUE="6887" />
    <ATTRIBUTE NAME="NeedClientAuth" VALUE="True" />
    <ATTRIBUTE NAME="ServerCertificateKey" VALUE="config/192.168.6.191.key"
/>
    <ATTRIBUTE NAME="ServerCertificateChain"
VALUE="config/192.168.6.191.cer;config/serverca.cer" />
    <ATTRIBUTE NAME="TrustCertificates" VALUE="config/clientrootca.cer" />
</SERVICE>
```

6.7.2.2.6 测试双向认证

访问应用，如果前面的操作都正确的话，应该可以看到Apusic的欢迎页面。同时状态栏上的小锁处于闭合状态，且证书为有效状态，表示您已经成功地与服务器建立了要求客户端验证的SSL安全连接。

6.7.2.3 国密配置

Apusic应用服务器支持国密算法SM2、SM3以及SM4，并可用于客户端方式访问基于国密算法的HTTPS协议。

6.7.2.3.1 配置

需在apusic.conf配置文件中的Muxer服务中添加SSLProtocol属性，设置值为："SMv1.1"(旧版本为SM1.1), 并分别配置秘钥库和信任库。

```
<SERVICE CLASS="com.apusic.net.Muxer">
    <ATTRIBUTE NAME="Port" VALUE="6888"/>
    <ATTRIBUTE NAME="Backlog" VALUE="1024"/>
    <ATTRIBUTE NAME="Timeout" VALUE="30"/>
    <ATTRIBUTE NAME="LookAheadTimeout" VALUE="30"/>
    <ATTRIBUTE NAME="MaxWaitingClients" VALUE="200"/>
    <ATTRIBUTE NAME="WaitingClientTimeout" VALUE="5"/>
    <ATTRIBUTE NAME="SecurePort" VALUE="6889"/>
    <ATTRIBUTE NAME="MutualAuthPort" VALUE="6887"/>
    <ATTRIBUTE NAME="NeedClientAuth" VALUE="false"/>

    <ATTRIBUTE NAME="SSLEnabled" VALUE="true"/>
    <ATTRIBUTE NAME="SSLProtocol" VALUE="SMv1.1"/>

    <ATTRIBUTE NAME="KeyStore" VALUE="config/keystore.p12"/>
```

```
    <ATTRIBUTE NAME="KeyStoreType" VALUE="PKCS12"/>
    <ATTRIBUTE NAME="KeyPassword" VALUE="123456"/>

    <ATTRIBUTE NAME="TrustStore" VALUE="config/trust.p12"/>
    <ATTRIBUTE NAME="TrustStoreType" VALUE="PKCS12"/>
    <ATTRIBUTE NAME="TrustStorePassword" VALUE="123456"/>
</SERVICE>
```

6.7.2.3.2 启动

使用**./startapusic.sh gm** 以国密模式启动AAS应用服务器。

6.7.2.3.3 访问

需要使用支持国密模式访问的浏览器访问，如360安全浏览器、红莲花浏览器等。使用HTTPS访问，如https://IP:6889/admin。

6.8 管理Apusic Web服务器

为更好提供基于Java的分布式企业应用，Apusic应用服务器中内置了一个高效、安全、可靠的Web服务器，提供完整的Web服务器的功能，使Apusic应用服务器在提供对静态页面和图形的同时，提供对JSP、Servlet的高效服务。

Apusic应用服务器中的Web服务器支持HTTP1.1标准。

6.8.1 管理与配置

Apusic安装目录下的config目录中，配置文件apusic.conf包含了Web服务器的配置段。缺省的配置段如下:

```
...
<SERVICE CLASS="com.apusic.servlet.http.WebService">
    <ATTRIBUTE NAME="MaxWaitingClients" VALUE="500" />
    <ATTRIBUTE NAME="WaitingClientTimeout" VALUE="5" />
    <ATTRIBUTE NAME="KeepAlive" VALUE="true" />
    <ATTRIBUTE NAME="KeepAliveTimeout" VALUE="15" />
    <ATTRIBUTE NAME="MaxKeepAliveRequests" VALUE="100" />
    <ATTRIBUTE NAME="MaxKeepAliveConnections" VALUE="300" />
    <ATTRIBUTE NAME="ServletReloadCheckInterval" VALUE="3" />
    <ATTRIBUTE NAME="EnableLog" VALUE="False" />
    <ATTRIBUTE NAME="LogFileName" VALUE="logs/access.log" />
    <ATTRIBUTE NAME="LogFileLimit" VALUE="1000000" />
    <ATTRIBUTE NAME="LogFileCount" VALUE="10" />
    <ATTRIBUTE NAME="SendFile" VALUE="False" />
</SERVICE>

...
```

可通过修改配置段中的可配置属性，对Web服务器进行管理。

Web服务器的属性见下表：

属性	描述	值类型	缺失值
MaxWaitingClients	如果当前并发用户数已达到最大值，则客户请求被保持到一个队列中，此参数决定此队列可保持的最大等待客户请求数	整型	500
WaitingClientTimeout	表示当队列已满时经过多长时间将请求抛弃	整型，单位为秒	5
KeepAlive	此属性决定是否保持与当前客户端的连接	true或false	true
KeepAliveTimeout	与当前客户端连接的超时数	整型，单位为秒	15
MaxKeepAliveRequests	客户端请求被保持到一个请求队列，此属性用于决定请求队列可保持的最大客户端请求数	整型	100
MaxKeepAliveConnections	表示系统同时保存的最大连接个数，超过这一个数时最近最少被使用的连接将被关闭	整型	300
ServletReloadCheckInterval	检查Servlet 是否被修改的时间间隔。如果Servlet的类被修改过，Apusic会重新装载此类。在开发环境下，建议将此值设置成3。此参数的值对性能影响较大，因此在生产环境下，建议将此值设置成-1(即不检查)或一个相对比较大的值。	整型，单位为秒	3
EnableLog	是否启用访问日志	true或false	false
LogFileName	相对于Apusic应用服务器安装目录的路径及日志记录文件名	字符串	logs/access.log
LogFileLimit	指定对每个日志文件可输出的最大近似大小(单位为字节)。如值为零，则文件可接受的输入没有限制	整型	0
LogFileCount	指定日志文件循环集合的大小	0~10	10
LogHandler	日志输出对象。目前支持"java.util.logging.FileHander"和 "com.apusic.logging.DateFileHandler"。使用前者时，根据"LogFileLimit"的决定一个文件的大小；当使用后者时，每天的访问日志输出到一个单独的日志文件。	字符串	java.util.logging.FileHander
LogFormat	日志输出格式。用于将日志记录转化为字符串形式的日志信息。	字符串	"%a %l %u %t %r %s %b"

FileCacheSize	文件缓存空间大小。当请求的静态文件未发生过变化时，返回缓存的文件，默认的缓存空间为1024KB。	整型，单位是KB	1024
EnableGZip	是否开启GZip。	默认值为"false", 合法取值有三种："true", "false", 或 "2048"。最后一种形式是一个数值，表示一个最小长度的字节数，只有当输出内容大于该长度时，才执行压缩。 如果配置为"true"，则最小长度默认值为2048个字节。	false
CompressableMimeTypes	开启GZip可压缩MIME类型	可选配置，只有在开启GZip时才打开此配置，默认值为"text/html,text/xml,text/plain,text/css",其取值为逗号分隔的字符串，表示这些类型的内容才会被压缩。	text/html,text/xml,text/plain,text/css
SendFile	表示对静态文件启用sendFile支持,默认为False	True或False	FALSE

下面是LogFormat目前支持的格式说明：

%a - 远端IP地址

%H - 请求协议

%m - 请求的方法（GET，POST，等）

%q - 查询字符串(如果存在，以 '?'开始)

%s - 响应的状态码

%t - 日志和时间，使用通常的Log格式

%l - 从identd返回的远端逻辑用户名（总是返回 '-'）

%u - 认证以后的远端用户（如果存在的话，否则为'-'）

%U - 请求的URI路径

%b - 发送的字节数，不包括HTTP头，如果为0，使用" - "

%B - 发送的字节数，不包括HTTP头

%A - 本地IP地址

%p - 收到请求的本地端口号

%r - 请求的第一行，包含了请求的方法和URI

%S - 用户的session ID

%v - 本地服务器的名称

%D - 处理请求的时间，以毫秒为单位

%T - 处理请求的时间，以秒为单位

%R - 引用地址

%g - User-agent

默认格式为：

```
%a %l %u %t %r %s %b
```

全格式为：

```
%a %l %u %t %r(%m %U %q %H) %s %b %B %A %p %S %v %R %g %D %T
```

括号内的意思是，如果有%r，则%m %U %q %H不生效。

6.8.2 配置Web应用

Web应用按照Servlet2.5规范进行部署和配置。Servlet2.5规范中描述了如何将静态的HTML内容、图片、JSP和Servlet等组件打包成为一个Web应用，Web应用可以对应用外的资源，如EJB、JSP Taglib等，进行访问。服务器可以部署多个Web应用，并提供运行环境，通常，可以通过在Web应用中定义的URI访问此Web应用。

6.8.2.1 部署Web应用

将打包好的web应用（war包或者目录）放置在Apusic应用服务器域主目录下applications目录中，Apusic应用服务器会自动部署此Web应用。部署Web应用还有其它方式，比如使用管理工具部署，或者直接修改域主目录下config目录中的server.xml文件，增加一个条目，指定Web应用路径（可以是相对applications目录的路径也可以是绝对路径），Apusic应用服务器启动时会加载此条目指定的Web应用。如Default应用的条目是：

```
<application name="default" base="applications/default" start="auto" />
```

server.xml中应用的完整配置项：

```
<application name="bookstore" base="applications/bookstore.war" start="auto"
config="" virtual-host="" base-context="" start="" loadon="" global-
```

```
session="" allowHosts="" denyHosts="" />
```

属性说明：

name:应用的名称。

base:应用的目录，可以是绝对路径，也可以是相对路径。如果是相对路径，则相对于%domain_home%。

start：应用启动的模式，值为auto和demand，auto表示服务器启动，应用就启动；demand则是通过管理员手动启动。

config：指定apusic-application.xml配置文件路径。

virtual-host：虚拟主机，多个虚拟主机使用逗号分割。

base-context：应用的基础上下文，和global-session配合可以到达多个应用共享session的功能(需要共享session的应用设置相同的base-context，而且global-session设置为true)。

loadon：如果存在多个应用，则决定应用加载的顺序，类型为数字，值小则优先加载。

global-session：是否为全局session，同一个企业应用中，多个web应用可以获取到其他应用的session属性值。

allowHosts：允许访问的机器ip列表。

denyHosts ：拒绝访问的机器ip列表。

6.8.2.2 配置web上下文根路径

在Apusic应用服务器中部署的Web应用，提供多种策略指定http访问该应用时的上下文根路径(context-root)。

- apusic-application.xml

参考Apusic的default应用的apusic-application.xml如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE apusic-application PUBLIC '-//Apusic//DTD Apusic Application
4.0.2//EN'
'http://www.apusic.com/dtds/apusic-application_4_0_2.dtd'>
<apusic-application>
    <module uri="public_html">
        <web>
            <context-root>/</context-root>
        </web>
    </module>
</apusic-application>
```

其中，<context-root>指定了该应用的上下文根路径为"/"，即可使用http://<hostname/访问该应用。可在web应用(war包或目录)内建一个META-INF目录，此目录与WEB-INF目录同级，并将apusic-application.xml文件放置在其中，即可生效。

- base-context

在域主目录下config目录中的server.xml文件中，可指定一个应用的base-context，如下：

```
<application name="default" base="applications/default" base-context="/app1"
start="auto"/>
```

当base-context不为空时，如果应用没有配置context-root，那么base-context即为该应用的context-root；如果该应用的web模块配置了context-root，那么该应用所有web模块的context-root会以base-context为前缀。比如app1的base-context为app1，app1下有个web模块的context-root为web1，那么访问该web模块的路径为http://hostname/app1/web1/。

- 应用名即上下文根路径

如果base-context和context-root都未配置，那么apusic会按照以下规则指定web应用的上下文根路径：如果应用是ear模块，那么指定apusic-application.xml中该web module的uri值为根路径；如果应用是web模块，那么web应用的根路径为server.xml中指定的应用名，即"name"属性的值。如果上述值以".war"结尾，那么去除".war"后的值为context-root。

6.8.3 配置虚拟主机

虚拟主机使用户可以配置Web服务器或集群，使某个Web应用只对指定主机名的HTTP请求提供响应。

一般，可以使用DNS指定一个或多个主机名，与主机名对应的Web服务器或集群IP地址，同时，指定被用于虚拟主机服务的Web应用。如果使用集群，负载均衡技术将会为应用提供更高的效率和可用性。

例如，某个Apusic应用服务器上部署了一个名为default的Web应用，当客户端通过浏览器发出对名为www.host.com的域名的请求时，由default应用提供响应。具体方法如下：

- 在网络中的DNS服务器上，配置一个由域名www.host.com到Apusic应用服务器主机IP地址的映射（相关配置方法请参考所使用的DNS服务器的文档）；
- 修改Apusic应用服务器安装目录下，位于config目录下的server.xml文件:

查找server.xml文件中，关于defult应用的配置段，一般，此配置段类似于：

```
...
<server>
    <application name="default" base="applications/default" start="auto" />

...
</server>

...
```

修改此配置段，在其中加入关于虚拟主机的属性定义，如下所示：

```
...
<server>
    <application name="default" base="applications/default" virtual-
host="www.host.com" start="auto" />

...
</server>

...
```

重新启动Apusic应用服务器，此时，通过在浏览器中访问www.host.com的内容应为default应用所提供的内容。

由于Apusic应用服务器中的Web服务器可以部署并运行多个Web应用，因此，可以在一个Apusic应用服务器上提供多个用于虚拟主机服务的Web应用。

例如，在上例的环境中的Apusic应用服务器，新部署了一个名为exampleApp的Web应用，希望使用此Web应用提供对域名www.host1.com的虚拟主机服务，只需重复上文的两个步骤，在DNS服务器中新增一个由域名www.host1.com到Apusic应用服务器主机IP地址的映射，再在server.xml文件中，修改关于exampleApp应用的配置段，然后重新启动应用服务器即可。

6.8.4 WebDAV配置

WebDAV是超文本传输协议(HTTP)的一组扩展，为Internet上计算机之间的编辑和文件管理提供了标准。在Apusic应用服务器中为了能够配置WebDAV，我们需要在应用的web.xml中加入如下内容:

```
...
<servlet>
    <servlet-name>WebDAV</servlet-name>
    <servlet-class>com.apusic.web.webdav.WebDAVServlet</servlet-class>
    <init-param>
        <param-name>readonly</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>showMetaResource</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>resourceRoot</param-name>
        <param-value>/</param-value>
    </init-param>
</servlet>
<servlet-mapping>
```

```
    <servlet-name>WebDAV</servlet-name>
    <url-pattern>/webdav/*</url-pattern>
</servlet-mapping>

...
```

参数	描述	值类型	缺省值
readonly	是否允许进行写操作	"True"或"False"	无
showMetaResource	是否显示JavaEE应用程序的META文件夹	"True"或"False"	无
resourceRoot	WebDAV所操作的根路径	字符串	"/"

6.9 管理Session服务

6.9.1 管理与配置

Apusic域主目录的config目录中，配置文件apusic.conf包含了Session服务的配置段。缺省的配置段如下:

```
...
<SERVICE CLASS="com.apusic.web.session.SessionService">
    <ATTRIBUTE NAME="DefaultSessionTimeout" VALUE="3600" />
    <ATTRIBUTE NAME="MaxSessionsInCache" VALUE="1024" />
    <ATTRIBUTE NAME="SessionInvalidateCheckInterval" VALUE="60" />
    <ATTRIBUTE NAME="Distributable" VALUE="False" />
    <ATTRIBUTE NAME="Replicable" VALUE="False" />
    <ATTRIBUTE NAME="SessionStick" VALUE="False" />
    <ATTRIBUTE NAME="PersisteSession" VALUE="False" />
    <ATTRIBUTE NAME="DestorySessionOnApplicationStop" VALUE="True" />
</SERVICE>

...
```

可通过修改配置段中的可配置属性，对Session服务器进行管理。

Session服务的属性见下表:

属性	描述	值类型	缺失值
DefaultSessionTimeout	默认Session超时时间。当web应用设置的超时时间小于0时，系统采用此值作为默认的Session超时时间。	整型，单位为秒	3600

MaxSessionsInCache	缓存Session空间的大小。当缓存中的数量超过此值时，Session会被交换到文件中。如果maxSessionInCache 小于 0 则所有的session都放在内存里，不会写出或抛弃，sessionStore无效；如果从正数改成负数或者负数改成正数都需要重启服务器，正数的大小在jmx里改动可及时生效。	整型	1024
SessionCreationPolicyOnMax	Override, 忽略最大限制，继续创建新Session，由后台线程负责清除多余的session（至于后台线程清除多余session的策略，由辅助参数 BackgroundPolicyOfOverrideOnMax控制）。这种策略相当于异步执行对超过最大限制的处理，而相对的ExpireOld和 SwapOutOld两种策略都是同步处理; NotAllow, 不允许超过最大限制，如超过，直接抛异常; ExpireOld, 如超过最大限制，先废弃掉多余的旧session（通常并未过期），再创建新的。这种选择暗示着SessionStore在运行中将不被需要，也许只是容器启动或停止时load或unload所有session时才需要SessionStore; SwapOutOld，如超过最大限制，先把内存中多余的旧session交换到持久存储SessionStore中，再创建新的。	Override或NotAllow"或ExpireOld或SwapOutOld	SwapOutOld
BackgroundPolicyOfOverrideOnMax	取值的含义同SessionCreationPolicyOnMax中的解释。	ExpireOld"或SwapOutOld	SwapOutOld
Anycasting	使用Session集群时除了需要配置ClusterService以外，在SessionService里增加了Anycasting选项，这个选项缺省为False，缺省使用广播方式，数据的复制包只发送一次到集群所有节点，如果Anycasting设为True在采用点对点通信，依次发送。	true或false	false
SessionInvalidateCheckInterval	检查Session是否有效的时间间隔	整型，单位为秒	3
Distributable	标识是否允许分布式Session，当此值为“true”时，Session被允许复制到Apusic负载均衡器中，以提供失效恢复功能（需保证Session中的对象实现了序列化接口）。	true或false	false
Replicable	标识是否允许在Apusic集群中复制Session。当此值和Distributable属性都为“true”时，Session被允许在启动了集群服务的Apusic节点中复制，以提供失效恢复功能	true或false	false

	（需保证Session中的对象实现了序列化接口）。		
SessionStick	标识是否提供Session粘滞功能。在Apusic集群中使用第三方的负载均衡器时，如果此值为“true”，则提供客户端与服务器之间的Session粘滞性。	true或false	false
PersisteSession	标识是否在停止Session服务时（比如关闭Apusic服务），将当前的所有Session存储到文件中，以保证在服务器重启后，Session还是可用的。	true或false	false
DestorySessionOnApplicationStop	标识是否在停止应用后销毁该应用相关的所有Session	true或false	true

6.9.2 配置Session序列化

aas session序列化提供了多种方式，有：jdk,fst,kryo,默认使用jdk方式.

在vm.options中配置 apusic.http.session.serializer=jdk 默认使用原有的序列化方式，即jdk方式.

可配置的选项有：jdk,fst,kryo.如需更改配置，直接修改apusic.http.session.serialize的值为可配置的任一选项即可。各序列化方式说明如下：

- jdk

序列化相关的类必须实现Serializable/Externalizable接口;

序列化生成的字节较大，速度较慢，兼容性好，稳定性好。

- fst

序列化相关的类必须实现Serializable/Externalizable接口，兼容jdk序列化(writeObject,readObject,readReplace,validation,putField,getField等等);

序列化生成的字节小，速度快，但目前使用人数较少。

- kryo

序列化的类无需实现Serializable/Externalizable接口, 不兼容jdk序列化;

序列化生成的字节较小，速度快。如果序列化的类没有无参构造函数，将混合使用jdk和kryo序列化。

6.9.3 配置Session存储

Apusic默认的Session存储是采用文件数据库的，当用户想更改Session存储方式时，可以将%DOMAIN_HOME%\config\apusic.conf中的文件数据库存储服务替换成其它存储方式，配置如下：

1. 默认的Session文件存储服务配置：

```
<SERVICE CLASS="com.apusic.web.session.FileSessionStoreService">
    <ATTRIBUTE NAME="Directory" VALUE="store/http_sessions" />
    <ATTRIBUTE NAME="RebuildStoreOnShutDown" VALUE="true" />
    <ATTRIBUTE NAME="RebuildStoreOnMaxFileSize" VALUE="1024" />
    <ATTRIBUTE NAME="RebuildStoreOnSchedule" VALUE="0 0 18 1 * *
*" />
</SERVICE>
```

其中：

Directory：为Sesssion存储目录。

RebuildStoreOnShutDown：是否在AAS关闭的时候进行Rebuild。

RebuildStoreOnMaxFileSize：确定在session文件达到多大的时候，进行Rebuild。单位为MB。

RebuildStoreOnSchedule：用于定时Rebuild，值是一个时间的表达式。

顺序	秒	分	时	日	月	年	周
范围	0-59	0-59	0-23	1-31	1-12		1-7

	提示
	时间表达式的说明每一字段可以是上述范围内的一个整数，或者为“”，*当为一个整数时，表示一个确定的时间，*“”表示该字段任意。各字段用空格隔开。 如：0 0 18 1 * * * 表示每年每月的一号下午6点整。 当最后的周字段被设置时，其会覆盖日字段，表示每周的周几。如：当前日期为2011-6-22，星期三，若时间格式为：“ 0 0 18 20 * * 1” 表示从当前日期，即6-22日开始，每周的周一下午6点整。也就是下一次时间为2011-6-27 18:0:0。

三个策略，可以同时配置，同时生效。也可以配置其中的任意几个。

若三个都没有配置，则使用RebuildStoreOnMaxFileSize策略，默认的session文件大小为1024MB。

2. 嵌入式数据库存储配置：

```
<SERVICE CLASS="com.apusic.web.session.DBSessionStoreService">
    <ATTRIBUTE NAME="Directory" VALUE="store/http_sessions" />
    <ATTRIBUTE NAME="RebuildStoreOnShutDown" VALUE="true" />
    <ATTRIBUTE NAME="RebuildStoreOnMaxFileSize" VALUE="1024" />
    <ATTRIBUTE NAME="RebuildStoreOnSchedule" VALUE="0 0 18 1 * *
*" />
</SERVICE>
```

其中属性同上（文件存储）。

3. 关系数据库存储配置：

```
<SERVICE CLASS="com.apusic.web.session.SQLSessionStoreService">
    <ATTRIBUTE NAME="DataSourceName" VALUE="java:/jdbc/apusic" />
    <ATTRIBUTE NAME="TableName" VALUE="t_httpsession" />
</SERVICE>
```

其中属性DataSourceName为数据源的JNDI名，TableName为Session存储的数据库中表的名称。

4. 分布式存储：

```
<SERVICE CLASS="com.apusic.web.session.CacheSessionStoreService">
    <ATTRIBUTE NAME="ServerAddress"
VALUE="192.168.6.66:11211,192.168.6.67:11211" />
    <ATTRIBUTE NAME="BackupServerAddress"
VALUE="192.168.6.76:11211,192.168.6.77:11211" />
    <ATTRIBUTE NAME="Weights" VALUE="1,2" />
    <ATTRIBUTE NAME="Failover" VALUE="True" />
    <ATTRIBUTE NAME="ConnectionPoolSize" VALUE="1" />
    <ATTRIBUTE NAME="BinaryProtocol" VALUE="False" />
    <ATTRIBUTE NAME="TokyoTyrant" VALUE="False" />
    <ATTRIBUTE NAME="ConsistentHash" VALUE="False" />
</SERVICE>
```

其中：

ServerAddress:分布式Session存储数据库节点列表；形如"节点1:port，节点2:port"的字符串，节点间用逗号隔开。

BackupServerAddress : 备份节点列表，格式同上，备份节点顺序和主节点顺序一一对应。

Weights : 节点权重，大于零。形如："1,2"的字符串，权重间用逗号隔开。

Failover：是否为失效转移模式。当为失效转移模式时，当一个节点挂掉后，它的key值将被分配到其它节点。若为非失效转移时，当某个节点挂掉的时候，不会从节点列表移除，请求也不会转移到下一个有效节点，而是直接将请求置为失败。

ConnectionPoolSize ： NIO连接池大小，默认为1。

BinaryProtocol : 是否使用二进制协议。默认为False。

TokyoTyrant ： 是否使用TokyoTyrant服务器，默认为False。

ConsistentHash : 是否使用一致性哈希，默认为False。

5. redis存储：

```
<SERVICE CLASS="com.apusic.web.session.RedisSessionStoreService">
    <ATTRIBUTE NAME="ConfigPath" VALUE="config/redis_session.properties" />
    <ATTRIBUTE NAME="ConnectionMode" VALUE="single" />
</SERVICE>
```

其中:

ConfigPath是redis客户端的参数配置文件

ConnectionMode表示采用的Redis部署方式。single（单机）、shard（分片） sentinel(哨兵)

redis_session.properties配置:

```
#common config
#maxTotal = 8
#maxWaitMillis = -1
#maxIdle = 8
#minIdle = 0
#testOnBorrow = false
#blockWhenExhausted = true
#evictionPolicyClassName = org.apache.commons.pool2.impl.DefaultEvictionPolicy
#fairness = false
#jmxEnabled = true
#jmxNameBase = pool
#jmxNamePrefix = pool
#lifo = true
#minEvictableIdleTimeMillis = 60000
#numTestsPerEvictionRun = -1
#softMinEvictableIdleTimeMillis = -1
#testOnCreate = false
#testOnReturn = false
#testWhileIdle = true
#timeBetweenEvictionRunsMillis = 30000

# single config
host = 192.168.235.249
port = 6379
#timeout = 2000#
password = 123456
```

公共配置

单机配置

```
# shard config
#shardInfoList = instance01,instance02,instance03
#instance01.host =
#instance01.port = 6379
#instance01.password =
#instance01.connectionTimeout =  2000
#instance01.soTimeout = 2000
#instance01.weight = 1

#instance02.host =
#instance02.port = 6379
#instance02.password =
#instance02.connectionTimeout =  2000
#instance02.soTimeout = 2000
#instance02.weight = 1

#instance03.host =
#instance03.port = 6379
#instance03.password =
#instance03.connectionTimeout =  2000
#instance03.soTimeout = 2000
#instance03.weight = 1

# sentinel config
#masterName = mymaster
#sentinelList = 192.168.101.44:26379,192.168.101.45:26379,192.168.101.46:26379
```

分片配置

哨兵配置

6.9.4 管理Session失效时间

Session的失效时间可以通过以下四种方式配置：

- 在DOMAIN_HOME/config/web.xml中配置Session的缺省失效时间，单位为分钟，例如：

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

- 在指定Web应用WEB-INF/web.xml中配置Session失效时间，单位为分钟，例如：

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

在Web应用中配置的Session失效时间只对这个Web应用起作用。如果没有为Web应用指定Session失效时间，将使用DOMAIN_HOME/config/web.xml作为缺省配置。

- 在程序中，使用HttpSession.setMaxInactiveInterval(int timeout)设置的超时时间，这种方式设置的值优先于前面两种方式。

- 在DOMAIN_HOME/config/apusic.conf的Session服务中配置失效时间，当前面三种方式设置的超时时间小于0时，系统采用此值作为默认的Session超时时间，参考上一节说明。

6.10 使用Apache作为Web代理

Apusic应用服务器能够通过下面的两种方式完成与Apache的集成：

6.10.1 HTTP Proxy

利用Apache自身提供的mod_proxy_http模块就可以完成与Apusic应用服务器的集成，而且Apusic应用服务器端不需要做任何设置。mod_proxy_http提供代理HTTP请求的功能。支持HTTP/0.9,HTTP/1.0,HTTP/1.1标准。

- 单机环境下的配置

需要在%APACHE_HOME%/conf/httpd.conf做如下配置：

```
...

LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
...

ProxyPass /${request_uri} http://192.168.6.119:6888/${request_uri}

...
```

- 多机环境下的负载均衡配置

  将Apache作为LoadBalance前置机有三种不同的部署方式，分别是：

  - 轮询均衡策略的配置

  进入%APACHE_HOME%/conf目录，看看httpd.conf中以下设置是否正确：

  ```
  ...

  LoadModule proxy_module modules/mod_proxy.so
  LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
  LoadModule proxy_http_module modules/mod_proxy_http.so
  ...
  ```

  上面的配置检查无误后，在httpd.conf文件末尾加入：

```
...

ProxyPass / balancer://proxy/ #注意这里以"/"结尾
<Proxy balancer://proxy>

BalancerMember http://192.168.6.119:6888/

BalancerMember http://192.168.6.120:6888/
</Proxy>

...
```

我们来观察上述的参数，我们这里假设为"ProxyPass / balancer://proxy/"，其中，"ProxyPass"是配置虚拟服务器的命令，"/"代表发送Web请求的URL前缀，如：http://myserver/或者http://myserver/aaa，这些URL都将符合上述过滤条件；"balancer://proxy/"表示要配置负载均衡，proxy代表负载均衡名；BalancerMember 及其后面的URL表示要配置的后台服务器，其中URL为后台服务器请求时的URL。以上面的配置为例，实现负载均衡的原理如下：

假设Apache接收到http://localhost/aaa请求，由于该请求满足ProxyPass条件（其URL前缀为"/"），该请求会被分发到后台某一个BalancerMember，譬如，该请求可能会转发到http://192.168.6.119:6888/aaa进行处理。当第二个满足条件的URL请求过来时，该请求可能会被分发到另外一台BalancerMember，譬如，可能会转发到http://192.168.6.120:6888/。如此循环反复，便实现了负载均衡的机制。

- 按权重分配均衡策略的配置

```
...

ProxyPass / balancer://proxy/ #注意这里以"/"结尾
<Proxy balancer://proxy>

BalancerMember http://192.168.6.119:6888/ loadfactor=3
BalancerMember http://192.168.6.120:6888/ loadfactor=1
</Proxy>

...
```

参数"loadfactor"表示后台服务器负载到由Apache发送请求的权值,该值默认为1，可以将该值设置为1到100之间的任何值。以上面的配置为例，介绍如何实现按权重分配的负载均衡，现假设Apache收到http://myserver/aaa 4次这样的请求，该请

求分别被负载到后台服务器，则有3次连续的这样请求被负载到BalancerMember为http://192.168.6.119:6888的服务器，有1次这样的请求被负载BalancerMember为http://192.168.6.120:6888后台服务器。实现了按照权重连续分配的均衡策略。

- 权重请求响应负载均衡策略的配置

```
...

ProxyPass / balancer://proxy/ lbmethod=bytraffic #注意这里以"/"结尾
<Proxy balancer://proxy>

BalancerMember http://192.168.6.119:6888/ loadfactor=3
BalancerMember http://192.168.6.120:6888/ loadfactor=1
</Proxy>

...
```

参数"lbmethod=bytraffic"表示后台服务器负载请求和响应的字节数，处理字节数的多少是以权值的方式来表示的。"loadfactor"表示后台服务器处理负载请求和响应字节数的权值，该值默认为1，可以将该值设置在1到100的任何值。根据以上配置是这么进行均衡负载的，假设Apache接收到http://myserver/aaa请求，将请求转发给后台服务器，如果BalancerMember为http://192.168.6.119:6888后台服务器负载到这个请求，那么它处理请求和响应的字节数是BalancerMember为http://192.168.6.120:6888 服务器的3倍(回想第二项均衡配置，是以请求数作为权重负载均衡的，第三项是以流量为权重负载均衡的，这是最大的区别)。

	**注意**
	每次修改httpd.conf，用apachectl –k restart重新启动Apache。

当多机环境下的负载均衡时，用户需要使用STICKY SESSION特性时，除了在Apusic中配置启用STICKY SESSION外，我们还需要在Apusic服务器的vm.options(%DOMAIN_HOME%/config目录下)文件中设置：

```
...

com.apusic.jvm.route=proxyserver1
...
```

同时apache上也要做相应的设置：

```
...
```

```
ProxyPass / balancer://proxy/ stickysession=JSESSIONID
<Proxy balancer://proxy>

BalancerMember http://192.168.6.119:6888/ route=proxyserver1
BalancerMember http://192.168.6.120:6888/ route=proxyserver2
</Proxy>

...
```

并且route属性必须和各Apusic服务器中的vm.option文件的route值相对应。

	**注意**
	上面的配置是基于Windows XP SP2 + Apache 2.2，Linux/Unix上的配置基本一致。配置完成后需要重新启动Apusic和Apache服务器。

6.10.2 AJP Proxy

AJP 是Apache提供的完成与其它服务器通讯的一种协议。在Apache中通过mod_proxy_ajp模块发送AJP数据,另外一端的服务器需要实现AJP协议，能够接受mod_proxy_ajp模块发送的AJP协议数据，在接受到AJP协议数据后做适当处理，并能够将处理结果以AJP协议方式发送回给mod_proxy_ajp模块 。

Apusic服务器能够完全支持AJP协议，当前实现的AJP协议版本为AJP/1.2,AJP/1.3。当前Apusic服务器缺省使用的是AJP/1.3协议

- 单机环境下的配置

需要在%APACHE_HOME%/conf/httpd.conf做如下配置：

```
...

LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_ajp.so
...

ProxyPass /${request_uri} ajp://192.168.6.119:8007/${request_uri}

...
```

在Apusic应用服务器域安装目录中的config目录下，在配置文件apusic.conf配置文件中加入如下内容：

```
...
<SERVICE CLASS="com.apusic.servlet.http.ajp.AJPEndpoint"
```

```
NAME="Endpoint:type=ajp,service=httpd">
    <ATTRIBUTE NAME="Port" VALUE="8007" />
</SERVICE>

...
```

- 多机环境下的负载均衡配置

需要在%APACHE_HOME%/conf/httpd.conf做如下配置：

```
...

LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_http_module modules/mod_proxy_ajp.so
...

ProxyPass /${request_uri} balancer://proxy/
<Proxy balancer://proxy>

BalancerMember ajp://192.168.6.119:8007/${request_uri}

BalancerMember ajp://192.168.6.120:8007/${request_uri}
</Proxy>

...
```

在Apusic应用服务器域安装目录中的config目录下，在配置文件apusic.conf配置文件中加入如下内容：

```
...
<SERVICE CLASS="com.apusic.servlet.http.ajp.AJPEndpoint"
NAME="Endpoint:type=ajp,service=httpd">
    <ATTRIBUTE NAME="Port" VALUE="8007" />
</SERVICE>

...
```

当多机环境下的负载均衡时，用户需要使用STICKY SESSION特性时，除了在Apusic中配置启用STICKY SESSION外，我们还需要在Apusic服务器的vm.options(%DOMAIN_HOME%/config目录下)文件中设置：

```
...

com.apusic.jvm.route=proxyserver1
...
```

	**注意**
	上面的配置是基于Windows XP SP2 + Apache 2.2，Linux/Unix上的配置基本一致。配置完成后需要重新启动Apusic和Apache服务器。

6.11 使用Microsoft IIS作为Web代理

Apusic 应用服务器可使用IIS作为Web代理，目前支持IIS/PWS 4.0 和5.0 版，不支持IIS 3.0。

6.11.1 理解ACP(Apusic Connector Protocol)

Apusic 与IIS 不是简单的集成，使用了Apusic定义的ACP(Apusic Connector Protocol)协议，除了可以使外部Web 服务器连接到Apusic 以外，ACP协议还具有可伸缩性(Scalibility)、负载均衡(load-balancing)和容错(fault tolerance)能力。ACP可以和多种Web 服务器连接，目前提供和IIS 的接口，使用IIS作为Apusic应用服务器的Web代理，具有以下特性：

- 可伸缩性：从一个由单台PC 服务器建立的简单站点到由几十台高性能Unix 主机组成的大容量商业站点都可以由Apusic 提供动力。例如，可以用10 台运行Apache Web Server 或Internet Information Server 的服务器通过循环DNS 或其他负载均衡技术组成Web 服务器集群。用30 台以上运行Apusic 的服务器组成应用服务器集群，两个集群之间通过ACP 连接成更大的集群系统；
- 负载均衡：ACP 将客户的请求转发给内部应用服务器，对内部服务器的选择是随机的，使负载得以平均分配。ACP 会正确处理HTTP session，同一个session 中的所有请求都路由到最初产生session 的服务器，即使路由出现错误，Apusic 的分布式session 管理器也会正确迁移session 数据。
- 容错：当任何一个应用服务器失效时，ACP 立即将请求转发给其他正常的应用服务器，不会出现单点故障，错误的恢复对客户端是透明的。ACP 维护一个由正常服务器组成的链表，当发现某个服务器出现故障时将其从链表中删除，并在后台试图对失效的服务器重新连接，一旦连接成功就将其放回到正常服务器表中。
- 高速缓存：ACP 将从应用服务器中获得的数据缓存起来，如果数据没有改变，将直接向客户端返回缓存中的数据，这样可以降低应用服务器的压力，提高整体性能。

6.11.2 安装

本节描述在Windows2000服务器上安装ACP，过程如下：

- 选择：开始->程序->管理工具->Internet服务管理器，打开IIS服务管理器；
- 停止IIS服务；

- 选择需要管理的Web站点，检查/scripts 虚拟目录是否存在，如果不存在则创建此虚拟目录，并赋予其执行权限。将Apusic应用服务器安装目录下，子目录\bin中的acp_isapi.dll 和acp_isapi.properties文件复制到该虚拟目录所对应的实际目录；
- 选择需要管理的Web站点，选择右键弹出菜单中的"属性"，在弹出的属性面板中的ISAPI筛选器中，添加实际目录中的acp_isapi.dll文件为筛选器，然后将此虚拟目录的执行许可设置为"脚本与可执行程序"；
- 设置acp_isapi.properties文件中运行Apusic应用服务器的主机与端口号，缺省的acp_isapi.propertie</span文件中包含了一个运行于本地的Apusic应用服务器，参照缺省设置，设置相应运行Apusic应用服务器主机的地址和端口号；
- 重新启动IIS；

	**注意**
	如果Windows服务器安装在64位机器上，必须使用APUSIC_HOME/bin目录下的acp_isapi_x64.dll文件，将其改名为acp_isapi.dll并替换同名文件

6.11.3 管理与配置

acp_isapi.properties文件中包含了所有Apusic -IIS 连接器的配置信息，此文件包含两部分内容：Apusic 服务器配置和URI 映射表。

可以使IIS同时和多个Apusic 服务器连接，你需要指定每个Apusic 服务器的主机名和端口号，例如：

```
servers = foo,bar
server.foo.host = 192.168.55.1
server.foo.port = 6888
server.bar.host = 192.168.55.2
server.bar.port = 6888
```

其中servers 是由逗号隔开的一系列服务器名，server.<name>.host 指定服务器所在机器的主机名或IP 地址，server.<name>.port 指定服务器的侦听端口号。

当客户向IIS 发送一个请求时，Apusic -IIS 连接器对请求的URI 进行分析，并将请求转发给某一个指定的Apusic 服务器。通过设置URI 映射表可以有选择地进行转发。 以下是几个典型的URI 映射：

```
servlet-mapping./servlet/* = foo
servlet-mapping.*.jsp = foo
servlet-mapping./examples/*.jsp = bar
```

以上映射将所有URI 以/servlet/开头或者以.jsp 结尾的请求转发给foo 服务器，而将所有以/examples开打头并且以.jsp结尾的请求转发给bar 服务器。

如果IIS 和Apusic 共用相同的物理目录，采用这种方式可以由IIS 处理静态文档或ASP 文件，而Servlet 和JSP 则由Apusic 来处理。但如果IIS 和Apusic 无法共享物理目录，或 Web 应用已经打包成war 文件，则必须使用/examples/*这样的映射将所有的请求都转发给

Apusic。

以下是一个高级的URI 映射：

```
servlet-mapping.[/mirror/foo]/* = foo
```

如果请求的URI 为/mirror/foo/blah/good.jsp， 则将前缀/mirror/foo 去掉，以 /blah/good.jsp 作为请求URI 转发给Apusic。在服务器名后面可以加上附加的路径信息，此路径将替换原来的前缀。例如：

```
servlet-mapping.[/mirror/foo]/* = foo/new
```

则将/mirror/foo/blah/good.jsp 转换为/new/blah/good.jsp。

采用以上技术可以使IIS 成为Apusic 服务器的代理，将不同前缀的请求转发给不同的内部服务器，同时起到了负载均衡的作用。

如果IIS 配置了多个虚拟主机，则可以使用以下映射方式：

```
servlet-mapping.//foo.com/* = foo
servlet-mapping.//bar.com/* = bar
```

注意映射字符串前面的两个斜杠('//')，当客户访问虚拟主机foo.com 时，请求被转发给 foo服务器，而访问虚拟主机bar.com 时，请求被转发给bar 服务器。

以上所有映射方式可以结合使用，具有很大的灵活性，例如：

```
servlet-mapping.//foo.com[/mirror/foo]/examples/*.jsp =foo/new
```

6.12 管理JDBC服务

Apusic应用服务器为基于Apusic运行的J2EE企业应用提供了一个易于管理和监控、高效的JDBC服务，Apusic的JDBC服务提供了数据库连接池、活动连接监控、JDBC事件监控等服务。下面，就这些服务的管理与配置及其功能进行说明。

6.12.1 理解数据库连接池

在一个未使用数据库连接池技术，而是应用程序直接管理数据库连接的企业应用中，数据库的物理连接和客户端的连接对象是一对一的。当连结对象关闭，物理连接也被相应关闭，因此，打开、初始化和关闭数据库的物理连接等等操作，充斥在每个运行中的客户端会话中。

连接池技术是通过将数据库的物理连接保存在缓存中来解决此问题的，这些缓存中的数据库的物理连接可被多个客户端会话重复使用。特别是在可用的数据库的物理连接数较少，客户端较多的多层企业应用中，使用连接池可以极大地提高应用程序的执行效率、可伸缩性。连接池的连接跟数据库的物理连接是一一对应关系，为方便起见，我们通常把连接池的连接称为物理连接。

数据库连接池在JDBC2.0扩展和JDBC3.0规范中进行了定义，但并未提供对应用服务器如何管理数据库连接池的算法。

Apusic应用服务器提供了一个高效、可靠的数据库连接池实现。提供对JDBC4.0规范的支持。

6.12.2 数据库连接池管理与配置

应用服务器通过在配置文件中定义的数据源（Data Source）生成数据库连接池，每个被定义的数据源指向一个数据库连接池，根据数据源定义中对初始连接数、最小连接数等属性的指定，应用服务器在JDBC服务启动时对连接池进行初始化。

可以通过编辑datasources.xml文件对数据库连接池进行配置：

6.12.2.1 编辑datasources.xml文件

Apusic应用服务器域主目录中，config子目录下的datasources.xml文件是定义数据库连接池的配置文件。datasources.xml按照文档类型定义文件(DTD)datasources.dtd进行编写。

6.12.2.1.1 DATASOURCE元素

datasources.xml由根元素datasources及其下属的一个或多个datasource元素构成，每个datasource元素对应一个数据源配置。Apusic应用服务器启动时，读取在datasources.xml中定义的datasource元素的设置，对相应的数据库连接池进行初始化。

datasource元素的属性如下表：

属性	描述	值类型	缺省值
name	用以区别于其他datasource元素的属性，此属性是必须定义的	符合xml规范中命名规则的名字	无
jndi-name	通过命名服务可访问到此数据源的JNDI名字，此属性是必须定义的	字符串	无
driver-class	用于连接特定数据库的驱动程序类名称（一般由数据库厂商提供），此属性是必须定义的	字符串	无
driver-classpath	driver-class属性中指定的驱动程序类所处的路径信息。如驱动程序类路径信息不在当前的环境变量"classpath"中，或驱动程序不在应用服务器目录下的lib中，则通过此属性来指定其路径信息，可以使用绝对路径如： c:driver_dirmy_jdbc_driver.jar 或 /usr/driver_dir/my_jdbc_driver.jar 来指定，亦可使用系统变量指定路径如： ${DOMAIN_HOME}/driver_dir/my_jdbc_driver.jar 来指定；反之，可以不指定此属性。	字符串	无
url	符合JDBC规范中URL语法的字符串，用于描述数据库的位置信息和通讯子协议，如： jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=pubs 此属性是必须定义的	字符串	无

min-spare-connections	连接池保持的最小备用物理连接数。当连接池中某个空闲物理连接的等待时间超过超时时间（idle-timeout），并且当前空闲连接数大于此值时，连接池会关闭此物理连接。	整型	5
max-spare-connections	连接池保持的最大备用物理连接数。当应用程序完成对连接的操作，如果连接池中的空闲物理连接数量小于此数时，应用程序所使用的物理连接将被放到连接池中作为备用连接，反之，应用服务器将关闭此物理连接。	整型	30
initial-connections	连接池在服务器启动时所创建的物理连接数。	整型	无
max-connections	连接池容量，即允许创建的最大连接数。连接池在连接数超出此值，并在最大等待时间内未得到被释放的数据库连接时，抛出资源不可用异常。	整型	-1。当该值小于或等于0时，系统默认设置为整数最大值
min-wait-time	最小等待时间。当没有空闲连接可用，并且连接池的连接数量超过最大空闲连接数，但小于连接池容量时，等待空闲连接的时间，若超过此时间未得到空闲连接，则创建一个新连接。	整型，单位是秒	2
max-wait-time	最大等待时间。当没有空闲连接可用，并且连接池的连接数量超过连接池容量时，等待空闲连接的时间，若超过此时间未能得到空闲连接，抛出资源不可用异常，客户端获取数据库连接失败。	整型，单位是秒	60
idle-timeout	物理连接等待超时时间。当连接池中的某个数据库物理连接等待被使用的实际时间超过此属性数值时，连接池自动关闭此数据库物理连接。	整型，单位是秒	300
test-before-reused	设置为true允许连接失效检测，false为不允许连接失效检测。	字符串	false
test-command	用于连接失效检测的SQL语句，通常是一个有效的SELECT语句。	字符串	无
isSameRM-override-value	用于判断两个XAResource后端的Resource Manager是否是同一个时直接采用提供的属性值。默认为“无”，表示既不是true也不是false，则在判断时采用数据库XAResource自己的判断得出结果。两个XAResource是否isSameRM决定了这两个XAResource是否以两个分支加入事务。如果isSameRM为 true，则两个XAResource实际上以同一个事务分支加入事务；如果isSameRM为false，则两个XAResource分别以不同的事务分支加入事务。该参数一般在数据源为Unshareable时使用，设为false可解决某些数据库在XAResource.start()方法中存在的资源无法关联到事务的问题。	字符串	无

6.12.2.1.2 PROPERTY

对于datasources.xml中定义的一个数据源元素，对应于一个JDBC规范中定义的DataSource的对象，JDBC规范中还定义了一套标识和描述此DataSource对象的属性。JDBC规范标准属性如下表：

属性名	属性描述	值类型	
databaseName	服务器上的特定数据库名称	字符串	
datasourceName	数据源名称；用于命名底层的XADataSource对象或ConnectionPoolDataSource对象	字符串	
description	数据源的描述	字符串	
networkProtocol	用于与数据库服务器交互的网络协议名称	字符串	
password	数据库密码	字符串	
portNumber	数据库服务器监听请求的端口号	字符串	
roleName	初始的SQL角色名	字符串	
serverName	数据库服务器名称	字符串	
user	用户账号名字	字符串	
isolation-level	用于指定数据库连接的事务隔离级别，如果不配置此属性，则采用数据库默认的事务隔离级别，可设置值为read-uncommitted，read-committed，repeatable-read，serializable。	字符串	无

这些属性值是否有效取决于JDBC驱动程序提供者的实现，JDBC规范中仅仅指定了提供者必须提供description属性。

对于上述属性，可以在datasources.xml文件中使用datasource元素的零个或多个子property元素来进行设置。

一个property元素必须包含两个属性，name属性用于说明需要设置的属性值的名称，value属性用于需要设置的值。如缺省的datasources.xml文件中使用property子元素指定了一个数据源所使用的数据库账号的名字和密码：

```
<datasource name="SampleDB" jndi-name="jdbc/sample" driver-
class="sun.jdbc.odbc.JdbcOdbcDriver" url="jdbc:odbc:apusic">
    <property name="user" value="scott" />
    <property name="password" value="tiger" />
</datasource>
```

6.12.2.1.3 REMOTE-ACL

Apusic 应用服务器中，应用程序可以远程访问数据库连接池，对于需要被远程客户访问的数据源，必须在对应的datasource元素中申明remote-acl元素，指定远程客户访问此数据源时必须具有的系统用户或组的名字列表，同时，在远程客户指定使用正确的用户身份信息：用户名，密码。

这样，当远程客户请求访问此数据源时，如远程客户请求中指定的用户名或密码可通过服务器的身份鉴定，则服务器根据数据源中指定的用户或组的名字，对远程客户请求进行授权。如远程客户请求中指定的用户名是remote-acl中指定的用户或属于remote-acl中指定的组，则授权远程客户进行访问。

下例定义了一个可被远程访问的数据源SampleDB，此数据源只授权给名为foo的用户或属于bar组的远程访问请求：

```
<datasource name="SampleDB" ... >
    <property
.../> ... <remote-acl>
        <user>foo</user>
        <group>bar</group>
    </remote-acl>
</datasource>
```

6.12.2.1.4 结果集缓存

对于Session Bean、JSP、Servlets为主的JavaEE应用中，频繁访问数据库而缺少对应的数据缓存等机制，往往会成为企业应用的瓶颈。因此，Apusic应用服务器提供了JDBC结果集缓存技术，通过将数据库返回的结果集保存在内存中可以大幅提高应用系统的性能，同时，使用结果集缓存对应用开发者是完全透明的，保证了应用的可移植特性。

```
<datasource name="SampleDB" ... >
    <property
.../> ... <property name="pre-fetch-size" value="100" />
</datasource>
```

6.12.2.1.5 语句缓存

Apusic应用服务器提供了JDBC语句缓存技术，同时使用预编译的查询语句，提高应用程序访问数据库的效率。

当应用程序通过使用定义了语句缓存的数据源访问数据库时，即可使用语句缓存中的预编译的查询语句对数据进行查询。

结果集缓存包含一个属性，即缓存的语句数量，此属性无缺省值。

下例中对SampleDB数据源定义了一个语句数量为20的语句缓存：

```
<datasource name="SampleDB" ... >
    <property .../>
    ...
    <property name="stmt-cache-size" value="20" />
</datasource>
```

6.12.2.1.6 连接失效检测

test-before-reused：设置为true允许连接失效检测，false为不允许连接失效检测。

test-command：用于连接失效检测的SQL语句，通常是一个有效的SELECT语句。当所用的数据库为Oracle时，不需要配置此参数，只需将test-before-reused设置为true，这时应用服务器会通过连接的createStatement判断所用的连接是否失效。当所用的数据库为SQL Server或Sybase时，必须在将test-before-reused设置为true的同时配置此参数，因为这两个数据库对Statement做了客户端缓冲，无法通过连接的createStatement判断所用的连接是否失效，这时需要执行一条具体的SQL语句来判断连接是否可用。

6.12.3 活动连接监控

所谓的活动连接，是指客户端程序从数据库连接池取到连接后，正在执行任务或者未释放的连接。通过Apusic应用服务器JDBC服务提供的活动连接监控，我们可以知道指定数据源的数据库连接池使用状况，比如当前连接池的空闲连接数，活动连接数等，并可查看到指定某个连接的调用栈。如果正在数据库连接池的连接总数一直在上涨并且空闲连接越来越少甚至一直为零，就有警惕是否发生连接泄露了。这时通过查看指定连接的栈，我们就可以查找客户程序中是否未关闭连接。

我们再来理解一下客户端连接与物理连接的区别。客户端连接是指客户端程序通过javax.sql.DataSource的getConnection()方法取得的连接，代表客户程序中的连接对象（java.sql.Connection），它不是真正的跟数据库之间的物理连接。由于在一个事务里，允许多个客户端连接共享一个物理连接，因此，客户端连接跟物理连接之间是多对一的关系。

由于数据库的连接资源是有限的，这就要求客户程序在用完连接后，应该主动释放此连接资源以供后来者使用。通常认为如果客户程序未主动使用java.sql.Connection.close()方法关闭连接，就会出现连接泄露，随着泄露连接数的增长，最终耗光了连接资源，系统就变为不可用了。很多情况下是这样的，但在使用Apusic数据库连接池时，Apusic 应用服务器对此做了容错处理，使得出现此类连接泄露的几率大大降低。这是因为在完成一次组件调用后，Apusic会检查此次调用中使用过的连接资源，首先取消物理连接与客户端连接的关联，然后把物理连接放回空闲连接池以供别的客户端调用使用。此时，上述未关闭的客户端连接变为非活动的，它不持有一个物理连接，因此即使客户程序忘了关闭连接也不会发生连接泄露的问题。当下次调用又使用了上次调用未关闭的连接时，Apusic的数据源连接服务会重新激活此连接，并重新为其关联一个物理连接，这时它又是一个活动连接了。

虽然如此，但在使用Apusic数据库连接池时，仍有可能会出现连接泄露的情况，如：此连接是在Apusic应用服务器所在的JVM外面通过远程调用获取的或者是在Apusic应用服务器中但连接是在客户自己启动的线程中取得的，通过这两种方式取得的连接，如果客户程序未主动执行java.sql.Connection.close()操作，则会出现连接泄露。

为了能够帮助用户查找并解决连接泄露的问题，Apusic应用服务器提供了连接监控服务。默认情况下，活动连接监控功能是关闭的，如果需要使用此功能，需编辑Apusic域主目录中，config子目录下的apusic.conf文件，其中相关配置段如下：

```
...
<SERVICE CLASS="com.apusic.jdbc.JDBCService">
    <ATTRIBUTE NAME="EnableActiveConnTrace" VALUE="False" />
</SERVICE>
...
```

将EnableActiveConnTrace属性的值改为True，重新启动应用服务器后，即可通过Apusic AdminConsole工具监控活动连接。Apusic AdminConsole的使用请查阅Admin Console的管理手册。

6.12.4 JDBC事件监控

JDBC是当前J2EE应用跟数据库系统交互的主要编程接口，因此，对JDBC的监控能更好的帮助程序员或系统管理员去发现JDBC应用中的性能、死锁、堵塞等问题。

Apusic 应用服务器的数据源监控提供了JDBC事件监控功能，所谓JDBC事件是指执行一条SQL语句或者提交一个事务等。当启动了监控后，可查看到应用系统中执行的SQL语句，执行时间，相关的数据库连接和事务等，并增强了JDBCTracerService的功能，可以对应用中的连接使用情况进行跟踪，超过时间没有释放则会进行提示；同时也提供一种强制关闭的方式，避免连接耗尽，影响应用的正常运行。相应的属性配置在apusic.conf文档里面。

```
...
<SERVICE CLASS="com.apusic.jdbc.trace.JDBCTracerService">
    <ATTRIBUTE NAME="TraceAllow" VALUE="False" />
    <ATTRIBUTE NAME="StackTraceAllow" VALUE="False" />
    <ATTRIBUTE NAME="checkAbandoned" VALUE="False" />
    <ATTRIBUTE NAME="removeAbandoned" VALUE="False" />
    <ATTRIBUTE NAME="checkAbandonedTimeout" VALUE="300" />
    <ATTRIBUTENAME="LogAllow"VALUE="True"/>

<ATTRIBUTENAME="LogEventType"VALUE="statement.slowExecute,statement.execute"/>
    <ATTRIBUTENAME="LongQueryTime"VALUE="2000"/>
</SERVICE>

...
```

TraceAllow：是否跟踪jdbc事件，true为跟踪，false为不跟踪。如果将StackTraceAllow属性设为True且TraceAllow也为True，则可查看JDBC事件发生时的调用栈。

checkAbandoned：是否开启检测连接池泄露，值设置为true则表示开启，其默认值为false，而检测的间隔时间和checkAbandonedTimeout属性的时间一致；

checkAbandonedTimeout：连接使用时间超过该值则认为该连接为泄露，单位为秒，默认值为300；

removeAbandoned :是否强制关闭泄露的连接。如果设置为true，则会自动关闭检测为超时的数据库连接；如果设置为false，则会在后台打印警告信息，默认值为false；

LogAllow:为True则会记录日志, 默认不启用慢查询日志追踪。

LongQueryTime:是超过该值才算慢查询（默认为2000单位毫秒）

LogEventType:日志事件类型，可为多个，用,隔开,类型有如下几种

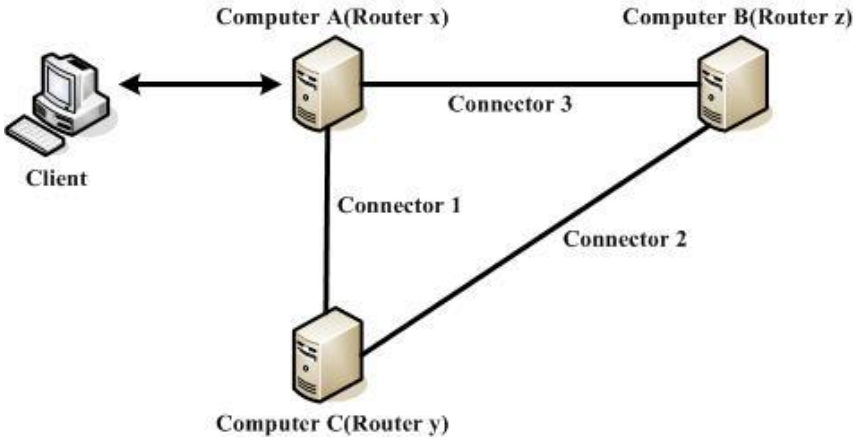
- statement.execute
- connection.close
- connection.setAutoCommit
- connection.commit
- transaction.begin
- transaction.complete
- transaction.timeout
- statement.slowExecute

6.13 消息服务配置

6.13.1 理解Apusic消息服务

JMS 是JavaEE 规范中提出的消息中间件服务(Java Message Service™)规范，提供应用程序间异步或同步的消息传递和管理服务，Apusic 应用服务器中包含了高效、可靠的消息服务。Apusic应用服务器中的消息服务接口完全遵循JMS API规范。提高了企业应用中各组件的可移植性、松耦合特性，同时更加高效可靠；为分布式企业应用异步交换关键业务数据和事件提供了可靠而灵活的服务。

Apusic消息服务提供对消息队列(Queue)和消息主题(Topic)的管理，消息的发送由消息服务负责完成，原理如下：



Computer A,B,C 为提供消息服务的Apusic应用服务器，每一个 Apusic 应用服务器称为一个Router，他们之间的连接被称为Connector，客户端进行消息发送时直接与Computer A 连接，客户端要把消息传送到Computer B的队列时，只需指明接收消息的主机名和队列名，然后发送消息即可，Computer A 中的消息服务将会查找配置文件中的设置，如从A 到B间有无通路连通，假如它们

之间有通路连接，消息服务就会寻找这些通路中最短的路径发送消息到指定服务器(Computer B)的指定队列，假如A 到B间无有效连接，消息将被保存在某一连通的服务器上，待有有效通路之后再发送消息。

Apusic应用服务器提供了两个主要的消息服务方面的特性：

- 全局事务

Apusic应用服务器提供了一个作为事务性资源管理器的JMS 提供者(Provider)，允许从JSP，Servlet， EJB 等应用组件中对消息服务进行事务性的访问。对消息的发送和接收可以和对其他资源的操作一起参与到一个JTA事务中

- Message-driven Bean

J2EE™1.3规范中提供的Enterprise Bean类型，Message-driven Bean，使多层分布式企业应用可以异步地使用消息

6.13.2 消息路由与存储

多个提供消息服务的应用服务器可以组成一个虚拟的JMS网络，每个提供消息服务的应用服务器可被视为一个JMS网络中的节点，而节点之间的连接则通过声明消息路由提供。只要用户连接到JMS网络中任何一个节点，即可向网络中的其他任意节点发送消息。

为提供消息服务的可靠性，和JMS网络中消息的暂时存储，Apusic应用服务器使用了一个可靠的消息存储机制。

Apusic消息服务消息路由与存储配置的部分存在于域主目录下config子目录中，文件中的相关配置段。apusic.conf</span文件中可对消息服务的消息存储目录和消息路由进行配置。

6.13.2.1 配置消息存储目录

Apusic应用服务器的消息存储的配置，通过编辑Apusic应用服务器域主目录下config子目录中的apusic.conf配置文件的相关配置段进行。缺省的配置段如下：

```
...
<SERVICE CLASS="com.apusic.jms.server.FileMessageStoreProvider">
    <ATTRIBUTE NAME="StoreDirectory" VALUE="store/jms" />
</SERVICE>

...
```

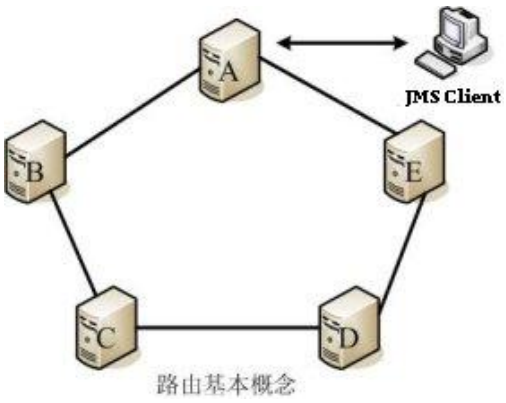
此配置段指定了消息服务的存储目录，其中，名为StoreDirectory属性的值指定了消息存储目录的路径，可使用绝对路径如：c:\message_store\或/usr/message_store来指定，亦可使用相对于域主目录的相对路径如：store /my_message_store来指定，如指定目录不存在，应用服务器将创建指定的目录。

6.13.2.2 配置消息路由

6.13.2.2.1 理解消息路由

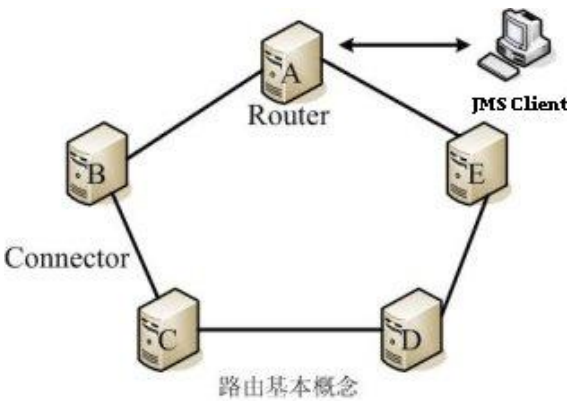
在一个由多台提供消息服务的 Apusic 应用服务器组成的网络上，每个提供消息服务的 Apusic 应用服务器可被视为一个网络中的节点，消息路由意味着节点之间的路径。只要用户连接到网络中任何一个节点，即可向网络中的其他任意节点发送消息，由消息服务根据消息路由提供消息在网络中的最佳传输机制。如下图所示：

消息路由图示-1



A、B、C、D、E 五个节点上都是提供消息服务的 Apusic 应用服务器，五个节点连通成一个网络，连接到 A 节点的客户端可以发消息到五个节点中的任意一个节点，Apusic 应用服务器会自动寻找一条最佳路径传递消息到目标节点。如从A 发消息到D，有两条连通路径：A->B->C->D 和A->E->D，其中A->E->D 经过的节点最少，该路径为最佳路径， Apusic 应用服务器将根据此路径对消息进行传递。

消息路由图示-2



在一个 Apusic 应用服务器 JMS 网络中，如上图所示，每个节点都是一个消息路由器(Router)，每个消息路由器有一个路由器名，以区别于其他的路由器。路由器名可以和主机名相同，也可以不同。

每两个节点之间的直接路径被称为路由连接器(RoutingConnector)，网络中每两个节点间的直接路径必须事先进行申明性的定义。每个路由连接器的申明都是双向的，如在A节点上申明到B的路由连接器，则说明从A节点到B节点之间是双向连通的，从A 可以到达B，从B也同样能够到达A。

假设上图中的五个节点A、B、C、D、E 是某个实际网络中五个提供消息服务的Apusic 应用服务器 ，实际网络中的主机名分别是computerA、computerB、computerC、computerD、computerE，在由这五个节点组成的 Apusic 应用服务器网络中，服务器配置的路由器名分别是routerA、routerB、routerC、routerD、routerE，要使这五个节点结形成类似于上图的 Apusic 应用服务器网络，即每个节点都有一条与下一个节点双向连通的路径，最后一个节点有一条与最开始的节点双向连通的路径，形成一个闭合的环，消息客户连接到环中的任意节点，都可向其他节点发送消息。则可以选择在每个节点上都申明一个到下一个节点的连接器，由于连接器是双向连通的，所以在下一个节点不需要申明一个到上一个节点的连接器。同时在每个节点上的远程路由需要申明跟它直接相连的其它节点的路由名。如：A节点上的远程路由需要申明B、E两个节点，表示由A节点可以直接连接到B、E两个节点上，并且不需要申明其它节点，这样A就能够向所有的节点发送消息了。同时需要申明由A到B的连接器，表示A节点跟B节点有一条双向连通的路径；节点B的远程路由则需要申明A、C两个节点，同时需要申明由B到C的连接器，如此类推。

6.13.2.2.2 配置和管理路由

对消息服务中路由的配置是通过Apusic应用服务器域主目录中子目录config下的apusic.conf配置文件进行的。通过apusic.conf</span中的相关配置段可以对消息路由器和路由连接器进行管理和配置。

- 路由器的配置

消息路由器的配置通过在apusic.conf文件中配置消息服务的服务配置段进行，其缺省配置段如下：

```
...
<SERVICE CLASS="com.apusic.jms.server.JMSServer">
</SERVICE>

...
```

通过对申明此服务的XML元素增加名为RemoteRouters的ATTRIBUTE子元素，在其值中列出相应的路由器名即可。以消息路由图示-2中的A节点为例，A节点上需要申明B、E两个节点，表示由A节点可以直接连接到B、E两个节点上，并且可以作为消息路由路径中的一个中转节点，则可以进行如下配置：

```
...
<SERVICE CLASS="com.apusic.jms.server.JMSServer">
    <ATTRIBUTE NAME="RemoteRouters" VALUE="computerB,computerE" />
</SERVICE>

...
```

其它B、C、D、E节点上的相关配置段如此类推进行配置。路由器名字间使用逗号分隔，路由器名字可以是其他主机的主机名。

一般，路由器的名字默认是服务器的主机名，但是，也可以通过使用对申明此服务的XML元素增加名为RouterName的ATTRIBUTE子元素，在其值中指定自身的路由器名。如上例中采用了主机名对路由连接器进行申明，如在五个节点上都申明了路由器名字，如A、B、C、D、E主机分别对应routerA、routerB、routerC、routerD、routerE，则上例的配置如下：

```
...
<SERVICE CLASS="com.apusic.jms.server.JMSServer">
    <ATTRIBUTE NAME="RouterName" VALUE="routerA" />
    <ATTRIBUTE NAME="RemoteRouters" VALUE="routerB,routerE" />
</SERVICE>

...
```

其它路由器的配置依次类推。

- 路由连接器的配置

每个JMS网络中的消息路由器都是通过路由连接器进行连接的，每个路由器可以拥有多个路由连接器连接到其它节点。对路由连接器的配置是通过在apusic.conf配置文件中增加路由连接器服务实现的。以消息路由图示-2中的A节点为例，需要申明到节点B的路由连接器，则可在apusic.conf文件中加入如下配置段：

```
...
<SERVICE CLASS="com.apusic.jms.routing.RoutingConnector"
NAME="Connector:Name=toB">
    <ATTRIBUTE NAME="RemoteHost" VALUE="computerB" />
    <ATTRIBUTE NAME="RemotePort" VALUE="6888" />
</SERVICE>

...
```

其中computerB是B的主机名，toB是此路由连接器区别于其他路由连接器的自由定义名字，6888是默认的JMS服务端口，具体使用中，可将其更改为实际的路由连接器名、主机名与端口。

消息路由图示-2中其它节点上的路由连接器配置如此类推。

6.13.3 管理消息服务

消息服务的管理是通过配置消息管理对象的属性完成的。消息管理对象包含了管理员生成的消息配置信息，之后由消息客户端使用。

消息服务中定义了两个管理对象：

- 连接工厂(ConnectionFactory)。消息客户通过此对象创建与消息服务的连接；
- 消息目的地(Destination)。消息客户用来指定发送消息的目的地或接收消息的来源地。

由于消息服务包含两种消息模型，即Point-to-Point(PTP)和Publish-and-Subscribe(Pub/Sub)模型，Apusic应用服务器对这两种消息模型提供了完整的支持，因此，对于连接工厂而言，提供了两种可配置的连接工厂类型，面向PTP消息模型的QueueConnectionFactory和面向Pub/Sub模型的TopicConnectionFactory；对于消息目的地而言，同样提供了两种可配置的消息接收类型站，面向PTP的队列（Queue）站和面向Pub/Sub的主题（Topic）站。

通过对存在于Apusic应用服务器域主目录下config子目录中，名为jms.xml的配置文件进行编辑，实现对Apusic应用服务器消息服务的管理。在此文件中，可以对连接工厂、消息目的地和消息服务安全策略进行配置。

jms.xml文件是一个xml文件，其文档类型定义(DTD)为jms-config_1_2.dtd。

Apusic应用服务器提供了jms.xml。

6.13.3.1 配置连接工厂

在jms.xml文件中，每一个连接工厂配置信息对应一个connection-factory标记申明的xml元素，每个connection-factory元素可包含使用以下三种标记所申明的子元素：

- description，可选标记，对此连接工厂的描述；
- display-name，必须申明的标记，用于区别于其它连接工厂；

- jndi-name，可选标记，通过JNDI，用于客户在服务器命名空间中查找此连接工厂。

实际应用中，当管理员为连接工厂分配JNDI名之后，消息客户即可使用JNDI在服务器的命名空间中对连接工厂进行查找并获得引用，之后通过连接工厂取得与消息服务的连接。

连接工厂的配置属性如下表：

属性	描述	值类型	缺省值
pooled	指定此连接工厂是否对其管理的连接使用连接池	“True”或“False”	“False”
secure	指定连接工厂所提供连接的通讯方式	“True”或“False”	“False”
anonymous	是否授权匿名用户访问此连接工厂	“True”或“False”	“True”
client-id	由于标识连接客户状态的标识符，通常被用于Pub/Sub模型中的持久订阅（Durable subscription)	字符串，此属性是可选的	无
default-delivery-mode	使用由此连接工厂生成的连接发送消息时，缺省的发送方式	“persistent”或“non-persistent”	“non-persistent”
default-priority	使用由此连接工厂生成的连接发送消息时，缺省的优先级	数字（0~9）	“4”
default-time-to-live	使用由此连接工厂生成的连接发送消息时，对于已发送的消息，消息系统保留此消息的缺省时间长度，单位为毫秒。	整型	0
min-pool-size	此连接工厂对应的连接池中，所保持的最少连接数	整型	5
max-pool-size	此连接工厂对应的连接池中，所保持的最大连接数	整型	30
idle-timeout	连接等待超时时间。当连接池中的某个连接等待被使用的实际时间超过此属性数值时，连接池自动关闭此连接	整型，单位是秒	300

6.13.3.2 配置消息目的地(Destination)

对应于PTP和Pub/Sub消息模型，Apusic应用服务器中的消息服务提供了两种消息目的地，队列（Queue）和主题（Topic）。实际应用中，管理员为消息目的地分配JNDI名，消息客户即可使用JNDI在服务器的命名空间中对消息目的地进行查找并获得引用，在通过连接工厂取得与消息服务的连接之后，消息客户即可向消息目的地同步或异步地发送或接收消息。

在jms.xml文件中，每一个消息目的地配置信息对应一个queue标记或topic标记申明的xml元素。

6.13.3.2.1 配置消息队列

在jms.xml文件中，每一个队列（Queue）配置信息对应一个queue标记申明的xml元素，每个申明的queue元素可包含三种标记所申明的子元素：

- description，可选标记，对此队列的描述；
- display-name，必须申明的标记，用于区别于其它队列；
- jndi-name，可选标记，通过JNDI，用于客户在服务器命名空间中查找此队列。

队列的配置属性如下表：

属性	描述	值类型	缺省值
cache-size	队列缓冲中保留的消息个数	整型	20
expiry-check-interval	系统检测消息队列中消息是否过期的时间间隔，单位是秒	整型	60

6.13.3.2.2 配置消息主题

在jms.xml文件中，每一个主题（Topic）配置信息对应一个topic标记申明的xml元素，每个申明的topic元素可包含三种标记所申明的子元素：

- description，可选标记，对此主题的描述；
- display-name，必须申明的标记，用于区别于其它主题；
- jndi-name，可选标记，通过JNDI，用于客户在服务器命名空间中查找此主题。

主题的配置属性如下表：

属性	描述	值类型	缺省值
cache-size	主题缓冲中保留的消息个数	整型	20
expiry-check-interval	系统检测消息主题中消息是否过期的时间间隔，单位是秒	整型	60

6.13.3.3 安全相关配置

JavaEE™体系中的JMS规范实际上并未包含有关安全方面的内容，因此，系统提供保证消息服务的安全和完整性的机制就极为重要。

Apusic应用服务器提供了对消息服务方面的安全管理，主要根据JavaEE™体系中的安全角色(Security Role)和消息客户的操作对消息服务进行保护。

消息服务中基于安全角色的授权方式是指，系统管理员可定义一组安全角色，每个被定义的安全角色对应于系统中的一组用户或组，然后，根据消息客户用户可对消息目的地(Destination)进行的操作（如对于队列，客户可执行发送、接收、浏览等操作）对前面定义的安全角色进行授权。

6.13.3.3.1 安全角色

在jms.xml文件中，每一个安全角色（Security Role）配置信息对应一个security-role标记申明的xml元素，每个申明的security-role元素可包含四种标记所申明的子元素：

- description，可选标记，对此安全角色的描述；
- role-name，必须申明的标记，用于区别于其它安全角色申明；
- principal或group，必须使用至少一个这两种标记申明的元素，可以是多个，用于将逻辑的安全角色对应于系统中的用户或组。

6.13.3.3.2 消息目的地(DESTINATION)访问许可

对于一个或者一组jms.xml中申明的消息目的地(Destination)和一个或者一组jms.xml中申明的安全角色，通过使用一个destination-permission标记申明的xml元素来设置这二者之间的对应关系，每个申明的destination-permission元素可包含三种标记所申明的子元素：

- description，可选标记，对此访问许可的描述；
- role-name，必须使用至少一个这种标记申明的元素，可以是多个，用于表示此访问许可所包含的在security-role中申明的安全角色；
- destination-method，必须使用至少一个这种标记申明的元素，可以是多个，用于表示此访问许可所包含的客户对消息目的地(Destination)能进行的操作。

  每个申明的destination-method元素可包含四种标记所申明的子元素：

  - description，可选标记，对此destination-method的描述；
  - queue-name或topic-name，必须有至少一个这两种标记申明的元素，而且它的值必须是对应的jms.xml中已申明的queue或topic元素，用于说明此访问许可包含的消息目的地(Destination)；
  - method-name，必须使用至少一个这种标记申明的元素，可以是多个，通过方法名，指定对指定队列（queue)或主题（topic)所能进行的客户操作。对于指定的队列（queue），每个method-name元素的值可以是send、receive或browse中之一；对于指定的主题（topic），每个method-name元素的值可以是publish、subscribe、subscribe-durable或unsubscribe中之一；

6.13.3.3.3 范例

下面范例授权安全角色foo可以对队列bar 进行接收和浏览：

- 申明队列

```
...
<queue>
    <queue-name>testQueue</queue-name>
</queue>

...
```

- 申明安全角色foo，并对应到管理员用户

```
...
<security-role>
    <role-name>foo</role-name>
    <principal>admin</principal>
</security-role>

...
```

- 设置访问许可

```
...
<destination-permission>
    <role-name>foo</role-name>
    <destination-method>
        <queue-name>testQueue</queue-name>
        <method-name>receive</method-name>
        <method-name>browse</method-name>
    </destination-method>
</destination-permission>

...
```

6.14 管理事务服务

6.14.1 配置事务服务

Apusic应用服务器提供事务服务，提供完全遵循JTA规范的实现，可以通过编辑在Apusic应用服务器域主目录中config子目录下的apusic.conf配置文件中的相关配置段，对事务服务的超时时间进行设置。

在配置事务服务前，应该熟悉JavaEE™规范中可参与到事务中的组件，如EJB，JMS，JDBC等。

apusic.conf配置文件中缺省的事务服务配置段如下：

```
...
<SERVICE CLASS="com.apusic.transaction.TransactionService">
    <ATTRIBUTE NAME="DefaultTimeout" VALUE="600" />
    <ATTRIBUTE NAME="EnableLog" VALUE="False" />
    <ATTRIBUTE NAME="RetryTimeout" VALUE="600" />
    <ATTRIBUTE NAME="RetryInterval" VALUE="60" />
    <ATTRIBUTE NAME="HeuristicDecision" VALUE="rollback" />
</SERVICE>

...
```

下表简述Apusic应用服务器中的事务服务的各项可配置属性：

属性	描述	属性值	缺省值
DefaultTimeout	从开始一个事务到系统强制其回滚的时间，以秒为单位	整型	600

		值	
EnableLog	打开事务日志。只有打开事务日志，才会启用分布式事务恢复功能	布尔值	默认为False（关闭事务日志）
RetryTimeout	发生通讯故障时重试的时间，以秒为单位	整型值	600（0表示会永远重试）
RetryInterval	发生通讯故障时重试的时间间隔，以秒为单位	整型值	60
HeuristicDecision	作为事务从属Coordinator的应用服务器如果在RetryTimeout指定的时间内无法从其上级Coordinator处获得事务完成方向的指示，会自行决定应该提交还是回滚。取值必须为“commit”或“rollback”两者之一。	字符串	rollback
TxServerID	事务管理器的ID，用于标识事务管理器的一个字符串	字符串	默认无需设置

6.15 管理命名服务

6.15.1 配置命名服务

Apusic应用服务器提供命名服务，提供完全遵循JNDI规范的实现，可以通过编辑在Apusic应用服务器域主目录中config子目录下的apusic.conf配置文件中的相关配置段，对命名服务进行设置。

在配置命名服务前，应该熟悉JNDI规范中的相关内容。 可参考：SUN的JNDI文档。

apusic.conf配置文件中缺省的命名服务配置段如下：

```
...
<SERVICE CLASS="com.apusic.naming.NameService">
</SERVICE>

...
```

下表简述Apusic应用服务器中的命名服务的各项可配置属性：

属性	描述	属性值	缺省值
AllowLookup	允许客户应用可以进行查找的命名空间	字符串	“*”
AllowBind	客户应用可以执行绑定操作的主机名	字符串	“localhost”

6.16 管理应用服务器域

6.16.1 应用服务器域的概念

应用服务器域是应用服务器实例的逻辑管理单元，这个单元是所有相关资源的集合。自Apusic应用服务器5.0版本以来，我们增加了应用服务器域的特性，能够在一份Apusic 服务器中存在多个域 。

6.16.2 应用服务器域的管理

应用服务器域的管理包括创建域，删除域，创建域模板三部分。当您安装Apusic应用服务器后，可以在%APUSIC_HOME%/bin下发现一个config.cmd（非Windows下为config）的命令,使用该命令能够完成应用服务器域的全部管理功能：

- 创建域：为该应用服务器创建一个域空间，来实现在一份Apusic应用服务器中运行多个实例；
- 删除域：为该服务器删除一个不需要的域空间；
- 创建域模板：创建一个域模板文件，用于在构建域的时候指定合适的模板。

6.17 应用服务器安全提供程序

Apusic应用服务器提供了灵活的,可扩展的安全框架，方便用户自定义自己的安全提供程序。

6.17.1 安全框架介绍

Apusic应用服务器安全框架的主要功能是提供简化的应用程序编程接口，安全和应用程序开发人员可以使用该接口来定义安全服务。在该上下文中，Apusic应用服务器安全框架还作为Apusic容器(Web和EJB)，资源容器和安全提供程序之间的中介。以下描述Apusic容器，资源容器和每个安全提供程序之间通过Apusic安全框架的交互：

- 身份验证

身份验证主要是为了完成用户凭证的有效性处理。在处理过程中，由用户发出请求，随后请求将会被分发到应用服务器的资源容器(Web容器或EJB容器)，资源容器会请求应用服务器的安全框架来保证受限的资源的正确访问，并要求用户提供有效的身份，一旦用户身份提交，相应的验证程序会被调用，来处理身份的有效性。

- 授权

授权主要是处理通过了身份验证的用户，对受限的资源时候是否具有相应的操作权限的控制。为了完成授权的处理，用户的请求同样会被分发到资源管理容器(Web容器或EJB容器)，同样资源容器会请求应用服务器的安全框架来进行授权，安全框架会调用相应的授权提供程序来进行授权。

- 角色映射

在Apusic应用服务器中角色的映射主要是在apusic-application.xml中定义的，信息大致如下：

```
<realm-name>AgentRealm</realm-name>
<security-role>
    <role-name>MANAGER_ROLE</role-name>
    <group>id=manager,ou=role,dc=examples,dc=com</group>
```

```
</security-role>
<security-role>
    <role-name>EMPLOYEE_ROLE</role-name>
    <group>id=employee,ou=role,dc=examples,dc=com</group>
</security-role>
```

6.17.2 内置安全提供程序

Apusic 应用服务器内置了三种安全提供程序实现，包含文件、JDBC以及LDAP存储，以方便用户根据自身业务进行选择。要使用这些安全提供程序，首先需要配置%DOMAIN_HOME%/config/security.xml，如何配置以下将逐个介绍。

6.17.2.1 文件存储配置

```
<realm>
    <realm-name>default</realm-name>
    <provider-type>FileStore</provider-type>
</realm>
```

6.17.2.2 JDBC存储配置

JDBC配置属性说明：

datasource-jndi-name： 连接用户与组信息所在数据库的数据源的JNDI名

user-table-name： 用户信息表名

usernameField： 用户字段名

passwordField： 密码字段名

needEncode： 指定密码是否需要编码

encoder-class-name： 用于对密码进行编码的全类名，该类必须实现接口：com.apusic.security.realm.Encoder

query-by-groupname-sql： 根据组名查询其包含用户的SQL语句

示例配置：

```
<realm>
    <realm-name>default</realm-name>
    <provider-type>RDBMS</provider-type>
    <attribute name="datasource-jndi-name" value="jdbc/mssqlserver" />
    <attribute name="usernameField" value="uname" />
    <attribute name="passwordField" value="pwd" />
    <attribute name="user-table-name" value="userbase" />
    <attribute name="query-by-groupname-sql" value="select uname from
```

```
groupbase where gname=?" />
</realm>
```

6.17.2.3 LDAP存储配置

LDAP配置属性说明：

host： LDAP服务器所在主机名或IP

port： LDAP服务器监听的端口号

rootDN： 登录LDAP服务器的根DN

password： 登录LDAP服务器的根DN对应的密码

searchBaseDN： 搜索用户的基DN

groupBaseDN： 搜索组的基DN

userField： 用户字段名

passwordField： 密码字段名

memberField： 组项中定义一个用户成员的字段名，此字段值对应用户的DN

注： 组字段固定名称为cn

假设有一个LDAP服务器，其中定义了一个名为manager的组，该组包含用户andy及jacky。

```
dn: cn=manager,ou=groups,dc=example,dc=com
objectClass: groupOfNames
objectClass: top
cn: manager
member: uid=andy,ou=DeptA,dc=example,dc=com
member: uid=jacky,ou=DeptA,dc=example,dc=com
```

配置：

```
<realm>
    <realm-name>ldap</realm-name>
    <provider-type>ldap</provider-type>
    <attribute name="host" value="localhost" />
    <attribute name="port" value="10389" />
    <attribute name="rootDN" value="uid=admin,ou=system" />
    <attribute name="password" value="secret" />
    <attribute name="searchBaseDN" value="ou=DeptA,dc=example,dc=com" />
```

```
    <attribute name="userField" value="uid" />
    <attribute name="passwordField" value="userpassword" />
    <attribute name="groupBaseDN" value="ou=groups,dc=example,dc=com" />
    <attribute name="memberField" value="member" />
</realm>
```

6.17.3 自定义安全提供程序

我们能够通过Apusic应用服务器的安全框架来自定义自己的安全提供程序，下面以一个例子来描述如何在Apusic应用服务器上开发一个自定义的安全提供程序

6.17.3.1 实体信息的存储与读取

这里讲述了如何开发一个有效的安全提供程序，这个例子描述的是通过配置文件存储用户和组的信息。用户的信息存储在user.conf文件中，组的信息存储在group.conf文件中。内容如下：

在group.conf文件中定义的组信息如下:

```
group=administrator
group=manager
group=employee
```

在user.conf文件中定义的用户信息如下:

```
user=admin password=admin group=administrator
user=director password=director group=manager
user=operater password=operator group=employee
```

上面介绍了实体信息的存储，下面我们可以通过SimpleStore.java来完成对实体信息的读取操作

```
// SimpleStore.java
package com.apusic.demo.security.store;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.security.Principal;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Vector;
```

```
public final class SimpleStore {
  // 从group.conf中读取的组信息
  private static final Map<String, SimpleGroup> groups = new HashMap<String,
SimpleGroup>();
  // 从user.conf中读取的用户信息
  private static final Map<String, Principal> users = new HashMap<String,
Principal>();
  private static final SimpleStore store = new SimpleStore();
  static {
    try {
      String groupEntry = null;
      String groupName = null;
      String userEntry = null;
      String user = null;
      String roles[] = null;
      String password = null;
      String elements[] = null;
      SimpleGroup group = null;
      SimplePrincipal principal = null;
      // 从group.conf中读取组信息
      Iterator iterator = readEntry("group.conf");
      while (iterator.hasNext()) {
        groupEntry = (String) iterator.next();
        if (groupEntry == null) {
          continue;
        }
        elements = groupEntry.split("=");
        if (elements.length == 2 && elements[0].equals("group")) {
          groupName = elements[1];
          groups.put(groupName, new SimpleGroup(groupName));
        }
      }
      // 从user.conf中读取用户信息，并设置用户所在的组
      iterator = readEntry("user.conf");
      while (iterator.hasNext()) {
        userEntry = (String) iterator.next();
        if (userEntry == null) {
          continue;
        }
        elements = userEntry.split("\\t");
```

```
        if (elements.length == 3 && elements[0].startsWith("user=") &&
elements[1].startsWith("password=") && elements[2].startsWith("group=")) {
          user = elements[0].substring(5);
          password = elements[1].substring(9);
          roles = elements[2].substring(6).split(",");
          principal = new SimplePrincipal(user, password);
          users.put(user, principal);
          for (String role : roles) {
            group = groups.get(role);
            if (group != null) {
              group.addMember(principal);
            }
          }
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static SimpleStore getStore() {
    return store;
  }

  public SimpleGroup getGroup(String group) {
    return groups.get(group);
  }

  public Principal getUser(String user) {
    return users.get(user);
  }

  private static Iterator readEntry(String file) throws IOException {
    Vector vector = new Vector();
    BufferedReader buffer = new BufferedReader(new FileReader(file));
    String entry = null;
    while (true) {
      entry = buffer.readLine();
      if (entry == null) {
        break;
```

```
      }
      if (entry.trim().length() == 0 || entry.startsWith("#")) {
        continue;
      }
      vector.add(entry);
    }
    vector.add(entry);
    return vector.iterator();
  }
}
```

6.17.3.2 实现身份验证提供程序

为了实现自己的身份验证提供程序，我们需要实现com.apusic.security.realm.AuthenticationProvider接口，并实现下面的方法,示例如下:

```
// SimpleAuthentication实现
package com.apusic.demo.security.provider;

import java.security.Principal;
import com.apusic.demo.security.store.SimplePrincipal;
import com.apusic.demo.security.store.SimpleStore;
import com.apusic.security.Password;
import com.apusic.security.User;
import com.apusic.security.auth.login.PasswordCredential;
import com.apusic.security.config.RealmConfig;
import com.apusic.security.realm.AuthenticationProvider;
import com.apusic.security.realm.InitialException;

public class SimpleAuthentication implements AuthenticationProvider {
  private SimpleStore store = SimpleStore.getStore();

  // 实现通过user.conf文件中查找是否存在对应的用户
  public boolean authenticate(Object user, Object password) {
    Principal principal = store.getUser((String) user);
    if (principal == null) {
      return false;
    }
    String pwd = new String(((PasswordCredential) password).getPassword());
    if (principal instanceof SimplePrincipal) {
      return ((SimplePrincipal) principal).getPassword().equals(pwd);
```

```
    }
    return false;
  }

  // 用户的Provider可以不实现该接口，一般内部使用
  public boolean authenticate(Object user, Object password, byte[] random) {
    return false;
  }

  // 资源的销毁
  public void destroy() {}

  // 实现通过user.conf文件中查找是否存在对应的用户
  public Object findUser(String userId) {
    User user = null;
    Password password = null;
    Principal principal = store.getUser(userId);
    if (principal == null) {
      return user;
    }
    if (principal instanceof SimplePrincipal) {
      password = new Password(((SimplePrincipal) principal).getPassword());
      user = new User(userId, password);
    }
    return user;
  }

  // 用来完成一些初始化工作，如读取在security.xml文件中配置的参数...等信息
  public void init(RealmConfig config) throws InitialException {}
}
```

6.17.3.3 实现授权提供程序

为了实现自己的授权验证提供程序，我们需要实现com.apusic.security.realm.AuthorizationProvider接口，并实现相关方法。示例如下：

```
//SimpleAuthorization实现
package com.apusic.demo.security.provider;

import java.security.acl.Group;
import com.apusic.demo.security.store.SimpleStore;
```

```
import com.apusic.security.config.RealmConfig;
import com.apusic.security.realm.AuthorizationProvider;
import com.apusic.security.realm.InitialException;

public class SimpleAuthorization implements AuthorizationProvider {
  //资源的销毁
  public void destroy() {}

  //实现通过group.conf文件中查找对应的组信息
  public Group getGroup(String name) {
    return SimpleStore.getStore().getGroup(name);
  }

  //用来完成一些初始化工作，如读取在security.xml文件中配置的参数...等信息
  public void init(RealmConfig config) throws InitialException {}
}
```

```
//SimpleGroup实现
package com.apusic.demo.security.store;

import java.security.Principal;
import java.security.acl.Group;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Enumeration;
import java.util.List;

public class SimpleGroup implements Group {
  private List<Principal> members;
  private String name = null;

  public SimpleGroup(String name) {
    this.name = name;
    members = new ArrayList<Principal>();
  }

  public boolean addMember(Principal user) {
    return members.add(user);
  }
```

```
  public boolean isMember(Principal member) {
    return members.contains(member);
  }

  public Enumeration<? extends Principal> members() {
    return Collections.enumeration(members);
  }

  public boolean removeMember(Principal user) {
    return members.remove(user);
  }

  public String getName() {
    return this.name;
  }
}
```

6.17.3.4 配置映射

完成了上面的验证提供程序和授权提供程序后，我们需要在%DOMAIN_HOME%/config/security.xml文件中添加该安全提供程序，配置如下：

```
<realm>
    <realm-name>SimpleRealm</realm-name>
    <provider-type>Simple Provider</provider-type>
    <authentication-
provider>com.apusic.demo.security.SimpleAuthentication</authentication-
provider>
    <authorization-
provider>com.apusic.demo.security.SimpleAuthorization</authorization-
provider>
</realm>
```

到现在我们就可以在我们自己应用的apusic-application.xml文件中的指定下面的信息,通过SimpleRealm实现安全验证和授权。

```
<realm-name>SimpleRealm</realm-name>
```

6.18 管理集群

6.18.1 集群的基本概念及相关术语

6.18.1.1 可扩展性（Scalability）

标识系统应对逐渐增加负载的能力，对一个大型系统，很难预知最终用户的数量和行为，可扩展性就是指系统能够快速适应负载的增加。通常解决这个问题有两种方式：

1. 通过增加CPU，内存或硬盘，提高系统处理能力。

2. 使用集群，用一组计算机共同分担繁重的工作，对最终用户来说，就好像一台计算机。

6.18.1.2 可用性（Availability）

标识系统供客户使用的时间百分比。可用性不涉及到系统服务客户的效率，它仅仅关注客户能否访问到服务。可用性就是指系统能够快速解决类似网络堵塞，网络延迟等问题，避免系统不可用的情况出现。

6.18.1.3 可靠性（Reliability）

标识系统是否一直能够同预期一样稳健地运行。一旦往系统中添加新的组件，系统的可靠性就容易得到破坏。可靠性就是指系统即使出现故障，都能够保证业务的稳健运行。容错是可靠性系统重要特征。

6.18.1.4 失效转移（Fail-Over）

失效转移是集群中实现可用性的一个重要技术。在集群中，当某个节点失效，服务可以转移到其他节点完成处理。这种转移可以通过硬件或软件实现。

6.18.1.5 负载均衡（Load Balance）

负载均衡是指通过一定的算法将请求合理分配到集群中多个节点，从而达到优化整个系统性能的作用。

6.18.1.6 会话粘滞（Session Stick）

会话粘滞是指在负载均衡模式下，用户的请求每次都被发送到第一次处理该请求的服务器上。这样可以避免集群中会话的频繁复制，提升集群的性能。

6.18.1.7 幂等性（Idempotent）

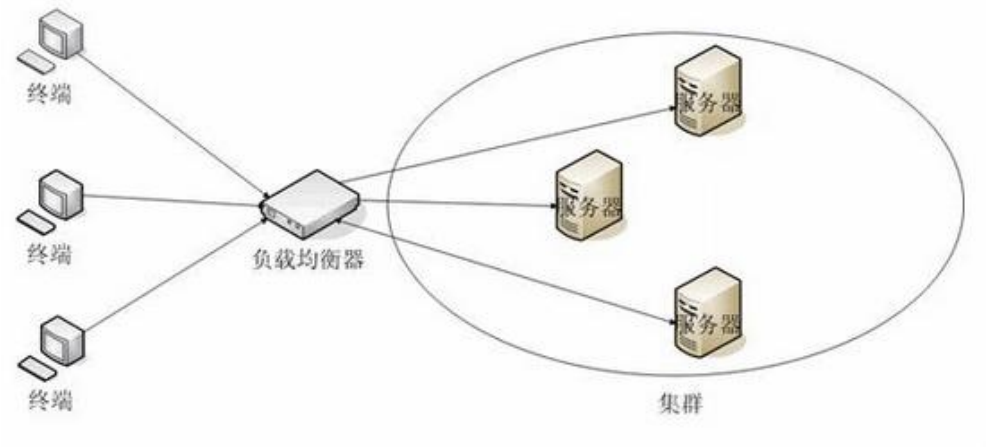
幂等性是指重复使用同样的参数调用同一方法时总能获得同样的结果。比如对同一资源的GET请求访问结果都是一样的。

6.18.2 J2EE集群

J2EE集群包括Web集群，JNDI集群，EJB集群，JMS集群等。当提及J2EE集群时，涉及的两个重要概念：负载均衡，失效转移。

6.18.2.1 负载均衡

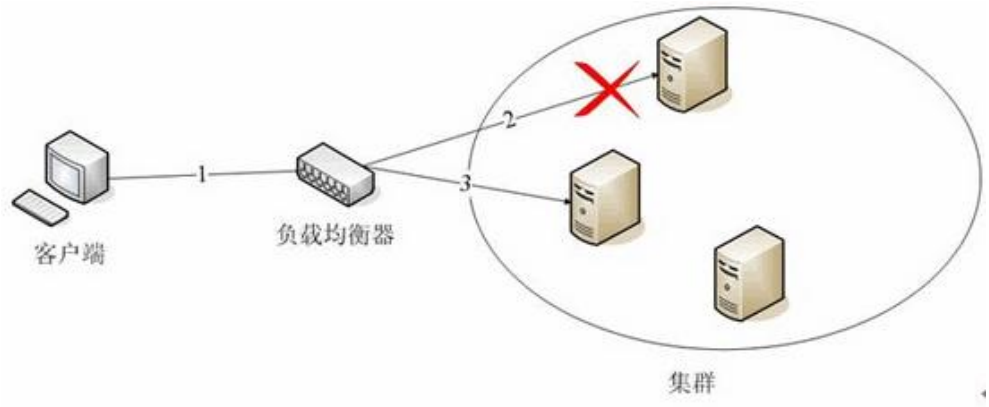
负载均衡是由多台服务器以对称的方式组成一个服务器集合，每台服务器都具有等价的地位，都可以单独对外提供服务而无须其他服务器的辅助。通过某种负载分担技术，将外部发送来的请求均匀分配到集群中的某一台服务器上，而接收到请求的服务器独立地回应客户的请求。负载均衡介于调用者和被调用者之间。如下图所示：



负载均衡器提供相应的算法（如随机算法，轮循算法，权重算法等）来分发和调度客户请求。目前存在各种类型的负载均衡器（硬件如：F5，软件如：apache server，微软的IIS等）。

6.18.2.2 失效转移

失效转移是为了保证集群可用性的一个重要技术手段。当客户端发起请求，处理该请求的服务器出现故障，负载均衡器可以将请求透明的重定向到另一台可用的服务器上。如下图：



在请求重定向到其他服务器时，用户总希望看到一致的应用状态，这将意味着应用状态必须备份到集群的某个地方。不同的服务器对于应用状态的备份采用不同的备份策略，比如：多服务器备份，配对备份，中央服务器备份等。

6.18.3 Apusic集群

Apusic应用服务器对集群服务提供了优异的支持，通过Apusic集群服务，企业应用能够获得高可用性及水平可扩展性.

Apusic集群主要包括Web集群、JNDI集群、EJB集群、JMS集群。

6.18.3.1 Web集群

一般来说，Web集群试图解决两个问题：客户请求的负载均衡和Session的高可用性。

Apusic Web集群为这两个问题提供了灵活、全面的解决方案。客户请求的负载均衡是指客户的请求依赖特定算法被合理地分配给多台Web Server来处理。Session的高可用性是指当某台Web Server失效，这台Web Server服务的客户的请求会被透明地转发给其它有效Web Server，而会话状态（Session）依然可用。我们把集群范围内具有高可用性的Session称为集群Session。

Apusic Session集群采取Instant Replication，即某节点的Session操作是即时传播（同步）到集群中的其它节点的；一些应用服务器厂商的集群Session同步采取非即时的方式，这会降低Session可用性。还有一个比较常见的场景是新节点加入工作中的Apusic集群时，新节点的Session会自动与集群同步。

接下来我们来看一下，使用Apusic LoadBalance和第三方LoadBalance作为负载均衡器时，分别如何实现客户请求的负载均衡和Session的高可用。

6.18.3.1.1 APUSIC负载均衡

Apusic 在web层实现了负载均衡。利用Apusic建立的负载均衡集群有一个很重要的特点就是Apusic实现了分布式Session管理，这也是实现Web服务器集群的关键，由于Web应用被分布到多台服务器上运行，因此保存在Session中的共享数据必须完全保持一致。Apusic没有使用共享数据库来保存Session数据，虽然这种方式能保证Session数据的一致性，但由于Session的变化是很频繁的，对数据库将造成很大的压力，最终将成为整个系统的瓶颈，Apusic使用一种分布式Session服务，每个服务器管理自己所产生的Session。当Web应用从一台服务器迁移到另一台服务器时，Session也会自动进行迁移，这样使得对Session的管理被均匀地分布到所有的服务器上，任何一台服务器失效并不会使Session丢失。对应用开发者来说，要保证分布式Session能够正常工作，在Session中只能保存实现了java.io.Serializable的数据，否则Session将无法完成迁移。

一. 负载均衡策略

采用Apusic作为负载均衡器时， Apusic提供三种负载均衡策略：

- 随机选择策略随机选择其中一台服务器处理请求。
- Round-Robin策略依次轮寻选择一台服务器处理请求。
- 权重策略按照权重的比例选择服务器处理请求。

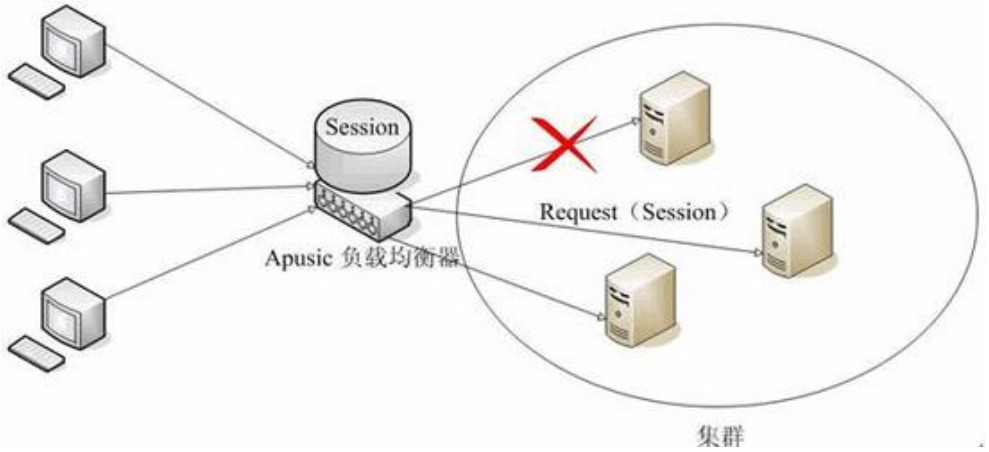
当然，用户也可以手动扩展自己的负载均衡策略，如何扩展负载均衡策略，请参考配置负载均衡。

二. Apusic负载均衡器的Session 复制策略

Apusic LoadBalancer使用了内存复制技术，将每个后置服务器的Session备份在自己的缓存中，即后台某个节点处理完客户端请求时，将当前Session复制到负载均衡器中。所以使用Apusic LoadBalance时，集群中每个节点必须开启SessionService中的复制服务（如何开启复制服务请参考配置负载均衡）。

三. 失效转移

采用Apusic LoadBalance时，由于Session缓存在Apusic LoadBalance的缓存中，当主节点失效时，Apusic LoadBalance从缓存取出当前的Session，将Session带到下一个节点进行处理，确保Session的可用性。如下图：



四. **配置负载均衡**

Apusic 负载均衡器是用Apusic应用服务器实现的，也就是说，负载均衡器和其他Apusic应用服务器没有多少区别，不同的是它上面没有安装应用系统，并且使用不同的配置文件，它只是专门提供负载均衡服务。客户通过访问负载均衡器的Web服务，来访问整个集群的资源。用户请求到达负载均衡器后，负载均衡器将它分配到某个Apusic应用服务器上，让这个应用服务器为此用户提供服务。

负载均衡器配置文件为$DOMAIN_HOME/config/loadbalancer.conf，缺省内容见附录,下面是其中LoadBalancer服务的片段：

```
<SERVICE CLASS="com.apusic.web.loadbalancer.LoadBalancer">
    <ATTRIBUTE NAME="ServerPort" VALUE="80" />
    <ATTRIBUTE NAME="BackendServers"
VALUE="192.168.6.66:6888,192.168.6.67:6888" />
    <ATTRIBUTE NAME="MaxWaitingClients" VALUE="500" />
    <ATTRIBUTE NAME="WaitingClientTimeout" VALUE="5" />
    <ATTRIBUTE NAME="KeepAlive" VALUE="True" />
    <ATTRIBUTE NAME="MaxKeepAliveRequests" VALUE="100" />
    <ATTRIBUTE NAME="KeepAliveTimeout" VALUE="15" />
    <ATTRIBUTE NAME="MaxKeepAliveConnections" VALUE="300" />
    <ATTRIBUTE NAME="EnableLog" VALUE="False" />
    <ATTRIBUTE NAME="LogFileName" VALUE="logs/access.log" />
    <ATTRIBUTE NAME="LogFileLimit" VALUE="1000000" />
    <ATTRIBUTE NAME="LogFileCount" VALUE="10" />
</SERVICE>
```

与Apusic应用服务器的配置文件config/apusic.conf比较一下，可以发现少了一些服务描述，但多了一个LoadBalancer服务。

当用这个配置文件启动Apusic应用服务器时，这台服务器就成为一个负载均衡器。

在配置文件中，最关键的属性BackendServers，定义了集群中的服务器，这是一个用逗号分割的地址列表，包括主机名和端口号。

Apusic负载均衡器默认情况下是会话粘滞（session-stick）的，同时采用轮循的策略选择可用节点。当然你也可以关闭会话粘滞，只要在LoadBalancer服务中，增加属性：

```
<ATTRIBUTE NAME="SessionStick" VALUE="false" />
```

就可以关闭会话粘滞。如果想采用其他策略来选择节点，可以增加属性：

```
<ATTRIBUTE NAME="BalancePolicy" VALUE="Random" />
```

其中Value值可以选择：Random，Round-Robin，LoadWeight.注意当用户选择LoadWeight的时候，还需要增加属性：

```
<ATTRIBUTE NAME="LoadWeight" VALUE="10,20,70" />
```

来标识权重的值。其值与服务器地址列表中的服务器一一对应。此外，用户还可以扩展自己的负载均衡策略，只要实现LoadBalancePolicy接口，同时增加属性：

```
<ATTRIBUTE NAME="BalancePolicyClass" VALUE="userclass " />
```

其中Value值为用户自定义的类。当两个属性BalancePolicy，BalancePolicyClass同时存在时，BalancePolicyClass的优先级高。

	**注意**
	要使用Session的分布式功能，集群中每台Apusic服务器的SessionService中属性：值必须为True。默认情况下为False。

当上面的配置完成后，负载均衡器可以通过命令行方式启动，不同的是需要增加-config参数指明负载均衡器的配置文件，例如：java -classpath E:\apusic\lib\apusic.jar;E:\apusic\lib\javac.jar;E:\apusic\lib\mejb.jar;E:\apusic\common\javaee.jar;com.apusic.server.Main -root E:\apusic\domains\mydomain -config /config/loadbalancer.conf

6.18.3.1.2 APUSIC HTTP SERVER负载均衡

Apusic Http Server是在Apache Http Server基础上对其负载均衡功能进行了优化而产生的Apusic Web服务器，其与Apusic应用服务器有更好的耦合性。详细请参考：使用Apusic Http Server作为负载均衡前置机

6.18.3.1.3 第三方负载均衡

Apusic集群对第三方的负载均衡（如：硬件负载均衡器F5， 软件负载均衡器Apache Server，微软的IIS等）提供良好的支持，用户只需要简单配置就可以使第三方的负载均衡器与Apusic完美结合在一起。

一. Session复制策略

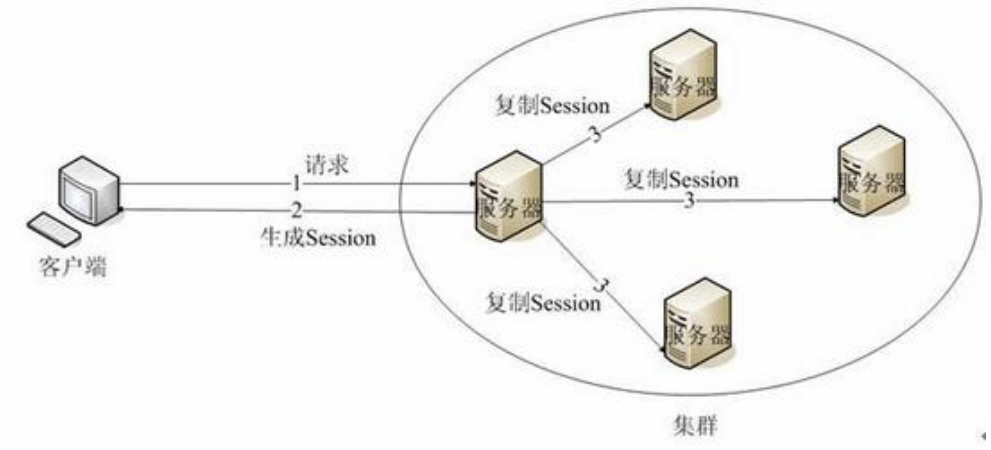
当采用第三方负载均衡器时，Apusic同样采用内存复制技术，与Apusic LoadBalancer不同的是，我们已经不能用第三方负载均衡器来备份状态，下面介绍两种其他方法。

第一种方式，使用IP多播技术:

在Apusic内部服务器之间进行内存复制的方案，这时候，就要开启Apusic应用服务器的集群服务。Apusic集群采用两种Session复制策略：多点复制，配对复制，用户可以根据自己的需要选择不同的复制策略。默认情况下为配对复制。

- 多点复制

多点复制即一个节点上的Session会即时复制到集群中其他节点。也就是说一个节点的Session在集群中同时存在多份备份。如下图：

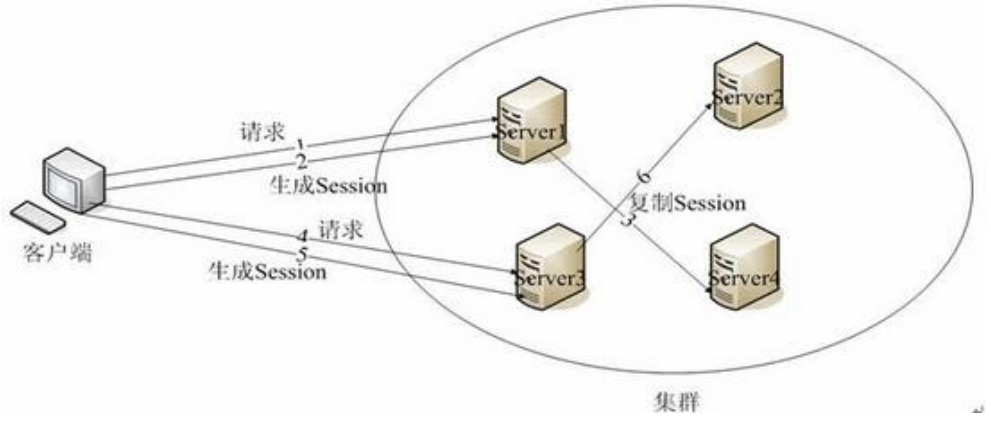


当集群中某个节点收到来自客户端的请求（1），服务器生成Session（2），在请求处理完后，Apusic服务器会将session复制到集群中其他节点（3）。

在使用多点复制，当并发量很大的情况下，容易造成网络风暴，大大降低系统性能。

- 配对复制

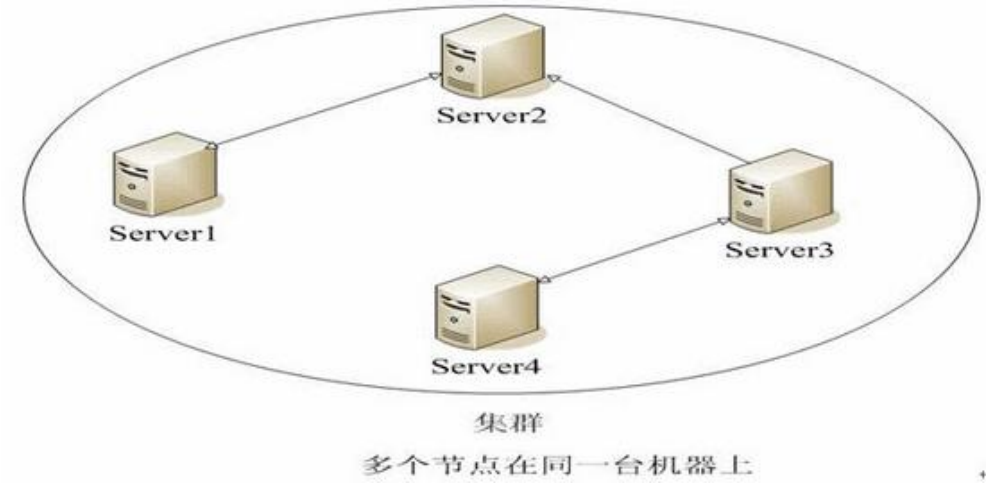
所谓的配对复制，即一个节点的Session会即时复制到与其配对的服务器上。也就是说对于任一个节点的Session，在集群中只存在一份备份。当客户使用第三方的负载均衡器，并且负载均衡器提供Session stick功能时，使用配对复制能够提高服务器性能，减少网络风暴，但如果第三方的负载均衡器不提供Session stick功能时，使用配对复制并不会比多点复制性能好。如下图：



图中，Server1与Server4互为备份，Server2与Server3互为备份。当Server1收到来自客户端的请求（1），Server1生成Session（2），当请求处理完成后，将Session复制到Server4（3）。当Server3收到客户端请求（4），Server3生成Session（5），当请求处理完成后，将Session复制到Server2（6）。此时Server4中备份了Server1中的Session，Sever2中备份了Server3中的Session。

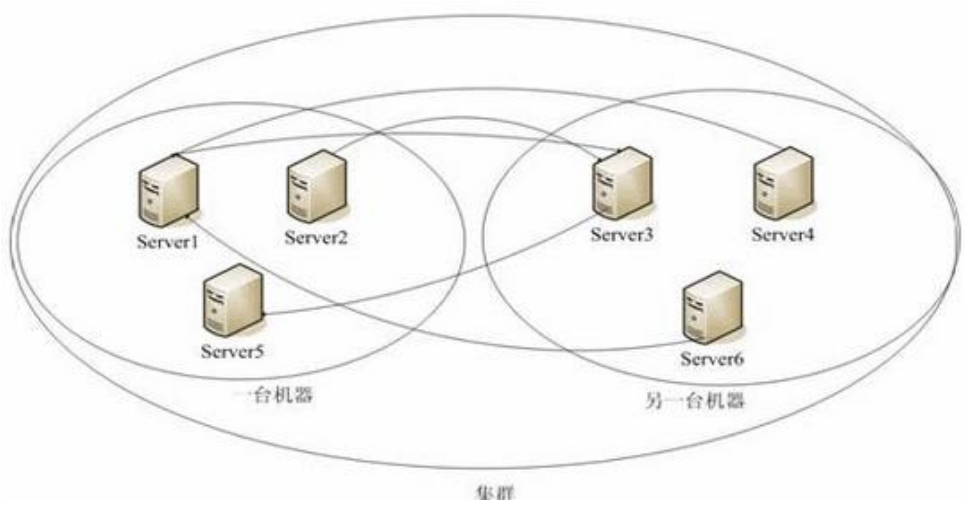
- 节点配对

Apusic提供两种方式设置节点配对：默认设置，手动设置。在默认设置的情况下，新加入节点寻找备份节点的策略为：首先找到第一个非本机的节点作为自己的备份节点，如果找不到，则会以本机的前一个节点作为自己的备份节点。如下图：



假设Server1，Server2，Server3，Server4依次加入集群，则Server1，Server2互为备份，Server3会以Server2为自己的备份，Server4以Server3为自己的备份。

当多个节点分布在不同的机器上时，如下图：



假设Server1，Server2，Server3，Server4，Server5，Server6依次加入集群，其中，Server1，Server2，Server5在同一台机器上，Server3，Server4，Server6在另一台机器上。当Server1 加入时，此时没有任何配对节点，Server2加入，Server1与Server2互为配对。当Server3加入，Server3选择Server1 为自己备份节点，此时Server1，Server2同时选择非本机的Server3为自己的备份节点。Server4加入，Server4选择Server1为自己的备份节点。Server5加入，Server5选择Server3为自己的备份节点，Server6加入，Server6选择Server1为自己的备份节点。

在默认设置的情况下有可能出现一台服务器成为多台服务器备份的情况，Apusic提供手动设置集群配对（如何手动设置节点配对，请参考配置Apusic应用服务器）。 假设用户有四台服务器Server1，Server2，Server3，Server4，设置的配对为Server1与Server2互为备份，Server3与Server4互为备份。当节点依次加入集群时，Server1加入，没有备份，Server2加入，由设置的配对关系知Server1与Server2互为备份，当Server3加入，由于Server4没有加入，遵循默认节点加入寻找规则，找到Server1作为自己的备份，当Server4加入，Server4根据设置找到Server3作为自己备份，Server3这时发现自己的配对节点加入集群，重新找到Server4作为自己的备份。

- 节点状态同步

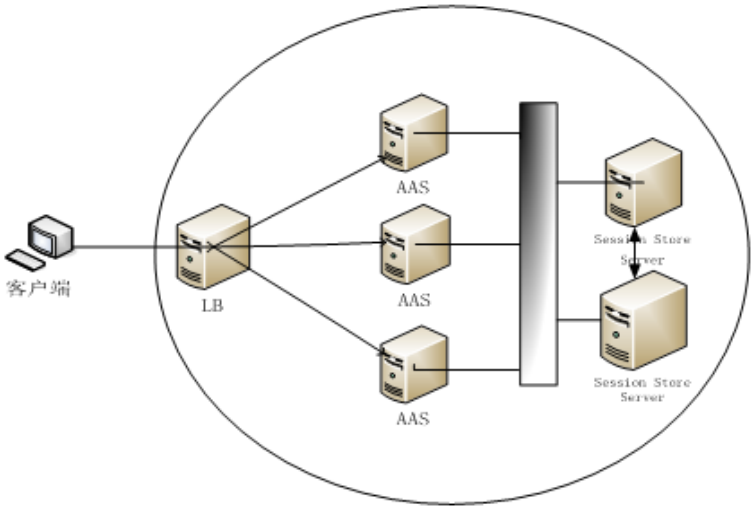
作为备份节点的服务器，必需要同步主服务器的状态（包括Session，Jndi等），这就涉及到节点状态同步。Apusic服务器提供节点状态的即时同步，即当主服务器选择了某个节点作为自己的备份服务器，这时主服务器会要求备份服务器节点同步主服务器节点的状态。

第二种方式，使用session分布式存储的：

Session 分布式存储主要是为了满足构建高水平扩展性及高可用性Web应用的需要。

Session分布式存储将Http Session集中存储在Key-Value DB中，由于Key-Value DB 提供了较高的读写性能及并发保护，给应用服务器水平扩展提供了有力支持。

Session分布式存储拓扑如下图所示：



在集群中，应用运行在多个应用服务器中，若需提供失效转移的高可用性能力，Session数据在应用服务器中需要进行共享。

Session 迁移的方式会造成网络风暴和服务器有效处理能力的下降，而在分布式存储中，Session 全部存储在Session Store Server中，节点间不再需要同步任何数据，当访问人数增加时，只需简单的添加应用服务器，提高系统的处理能力，实现了高可扩展性，且如果某个节点宕机后，其他节点将取代它，从Server中取得Session数据，继续为客户服务，实现高可靠性。

所以，分布式存储较适合于集群节点较多，且要求失效转移的高可用性、系统高扩展性的应用场合。

二. 配置Web集群

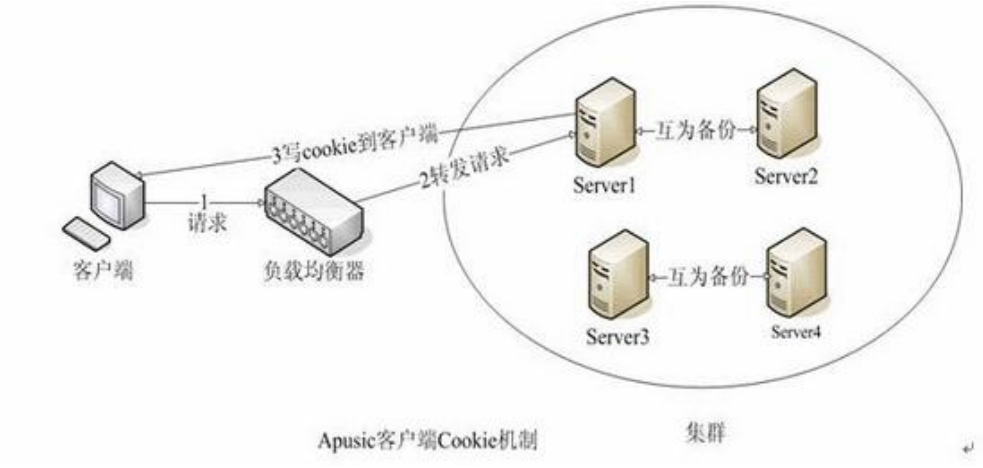
具体配置信息请参考"配置Apusic应用服务器"

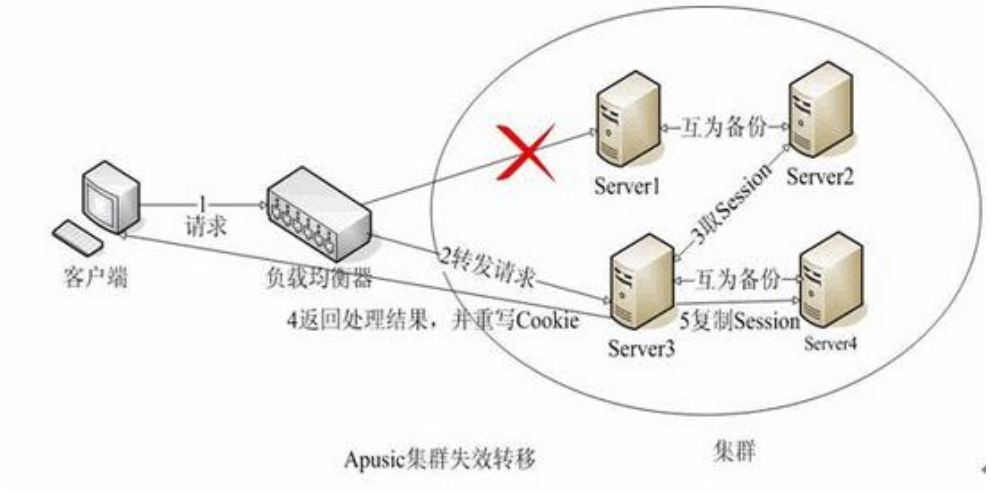
三. Web集群失效转移

Web 集群的失效转移主要着眼于当处理请求的主服务器离开，当前用户的Session是否能够正确转移到其他的服务器上进行处理。在多点复制的情况下，主服务器离开，请求随机转发到其他服务器上，由于其他服务器存在当前的Session，这时可以继续处理用户的请求。在配对复制的情况下可能存在两种情况：

- 主服务器离开，请求被转发到备份服务器。这种情况和多点复制一样，备份服务器可以继续处理用户的请求。

- 主服务器离开，请求被转发到非备份服务器，这种情况下，非备份服务器会从备份服务器那将Session复制过来，继续处理请求。 那么当前服务器如何能够找到备份服务器呢？Apusic使用客户端Cookie机制，存储主服务器及备份服务器名称。当主服务器离开，请求发送到非备份服务器，非备份服务器通过Cookie里面的值知道备份服务器。如下图：





6.18.3.1.4 APUSIC SESSION复制粒度

对不同的服务器厂商，对Session的复制粒度各有不同，典型的有三种：

- 复制全部Session 每次备份所有的Session，这种方案最简单。
- 复制新建及被修改的Session 当一个Session被创建或者被修改，服务器会将该Session复制到备份服务器。这种方案相比第一种方案能够大幅度提高性能。但是如何界定一个Session被修改，通常认为调用Session.setAttribute()，Session.removeAttribute（）时，我们认为Session被修改。另外，对于被访问过但未修改的Session，Apusic会更新其备份服务器上Session的访问时间。
- 复制发生变化的属性在Session发生变化时，不是备份整个Session，只对单个属性备份，这种方案能够取得更好的性能和更好的网络消耗。但是这时必须确保属性是可序列化的。同时也存在交叉引用的情况。

Apusic在Session的复制上，采用了第二种复制粒度。即对新建的和被修改Session，Apusic服务器才复制到备份服务器。另外备份Session发生的时机为完成每次请求后，Apusic服务器自动将修改的Session备份到备份服务器。在备份Session过程中，如果发现

Session中的某个属性不可序列化，则默认丢弃，同时后台会有警告提示信息。当某个实现序列化的属性中存在非序列化的对象时，会在序列化过程中出错。

6.18.3.1.5 WEB集群方案

对于只需要负载均衡集群的用户，有三种方案：

- 前端，采用硬件负载均衡器；后端，部署多台Apusic应用服务器。
- 前端，部署Apusic LoadBalancer；后端，部署多台Apusic应用服务器。
- 前端，采用第三方软件负载均衡器或者Apusic Http Server；后端，部署多台Apusic应用服务器。

对于同时需要负载均衡和Session高可用的用户，则需要在多台Apusic应用服务器之间共享Session数据。当一台服务器失效后，可以把用户请求重定向到另外一台服务器进行服务，不会导致数据的丢失。服务器之间共享Session数据有2种方式：

启用Apusic应用服务器的集群服务和Session迁移功能。

使用分布式存储对Session进行存储。

这两种方式如何配置，请参考"配置Apusic应用服务器"。

以下着重讲述当前比较流行的Apache Server作负载均衡器，在Windows环境下如何配置Apusic集群。

1. 到Apache网站http://httpd.apache.org/下载Apache Server，目前版本为2.26.
2. 安装Apache Server,假设安装目录为：C:\Program Files\Apache Software Foundation\Apache2.2。到conf目录下打开文件httpd.conf,确保选项：

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_connect_module modules/mod_proxy_connect.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

处于开启状态。同时在文件末尾加入如下配置：

```
<VirtualHost *:80>

ProxyRequests off
ProxyPass / balancer://test/
<Proxy balancer://test>

BalancerMember http://localhost:6888 loadfactor=1
BalancerMember http://localhost:5888 loadfactor=1
BalancerMember http://localhost:4888 loadfactor=1
```

```
BalancerMember http://localhost:3888 loadfactor=1
</Proxy>
</VirtualHost>
```

其中80端口是用户安装时配置的http协议监听端口，确认是否为80端口，可以查看Listen选项。BalancerMember为后置机节点，后面的值为后置机的地址和端口。Loadfactor为负载权重。当用户想使用会话粘滞（Session-Stick），可以在ProxyPass / balancer://test/ 后面加入stickysession=JSESSIONID，当用户想使用失效转移时要在后面加入nofailover=off，同时在每个BalancerMember最后面加入route=serverName。最终的配置形式如下：

```
<VirtualHost *:80>

ProxyRequests off
ProxyPass / balancer://test/ stickysession=JSESSIONID nofailover=off
<Proxy balancer://test>

BalancerMember http://localhost:6888 loadfactor=1 route=server3
BalancerMember http://localhost:5888 loadfactor=1 route=server2
BalancerMember http://localhost:4888 loadfactor=1 route=server1
BalancerMember http://localhost:3888 loadfactor=1 route=server0
</Proxy>
</VirtualHost>
```

3. 配置Apusic应用服务器，如何配置apusic集群服务，请参考配置Web集群。

当用户想使用Session Stick时，只需要在apusic.conf文件中的SessionService中增加如下属性：

```
<ATTRIBUTE NAME="SessionStick" VALUE="true" />
```

4. 所有配置完成后，启动Apache Server和Apusic应用服务器.

注意：同时启动几台Apusic节点可能造成节点握手不成功，导致后加入节点不能正确加入集群，建议启动节点要依次加入。

5. 当使用Session配对复制时，用户可以通过Apusic AdminConsole设置节点之间配对关系。

6.18.3.1.6 SESSION分布式存储

一. 为什么需要Session分布式存储

Apusic应用服务器已经提供了文件存储，BerkelyDB存储及数据库存储Session的方式，这都极大地满足了Session钝化及备份的需求，然而，为什么还需要Session分布式存储呢?

Session分布式存储主要是为了满足构建高水平扩展性及高可用性Web应用的需要。若构建高水平扩展性、高可用性的Web应用，一般而言对Http Session有两种处理方案：

1. 基于数据库的Session共享，实现分布式应用间Session共享。此种方案把Session信息存储到数据库表，这样实现不同应用服务器间Session信息的共享。
2. Session迁移。Session迁移即一个节点上的Session会即时复制到集群中其他节点。

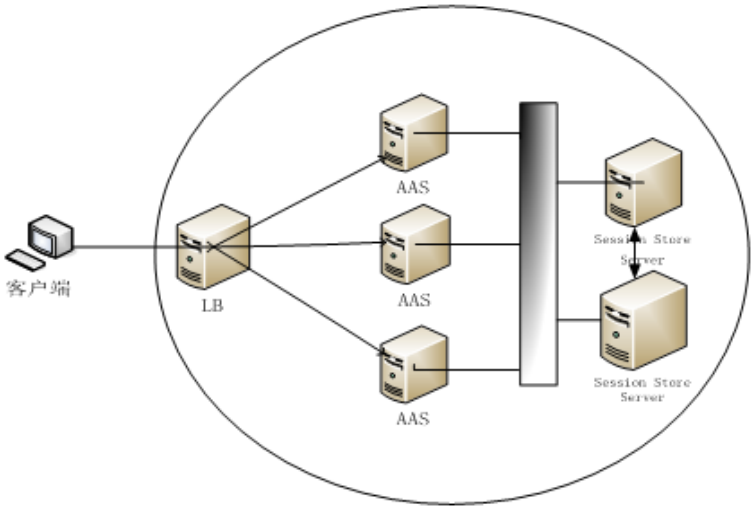
以上两种方案都有其固有的缺点：前者的缺点是由于数据库服务器相对于应用服务器更难扩展且资源更为宝贵，在高并发的Web应用中，最大的性能瓶颈通常在于数据库服务器。

因此如果将Session存储到数据库表，频繁的增加、删除、查询操作很容易造成数据库表争用及加锁，最终影响业务。后者的缺点为：

当并发量很大的情况下，容易大量Session数据传输，造成网络风暴，同时服务器也需要不断对数据进行处理，减弱了服务器服务客户的有效处理能力，大大降低系统性能。

Session分布式存储较好的解决了以上问题，Http Session集中存储在Key-Value DB中，由于Key-Value DB 提供了较高的读写性能及并发保护，给应用服务器水平扩展提供了有力支持。

Session分布式存储拓扑如下图所示：



二. 应用场景及优点

Session 分布式存储主要用于构建满足可水平扩展及高可用Web应用集群。Web应用集群各个节点都把Http Session集中存储在一个或多个内存数据库中，避免各个节点相互同步HttpSession信息而造成的网络风暴，而各个节点也不需要对同步过来的HttpSession信息进行处理，提升了节点的服务有效处理能力。内存数据库以key-value的形式保存HttpSession信息，有很高的读写性能。由于集群各个节点HttpSession信息都使用相同的存储，如果需要在集群增加一个节点，只需要在新节点部署应用，然后把HttpSession信息存储到相同的存储即可，这就完成了给应用集群水平扩展，对其他节点毫无影响。

三. 配置分布式存储

配置Apusic应用服务器，如何配置apusic集群服务，请参考"配置Session存储"。

四. 分布式Session存储服务器安装

Apusic应用服务器支持TokyoTyrant/TokyoCabinet 及Memcached-like 存储服务器。

Apusic应用服务器推荐使用TokyoTyrant，但其现阶段只支持Linux系统；Windows下可以考虑Membase，需将apusic.conf配置文件中TokyoTyrant设置为False。

Tokyo tyrant安装：

请先安装tokyo cabinet,再安装tokyo tyrant.后者依赖前者

1. 编译安装tokyo cabinet数据库

```
wget http://fallabs.com/tokyocabinet/tokyocabinet-1.4.47.tar.gz
tar zxvf tokyocabinet-1.4. 47.tar.gz
cd tokyocabinet-1.4. 47/
./configure
make
make install
```

2. 编译安装tokyotyrant

```
wget http://tokyocabinet.sourceforge.net/tyrantpkg/tokyotyrant-1.1.29.tar.gz
tar zxvf tokyotyrant-1.1.29.tar.gz
cd tokyotyrant-1.1.29/
./configure
make
make install
```

3. Tokyo tyrant 启动

启动tokyotyrant的主进程（ttserver）

- 单机模式

```
ulimit -SHn 51200
ttserver -host 127.0.0.1 -port 11211 -thnum 8 -dmn -pid
/ttserver/ttserver.pid -log
/ttserver/ttserver.log -le -ulog /ttserver/ -ulim 128m -sid 1 -rts
/ttserver/ttserver.rts /ttserver/database.tch
```

- 双机互为主辅模式（相互备份）

服务器192.168.1.91:

```
ulimit -SHn 51200
ttserver -host 192.168.1.91 -port 11211 -thnum 8 -dmn -pid
/ttserver/ttserver.pid -log
/ttserver/ttserver.log -le -ulog /ttserver/ -ulim 128m -sid 91 -mhost
192.168.1.92 -mport 11211
rts /ttserver/ttserver.rts /ttserver/database.tch
```

服务器192.168.1.92:

```
ulimit -SHn 51200
ttserver -host 192.168.1.92 -port 11211 -thnum 8 -dmn -pid
/ttserver/ttserver.pid -log
/ttserver/ttserver.log -le -ulog /ttserver/ -ulim 128m -sid 92 -mhost
192.168.1.91 -mport 11211
rts /ttserver/ttserver.rts /ttserver/database.tch
```

- 参数说明

```
ttserver -host -port -thnum -tout [-dmn] -pid -log [-ld|-le] -ulog -ulim [-
uas] -sid -mhost -mport -rts [dbname]
```

host name : 指定需要绑定的服务器域名或IP地址。默认绑定这台服务器上的所有IP地址。

port num : 指定需要绑定的端口号。默认端口号为1978

thnum num : 指定线程数。默认为8个线程。

tout num : 指定每个会话的超时时间（单位为秒）。默认永不超时。

dmn : 以守护进程方式运行。

pid path : 输出进程ID到指定文件（这里指定文件名）。

log path : 输出日志信息到指定文件（这里指定文件名）。

ld : 在日志文件中还记录DEBUG调试信息。

le : 在日志文件中仅记录错误信息。

ulog path : 指定同步日志文件存放路径（这里指定目录名）。

ulim num : 指定每个同步日志文件的大小（例如128m）。

uas : 使用异步IO记录更新日志（使用此项会减少磁盘IO消耗，但是数据会先放在内存中，不会立即写入磁盘，如果重启服务器或ttserver进程被kill掉，将导致部分数据丢失。一般情况下不建议使用）。

sid num : 指定服务器ID号（当使用主辅模式时，每台ttserver需要不同的ID号）

mhost name : 指定主辅同步模式下，主服务器的域名或IP地址。

mport num : 指定主辅同步模式下，主服务器的端口号。

rts path : 指定用来存放同步时间戳的文件名。

dbname：数据库名字。Tokyo Tyrant使用数据库名字后缀指定数据库类型。

下面我们再来看下数据库类型的详细配置:

如果数据库名为"*"，表示内存hash数据库。

如果数据库名为"+"表示内存tree数据库。

如果数据库名为".tch",则数据库为hash数据库。

如果数据库名的后缀为".tcb"，数据库将为B+ tree数据库。

如果数据库名的后缀为".tcf"。则数据库将为fixed-length数据库。

如果数据库名的后缀为".tct",则数据将为一个table数据库（有表的概念）。

停止tokyotyrant（ttserver）

ps -ef | grep ttserver,找到ttserver的进程号并kill，例如：kill -TERM 2159

五. **安装中可能遇到问题及解决办法**

在Ubuntu，OpenSuse中可能因为依赖而导致不成功，具体问题如下：

1. 缺少bzlib.h文件

解决方法：下载及安装这两个安装包

i、bzip2-1.0.4.tar.gz

ii、zlib-1.2.3.tar.gz

2. tc make file时候，当碰到依赖libbz2.a或者libz.a这两个库的时候会出现编译错误。

```
//*****************************************************************************//

/usr/bin/ld: /usr/local/lib/libbz2.a(bzlib.o): relocation R_X86_64_32S
against `a local symbol' can not be used
when making a shared object; recompile with -fPIC
/usr/local/lib/libbz2.a: could not read symbols: Bad value
collect2: ld 返回 1
make: *** [libtokyocabinet.so.8.22.0] 错误 1
//*****************************************************************************//
```

```
/usr/bin/ld: /usr/local/lib/libz.a(crc32.o): relocation R_X86_64_32 against
`a local symbol' can not be used when
making a shared object; recompile with -fPIC
/usr/local/lib/libz.a: could not read symbols: Bad value
make: *** [libtokyocabinet.so.8.22.0] 错误 1
//*****************************************************************************//
```

解决方法：

1. 删除文件libbz2.a，libz.a

如果发现 libbz2.a: could not read symbols: Bad value,就应该把/usr/local/lib中的libbz2.a删掉。进入解压缩的zlib-1.2.3目录，用make clean命令清理一下。

同样，/usr/local/lib/libz.a: could not read symbols，操作同上，删掉libz.a，把bzip2-1.0.4目录的编译文件清理一下。如果找不到这两个文件的位置，可以在终端敲上：find -name libbz2.a。

2. 修改zlib-1.2.3的Makefile文件

把gcc的编译参数加上 -fPIC，原文：CFLAGS=-O3 -DUSE_MMAP修改为：CFLAGS=-O3 -DUSE_MMAP -fPIC。如果还是过不去，劝你硬来CC=gcc 直接后面跟上-fPIC让他们全部独立编译。

重申：如果你之前编译过了，一定要用make clean清掉，否则还是徒劳。最后make 还有 make install。

3. 修改bzip2-1.0.4的Makefile文件

```
CC=gcc -fPIC
AR=ar
RANLIB=ranlib
LDFLAGS=

BIGFILES=-D_FILE_OFFSET_BITS=64
CFLAGS=-fPIC -Wall -Winline -O2 -g $(BIGFILES)
```

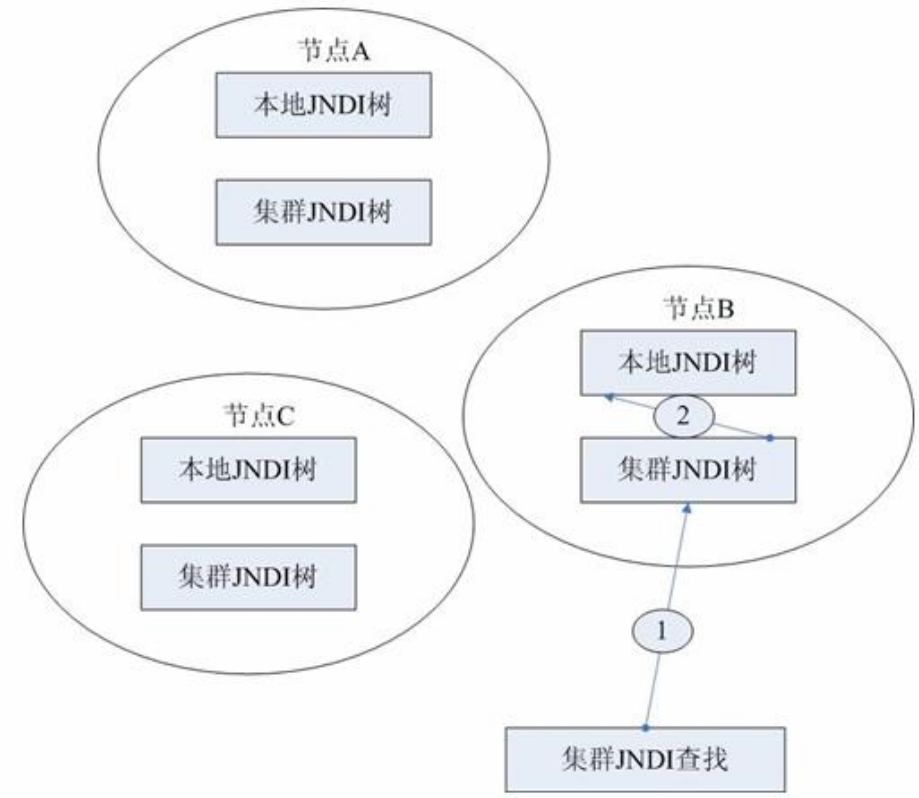
4. 重新安装Tokyo cabinet

6.18.3.2 JNDI集群

JNDI作为JEE的基础技术，是J2EE中提供使用资源的间接层。JNDI集群往往是其它上层集群技术的必要条件。

6.18.3.2.1 本地JNDI和JNDI集群

我们知道，JNDI命名是以层次结构来组织的，在示意图中，往往用树来表达，所以称为JNDI树。



那么，JNDI集群又是怎么回事呢?

JNDI集群是Apusic服务器提供的一个服务，一旦启用这个服务，集群的各节点（指Apusic服务器）都拥有一个集群JNDI树，它有别于本地JNDI树，事实上，在JNDI集群节点中，这两棵树是同时存在的，那么它们的区别和联系又在哪里?

各节点的本地JNDI树是彼此独立的，遵循J2EE JNDI标准语义、习惯用法，而它们的集群JNDI树之间会遵循如下规则协作。

- 在集群中，某个服务器节点集群JNDI树上的操作都会传播到其它节点的集群JNDI树上。譬如在某节点bind("a","a")，那么其它节点的集群JNDI树上也会存在名值对("a","a")。当然，这里可能出现名字冲突的问题，譬如，A节点首先bind("a","a")，并且传播到其它节点，随后，B节点bind("a","aa")，那么会出现什么情况呢？对于这种情况，我们只会将B节点的a与aa绑定，而集群内的其它节点保持不变。如果你需要集群范围所有的集群JNDI树上的名值对("a","a")全部更新为("a","aa")，你需要使用rebind方法。
- 当你持有一个集群JNDI上下文，并且使用它lookup的时候，它首先在集群JNDI树上搜索，搜不到，还会到本地JNDI树上搜索（如上图所示）。

6.18.3.2.2 JNDI 负载均衡和JNDI失效恢复

什么是JNDI负载均衡（LoadBalance）和JNDI失效恢复（Fail-Over）？它试图解决的是什么问题?

场景：某公司C部署了一台Apusic服务器S1，并且通过本地JNDI发布了服务（n,s），系统客户端遵循常规JNDI用法，指定S1为provider。显然，服务器都存在失效的可能，而且，随着业务的扩展，服务量的增加，一台服务器，已经不能满足需求，服务质量（系统响应时间和系统可用性）急待提高。公司又增加两台服务器S2、S3，S2、S3同样通过本地JNDI发布了服务(n,s)，此时，系统客户端怎样和S1、S2、S3交互成为问题的关键所在。理想的情况是，系统客户端的服务请求被合理地分配到S1、S2、S3上，一旦，其中一台或多台服务器失效，服务请求会透明地、合理地被分配到剩下的有效的服务器上。

上述场景中的服务请求被合理地分配到多台服务器的行为就称作JNDI负载均衡，而一台或多台服务器失效后，服务请求通明、合理地被分配到剩下地有效的服务器上的行为就称作JNDI失效恢复.

6.18.3.2.3 JNDI集群配置

启用Apusic JNDI集群服务，你需要在%DOMAIN_HOME%\config\apusic.conf中配置ClusterNameService服务，片断如下:

```
<SERVICE CLASS="com.apusic.naming.cluster.ClusterNameService" />
```

当然，由于ClusterService服务是ClusterNameService服务的基础，所以，你需要保证在%DOMAIN_HOME%\config\apusic.conf中已经配置ClusterService服务(如何配置ClusterService服务请参考配置Web集群).

上面提及，无论使用本地JNDI还是集群JNDI，首先我们必须获取相应的JNDI上下文（一般都是初始上下文）。本地JNDI的使用遵循JNDI标准，示例代码片断:

```
Hashtable env = new Hashtable();
//You can specify Apusic-specific properties here.
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.apusic.naming.jndi.CNContextFactory");
env.put(Context.PROVIDER_URL,"iiop://192.168.6.55");
//something omitted.
...
Context ctx = new InitialContext(env);
```

如果你试图获取集群JNDI初始上下文而不是本地JNDI初始上下文，那么你需要在客户端设定环境变量:

```
apusic.naming.clustering=true
```

那么在哪里设定上述环境变量呢？你可以使用命令行参数；你也可以通过编码在JNDI初始上下文属性中指定；甚至，你可以将之写入JNDI实现jar包的jndi属性配置文件中（不推荐）。

下面是第二种方式的示例代码片断:

```
Hashtable env = new Hashtable();
//You can specify Apusic-specific properties here.
env.put("apusic.naming.clustering","true");
env.put(Context.INITIAL_CONTEXT_FACTORY,"
com.apusic.naming.jndi.CNContextFactory);
env.put(Context.PROVIDER_URL,"iiop://192.168.6.55");
//something omitted.
...
Context ctx = new InitialContext(env);
```

为了启用Apusic JNDI负载均衡和失效恢复能力，你只需要在客户端设定环境变量java.naming.provider.url=<provider-list>.你可以通过多种方式来设定。推荐方式是在JNDI初始上下文的PROVIDER_URL属性中指定<provider-list>，<provider-list>格式示例如下：

```
iiop://S1:6888,S2:6888,S3:6888
```

显然，硬编码总是缺乏灵活性，所以，Apusic还提供了JNDI provider的发现(discovery)功能，你不需要通过PROVIDER_URL属性指定任何provider，你只需要在客户端设定如下环境变量：

```
apusic.naming.discovery=true
apusic.naming.discovery.address=230.0.0.2
apusic.naming.discovery.port=1500
apusic.naming.discovery.timeout=5000
apusic.naming.discovery.cluster_name= ApusicCluster
```

其中，apusic.naming.discovery=true是必须的，剩下的四个环境变量，除apusic.naming.discovery.cluster_name，其它三个如果省略，那么如上默认值会被采用。当然，别忘了在%DOMAIN_HOME%\config\apusic.conf中配置DiscoveryService服务，片断如下：

```
<SERVICE CLASS="com.apusic.net.DiscoveryService">
    <ATTRIBUTE NAME="GroupAddress" VALUE="230.0.0.2" />
    <ATTRIBUTE NAME="GroupPort" VALUE="1500" />
</SERVICE>
```

6.18.3.3 EJB集群

Apusic 应用服务器支持EJB3.0，同时兼容EJB2.x、EJB1.x。EJB集群是解决大规模、具有厚重而复杂的商业逻辑、高性能、高可用、高水平可扩展的JavaEE系统的良好方案。Apusic为SessionBean、EntityBean集群提供简洁有效、独具特色的支持。

Apusic EJB集群主要面向两个问题：负载均衡和高可用性。值得注意的是，EJB集群一般都是针对EJB远程调用而言，而非本地调用（包括集群节点内部使用远程接口，因为对于这种情况，Apusic会自动优化成本地调用）。以下叙述，对于Home接口、Bean接口，如果如果没特殊说明，均指远程的。

6.18.3.3.1 EJB负载均衡和EJB高可用性

首先，以EJB2.x编程模型为例，我们看看SessionBean、EntityBean的典型应用模式。

一个典型的SessionBean应用代码片断：

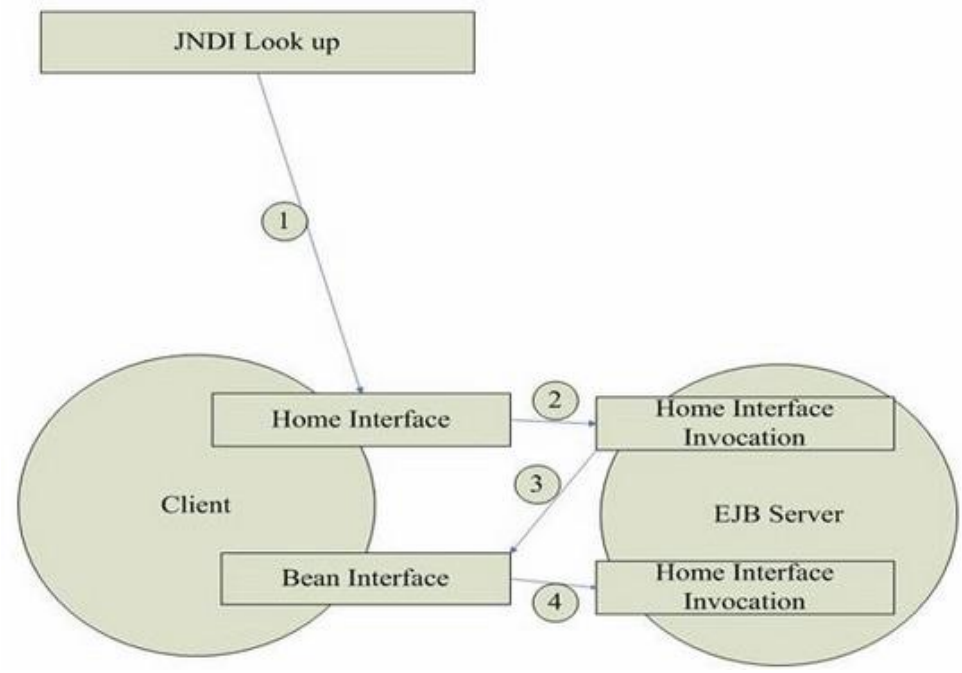
```
...
Context ctx = ...;
CounterHome counterH = (CounterHome)
PortableRemoteObject.narrow(ctx.lookup("ejb/Counter"),CounterHome.class);
Object o = counterH.create(/*may be ,some initial states for stateful session
bean.*/);
```

```
Counter counter = PortableRemoteObject.narrow(o,Counter.class);
counter.add(1);
counter.minus(100);
...
```

一个典型的EntityBean应用代码片断:

```
...
Context ctx = ...;
AccountHome accountH = (AccountHome)
PortableRemoteObject.narrow(ctx.lookup("ejb/Account"),AccountHome.class);
Object o = accountH. findByPrimaryKey(new AccountPK("ISN"));
Account account = PortableRemoteObject.narrow(o,Accout.class);
account.deposit(1);
account.withdraw(100);
...
```

从上面代码片断我们可以看出，典型的EJB应用，首先获取Home接口，然后通过Home接口获取Bean接口，最后通过Bean接口调用业务方法完成商业计算（如下图）：



那么，EJB负载均衡、EJB高可用性是怎么回事?

沿袭负载均衡一贯的语义，EJB负载均衡指EJB调用通过特定算法分配到多台应用服务器的行为。这里的EJB调用指对EJB Bean接口的调用。然而，Apusic EJB集群还支持Home接口的负载均衡。

目前，Apusic EJB集群支持的负载均衡算法有:

- RANDOM 随机选择集群有效节点中的一个.
- ROUND_ROBIN 循环地、依次地选择集群有效节点中的一个.
- WEIGHTED 根据集群节点预先设定的负载权重进行选择。权重通过服务ClusterService的LoadWeight属性设置，范围是[1,100]。一般来说，WEIGHTED用于非对称集群，譬如，集群中有两台服务器S1、S2，S1计算能力强于S2，那么可以根据计算能力的比例为S1、S2设置恰当的权重值，从而，使得计算能力强的服务器接收、处理更多的请求。
- STICKY 在获取Bean接口、通过Bean接口调用业务方法的时候，随机选择一个节点，此后，获取Bean接口、调用Bean接口的业务方法的调用全部发送到这个节点，除非这个节点失效。

EJB高可用性指在某集群节点失效的时候，对它的EJB Home接口的调用或者Bean接口的调用会透明地转移到其它有效节点。

6.18.3.3.2 EJB集群配置

事实上，EJB集群的配置是相当简单的，你只需要在apusic-application.xml的Bean配置中添加如下格式的片断：

```
<cluster-config>
    <clustered>true</clustered>
    <home-load-balance-policy>ROUND_ROBIN</home-load-balance-policy>
    <object-load-balance-policy>ROUND_ROBIN</object-load-balance-policy>
</cluster-config>
```

其中<home-load-balance-policy>配置Home接口的负载均衡算法。<object-load-balance-policy>配置Bean接口的负载均衡算法。当然，以上两个配置是可选的，你可以简单添加片断：

```
<cluster-config>
    <clustered>true</clustered>
</cluster-config>
```

此时，Apusic会根据Bean的类型决定Home接口和Bean接口的负载均衡算法。决策为所有类型的Bean，Home接口采用ROUND_ROBIN算法；SLSB和只读EntityBean的Bean接口采用ROUND_ROBIN算法；SFSB和非只读EntityBean的Bean接口采用STICKY算法。以下是集群SLSB的apusic-application.xml的片断:

```
<apusic-application>
    <module uri="">
        <ejb>
            <session ejb-name="GreetingBean">
                <jndi-name>ejb/GreetingHome</jndi-name>
                <local-jndi-name>ejb/GreetingLocalHome</local-jndi-name>
                <security-mechanisms />
                <cluster-config>
                    <clustered>true</clustered>
```

```
                    <home-load-balance-policy>ROUND_ROBIN</home-load-balance-
policy>
                    <object-load-balance-policy>ROUND_ROBIN</object-load-
balance-policy>
                </cluster-config>
            </session>
        </ejb>
    </module>
</apusic-application>
```

同时，别忘了，在%DOMAIN_HOME%\config\apusic.conf中配置ClusterService服务。

6.18.3.3.3 EJB集群实践

- 关于Home接口和Bean接口负载均衡算法的选择

  理论上，任何类型Bean的Home接口和Bean接口都可以采用RANDOM、ROUND_ROBIN、WEIGHTED、STICKY算法。不过，显然，集群节点计算能力不对称、各种Bean的特性、各种Bean集群的特定实现方式等等因素都会影响甚至决定均衡算法的选择。

  - 如果集群节点计算能力不对称，优先考虑WEIGHTED算法。
  - 由于Apusic SFSB、非只读EntityBean集群的特定实现方式，所以它们的Bean接口最好采用STICKY算法。

- 关于集群节点内部的EJB调用

上面，我们提到，一般来说，EJB集群针对远程调用而言。同时，存在这样的事实：对于节点内部的EJB远程调用，Apusic会自动将之优化成本地调用。所以，包括Web-EJB、EJB-EJB等方式在内的集群节点内部的EJB调用（无论使用远程接口还是本地接口），最终都变成本地调用，同时负载均衡和高可用性特性消失。

- 关于EntityBean一定程度的状态同步

Apusic EJB 容器支持EJB2.1规范提及的A、B、C三种commit-time option。很显然，基本上，EntityBean集群场景是不符合option A的假定，更重要的是，Apusic EJB集群依赖数据库同步，所以，一般来说，concurrency-strategy设定为Exclusive并且force-refresh设定为False，这样的方案绝对不应采用，而Concurrency-strategy采用Parallel策略是推荐方案。

- 关于安全角色的传播

要使节点间EJB调用的安全角色能够传播，必须使用一致的认证密钥，即%DOMAIN_HOME%\config\authkey.dat必须一致。可以先启动一个Apusic服务器，生成config/authkey.dat文件，然后将此文件复制到其他服务器。

6.18.3.4 JMS集群

JMS是JavaEE相当重要的一部分，它为开发异步、可靠、高性能、灵活的系统提供支持。目前广泛应用于EAI、EDI等领域。Apusic JMS集群主要包括消息路由和集群队列功能。

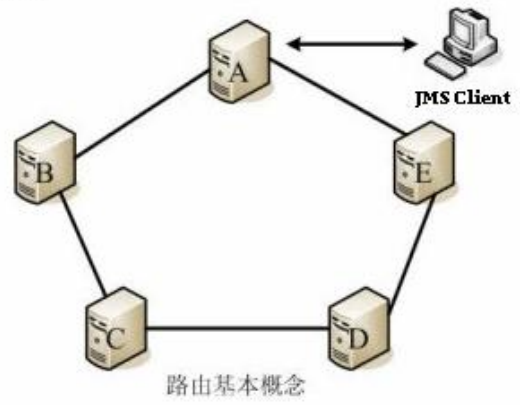
6.18.3.4.1 消息路由

消息路由一般用来解决发送消息的客户端和消息目的地不能直接连通的情况下，依赖特定算法，选择JMS网络中的一个或多个中间消息路由器（Router）组成的能够到达消息目的地的最优通路并负责转发消息。

在一个由多台提供消息服务的 Apusic 应用服务器组成的网络上，每个提供消息服务的 Apusic 应用服务器可被视为一个网络中的节点，消息路由意味着节点之间的路径。只要用户连接到网络中任何一个节点，即可向网络中的其他任意节点发送消息，由消息服务根据消息路由提供消息在网络中的最佳传输机制。

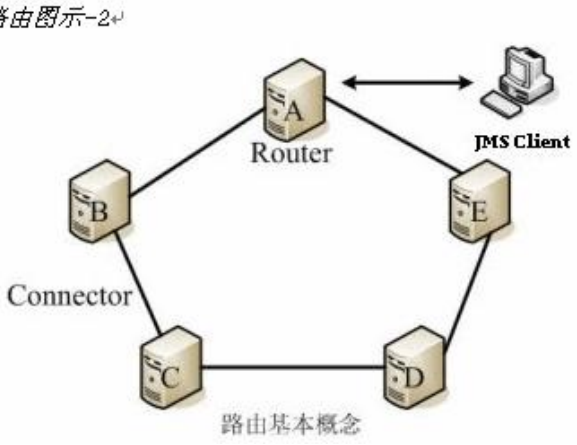
如下图所示：

消息路由图示-1



A、B、C、D、E五个节点上都是提供消息服务的Apusic应用服务器，五个节点连通成一个网络，连接到A节点的客户端可以发消息到五个节点中的任意一个节点，Apusic应用服务器会自动寻找一条最佳路径传递消息到目标节点。如从A发消息到D，有两条连通路径：A->B->C->D和A->E->D，其中A->E->D经过的节点最少，该路径为最佳路径，Apusic应用服务器将根据此路径对消息进行传递。

消息路由图示-2



在一个Apusic应用服务器JMS网络中，如上图所示，每个节点都是一个消息路由器(Router)，每个消息路由器有一个路由器名，以区别于其他的路由器。路由器名可以和主机名相同，也可以不同。

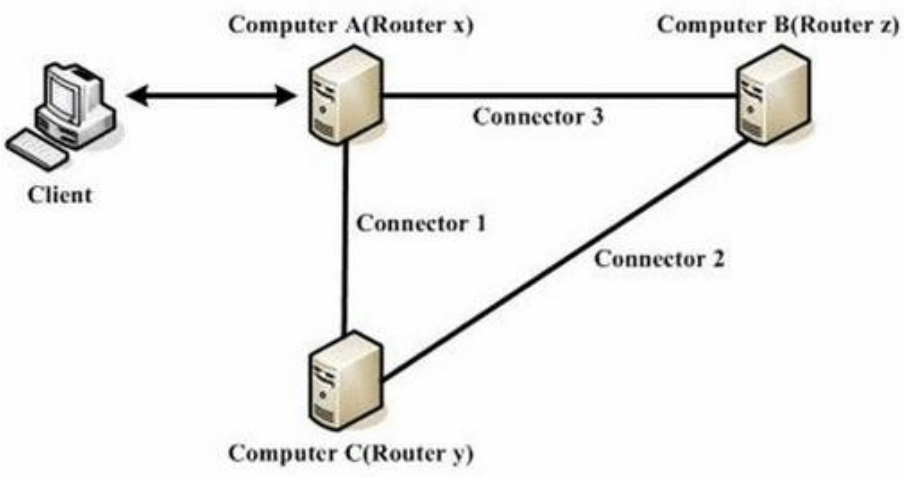
每两个节点之间的直接路径被称为路由连接器(RoutingConnector)，网络中每两个节点间的直接路径必须事先进行申明性的定义。每个路由连接器的申明都是双向的，如在A节点上申明到B的路由连接器，则说明从A节点到B节点之间是双向连通的，从A可以到达B，从B也同样能够到达A。

假设上图中的五个节点A、B、C、D、E是某个实际网络中五个提供消息服务的Apusic应用服务器，实际网络中的主机名分别是computerA、computerB、computerC、computerD、computerE，在由这五个节点组成的Apusic应用服务器网络中，服务器配置的路由器名分别是routerA、routerB、routerC、routerD、routerE，要使这五个节点结形成类似于上图的Apusic应用服务器网络，即每个节点都有一条与下一个节点双向连通的路径，最后一个节点有一条与最开始的节点双向连通的路径，形成一个闭合的环，消息客户连接到环中的任意节点，都可向其他节点发送消息。则可以选择在每个节点上都申明一个到下一个节点的连接器，由于连接器是双向连通的，所以在下一个节点不需要申明一个到上一个节点的连接器。同时在每个节点上的远程路由需要申明跟它直接相连的其它节点的路由名。如：A节点上的远程路由需要申明B、E两个节点，表示由A节点可以直接连接到B、E两个节点上，并且不需要申明其它节点，这样A就能够向所有的节点发送消息了。同时需要申明由A到B的连接器，表示A节点跟B节点有一条双向连通的路径；节点B的远程路由则需要申明A、C两个节点，同时需要申明由B到C的连接器，如此类推。

一. 理解Apusic消息服务

JMS是JavaEE规范中提出的消息中间件服务(Java Message Service™)规范,提供应用程序间异步或同步的消息传递和管理服务，Apusic应用服务器中包含了高效、可靠的消息服务。Apusic应用服务器中的消息服务接口完全遵循JMS API规范。提高了企业应用中各组件的可移植性、松耦合特性，同时更加高效可靠；为分布式企业应用异步交换关键业务数据和事件提供了可靠而灵活的服务。

Apusic消息服务提供对消息队列(Queue)和消息主题(Topic)的管理，消息的发送由消息服务负责完成，原理如下：



ComputerA,B,C为提供消息服务的Apusic应用服务器，每一个Apusic应用服务器称为一个Router，他们之间的连接被称为Connector，客户端进行消息发送时直接与Computer A 连接，客户端要把消息传送到Computer B 的队列时，只需指明接收消息的主机名和队列名，然后发送消息即可，Computer A 中的消息服务将会查找配置文件中的设置，如从A 到B间有无通路连通，假如它们之间有通路连接，消息服务就会寻找这些通路中最短的路径发送消息到指定服务器(Computer B) 的指定队列，假如A 到B 间无有效连接，消息将被保存在某一连通的服务器上，待有有效通路之后再发送消息。

Apusic应用服务器提供了两个主要的消息服务方面的特性：

- 全局事务

Apusic应用服务器提供了一个作为事务性资源管理器的JMS提供者(Provider)，允许从JSP，Servlet，EJB等应用组件中对消息服务进行事务性的访问。对消息的发送和接收可以和对其他资源的操作一起参与到一个JTA事务中。

- Message-drivenBeanJ2EE™1.3规范中提供的EnterpriseBean类型，Message-drivenBean,使多层分布式企业应用可以异步地使用消息。

二. 配置消息路由与存储

多个提供消息服务的应用服务器可以组成一个虚拟的JMS网络，每个提供消息服务的应用服务器可被视为一个JMS网络中的节点，而节点之间的连接则通过声明消息路由提供。只要用户连接到JMS网络中任何一个节点，即可向网络中的其他任意节点发送消息。

为提供消息服务的可靠性，和JMS网络中消息的暂时存储，Apusic应用服务器使用了一个可靠的消息存储机制。

- 配置消息存储目录

Apusic消息路由与存储配置的部分存在于域主目录下config子目录中，apusic.conf文件中的相关配置段。缺省的配置段如下：

```
...
<SERVICE CLASS="com.apusic.jms.server.FileMessageStoreProvider">
    <ATTRIBUTE NAME="StoreDirectory" VALUE="store/jms" />
</SERVICE>

...
```

此配置段指定了消息服务的存储目录，其中，名为StoreDirectory属性的值指定了消息存储目录的路径，可使用绝对路径如：c:\message_store\或/usr/message_store来指定，亦可使用相对于域主目录的相对路径如：store /my_message_store来指定，如指定目录不存在，应用服务器将创建指定的目录。

- 配置消息路由

  对消息服务中路由的配置是通过Apusic应用服务器域主目录中子目录config下的apusic.conf 配置文件进行的。通过apusic.conf中的相关配置段可以对消息路由器和路由连接器进行管理和配置。

  - 路由器的配置

  消息路由器的配置通过在apusic.conf文件中配置消息服务的服务配置段进行，其缺省配置段如下：

  ```
  <SERVICE CLASS="com.apusic.jms.server.JMSServer">
  </SERVICE>
  ```

  通过对申明此服务的XML元素增加名为RemoteRouters的ATTRIBUTE子元素，在其值中列出相应的路由器名即可。以消息路由图示-2中的A节点为例，A节点上需要申明B、E两个节点，表示由A节点可以直接连接到B、E两个节点上，并且可以作为消息路由路径中的一个中转节点，则可以进行如下配置：

  ```
  <SERVICE CLASS="com.apusic.jms.server.JMSServer">
      <ATTRIBUTE NAME="RemoteRouters" VALUE="computerB,computerE" />
  </SERVICE>
  ```

  其它B、C、D、E节点上的相关配置段如此类推进行配置。路由器名字间使用逗号分隔，路由器名字可以是其他主机的主机名。

一般，路由器的名字默认是服务器的主机名，但是，也可以通过使用对申明此服务的XML元素增加名为RouterName的ATTRIBUTE子元素，在其值中指定自身的路由器名。如上例中采用了主机名对路由连接器进行申明，如在五个节点上都申明了路由器名字，如A、B、C、D、E主机分别对应routerA、routerB、routerC、routerD、routerE，则上例的配置如下：

```
<SERVICE CLASS="com.apusic.jms.server.JMSServer">
    <ATTRIBUTE NAME="RouterName" VALUE="routerA" />
    <ATTRIBUTE NAME="RemoteRouters" VALUE="routerB,routerE" />
</SERVICE>
```

其它路由器的配置依次类推。

- 路由连接器的配置

每个JMS网络中的消息路由器都是通过路由连接器进行连接的，每个路由器可以拥有多个路由连接器连接到其它节点。对路由连接器的配置是通过在apusic.conf配置文件中增加路由连接器服务实现的。以消息路由图示-2中的A节点为例，需要申明到节点B的路由连接器，则可在apusic.conf文件中加入如下配置段：

```
<SERVICE CLASS="com.apusic.jms.routing.RoutingConnector"
NAME="Connector:Name=toB">
    <ATTRIBUTE NAME="RemoteHost" VALUE="computerB" />
    <ATTRIBUTE NAME="RemotePort" VALUE="6888" />
</SERVICE>
```

其中computerB是B的主机名，toB是此路由连接器区别于其他路由连接器的自由定义名字，6888是默认的JMS服务端口，具体使用中，可将其更改为实际的路由连接器名、主机名与端口。

三. **管理消息服务**

消息服务的管理是通过配置消息管理对象的属性完成的。消息管理对象包含了管理员生成的消息配置信息，之后由消息客户端使用。

消息服务中定义了两个管理对象：

- 连接工厂(ConnectionFactory)。消息客户通过此对象创建与消息服务的连接.
- 消息目的地(Destination)。消息客户用来指定发送消息的目的地或接收消息的来源地。

由于消息服务包含两种消息模型，即Point-to-Point(PTP)和Publish-and-Subscribe(Pub/Sub)模型，Apusic应用服务器对这两种消息模型提供了完整的支持，因此，对于连接工厂而言，提供了两种可配置的连接工厂类型，面向PTP消息模型的QueueConnectionFactory和面向Pub/Sub模型的TopicConnectionFactory；对于消息目的地而言，同样提供了两种可配置的消息接收类型站，面向PTP的队列（Queue）站和面向Pub/Sub的主题（Topic）站。

通过对存在于Apusic应用服务器域主目录下config子目录中，名为jms.xml的配置文件进行编辑，实现对Apusic应用服务器消息服务的管理。在此文件中，可以对连接工厂、消息目的地和消息服务安全策略进行配置。

jms.xml文件是一个xml文件，其文档类型定义(DTD)为jms-config_1_2.dtd。

1. 配置连接工厂

在jms.xml文件中，每一个连接工厂配置信息对应一个connection-factory标记申明的xml元素，每个connection-factory元素可包含使用以下三种标记所申明的子元素：

- description，可选标记，对此连接工厂的描述 .
- display-name，必须申明的标记，用于区别于其它连接工厂.
- jndi-name，可选标记，通过JNDI，用于客户在服务器命名空间中查找此连接工厂.

实际应用中，当管理员为连接工厂分配JNDI名之后，消息客户即可使用JNDI在服务器的命名空间中对连接工厂进行查找并获得引用，之后通过连接工厂取得与消息服务的连接。

连接工厂的配置属性如下表：

属性	描述	值类型	缺省值
pooled	指定此连接工厂是否对其管理的连接使用连接池	“True”或“False”	“False”
secure	指定连接工厂所提供连接的通讯方式	“True”或“False”	“False”
anonymous	是否授权匿名用户访问此连接工厂	“True”或“False”	“True”
client-id	由于标识连接客户状态的标识符，通常被用于Pub/Sub模型中的持久订阅（Durable subscription)	字符串，此属性是可选的	无
default-delivery-mode	使用由此连接工厂生成的连接发送消息时，缺省的发送方式	“persistent”或“non-persistent”	“non-persistent”
default-priority	使用由此连接工厂生成的连接发送消息时，缺省的优先级	数字（0~9）	4
default-time-to-live	使用由此连接工厂生成的连接发送消息时，对于已发送的消息，消息系统保留此消息的缺省时间长度，单位为毫秒。	整型	0
min-pool-size	此连接工厂对应的连接池中，所保持的最少连接数	整型	5
max-pool-size	此连接工厂对应的连接池中，所保持的最大连接数	整型	30
idle-timeout	连接等待超时时间。当连接池中的某个连接等待被使用的实际时间超过此属性数值时，连接池自动关闭此连接	整型（单位秒）	300

2. 配置消息目的地

对应于PTP和Pub/Sub消息模型，Apusic应用服务器中的消息服务提供了两种消息目的地，队列 （Queue）和主题（Topic）。实际应用中，管理员为消息目的地分配JNDI名，消息客户即可使用JNDI在服务器的命名空间中对消息目的地进行查找并获得引用，在通过连接工厂取得与消息服务的连接之后，消息客户即可向消息目的地同步或异步地发送或接收消息。

在jms.xml文件中，每一个消息目的地配置信息对应一个queue标记或topic标记申明的xml元素。

- 配置消息队列（Queue）

在jms.xml文件中，每一个队列（Queue）配置信息对应一个queue标记申明的xml元素，每个申明的queue元素可包含三种标记所申明的子元素：

- description，可选标记，对此队列的描述；
- display-name，必须申明的标记，用于区别于其它队列；
- jndi-name，可选标记，通过JNDI，用于客户在服务器命名空间中查找此队列。

队列的配置属性如下表：

属性	描述	值类型	缺省值
cache-size	队列缓冲中保留的消息个数	整型	20
expiry-check-interval	系统检测消息队列中消息是否过期的时间间隔，单位是秒	整型	60

- 配置消息主题（Topic）

  在jms.xml文件中，每一个主题（Topic）配置信息对应一个topic标记申明的xml元素，每个申明的topic元素可包含三种标记所申明的子元素：

  - description，可选标记，对此主题的描述
  - display-name，必须申明的标记，用于区别于其它主题
  - jndi-name，可选标记，通过JNDI，用于客户在服务器命名空间中查找此主题

主题的配置属性如下表：

属性	描述	值类型	缺省值
cache-size	队列缓冲中保留的消息个数	整型	20
expiry-check-interval	系统检测消息队列中消息是否过期的时间间隔，单位是秒	整型	60

- 安全相关配置

  JavaEE™体系中的JMS规范实际上并未包含有关安全方面的内容，因此，系统提供保证消息服务的安全和完整性的机制就极为重要。

  Apusic应用服务器提供了对消息服务方面的安全管理，主要根据JavaEE™体系中的安全角色(SecurityRole)和消息客户的操作对消息服务进行保护。

  消息服务中基于安全角色的授权方式是指，系统管理员可定义一组安全角色，每个被定义的安全角色对应于系统中的一组用户或组，然后，根据消息客户用户可对消息目的地(Destination)进行的操作（如对于队列，客户可执行发送、接收、浏览等操作）对前面定义的安全角色进行授权。

  - 安全角色

在jms.xml文件中，每一个安全角色（Security Role）配置信息对应一个security-role标记申明的xml元素，每个申明的security-role元素可包含四种标记所申明的子元素：

- description，可选标记，对此安全角色的描述 。
- role-name，必须申明的标记，用于区别于其它安全角色申明。
- principal或group，必须使用至少一个这两种标记申明的元素，可以是多个，用于将逻辑的安全角色对应于系统中的用户或组。

- 消息目的地(Destination)访问许可

对于一个或者一组jms.xml中申明的消息目的地(Destination)和一个或者一组jms.xml中申明的安全角色，通过使用一个destination-permission标记申明的xml元素来设置这二者之间的对应关系，每个申明的destination-permission元素可包含三种标记所申明的子元素：

- description，可选标记，对此访问许可的描述 。
- role-name，必须使用至少一个这种标记申明的元素，可以是多个，用于表示此访问许可所包含的在security-role中申明的安全角色。
- destination-method，必须使用至少一个这种标记申明的元素，可以是多个，用于表示此访问许可所包含的客户对消息目的地(Destination)能进行的操作。

每个申明的destination-method元素可包含四种标记所申明的子元素：

- description，可选标记，对此destination-method的描述 。
- queue-name或topic-name，必须有至少一个这两种标记申明的元素，而且它的值必须是对应的jms.xml中已申明的queue或topic元素，用于说明此访问许可包含的消息目的地(Destination)。
- method-name，必须使用至少一个这种标记申明的元素，可以是多个，通过方法名，指定对指定队列（queue)或主题（topic)所能进行的客户操作。对于指定的队列（queue），每个method-name元素的值可以是send、receive或browse中之一；对于指定的主题（topic），每个method-name元素的值可以是publish、subscribe、subscribe-durable或unsubscribe中之一。

- 范例

下面范例授权安全角色foo可以对队列bar 进行接收和浏览：

- 申明队列

```
...
<queue>
    <queue-name>testQueue</queue-name>
</queue>
```

```
...
```

- 申明安全角色foo，并对应到管理员用户

```
...
<security-role>
    <role-name>foo</role-name>
    <principal>admin</principal>
</security-role>

...
```

- 设置访问许可

```
...
<destination-permission>
    <role-name>foo</role-name>
    <destination-method>
        <queue-name>testQueue</queue-name>
        <method-name>receive</method-name>
        <method-name>browse</method-name>
    </destination-method>
</destination-permission>

...
```

6.18.3.5 集群队列

基于JMS的系统同样存在负载均衡和可用性的问题。集群队列基于消息路由技术，它能够很好的解决这两个问题。

集群队列是指JMS网络中的任何一个节点上定义的消息队列都将被其他节点所共享，通过任何节点向一个集群队列发送消息都是等效的，对客户来说无法察觉是否正在使用集群，集群中网络拓扑结构发生变化对客户端也没有任何影响。

集群队列具有高可用性：JMS网络中单个节点失效或部分网络无法连通时并不影响集群队列的使用.

集群队列具有负载均衡特性：在发送消息时，根据各节点的负荷情况，负载会被合理分配，从而使节点处理能力和网络带宽被充分利用。

6.18.3.5.1 配置集群队列

要使用集群队列，你只需要在jms.xml中，为某个队列添加clustered属性，示例片断：

```
<queue clustered="true">
    <queue-name>clusteredQueue</queue-name>
</queue>
```

6.18.3.5.2 使用集群队列

典型的JMS应用中，无论发送、接收消息，都需要通过某种方式获取目的地（Destination，为Queue或者Topic），一般，我们通过JNDI来获取，示例代码片断如下：

```
Context ctx = ...;

Session ssn = ...;

Queue queue = (Queue)ctx.lookup(“TestQueue”);

...
```

那么我们又怎样获取集群队列呢？其实，很简单，我们需要通过JMS Session接口的方法Queue createQueue(String queueName)来获取，以上面注册的clusteredQueue为例，代码片断如下：

```
Session ssn = ...; //get session instance
Queue queue = ssn.createQueue(“clusteredQueue”);
...
```

6.19 管理sip服务

6.19.1 SIP概述

SIP（Session Initiation Protocol，会话初始协议）是由IETF制定的多媒体通信协议。它是一个基于文本的应用层控制协议，定义了如何在通信设备之间相互连接和信息交换，用于创建、修改和释放一个或多个参与者的会话。SIP广泛应用于电路交换、下一代网络以及IP多媒体子系统的网络中，可以支持并应用于语音、视频、数据等多媒体业务，同时也可以应用于呈现、即时消息等特色业务。

SIP特点包括

- 稳定性：该协议已经使用了多年，现在十分稳定。
- 标准化：随着整个通信行业都在向SIP靠拢，SIP已经讯速成为一种标准
- 安全性：它提供像加密(SSL、S/MIME)和身份验证等一组安全功能，还提供其他安全性功能实现对SIP进行扩展。
- 兼容性：参照HTTP协议定义的，适用基于IP的网络；采用了URI、DNS和MIME并与其它IP应用兼容。

- 灵活性：具有良好的可扩展特性，可以方便地增加定义，嵌入各种用户终端并迅速实现新功能；有较强的互操作能力，良好的开放性。

SIP主要功能包括

- 用户定位：确定被叫用户通信所使用的终端系统的位置
- 用户能力协商：确定所用媒体类型和媒体参数
- 用户可用性判断：确定被叫用户是否空闲以及是否愿意加入会话
- 呼叫管理：包括传输和终止会话、修改呼叫参数和调用服务

6.20 SIP的体系结构

SIP体系结构包括以下四个主要实体：

- 用户代理（User Agent）

SIP 用户代理是一个SIP逻辑网络端点，用于创建、发送、接收SIP消息并管理一个SIP会话。SIP用户代理可分为用户代理客户端UAC（User Agent Client）和用户代理服务端UAS（User Agent Server）。UAC创建并发送SIP请求，UAS接收处理SIP请求，发送SIP响应。UAC和UAS是逻辑上的两个部分，每个终端系统都包含了UAC和UAS的功能。

- 代理服务器（Proxy Server）

SIP代理服务器在网络上位于SIP UAC和UAS之间，用于帮助UAC和UAS间的消息路由。代理服务器也可以执行路由策略控制（比如，检查SIP消息的合法性，确认消息是否允许被路由）。代理服务器具有解析能力，负责接收用户代理发来的请求，根据网络策略将请求发给相应的服务器，并根据应答对用户做出响应，也可以将收到的消息改写后再发出。

- 注册服务器（Register Server）

注册服务器是一个接收注册的SIP服务器，用以进行管理以及特定的服务。通过注册过程接收客户当前的位置信息，并对定位服务器进行添加、修改、查询等操作。通常与代理服务器或重定向服务器放在一起。

- 重定向服务器（Redirect Server）

负责规划SIP呼叫路由。它将获得的呼叫的下一跳地址信息告诉呼叫方，呼叫方由此地址直接向下一跳发出申请，而重定向服务器则退出这个呼叫控制过程。重定向服务器只提供地址解析服务，类似于DNS。

- 定位服务器（Location Server）

定位服务器不是SIP实体，但是它是任何实用SIP协议的体系结构中非常重要的一部分。位置服务器存储并且向用户返回可能的位置信息。它可以利用从注册服务器或者其他数据库得来的信息。大部分的注册服务器接收到位置信息时即刻将这些信息上载到定位服务器。

6.20.1 SIP协议结构

SIP 消息是客户机和服务器之间通信的基本信息单元，它是一个基于UTF-8的文本编码协议。SIP消息主要分为请求消息和响应消息，不管请求消息还是响应消息，它都是由一个起始行、若干个头字段和一个消息体组成，其中的消息体是可选，其格式遵循RFC2822因特网文本消息格式标准。

SIP协议结构与HTTP协议结构非常相似，其消息格式如下：

起始行

消息头部（若干个头字段）

空行

消息体（SDP）

其中的起始行对于请求消息是请求行，对于响应消息是状态行。

下图是一个sip客户端向服务端注册时的请求消息和响应消息：



- 请求行

请求消息的起始行为请求行。请求行的格式由方法名、请求URI和协议版本组成，各部分之间用一个空格字符进行分隔。请求行必须用回车换行字符表示行终结。请求行的格式如下：

格式：Request-Line = Method Request-URI SIP-Version

Method：方法表示请求的类型，核心的类型有6种，INVITE，ACK，BYE，CANCEL，REGISTER和OPTIONS

Request-URI：请求的URI表示此请求将要被发送的目标地址

SIP-Version：协议版本，一般为SIP/2.0

例子中的请求方法是REGISTER，Request-URI为sip:192.168.101.47:5080，SIP-Version为SIP/2.0。常见的几种请求方法的说明如下表：

请求消息	消息含义
INVITE	发起会话请求，邀请用户加入一个会话。
ACK	证实已收到对于 INVITE 请求的最终响应。该消息仅和 INVITE 消息配套使用。
BYE	释放已建立的呼叫。
CANCEL	释放尚未建立的呼叫，对于已完成的请求（即已收到最终响应的请求）则没有影响。
REGISTER	向 SIP 网络服务器登记用户位置信息。
OPTIONS	查询服务器的能力信息。

- 状态行

响应消息的起始行为状态行，状态行由协议版本、状态码和状态原因短语组成，各个部分之间用一个空格字符进行分隔。状态行的格式如下所示：

格式： Status-Line = SIP-Version Status-Code Reason-Phrase

SIP-Version：协议版本，一般为SIP/2.0

Status-Code：状态码，该参数为一个3为的十进制整数，来报告事务的状态

Reason-Phrase：状态的原因短语

例子中协议版本为SIP/2.0，状态码为200，原因短语是OK

关于返回状态码：

1XX响应消息：呼叫进展响应，表示已经接收到请求消息，正在对其进行处理；

2XX响应消息：成功响应，表示请求已经被成功接受、处理；

3XX响应消息：重定向响应，表示需要采取进一步动作，以完成该请求；

4XX响应消息：客户端出错，表示请求消息中包含语法错误或者服务器端不能完成对该请求消息的处理，如488消息表示“此处不接受”；

5XX响应消息：服务器端出错，表示SIP服务器故障，不能完成对正确消息的处理；

6XX响应消息：全局故障，表示请求不能在任何SIP服务器上实现。

- 消息头部

头字段提供了关于请求（或应答）的消息和关于这些消息所包含的消息体的信息。一些头字段可以在请求或应答两种消息中使用，而其他的头字段只能单独特定地用于请求（或应答）。头字段由头字段名、后边跟着一个冒号、再后面跟着头字段值组成。头字段格式如下所示：

field-name：field-value

下面介绍常见的字段：

Via：Via 头字段指定目前请求消息经过的路径，同时指定响应也要按该路径返回。该字段值中的branch ID参数是一个事务标识符，代理服务器用它来检测环路。Via头字段包含一个用来发送消息的传送协议和客户端的主机名或者网络地址，该头字段还可能包含一个接收响应的端口号。Via头字段的缩写形式为v。

Max-forwards：在RFC3261中规定，Max-Forwards（最大转发次数）头字段必须和任何方法一起规定向下游转发消息的代理服务器和网关的个数。当某客户端沿着某条链路发送请求消息的时候，使用该字段可以有效地防止链路中出错或者发生回环。Max-forwards头字段的值是一个0-255的整数，指示了该请求还允许被转发的次数。转发该请求时，每经过一个服务器该值就减一。一般默认为70。

To：该头字段指定了请求的逻辑接收者。"display-name"参数用于人机接口，为可选。"tag"参数一般用于标识对话。它的缩写形式为t。

From：From头字段用于指示请求的发起者，也就是发送请求（不是对话）的源地址。这可能与对话的发起者并不同。当被叫发出请求时，From字段中就是使用被叫的地址。From的缩写形式是f。

Call-ID：Call-ID头字段唯一地标识某个客户端的某个特定的会话或所有的注册请求。一个多媒体会议可以发起几个Call-ID不同的呼叫。Call-ID区分大小写并逐字节比较，缩写形式为i。

Cseq：命令序列头字段Cseq位于请求消息中，包含两个字段：一个无符号整数字段和一个方法名。该头字段用于把某对话中的事务进行排序且提供了一种唯一标识某事务的方法（即INVITE、ACK等method），并能够区分某请求是新的请求还是重发的请求。如果两个Cseq的数字序列以及方法都相等那么这两个Cseq就是等价的。

Contact：Contact字段的值含有一个URI，UA可根据这个地址，直接找到另一个UA，从而避开SIP服务器，当Contact头字段包含一个显示名称的时候，带有所有的URI参数的URI应放于三角括号<>中，Contact头字段的缩写是m。

Record-Route：Record-Route头字段由代理服务器插入请求消息中，这样可以使该对话中将来的请求仍能经过该代理服务器。

Route：Route头字段有一个代理服务器列表，用来指定请求消息的路由

Content-length：消息体字段长度

Server：服务器容器信息。

6.20.1.1 消息体与SDP

请求和应答都可能含有消息体，它被一个空行和消息头分开。被SIP消息携带的消息体通常是会话描述符（即SDP），它就象附件一样包含在SIP消息中。

会话描述协议（SDP，Session Description Protocol）规定了对描述会话的必要信息怎样进行编码。两个SIP实体可以通过携带SDP消息体来使它们之间的多媒体会话达成一致。

SDP用于构建INVITE和200 OK响应消息的消息体，供主/被叫用户交换媒体信息。

SDP包括以下一些方面：

- 会话的名称和目的
- 会话存活时间
- 包含在会话中的媒体信息，包括：

媒体类型(video, audio, etc)

传输协议(RTP/UDP/IP, H.320, etc)

媒体格式(H.261 video, MPEG video, etc)

多播或远端（单播）地址和端口

- 为接收媒体而需的信息(addresses, ports, formats and so on)
- 使用的带宽信息
- 可信赖的接洽信息（Contact information）

SDP会话描述是基于文字的，一个会话描述由一些类似如下形式的文字行组成：Type=value

类型域为一个单独字符，而值域的格式则取决于它前面的类型语。一个SDP描述含有会话级信息和媒体级信息，会话级信息应用于整个会话，媒体级信息作用于特殊的媒体流。一个SDP会话描述以会话级信息和媒体级信息开始，如果任意一个出现，另外一个就接着在后面出现。会话级部分以v = 0开始，v代表类型，0为值，意思是协议版本号为0(SDP版本0)。接下来的行直到媒体流部分或者会话描述的终点，提供了整个会话的信息。

6.20.1.2 SIP与Java

Java 在支持SIP上发布了许多相关的规范，如Java APIs for Integrated Networks、SIP Servlet API 、JAIN SIP Lite 、SIP API for J2ME 、180)，JAIN SIMPLE Presence 、Java Media Framework for RTP、JAIN SIMPLE Instant Messaging、JAIN SDP、SIP for J2ME等，而SIP Servlets规范提供了一系列的Java API和一个基于容器(Container)/应用服务器(Application Server)的开发模型，用于提高服务器端SIP应用的开发效率。

SIP Servlets同样基于Java Servlet架构，其API归属于javax.servlet.sip包和javax.servlet.http同样扩展自javax.servlet.。不同的是，HTTP Servlets通过Servlet架构实现了HTTP协议，而SIP Servlet实现了SIP协议。一组SIP Servlets连同资源和部署描述文件打包后部署并运行在一个容器或SIP应用服务器上，容器提供了例如会话状态管理、事务管理、重发、网络连接、消息调度、线程管理、资源管理以及应用程序管理等服务。

下表是Http servlet和Sip servlet的比较：

比较项	HTTP	SIP
Servlet Class	HttpServlet	SipServlet
Session	HttpSession	SipSession
Application package	war	sar
Deployment Descriptor	web.xml	sip.xml

6.20.2 配置Apusic SIP服务

Apusic应用服务器对企业规范进行扩展，实现SIP协议和融合应用的快速开发和部署能力，完全兼容SIP Servlets1.1规范（JSR289）。

要在Apusic应用服务器中使用SIP，需要在apusic.conf中配置相应的服务，配置在应用部署服务（J2EEDeployer）之前，避免应用启动时找不到需要的服务。

首先配置SipService，如下所示：

```
...
<SERVICE CLASS="com.apusic.web.sip.SipService">
    <ATTRIBUTE NAME="DarConfigurationFileLocation" VALUE="config/sip/sip-dar.properties" />
    <ATTRIBUTE NAME="SipStackPropertiesFileLocation" VALUE="config/sip/sip-stack.properties" />
</SERVICE>

...
```

SipService主要是路由文件和Sip协议栈文件的配置。容器根据路由文件配置把请求分发到相应的应用程序。如果路由配置不对，将导致请求不会进入到应用程序。

参数配置项：

参数名称	默认值	类型	说明
DarConfigurationFileLocation	无	String	路由配置文件
SipStackPropertiesFileLocation	无	String	sip协议栈参数配置文件
SipApplicationDispatcherClassName	org.mobicents.servlet.sip.core.SipApplicationDispatcherImpl	String	执行路由算法和应用选择过程
GatherStatistics	True	Boolean	是否允许收集数据
CanceledTimerTasksPurgePeriod	0	Integer	定期清理容器计时器资源
BaseTimerInterval	500	Integer	单位毫秒，重新传输INVITE请求的最小时间
T2Interval	4000	Integer	单位毫秒，最大重新传输非INVITE请求和INVITE回复
T4Interval	5000	Integer	单位毫秒，消息在网络上最大时间
TimerDInterval	32000		单位毫秒，针对UDP，等待重新传输的时间
DispatcherThreadPoolSize	15	Integer	处理sip消息的线程
AdditionalParameterableHeaders	无	String	添加sip头消息

BypassResponseExecutor	True	Boolean	是否允许在线程池下分发消息
DialogPendingRequestChecking	false	Boolean	是否检测sip事务重叠
CallIdMaxLength	0	Integer	sip头部Call-Id最大长度
TagHashMaxLength	0	Integer	sip头部From和To的最大程度
HttpFollowsSip	false	Boolean	是否sip粘滞
UsePrettyEncoding	true	Boolean	是否允许sip消息拆分为多行

其次是配置SipConnectorService，如下：

```
...
<SERVICE CLASS="com.apusic.web.sip.SipConnectorService"
NAME="apusic:service=SipConnector,name=udp">
    <ATTRIBUTE NAME="Port" VALUE="5080" />
    <ATTRIBUTE NAME="SignalingTransport" VALUE="udp" />
</SERVICE>
<SERVICE CLASS="com.apusic.web.sip.SipConnectorService"
NAME="apusic:service=SipConnector,name=tcp">
    <ATTRIBUTE NAME="Port" VALUE="5080" />
    <ATTRIBUTE NAME="SignalingTransport" VALUE="tcp" />
</SERVICE>
<SERVICE CLASS="com.apusic.web.sip.SipConnectorService"
NAME="apusic:service=SipConnector,name=tls">
    <ATTRIBUTE NAME="Port" VALUE="5081" />
    <ATTRIBUTE NAME="SignalingTransport" VALUE="tls" />
</SERVICE>
<SERVICE CLASS="com.apusic.web.sip.SipConnectorService"
NAME="apusic:service=SipConnector,name=ws">
    <ATTRIBUTE NAME="Port" VALUE="5082" />
    <ATTRIBUTE NAME="SignalingTransport" VALUE="ws" />
</SERVICE>

...
```

SipConnectorService主要是配置监听的端口和使用的协议，SipConnectorService可以配置多个，其中udp和tcp使用的端口可以是同一个，ws（WebSocket）和tls的端口必须唯一。如果只是用了udp，只配置一个SipConnectorService即可，无需配置多个。

参数配置项：

参数名称	默认值	类型	说明
Port	无	Integer	监听sip消息的端口
SignalingTransport	无	String	可选值udp/tcp/ws/tls
ProtocolHandler	org.mobicents.servlet.sip.apusic.SipProtocolHandler	String	使用默认值即可，无需改变
IpAddress	无	String	ip地址，如果有多个可手动指定
UseStun	false	Boolean	是否使用stun协议
StunServerAddress	无	String	使用stun协议时的ip地址
StunServerPort	0	Integer	使用stun协议时的端口
StaticServerAddress	无	String	Load-banlance端的地址
StaticServerPort	0	Integer	Load-banlance端的端口
UseStaticAddress	false	Boolean	是否启用上面两项

6.21 配置和管理工具

6.21.1 appctl工具

Apusic应用服务器目录下的bin子目录中，提供了appctl工具，在命令行下，可使用appctl 实用命令行工具来部署、卸载、重启动、输出应用列表等操作，同时，appctl 也可用于远程部署。下面介绍appctl 命令行工具的使用：

- 用法:

```
appctl [options] <command>
```

- 参数:

-s server-host-name (default is localhost:6888)

-p administrative-password (prompt from console if not provided)

- 命令列表：
    - install <应用名> <应用档案文件(.ear)> [-remote] [-vhost <虚拟主机名>] [-context <虚拟目录名>]
    - uninstall <应用名>
    - list
    - start <应用名>
    - stop <应用名>
    - restart <应用名>

- list-module <应用名>
- list-role <应用名>
- assign-role <应用名> <角色名> [[-u] <用户>...] [-g <组>...]
- unassign-role <应用名> <角色名>

- 示例:

输出应用列表。如希望输出当前本机运行于Apusic 服务器中的应用，可使用如下命令行:

```
appctl list
```

如输出IP 为xxx.xxx.xxx.xxx 的远程主机运行的Apusic 服务器中的应用，可使用如下命令行:

```
appctl -s xxx.xxx.xxx.xxx list
```

安装应用。安装your_app.ear应用包安装到本机运行的Apusic应用服务器。如果该应用已经存在，appctl会重新部署应用:

```
appctl install your_application_name your/packge/your_app.ear
```

卸载应用。卸载安装到本机运行的Apusic 服务器上的your_application_name 应用:

```
appctl uninstall your_application_name
```

远程安装应用。安装your_app.ear 应用包安装到远程运行的Apusic应用服务器:

```
appctl -s xxx.xxx.xxx.xxx install your_application_name
your/packge/your_app.ear -remote
```

	**注意**
	安装应用到远程运行的Apusic应用服务器必须加-s指定服务器的IP地址，并在应用档案文件后增加-remote参数，例如: appctl -s xxx.xxx.xxx.xxx install your_application_name your/packge/your_app.ear -remote appctl会将应用档案文件上传到远程服务器的${DOMAIN_HOME}/upload目录下，然后在server.xml中登记一项。

6.21.2 apclient工具

Apusic 应用服务器目录下的bin子目录中，提供了apclient工具。在命令行下，可使用apclient实用命令行工具来运行应用客户端程序。apclient会启动应用客户端容器，使应用客户端运行在客户端容器中，为客户端提供运行时环境，如JNDI包装等。下面介绍 apclient 命令行工具的使用：

- 用法：

```
apclient [-url <url>] [-noauth] -jar jarfile [-name app-name] [args...]
```

或

```
apclient [-url <url>] [-noauth] main-class [args...]
```

- 参数：

  url <url>：指定服务器url，该url具有以下格式：

  ```
  iiop://user:password@host:port
  ```

  例如：

  ```
  iiop://admin:admin@localhost:6888
  ```

  运行在客户端容器中的应用客户端会自动使用JAAS用户认证登录服务器，如果指定了url则不会弹出登录窗口。apclient使用$DOMAIN_HOME/config/clientauth.cfg作为登录配置文件。

  - noauth： 不弹出登录窗口，以匿名身份访问服务器 。
  - name <app-name>：当指定的JAR文件中包含多个应用客户端模块时，通过app-name确认调用哪个客户端模块。一般，app-name为客户端JAR的文件名。

- 示例：

运行应用客户端，以admin身份访问服务器：

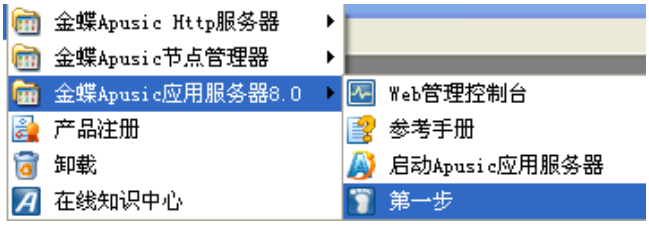
```
apclient iiop://admin:admin@localhost:6888 -jar appclient.jar
```

运行应用客户端，以匿名身份访问服务器：

```
apclient -noauth -jar appclient.jar
```

6.21.3 firststep工具

Apusic 应用服务器目录下的bin子目录中，提供了firststep工具。在命令行下，可使用firststep工具打开apusic服务器启动向导。通过此向导，可以启动Sample域下的Apusic应用服务器（带有示例应用，位于%APUSIC_HOME%/domains/sample/目录下）、打开示例应用（需先启动Sample域下的服务器）、开启Apusic管理控制台、开启Web管理控制台、开启Apusic信息中心、进入产品注册网页等。



6.21.4 dep工具

Apusic应用服务器目录下的bin子目录中，提供了dep命令脚本。在命令行下，可使用dep命令打开Apusic服务器部署工具。Apusic部署工具为部署基于EJB2.X的应用提供支持，包括对资源适配器(RAR)的配置，并对Apusic部署描述文件提供支持。注意dep工具不适用于没有部署描述文件的EJB3.0应用。

6.21.5 adminmain工具

从Apusic应用服务器4.0版开始，为增加管理和配置服务器的自动化功能，提供基于命令行的管理工具。通过命令行管理工具可以完成通过图形管理工具的管理和配置功能，可对服务器中的服务进行配置，并可察看各配置项的帮助信息。

Apusic应用服务器在运行时包含多个服务，每个服务包含一些可配置项，性能参数和管理操作。管理命令行管理工具提供通过命令行对应用服务器进行远程察看性能参数，修改可配置项，调用管理操作的方式。

6.21.5.1 选项

- -s 本选项用于指定管理工具连接的主机名，默认为localhost:6888
- -p 本选项用于指定被连接主机的管理员密码，默认为admin

6.21.5.2 用法

在Apusic安装目录的bin目录下($APUSIC_HOME/bin)，执行下面所述的命令完成相关任务。

- 显示指定服务器当前可用的服务列表

```
AdminMain -s <主机名:端口> -p <管理员密码> list
```

例如:

```
AdminMain -s localhost:6888 -p admin list
```

将返回本机监听6888端口的服务器的服务列表

- 显示指定服务的可配置项，性能参数和管理操作列表

```
AdminMain -s <主机名:端口> -p <管理员密码> list <服务名>
```

例如：

```
AdminMain -s localhost:6888 -p admin list http
```

将返回http服务包含的可配置项

- 将显示指定服务的配置项的描述和参数信息

```
AdminMain -s <主机名:端口> -p <管理员密码> list <服务名> <配置项名>
```

例如：

```
AdminMain -s localhost:6888 -p admin list http MaxClients
```

将返回http服务MaxClients条目的描述和参数信息

- 察看性能参数，修改配置项值或者调用管理操作

  ```
  AdminMain -s <主机名:端口> -p <管理员密码> <服务名> <配置项名> <[参数1]...>
  ```

  例如：

  - 使用如下命令显示可配置项的当前值，如下

  ```
  AdminMain -s localhost:6888 -p admin http MaxClients
  ```

  将显示http服务中MaxClients配置项的值

  - 如命令行根据可配置项的说明指定了参数列表，则修改当前可配置项的值为参数列表中指定的值，如下

  ```
  AdminMain -s localhost:6888 -p admin http MaxClients 200
  ```

  将修改http服务中 MaxClients 的值为200，并在修改后显示当前 MaxClients 的值

  - 使用如下命令调用管理操作

  ```
  AdminMain -s localhost:6888 -p admin http restart
  ```

将重新起动http服务，管理操作的调用类似于查询可配置项，不同的是查询配置项仅仅返回配置项的值，而管理操作的调用将使服务执行管理操作指定的行为

6.21.5.3 关于服务的说明

- 服务的运行、停止和重启动

大多数服务涉及运行和停止状态，因此，在对服务配置之前，需要先调用stop管理操作，并且在修改服务配置之后，调用start管理操作以启动服务，以免带来服务配置在运行状态中被修改带来的逻辑不一致的问题。

- 关于JDBC连接池的数据源属性

  JDBC数据源的多数配置均通过设置数据源的属性进行，可以通过调用指定数据源的setProperty, getProperty, removeProperty 对数据源进行配置。Apusic支持的数据源属性有如下：

  - user

  指定数据源访问数据库所使用的数据库用户名

  - password

  指定数据源访问数据库所使用的数据库用户密码

  - charset-fixup

  如果使用Oracle数据库，在存储中文时存在问题，可使用此属性，指定中文字符集名称以修正此问题，如"gbk"或"gb2312"

  - stmt-cache-size

  用于指定数据源能够在缓存中保存的SQL语句数

  - test-before-reused

  在连接池将曾经出错的连接返回客户请求之前，是否对连接进行测试，值为true或false

  - test-command

  如果test-before-reused的值为true，则连接池使用此属性指定的SQL语句对连接进行测试

- 关于动态资源

  Apusic服务包含很多动态资源，如消息队列，数据源等等，因此，在使用管理命令行时，动态资源的名字通常会在其名称之前，包含所属服务名，两者之间使用“/”分隔，对动态资源的配置和管理，与对服务的配置和管理没有区别。

6.21.6 JSPC工具

Apusic 的JSP编译器会自动的处理JSP页面，所以通常不需要直接使用JSP编译器。但是在一些特殊的情况下，直接使用Apusic的JSP编译器对JSP源文件进行预编译会非常有用。Apusic提供了jspc工具对JSP文件进行预编译，它的工作原理是首先对JSP源文件进行解析生成java文件，然后使用标准的Java编译器编译。

使用步骤如下所述：

- 编辑$APUSIC_HOME/bin/jspc.xml。需要修改的几个属性说明如下：

```
<!-- 需预编译的web模块路径 -->
<property name="src"
value="${apusic.home}/domains/mydomain/applications/default/public_html"/>
```

将value属性修改为需预编译的web模块的路径，默认是mydomain下的default应用

```
<!-- 预编译的class文件的存放路径 -->
<property name="dest"
value="${apusic.home}/domains/mydomain/applications/default/public_html/WEB-
INF/classes"/>
```

将value属性修改为预编译的class文件的存放路径，注意：编译生成的类文件一定要存放在WEB-INF\classes目录下

```
<!-- 编译依赖的jar文件 -->
<classpath>
    <pathelement location="${apusic.home}/lib/apusic.jar" />
    <pathelement location="${apusic.home}/lib/operamasks-impl.jar" />
    <pathelement location="${apusic.home}/common/javaee.jar" />
</classpath>
```

添加所要编译的jsp依赖的jar文件。

- 在命令行提示符下进入$APUSIC_HOME/bin目录，执行jspc即可开始预编译

<apusicjspc></apusicjspc>可配置的属性包括：

选项	描述
srcDir	需预编译的web模块路径
destDir	预编译的class文件的存放路径
keep	是否保留预编译过程中生成的java文件，默认为false
verbose	输出冗余信息，默认为false
nopkg	生成的类中不包含包名，默认为false
encoding	指定缺省的JSP字符编码
compiler	指定Java编译器

由于已经对JSP进行了预编译，所以应该在WEB-INF\web.xml部署描述文件中作适当设置不再编译JSP文件：

```
<context-param>
    <param-name>jsp.nocompile</param-name>
    <param-value>true</param-value>
</context-param>
```

如果想将jspc引入项目的ant构建环境中，只需要仿照$APUSIC_HOME/bin/jspc.xml编写构建任务，并将$APUSIC_HOME/lib/ant/ant-apusic.jar加入项目构建的classpath中。

对JSP进行预编译处理后可以按照正常的步骤部署Web应用。

6.22 使用Jconsole监控

JDK1.6 提供JMX remote的管理工具Jconsole，可以监控Java运行程序的内存使用情况、活动线程数量、类装载的数量、MBeans的状态、虚拟机的各种信息等，还可以执行MBean公开的方法或强制进行垃圾回收。因为Apusic应用服务器实现标准的JMX接口，因此也可以使用Jconsole进行监控。

6.22.1 用Jconsole监控和操作Apusic应用服务器

需要开启控制，则需要设置允许访问的ip列表。在Muxer服务中增加如下的属性配置：

```
    <ATTRIBUTE NAME="EnableMuxAllow" VALUE="true"/>
    <ATTRIBUTE NAME="MuxAllowIPs" VALUE="127.0.0.1"/>
```

EnableMuxAllow：表示是否开启这个功能，true表示开启，默认值为false；

MuxAllowIPs：设置允许访问的ip地址列表，多个ip以逗号分割，默认本机可以访问。

监控命令：执行Jdk的jconsole命令，使用Advanced登录窗口，输入指定的JMX URL:

```
service:jmx:iiop:///jndi/corbaname::1.2@hostname:6888#jmx/rmi/RMIConnectorServer
```

其中的`hostname`根据实际地址填写，进入后选择MBeans下的apusic，可以查看apusic应用服务器的各种参数，也可做控制apusic应用服务器的部分操作。

7 附录 A. 附录

A.1. 附录：jserv.conf ----------

```
#**************************************************************

#* Apusic JServ Configuration File *

#**************************************************************

#

# Note: this file should be appended or included into your
# apache_home\\conf\\httpd.conf
# Tell Apache on win32 to load the Apache JServ communication
# module LoadModule
LoadModule jserv_module "libexec\\ApacheModuleJServ-1.1-1.3.9.dll"

# Tell Apache on Unix to load the Apache JServ communication module
# For shared object builds only!!!

#LoadModule jserv_module libexec/mod_jserv.so
<IfModule mod_jserv.c>

ApJServManual on
#ApJServDefaultProtocol ajpv12
ApJServSecretKey DISABLED
ApJServMountCopy on
ApJServDefaultHost localhost
ApJServDefaultPort 8007
# ApJServMount [virtual_directory_in_apache]
[virtual_directory_in_apusic]

ApJServMount / /

#Mapping .jsp file to apusic
AddType text/jsp .jsp
#AddHandler jserv-servlet .jsp
</IfModule>
```

```
#*****************************************************************

#* END OF FILE JSERV.CONF *

#*****************************************************************
```

8 A.2. 附录：缺省的apusic.conf配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<CONFIG>
    <SERVICE CLASS="com.apusic.util.ThreadPoolService"
NAME="apusic:service=ThreadPool,name=default">
        <ATTRIBUTE NAME="MinSpareThreads" VALUE="5" />
        <ATTRIBUTE NAME="MaxSpareThreads" VALUE="30" />
        <ATTRIBUTE NAME="MaxThreads" VALUE="-1" />
        <ATTRIBUTE NAME="MaxQueueSize" VALUE="500" />
        <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.util.ThreadPoolService"
NAME="apusic:service=ThreadPool,name=MuxHandler">
        <ATTRIBUTE NAME="MinSpareThreads" VALUE="0" />
        <ATTRIBUTE NAME="MaxSpareThreads" VALUE="10" />
        <ATTRIBUTE NAME="MaxThreads" VALUE="10" />
        <ATTRIBUTE NAME="MaxQueueSize" VALUE="500" />
        <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.util.ThreadPoolService"
NAME="apusic:service=ThreadPool,name=HTTPHandler">
        <ATTRIBUTE NAME="MinSpareThreads" VALUE="5" />
        <ATTRIBUTE NAME="MaxSpareThreads" VALUE="30" />
        <ATTRIBUTE NAME="MaxThreads" VALUE="30" />
        <ATTRIBUTE NAME="MaxQueueSize" VALUE="500" />
        <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.util.ThreadPoolService"
NAME="apusic:service=ThreadPool,name=JMSHandler">
        <ATTRIBUTE NAME="MinSpareThreads" VALUE="0" />
        <ATTRIBUTE NAME="MaxSpareThreads" VALUE="30" />
        <ATTRIBUTE NAME="MaxThreads" VALUE="30" />
        <ATTRIBUTE NAME="MaxQueueSize" VALUE="500" />
        <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.util.ThreadPoolService"
NAME="apusic:service=ThreadPool,name=ORBWorker">
        <ATTRIBUTE NAME="MinSpareThreads" VALUE="5" />
```

```
        <ATTRIBUTE NAME="MaxSpareThreads" VALUE="30" />
        <ATTRIBUTE NAME="MaxThreads" VALUE="150" />
        <ATTRIBUTE NAME="MaxQueueSize" VALUE="500" />
        <ATTRIBUTE NAME="IdleTimeout" VALUE="300" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.net.Muxer">
        <ATTRIBUTE NAME="Port" VALUE="6888" />
        <ATTRIBUTE NAME="Backlog" VALUE="50" />
        <ATTRIBUTE NAME="Timeout" VALUE="300" />
        <ATTRIBUTE NAME="MaxWaitingClients" VALUE="200" />
        <ATTRIBUTE NAME="WaitingClientTimeout" VALUE="5" />
        <ATTRIBUTE NAME="SSLEnabled" VALUE="True" />
        <ATTRIBUTE NAME="SecurePort" VALUE="6889" />
        <ATTRIBUTE NAME="KeyStore" VALUE="config/sslserver" />
        <ATTRIBUTE NAME="KeyPassword" VALUE="keypass" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.corba.ORBService">
    </SERVICE>
    <SERVICE CLASS="com.apusic.naming.NameService">
    </SERVICE>
    <SERVICE CLASS="com.apusic.transaction.TransactionService">
        <ATTRIBUTE NAME="DefaultTimeout" VALUE="600" />
        <ATTRIBUTE NAME="EnableLog" VALUE="False" />
        <ATTRIBUTE NAME="RetryTimeout" VALUE="600" />
        <ATTRIBUTE NAME="RetryInterval" VALUE="60" />
        <ATTRIBUTE NAME="HeuristicDecision" VALUE="rollback" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.jdbc.JDBCService">
        <ATTRIBUTE NAME="EnableActiveConnTrace" VALUE="False" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.security.SecurityService">
    </SERVICE>
    <SERVICE CLASS="com.apusic.jdbc.trace.JDBCTracerService">
        <ATTRIBUTE NAME="StackTraceAllow" VALUE="False" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.connector.JCAService">
    </SERVICE>
    <SERVICE CLASS="com.apusic.ejb.EJBService">
        <ATTRIBUTE NAME="SessionCacheSize" VALUE="1000" />
        <ATTRIBUTE NAME="EntityCacheSize" VALUE="1000" />
```

```
        <ATTRIBUTE NAME="SessionStoreDirectory" VALUE="store/ejb" />
        <ATTRIBUTE NAME="DefaultEntityPersistenceDataSource"
VALUE="jdbc/sample" />
        <ATTRIBUTE NAME="DefaultMessageDrivenConnectionFactory"
VALUE="jms/ConnectionFactory" />
        <ATTRIBUTE NAME="Cmp11Promotion" VALUE="True" />
        <ATTRIBUTE NAME="EnablePassByReference" VALUE="False" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.ejb.timer.EJBTimerService">
        <ATTRIBUTE NAME="MaxRedeliveries" VALUE="1" />
        <ATTRIBUTE NAME="RedeliveryInterval" VALUE="5000" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.web.WebService">
        <ATTRIBUTE NAME="MaxWaitingClients" VALUE="500" />
        <ATTRIBUTE NAME="WaitingClientTimeout" VALUE="5" />
        <ATTRIBUTE NAME="KeepAlive" VALUE="true" />
        <ATTRIBUTE NAME="KeepAliveTimeout" VALUE="15" />
        <ATTRIBUTE NAME="MaxKeepAliveRequests" VALUE="100" />
        <ATTRIBUTE NAME="MaxKeepAliveConnections" VALUE="300" />
        <ATTRIBUTE NAME="NumberSelectors" VALUE="1" />
        <ATTRIBUTE NAME="ServletReloadCheckInterval" VALUE="3" />
        <ATTRIBUTE NAME="EnableLog" VALUE="False" />
        <ATTRIBUTE NAME="LogFileName" VALUE="logs/access.log" />
        <ATTRIBUTE NAME="LogFileLimit" VALUE="1000000" />
        <ATTRIBUTE NAME="LogFileCount" VALUE="10" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.web.session.SessionService">
        <ATTRIBUTE NAME="DefaultSessionTimeout" VALUE="3600" />
        <ATTRIBUTE NAME="MaxSessionsInCache" VALUE="1024" />
        <ATTRIBUTE NAME="SessionInvalidateCheckInterval" VALUE="60" />
        <ATTRIBUTE NAME="Distributable" VALUE="False" />
        <ATTRIBUTE NAME="Replicable" VALUE="False" />
        <ATTRIBUTE NAME="SessionStick" VALUE="False" />
        <ATTRIBUTE NAME="PersisteSession" VALUE="False" />
        <ATTRIBUTE NAME="DestorySessionOnApplicationStop" VALUE="True" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.web.session.FileSessionStoreService">
        <ATTRIBUTE NAME="Directory" VALUE="store/http_sessions" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.web.http.mux.MUXEndpoint"
```

```
NAME="Endpoint:type=mux,service=httpd">
    </SERVICE>
    <SERVICE CLASS="com.apusic.web.http.acp.ACPEndpoint"
NAME="Endpoint:type=acp,service=httpd">
        <ATTRIBUTE NAME="Timeout" VALUE="300" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.jms.store.file.FileMessageStoreProvider">
        <ATTRIBUTE NAME="StoreDirectory" VALUE="store/jms" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.jms.server.JMSServer">
        <ATTRIBUTE NAME="RoutingRedeliveryInterval" VALUE="60" />
        <ATTRIBUTE NAME="RoutingMaxRedeliveryCount" VALUE="100" />
        <ATTRIBUTE NAME="TransactedDelivery" VALUE="False" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.jms.server.JMSRegistry">
    </SERVICE>
    <SERVICE CLASS="com.apusic.deploy.runtime.J2EEDeployer">
        <ATTRIBUTE NAME="EnableAutoDeployment" VALUE="True" />
        <ATTRIBUTE NAME="WatchedDirectories" VALUE="applications" />
        <ATTRIBUTE NAME="UploadDirectory" VALUE="upload" />
    </SERVICE>
    <SERVICE CLASS="com.apusic.jmx.adaptors.rmi.RMIAdaptorService"
NAME="Adaptor:type=rmi">
    </SERVICE>
    <SERVICE CLASS="com.apusic.tools.admin.AdminService">
    </SERVICE>
    <SERVICE CLASS="com.apusic.util.ClassLoaderViewer">
    </SERVICE>
</CONFIG>
```

9 A.3. 附录：缺省的datasources.xml配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE datasources
PUBLIC '-//Apusic//DTD Apusic Data Source Configuration 1.0//EN'

'http://www.apusic.com/dtds/datasources_1_0.dtd'>
<datasources>
    <!--datasource name="SampleDB"

jndi-name="jdbc/sample"

driver-class="sun.jdbc.odbc.JdbcOdbcDriver"

url="jdbc:odbc:apusic"

>
<property name="user" value="scott"/>
<property name="password" value="tiger"/>
</datasource-->
</datasources>
```

10 A.4. 附录：缺省的jms.xml配置文件

```
<?xml version="1.0"?>
<!DOCTYPE jms-config
PUBLIC "-//Apusic//DTD Apusic JMS Configuration 1.2//EN"

"http://www.apusic.com/dtds/jms-config_1_2.dtd">
<!--

Use this file to configure one or more queues and topics used by JMS
aware applications.

>
<jms-config>
<!-- Administered connection factories -->
<connection-factory secure="false" anonymous="true" default-delivery-
mode="persistent" default-priority="4"
    default-time-to-live="0" idle-timeout="300" min-wait-time="2" max-wait-
time="60" min-spare-connections="5"
    max-spare-connections="30" max-connections="-1">
    <description>The default connection factory</description>
    <display-name>ConnectionFactory</display-name>
    <jndi-name>jms/ConnectionFactory</jndi-name>
</connection-factory>
<!-- Queues and Topics -->
<queue cache-size="64" expiry-check-interval="60" clustered="false">
    <description>Undeliverable Message Queue</description>
    <queue-name>SYSTEM.DEAD_LETTER</queue-name>
    <jndi-name>SYSTEM.DEAD_LETTER</jndi-name>
</queue>
<queue clustered="false">
    <queue-name>testQueue</queue-name>
    <jndi-name>testQueue</jndi-name>
</queue>
<queue clustered="false">
    <queue-name>testReplyQueue</queue-name>
    <jndi-name>testReplyQueue</jndi-name>
</queue>
<topic>
```

```
    <topic-name>testTopic</topic-name>
    <jndi-name>testTopic</jndi-name>
</topic>
<!-- Systems queues are allowed only by system administraters. -->
<destination-permission>
    <role-name>everyone</role-name>
    <destination-method>
        <queue-name-pattern>SYSTEM.*</queue-name-pattern>
        <deny-all />
    </destination-method>
    <destination-method>
        <queue-name-pattern>SYSTEM.DEAD_LETTER</queue-name-pattern>
        <allow-all />
    </destination-method>
    <destination-method>
        <queue-name-pattern>SYSTEM.FILE_TRANSPORT_REQUEST_LETTER</queue-name-
pattern>
        <allow-all />
    </destination-method>
    <destination-method>
        <queue-name-pattern>SYSTEM.FILE_TRANSPORT_REPLY_LETTER</queue-name-
pattern>
        <allow-all />
    </destination-method>
    <destination-method>
        <queue-name-pattern>SYSTEM.FILE_TRANSPORT_LETTER</queue-name-pattern>
        <allow-all />
    </destination-method>
</destination-permission>
<destination-permission>
    <role-name>admin</role-name>
    <destination-method>
        <queue-name-pattern>SYSTEM.*</queue-name-pattern>
        <allow-all />
    </destination-method>
</destination-permission>
<!-- Systems topics are allowed only by system administraters. -->
<destination-permission>
    <role-name>everyone</role-name>
    <destination-method>
```

```
        <topic-name-pattern>SYSTEM.*</topic-name-pattern>
        <deny-all />
    </destination-method>
</destination-permission>
<destination-permission>
    <role-name>admin</role-name>
    <destination-method>
        <topic-name-pattern>SYSTEM.*</topic-name-pattern>
        <allow-all />
    </destination-method>
</destination-permission>
<security-role>
    <role-name>everyone</role-name>
    <group>everyone</group>
</security-role>
<security-role>
    <role-name>admin</role-name>
    <principal>admin</principal>
</security-role>
</jms-config>
```

11 A.5. 附录：缺省的logging.xml配置文件

```
<?xml version="1.0"?>
<!DOCTYPE logging SYSTEM "http://www.apusic.com/dtds/logging.dtd">
<!-- Logging configuration -->
<logging>
    <handler class="com.apusic.logging.ConsoleHandler">
        <property name="level" value="ALL" />
        <formatter class="com.apusic.logging.SimpleFormatter" />
    </handler>
    <handler class="com.apusic.tools.admin.WebConsoleHandler">
        <property name="level" value="ALL" />
        <formatter class="com.apusic.logging.SimpleFormatter" />
    </handler>
    <handler class="java.util.logging.FileHandler">
        <property name="level" value="ALL" />
        <property name="limit" value="1000000" />
        <property name="count" value="10" />
        <property name="pattern" value="%h/logs/apusic.log" />
        <property name="append" value="true" />
        <formatter class="com.apusic.logging.SimpleFormatter" />
    </handler>
    <!-- Set the logging level for the root of the namespace. -->
    <!-- This becomes the default logging level for all Loggers. -->
    <logger name="" level="WARNING" />
    <!-- The default logging level for server services. -->
    <logger name="apusic" level="WARNING" />
    <logger name="apusic.server" level="INFO" />
    <logger name="apusic.service" level="INFO" />
</logging>
```

12 A.6. 附录：缺省的server.xml配置文件

```
<?xml version="1.0"?>
<!DOCTYPE server
PUBLIC "-//Apusic//DTD Apusic Server Application 1.1//EN"

"http://www.apusic.com/dtds/server_1_1.dtd">
<server>
    <application name="default" base="applications/default" start="auto" />
</server>
```

13 A.7. 附录：缺省的auth.cfg配置文件

```
com.apusic.security.ServerLogin {

/*

* This LoginModule is used to authenticate server principal.

* You can set following options to customize this module:

*

* useTicketCache Set this to true, if you want the TGT
* to be obtained from the ticket cache.

* Set this option to false if you do not
* want this module to use the ticket cache.

*

* ticketCache Set this to the name of the ticket cache
* that contains user's TGT. If this is set,

* useTicketCache must also be set to true.

*

* useKeyTab Set this to true if you want the module to
* get the principal's key from the keytab.

* If keytab is not set then the module will
* locate the keytab from the Kerberos
* configuration file.

*

* keyTab Set this to the file name of the keytab to get
* principal's secret key. If this is set,
```

```
* useKeyTab must also be set to true.

*

* storeKey Set this to true if you want the principal's
* key to be stored in the Subject's private
* credentials. This option is madatory, you
* should not set it to false.

*/

com.sun.security.auth.module.Krb5LoginModule required storeKey=true;

};

// The default login configuration used by server applications
other {

/*

* A simple server login module, which retrieves principal and
* credentials from current security context, and store them
* into Subject's principal set and private credential set.

*/

com.apusic.security.auth.login.ServerLoginModule required;

};
```

14 A.8. 附录：缺省的clientauth.cfg配置文件

```
// The login configuration used by application clients
client {

com.apusic.security.auth.login.ClientPasswordLoginModule required;

};

deploytool {

com.apusic.security.auth.login.ClientPasswordLoginModule required;

};
```

15 A.9. 附录：缺省的acp_isapi.properties配置文件

```
servers = local
server.local.host = 127.0.0.1
server.local.port = 6888
servlet-mapping./servlet/* = local
servlet-mapping.*.jsp = local
servlet-mapping./examples/* = local
```

A.10. 附录：logging.dtd ----------

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT logging (handler*, logger*)>
<!ELEMENT handler (property*, filter?, formatter?)>
<!ATTLIST handler
class CDATA #REQUIRED
>
<!ELEMENT filter EMPTY>
<!ATTLIST filter
class CDATA #REQUIRED
>
<!ELEMENT formatter EMPTY>
<!ATTLIST formatter
class CDATA #REQUIRED
>
<!ELEMENT property EMPTY>
<!ATTLIST property
name CDATA #REQUIRED
value CDATA #REQUIRED
>
<!ELEMENT logger (handler*)>
<!ATTLIST logger
name CDATA #REQUIRED
level CDATA #REQUIRED
useParentHandlers (true|false) "true"

>
```

16 部分VI 开发指南

16.1 Web开发手册

16.1.1 Apusic JSP 开发

16.1.1.1 JSP概述

JavaServer Pages (JSP)技术允许开发人员轻松的创建静态或动态的Web内容。JSP不仅具有象Servlet一样创建动态Web页面的能力，而且更接近创建静态内容的方式。JSP使得页面的动态内容与它的表示进一步分离。JSP的设计迎合了两个不同的技术层面：页面的图形设计和创建动态内容的软件开发。JSP的主要特性包括：

- JSP是基于文本的文档，用来处理客户端请求（request）和构造响应（response）
- 构造server端访问的对象
- 定义扩展JSP语言的机制

Apusic JSP建立在Sun Microsystem提出的JSP2.1规范之上，完全符合JSP2.1规范，同时兼容JSP2.0与JSP1.2规范。

16.1.1.1.1 什么是JSP

JSP是以文本为基础的文档，它包含两种类型的文本：静态模板数据和JSP元素。

- 静态模板数据：表达成文本格式内容，如HTML、XML等
- JSP元素：用来构造动态内容

JSP页面定义了实现JSP页面语义的实现类。实现类是Servlet的子类，容器在用户访问JSP页面时自动转向到对应的实现类来处理用户的请求。

16.1.1.1.2 JSP如何处理请求

下面描述了一个典型的JSP请求是怎样被处理的：

1. 浏览器从服务器请求一个JSP页面（JSP页面文件以.jsp作为后缀）
2. Apusic服务器接受请求
3. JSP容器查找被请求的JSP文件，然后利用JSP编译器将该文件转换为一个实现了javax.servlet.jsp.JspPage接口的servlet类。jsp文件只有在需要时才会被编译；一般在第一次被请求或文件被修改后。这样，以前编译的JSP servlet类就可以被重复使用，使得以后的响应更快
4. 产生的JspPage servlet类被调用来处理浏览器请求

16.1.1.2 Apusic JSP变动及其升级

本章节主要介绍Apusic JSP部分跟随规范变动和升级的内容，具体开发内容的变动体现在开发指南部分

16.1.1.2.1 从JSP2.0到JSP2.1的主要变动

JSP 2.1支持统一表达语言Unified Expression Language

在JSP2.0的表达式语言基础上，JSP2.1引入了统一表达式语言（Unified EL)。它是JSP2.0所提供的表达式语言与为JavaServer Faces技术而创的表达式语言的结合体。

JSP2.0 中的表达式语言允许网页设计者使用简单的表达式来动态读取JavaBean的数据。JSP支持简单的请求(request)/响应(response)生命周期。在此周期中，当页面被执行时，HTML标记会被即时生成。因此，JSP2.0所提供的简洁、只读的表达式语言非常适合JSP应用。

另一方面，JavaServer Faces技术提供了一个多阶段的生命周期来支持其精密的用户界面组件模型。此模型允许转换与校验组件数据，把组件数据传递到对象，以及处理组件事件。为了支持这些功能，JavaServer Faces技术所使用的表达式语言需要提供以下功能：

- 表达式的延迟演算
- 允许对数据赋值
- 允许调用方法

合并这两种表达式语言有两个主要原因：第一，网页开发者就可以自由地混合JSP与JavaServer Faces标签，而不需要担心两种技术由于生命周期不同而造成冲突。第二，令其他基于JSP的技术也能运用JSF所使用的新技术。虽然标准JSP标签库与静态内容仍旧只能使用JSP2.0中的技术，但JSP自定义标签开发人员在开发新标签时则可以从Unified EL的新特性中得益。

16.1.1.2.2 从JSP1.2到JSP2.0的主要变动

一. JDK要求

JSP 1.2可以在Java 2标准版1.3版本运行，而JSP 2.0要求使用Java 2标准版1.4或更新版本，JSP 2.0使用由Servlet 2.4规定的Web程序部署描述格式。

二. JSP 2.0支持表达语言expression language

JSP 2.0支持表达语言(expression language)。JSTL表达式语言可以使用标记格式方便地访问JSP的隐含对象和JavaBeans组件，JSTL的核心标记提供了流程和循环控制功能。自制标记也有自定义函数的功能，因此基本上所有seriptlet能实现的功能都可以由JSP替代。请参考："表达式语言"

三. JSP 2.0中加入了JSP fragment

JSP 2.0中加入了JSP fragment，它的基本特点是可以使处理JSP的容器推迟评估JSP标记属性。一般JSP是首先评估JSP标记的属性，然后在处理JSP标记时使用这些属性，而JSP fragment提供了动态的属性。

四. JSP 2.0中加入了SimpleTag

JSP 2.0中加入了新的Tag处理方法(javax.servlet.jsp.tagext.SimpleTag)，它定义了用来实现简单标记的接口。SimpleTag接口不使用doStartTag()和doEndTag()方法，而提供了一个简单的doTag()方法。需要在一个自制标记中实现的所有逻辑过程、循环和对标记体的评测等都在这个方法中实现。

16.1.1.3 Apusic JSP开发指南

16.1.1.3.1 JSP页面的生命周期

JSP象Servlet一样处理客户端的请求（request）。因此，JSP页面的生命周期和许多功能都决定于Servlet技术。关于Servlet的讨论请参考"Apusic Http Servlets 开发"。

当一个请求（request）映射到JSP页面时，请求（request）首先被特殊的Servlet处理，检查JSP页面和它对应的实现类。如果发现实现类比JSP页面旧，那么就把JSP页面转化成一个servlet类并编译。在这个过程中，JSP比Servlet具有优势的一个地方是编译是自动进行的。

在转化阶段，对JSP页面中每一种类型的数据处理都不一样。模板数据被转化成代码然后写入返回给客户端的输出流中。JSP元素被作如下处理：

- 伪指令（Directives）被用作控制 Web 容器如何转化和执行JSP页面。请参考 “伪指令元素 (Directive Elements)”。
- 脚本元素（Scripting Elements）被插入到JSP页面对应的实现类中。请参考 “脚本元素 (Scripting Elements)”。
- 动作（Actions）被转化成调用 JavaBeans或 Java Servlet API。请参考 “动作元素(Action Elements)”。

如果一个JSP页面命名为test，那么JSP的实现类的源文件就保存在：

APUSIC_HOME/domains/<domain_name>/deploy/<webapp_name>/tmpfiles/jsp/_jspx_test__jsp.java

转化和编译阶段都可能产生错误，但只有在第一次访问页面时才被发现。如果错误发生在页面转化过程中，服务器将抛出ParseException异常，并且Servlet源文件为空或不完整。如果错误发生在页面的编译期，服务器会抛出JasperException异常和错误所在的Servlet源文件的位置。一旦JSP被转化和编译，JSP页面的实现类基本上和Servlet的声明周期相同。请参考“Servlet生命周期”。

- 如果JSP页面的实现类不存在，那么容器：

1. 转载实现类
2. 实例化
3. 调用jspInit方法初始化

- 调用_jspService方法，处理请求（request）和响应(response)
- 如果容器需要删除JSP页面的实现类，调用jspDestroy方法

16.1.1.3.2 初始化和结束一个JSP页面

开发人员能够通过覆盖JspPage接口的jspInit方法来定制JSP页面的初始化过程，允许它读取持久性配置文件、初始化资源和执行任何一次性的动作。开发人员可以调用jspDestroy释放资源。在JSP页面中定义方法，请参考 “脚本元素 (Scripting Elements)”。

16.1.1.3.3 创建静态内容

开发人员可以象创建普通页面一样简单的在JSP页面中创建静态内容。静态内容表现为文本格式，如HTML、WML和XML，缺省的格式是HTML。

16.1.1.3.4 创建动态内容

开发人员可以通过在JSP元素中访问Java对象来创建动态内容。

一. 使用内置对象

在JSP页面中可以访问不同的对象，包括EJB、JavaBeans等。JSP技术使一些对象自动可用，开发人员也可以创建与应用相关的对象。内置对象是由Web容器创建的、包含了与请求（request）响应（response）和应用相关信息的对象，大部分的内置对象都在Java

Servlet技术规范中定义。JSP页面的作者能够访问确定的内置对象，从页面的开始这些内置对象都是可用的，可以在脚本元素(Scripting Elements)中通过脚本变量访问。使用页面声明标签（<%! %>）定义的方法无法直接使用内置对象，需要通过参数传入

每一个内置对象都对应核心 Java API或 Java Servlet API 的类或接口。

变量名	类型	描述
request	javax.servlet.ServletRequest的子类，如javax.servlet.http.HttpServletRequest	request触发了 JSP页面的执行。作用范围：request。
response	javax.servlet.ServletResponse的子类，如javax.servlet.http.HttpServletResponse	返回给客户端的响应（response）。在JSP页面中不常使用。作用范围：page
pageContext	javax.servlet.jsp.PageContext	JSP页面的执行环境。提供了管理不同作用范围属性的API，在扩展标记中被大量使用。作用范围：page
session	javax.servlet.http.HttpSession	为客户端创建的 session对象。作用范围：session
application	javax.servlet.ServletContext	在同一个Web应用中的Servlet 和其他Web组件的运行环境。从ServletConfig对象获得。作用范围：application
out	javax.servlet.jsp.JspWriter	输出流。作用范围：page
config	javax.servlet.ServletConfig	JSP页面从 ServletConfig对象获得初始化信息。作用范围：page
page	java.lang.Object	JSP页面实现类处理当前请求的实例。等同于“this”。作用范围：page

“作用范围”的概念在下一节介绍。

二. 对象的作用范围

当开发人员在JSP页面中实例化一个对象，可能希望它可以被Web应用中的其他对象访问。为了控制对象的可访问性，JSP规范定义了一系列作用范围对象，在其中可以放置对其他对象的引用（reference）。这些作用范围是：page、request、session和application。在运行过程中，作用范围被相应的Java对象实现。

作用范围	对象类型	描述
page	javax.servlet.jsp.PageContext	作用范围为page的对象只能在创建它的当前页面被访问。当响应（response）返回给客户端或请求被转向其他地方时，所有对该对象的引用都将释放。作用范围为page的对象其引用（reference）存储在PageContext对象中。
request	javax.servlet.ServletRequest	作 用范围为request的对象可以在处理同一请求（request）的JSP页面中被访问。请求（request）处理结束后，所有对该对象的引用都将释 放。如果请求（request）在同一运行时间被转移给其他Web资源，对象仍然

		可访问。作用范围为request的对象其引用（reference）存 储在request对象中。
session	javax.servlet.http.HttpSession	作用范围为session的对象可以在同一session的JSP页面中被访问。当session结束后，所有对该对象的引用都将释放。作用范围为session的对象其引用（reference）存储在session对象中。
application	javax.servlet.ServletContext	作 用范围为application的对象可以在同一Web应用的JSP页面中被访问，对象的引用（reference）存储在application对象 中。当运行环境收回ServletContext时，所有存储在application对象中对象引用（reference）都将被释放。

使用useBean动作时可以定义对象的作用范围，选择一个范围属性来确定JavaBeans实例在什么范围可用，如：

```
<jsp:useBean id="myCart" scope="session" class="Cart">
```

范围属性的实现对象在JSP页面中自动可用，表现为脚本变量，在脚本元素中使用。这些脚本变量和范围属性使用同样的名字：page、request、session和application。例如下面的代码片断使用了内置的request对象。因为使用了session变量来存储Cart对象，那么它可以被同一session作用范围的其他JSP页面访问。

```
<%
CartLineItem lineItem = new CartLineItem();
lineItem.setID(request.getParameter("cdId"));
lineItem.setCDTitle(request.getParameter("cdTitle"));
lineItem.setPrice(request.getParameter("cdPrice"));
myCart.lineItems.addElement(lineItem);
session.putValue("myLineItems", myCart.getLineItems());
%>
```

	**注意**
	缺省情况下，JSP页面可以访问session作用范围。但如果页面的伪指令page设置session的属性为false，那么页面将不能访问当前的Http session，而且也不能使用session内置对象。

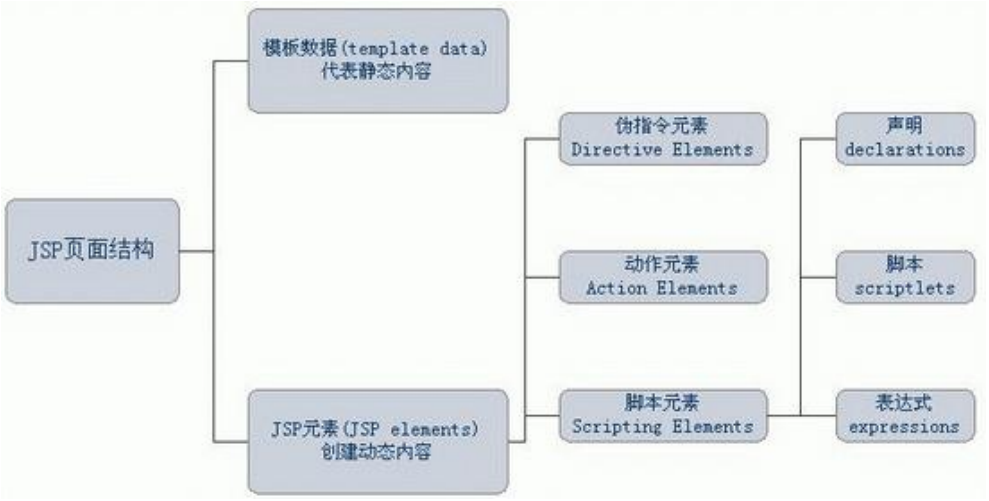
例如，当页面包含下面的代码，前面的useBean动作和脚本片段都将非法：

```
<%@ page session="false" %>
```

三. JSP元素(Element)

JSP 包含模板数据（template data）和元素（elements）。元素能被Web容器识别，提供了创建动态内容的能力。模板数据不需要识别，例如HTML和XML代码，直接被返回到客户端。模板数据通常用来产生静态内容和格式化动态内容。JSP元素分为三个类别：伪指令元素(Directive Elements)、动作元素（Action Elements）和脚本元素（Scripting Elements）。

JSP页面结构图



(1). **伪指令元素 (Directive Elements)**

伪指令提供了跟任何请求（request)无关的、关于JSP页面的全局声明信息。例如可以在页面引入包（import package），也可以指示页面关联当前的Http Session。伪指令不能向响应（response）中输出信息，它只是为JSP页面转化成相应Servlet的转化阶段提供信息。伪指令的语法：

```
<%@ directive { attr=”value” }* %>
```

例如，下面的伪指令引入了java.util包和关联JSP页面到当前的Http Session。

```
<%@ page import="java.util.*" session="true"%>
```

JSP规范定义了三种伪指令：page、include和taglib。

伪指令	描述	语法
include	包含其他文件的内容	<%@ include file=" relativeURL "%>
page	定义JSP页面的属性	<%@ page page_directive_attr_list %> page_directive_attr_list ::= {language="scriptingLanguage"} {extends="className" } { import="importList" } { session="true|false" } {buffer="none|sizekb" } {autoFlush="true|false" } {isThreadSafe="true|false" }

		{ info="info_text"} { errorPage="error_url" } {isErrorPage="true|false" } {contentType="ctinfo" } {pageEncoding="peinfo" }
taglib	为自定义标记设置标记库和前缀。	<%@ taglib uri=" URI " prefix=" tagPrefix"%>

(2). **动作元素** (Action Elements)

动作元素可以影响当前的输出和使用、改变、创建对象。动作元素提供了处理请求（request）阶段的信息。动作元素可以是标准的，定义在JSP规范中；也可以是客户自定义的，请参考"JSP扩展标记"。动作元素遵循XML的语法。例如，jsp:include动作可以插入JSP页面到当前的页面：

```
<jsp:include page="header.jsp" flush="true" />
```

jsp:useBean动作可以创建JavaBean对象：

```
<jsp:useBean id="cBean" scope="application" class="Expns.CBean">
<%
cBean.getConnected();
cBean.getEngine();
%>
</jsp:useBean>
```

下面列出了JSP规范定义的标准动作 。

动作元素	描述	语法
<jsp:forward>	传递请求（request）到其他Web资源	<jsp:forward page="relativeURLspec"/> 和 <jsp:forwardpage="urlSpec"> { <jsp:param .... />}* </jsp:forward>
<jsp:getProperty>>	插入JavaBeans的属性值到结果中	<jsp: getProperty name=" beanInstanceName "property=" propertyName "/>
<jsp:include>	包含静态文件或Web组件的执行结果	<jsp:include page="urlSpec"flush="true|false"/> 和 <jsp:include page="urlSpec"flush="true|false"> { <jsp:param ..../> }* </jsp:include>
<jsp:plugin>	引起applet或bean的执行	<jsp:plugintype="bean|applet" code="objectCode" codebase="objectCodebase" {align="alignment" }

		{ archive="archiveList"} { height="height" } {hspace="hspace" } { jreversion="jreversion"} { name="componentName" } {vspace="vspace" } { width="width" } { nspluginurl="url" } {iepluginurl="url" } > {<jsp:params> {<jsp:paramname="paramName" value=”paramValue" />}+ </jsp:params> } {<jsp:fallback> arbitrary_text </jsp:fallback>} </jsp:plugin>
<jsp:setProperty>	设置JavaBeans的属性	<jsp: setProperty name=" beanInstanceName" { property="*" | property="propertyName " [param=" parameterName "]| property=" propertyName " value="{ string | <%=expression %>}" } />

<jsp:useBean>	定位或实例化JavaBeans，指定名字和作用范围。	<jsp: useBean id=" beanInstanceName "
		scope=" page |request| session| application"
		{
		class=" package. class " [type="package. class "] |
		beanName="{ package. class| <%= expression %>}"
		type=" package.class "|
		type=" package. class "
		}
		{/> | > other elements
		</ jsp: useBean> }

<jsp:param>	提供 键/值信息	<jsp:param name="name" value="value" />
<jsp:params>	是<jsp:plugin>的一部分	
<jsp:fallback>	是<jsp:plugin>的一部分	

(3). **脚本元素** (Scripting Elements)

脚本元素使开发人员能够在JSP页面中嵌入Java代码。脚本元素通常用来维护对象，也可以通过执行计算来影响内容的产生。有三种类型的脚本元素：声明（declarations）、脚本（scriptlets）和表达式（expressions）。

声明（declarations）用来在JSP页面中声明变量和方法。声明不会产生任何输出，在JSP页面初始化时被初始化。JSP页面的回调方法jspInit()与jspDestroy()应在此定义。例子：

```
<%! int i; %>
<%! int i = 0; %>
<%! public String f(int i) { if (i<3) return("..."); ... } %>
```

语法：

```
<%! declaration(s) %>
```

脚本（scriptlets）允许开发人员嵌入任何有效的Java代码。可以使用在声明中定义的变量和方法。脚本在请求（request）的时刻执行。一个简单的例子，产生日期和时间：

```
<% if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM)
{%>
Good Morning
<% } else { %>
Good Afternoon
<% } %>
```

在脚本中同样能够使用本地变量：

```
<% int i; i= 0; %>

Hi, the value of i is <% i++ %>
```

语法：

```
<% scriptlet %>
```

表达式（expressions）使开发人员能够在页面中嵌入任何有效的Java表达式。在请求（request）时刻Web容器计算表达式的值并转换结果为字符串（String），然后把结果字符串写入HttpResponse对象。例如插入当前日期：

```
<%= (new java.util.Date()).toLocaleString() %>
```

语法：

```
<%= expression %>
```

16.1.1.3.5 表达式语言

JSP2.0的一个首要特点就是表达式语言，表达式语言能使方便地访问JavaBeans组件成为可能。例如，表达式语言能使用简单的语法${name}，${name.foo.bar}，访问对象和对象内部属性。

举例：test属性的值为session-scoped范围内cart对象中numberOfItems属性值是否大于0

```
<c:if test="${sessionScope.cart.numberOfItems > 0}">

...
</c:if>
<c:if test="${sessionScope.cart.numberOfItems > 0}">

...
</c:if>
```

注意：表达式语言中的属性与脚本(Scriptlets)中的变量并不互通。若要在脚本(Scriptlets)中访问表达式语言中的属性，需要使用相应scope的getAttribute()方法访问；若要在表达式语言中使用脚本中的变量，则需要在脚本中使用相应scope(例如page)的setAttribute()方法设置属性

一. 使表达式语言失效

在JSP2.0之前，因为--${}--不是保留的关键字和符号，在以前的系统中，可能存在使用了该符号。为了保持向前兼容的功能，可以使表达式语言失效，语法如下：

```
<%@ page isELIgnored ="true|false" %>
```

属性为true时，表达式语言失效，属性为false时，表达式语言生效，系统默认是false。

二. 使用表达式语言

表达式语言能使用在

- 静态文字
- 任何能够接受表达式语言的标签属性，包括标准或定制的标签。

三种方式设定标签属性值

- 简单表达式 ，<some:tag value="${expr}"/>，属性值被强制转换成属性值期望的类型。
- 有一个或多个表达式，且有文字夹杂，<some:tag value="some${expr}${expr}text${expr}"/>，这种情况先转换成字符串，然后被强制转换成属性值期望的类型。
- 直接赋值，<some:tag value="sometext"/>。

表达式通常根据期望类型的上下文来设定标签属性值，当不能准确匹配时，通常会进行强制转换。例如：${1.2E4 + 1.4}给一个float标签赋值时，会通过以下转换：

```
Float.valueOf("1.2E4 + 1.4").floatValue()
```

三. **变量**

web容器寻找表达式中出现的变量时，根据page, request,session,application范围和次序来寻找变量，当变量名为系统对象时，返回系统对象。

主要系统对象：

- pageContext：可提供访问servletContext，session，request，response等对象。
- 其它一些对象：

名称	解释
param	将请求参数名称映射到单个字符串参数值（通过调用 ServletRequest.getParameter(String name) 获得）。getParameter (String) 方法返回带有特定名称的参数。表达式$(param.name) 相当于 request.getParameter (name)。
paramValues	将请求参数名称映射到一个数值数组（通过调用 ServletRequest.getParameter(String name) 获得）。它与 param 隐式对象非常类似，但它检索一个字符串数组而不是单个值。表达式${paramvalues.name) 相当于 request.getParamterValues(name)。
header	将请求头名称映射到单个字符串头值（通过调用 ServletRequest.getHeader(Stringname) 获得）。表达式 ${header.name} 相当于request.getHeader(name)。
headerValues	将请求头名称映射到一个数值数组（通过调用 ServletRequest.getHeaders(String)获得）。它与头隐式对象非常类似。表达式 ${headerValues.name} 相当于request.getHeaderValues(name)。
cookie	将 cookie 名称映射到单个 cookie 对象。向服务器发出的客户端请求可以获得一个或多个cookie。表达式 ${cookie.name.value} 返回带有特定名称的第一个 cookie值。如果请求包含多个同名的 cookie，则应该使用 ${headerValues.name} 表达式。
initParam	将上下文初始化参数名称映射到单个值（通过调用ServletContext.getInitparameter(String name) 获得）。

四. **文字**

JSP 表达式语言定义可在表达式中使用的以下文字：

文字	文字的值
Boolean	true 和 false
Integer	与 Java 类似。可以包含任何正数或负数，例如 24、-45、567
Floating Point	与 Java 类似。可以包含任何正的或负的浮点数，例如 -1.8E-45、4.567
String	任何由单引号或双引号限定的字符串。对于单引号、双引号和反斜杠，使用反斜杠字符作为转义序列。必须注意，如果在字符串两端使用双引号，则单引号不需要转义。

Null	Null

五. 操作符

类型	操作符
算术型	+、-（二元）、*、/、div、%、mod、-（一元）
逻辑型	and、&&、or、
关系型	==、eq、!=、ne、、gt、<=、le、>=、ge。可以与其他值进行比较，或与布尔型、字符串型、整型或浮点型文字进行比较。
空	空操作符是前缀操作，可用于确定值是否为空。
条件型	A ?B :C。根据 A 赋值的结果来赋值 B 或 C。

六. 保留关键字

这些关键字不一定现在就已经用到，但是将来可能用到

```
and eq gt true instanceof
or ne le false empty
not lt ge null div mod
```

七. 举例列表

表达式	结果
${1 > (4/2)}	False
${4.0 >= 3}	True
${100.0 == 100}	True
${(10*10) ne 100}	False
${'a' < 'b'} true ${'hip' gt 'hit'}	False
${4 > 3}	True
${1.2E4 + 1.4}	12001.4
${3 div 4}	0.75
${10 mod 4}	2
${!empty param.Add}	当request中Add的属性值为空字串或null时返回真

${pageContext.request.contextPath}	The context path
${sessionScope.cart.numberOfItems}	The value of the numberOfItems property of the session-scoped attribute named cart
${param['mycom.productId']}	返回request中mycom.productId的属性值
${header["host"]}	The host
${departments[deptName]}	departments map中的deptName键值
${requestScope['javax.servlet.forward.servlet_path']}	The value of the request-scoped attribute named javax.servlet.forward.servlet_path

16.1.1.3.6 使用JAVABEANS

JavaBeans 组件是Java的类，能够很容易的重用和组装在一起形成应用。任何Java类只要遵循一定的设计惯例就可能成为JavaBeans组件。JSP技术通过JSP元素支持JavaBeans组件，可以创建和初始化beans，设置和得到属性。本节提供了JavaBeans组件的基本信息和在JSP页面中访问JavaBeans组件。

一. JavaBeans 设计惯例

JavaBeans的设计惯例约定了类的属性和访问属性的public方法的定义方式。JavaBeans的属性能够：

- 读 / 写、只读、只写
- 包含单值或数组

属性不一定要通过实例变量被实现，但属性必须能够通过pubilc方法访问。对public方法的要求是：

- 对应每个可读的属性，必须有方法PropertyClass getProperty() { ...}得到属性
- 对应每个可写的属性，必须有方法setProperty(PropertyClass pc) { ...}设置属性

除了属性方法，JavaBeans组件还必须具有无参数的公共构造函数。下面是一个简单的JavaBeans范例程序：

```
public class Currency {
  private Locale locale;
  private double amount;

  public Currency() {
    locale = null;
    amount = 0.0;
  }

  public void setLocale(Locale l) {
    locale = l;
  }
```

```
  public void setAmount(double a) {
    amount = a;
  }

  public String getFormat() {
    NumberFormat nf = NumberFormat.getCurrencyInstance(locale);
    return nf.format(amount);
  }
}
```

二. 创建和使用JavaBeans

在JSP页面中使用jsp:useBean标记声明JavaBeans，形式如下：

```
<jsp:useBean id="beanName" class="fully_qualified_classname" scope="scope" />
```

jsp:userBean 标记声明页面将使用JavaBeans，并设置JavaBeans的作用范围：application、session、request和page。如果bean不存在，那么声明将创建bean然后把它存储在属性中设置的作用范围对象中。id属性决定了bean的名字，作为在JSP页面中引用bean对象的标识符。

三. 设置JavaBeans的属性

可以通过两种方式设置JavaBeans的属性：使用jsp:setProperty标记或者使用脚本，

```
<% beanName.setPropName(value); %>
```

jsp:setProperty的语法依赖于属性值的来源，下面总结了使用 jsp:setProperty 设置 JavaBeans属性的不同方式：

属性值的来源	语法
String常量	<jsp:setProperty name="beanName" property="propName" value="string constant"/>
Request参数	<jsp:setProperty name="beanName" property="propName" param="paramName"/>
Request参数名匹配bean属性	<jsp:setProperty name="beanName" property="propName"/> <jsp:setProperty name="beanName" property="*"/>
表达式	<jsp:setProperty name="beanName" property="propName" value="<%= expression %>"/>
	注意
	beanName必须和useBean标记中的id属性值相同JavaBeans组件中必须有setPropName方法

	paramName必须是请求（request）参数名

下面的代码范例声明了前面定义的JavaBeans，然后设置它的属性：

```
<jsp:useBean id="currency" class="util.Currency" scope="session">
    <jsp:setProperty name="currency" property="locale" value="<%=
request.getLocale() %>" />
</jsp:useBean>
<jsp:setProperty name="currency" property="amount" value="
<%=cart.getTotal()%>" />
```

属性可以设置为字符串（String）常量或请求（request）参数。由于这两种类型都是字符串（String），Web容器会自动的转换字符串（String)为正确的属性类型。下表列出了字符串如何转换为目标类型。

目标类型	转换
Bean属性	setAsText(string-literal)
boolean或Boolean	java.lang.Boolean.valueOf(String)
byte或Byte	java.lang.Byte.valueOf(String)
char或Character	java.lang.String.charAt(0)
double或Double	java.lang.Double.valueOf(String)
int或Integer	java.lang.Integer.valueOf(String)
float或Float	java.lang.Float.valueOf(String)
long或Long	java.lang.Long.valueOf(String)
short或Short	java.lang.Short.valueOf(String)
Object	String(string-literal)

四. 得到JavaBeans 的属性

可以通过几种方式获得JavaBeans组件的属性值。其中两种方式（jsp:getProperty标记和表达式）是转化属性值为String然后输出属性值到当前的内置对象out：

- `<jsp:getProperty name="beanName" property="propName" />`
- `<%= beanName.getPropName() %>`

对于这两种方法，beanName必须和useBean标记中的id属性值相同，而且JavaBeans中必须定义了getPropName方法。

如果需要获得属性值但并不转化它，也不把它输出到内置对象out，那么必须使用脚本，例如：

```
<% Object o = beanName.getPropName(); %>
```

下面的范例代码展示了如何获得前面声明的JavaBeans的属性值，

```
<jsp:getProperty name="currency" property="format" />
```

或，

```
<%= currency.getFormat() %>
```

16.1.1.3.7 在JSP页面中包含其他内容

在JSP页面中包含其他Web资源有两种方式：使用include伪指令和<jsp:include>动作。include伪指令是在JSP页面转化到对应的servlet的过程中被处理的，它的效果就是在JSP页面中插入另外一个文件的文本内容，不管文件是静态内容还是JSP页面。一般使用include伪指令包含banner内容和版权（copyright）信息。include伪指令的语法是：

```
<%@ include file="filename" %>
```

<jsp:include>动作是在JSP页面的执行过程中被处理的。<jsp:include>动作允许在JSP页面中包含静态或者动态的资源，而包含静态和动态资源的结果十分不同。如果资源是静态的，那么它的内容被插入到包含它的JSP页面中；如果资源是动态的，那么首先把请求（request）传递给它，然后执行，最后把执行结果插入到包含它的JSP页面中。<jsp:include>动作的语法是：

```
<jsp:include page="includedPage" />
```

include 机制	描述	语法
include伪指令	不管是静态还是动态资源，都是插入文件的文本内容	<%@ include file="filename" %>
<jsp:include>动作	静态资源，插入文件的文本内容动态资源，插入JSP页面的执行结果	<jsp:include page="includedPage" />

16.1.1.3.8 传递控制到其他WEB组件

从JSP页面传递控制到其他Web组件的方法是使用Java Servlet API。在JSP 页面中使用jsp:forward动作可以达到传递控制的目的，例如：

```
<jsp:forward page="/main.jsp" />
```

16.1.1.3.9 在JSP页面中包含APPLET

可以使用jsp:plugin元素在JSP页面中包含applet或JavaBeans组件。jsp:plugin产生适当的依赖于客户端浏览器HTML代码（<object>或<embed>），如果有必要，则会下载Java插件和客户端组件并执行。<jsp:plugin>标记被<object>或<embed>标记代替，

<jsp:plugin>的属性标记提供了配置信息。<jsp:param>标记指出了applet或JavaBeans的参数，<jsp:fallback>标记指出了当applet或JavaBeans不被支持时客户端显示的内容。下面是一个在JSP页面中包含applet的例子：

```
<jsp:plugin type=applet code="Molecule.class" codebase="/html" >
    <jsp:params>
        <jsp:param
name="molecule"

value="molecules/benzene.mol"/>
    </jsp:params>
    <jsp:fallback>
        <p> unable to start plugin </p>
    </jsp:fallback>
</jsp:plugin>
```

jsp:plugin标记的语法是：

```
<jsp:plugin
type="bean|applet"
code="objectCode"
codebase="objectCodebase"
{ align="alignment" }
{ archive="archiveList" }
{ height="height" }
{ hspace="hspace" }
{ jreversion="jreversion" }
{ name="componentName" }
{ vspace="vspace" }
{ width="width" }
{ nspluginurl="url" }
{ iepluginurl="url" } >
{ <jsp:params>
    { <jsp:param name="paramName" value="paramValue" /> }+
</jsp:params> }
{ <jsp:fallback> arbitrary_text </jsp:fallback> }
</jsp:plugin>
```

16.1.1.3.10 使用APUSIC JSP预编译

Apusic 的JSP编译器会自动的处理JSP页面，所以通常不需要直接使用JSP编译器。但是在一些特殊的情况下，直接使用Apusic的JSP编译器对JSP源文件进行预编译会非常有用。Apusic提供了jspc工具对JSP文件进行预编译，它的工作原理是首先对JSP源文件进行解析生

成java文件，然后使用标准的Java编译器编译。

详情请参考"JSPC工具"

对JSP进行预编译处理后可以按照正常的步骤部署Web应用。

16.1.1.3.11 启用JSP热加载

如果要开启jsp热加载，应该在将 `ServletReloadCheckInterval` 参数更改之后，在 `web.xml` 中添加参数 `jsp.reload.noly` 。

```
<context-param>
<description>
Only JSP hot loading
</description>
<param-name>jisp.reload.only</param-name>
<param-value>true</param-value>
</context-param>
```

16.1.2 Apusic Http Servlets 开发

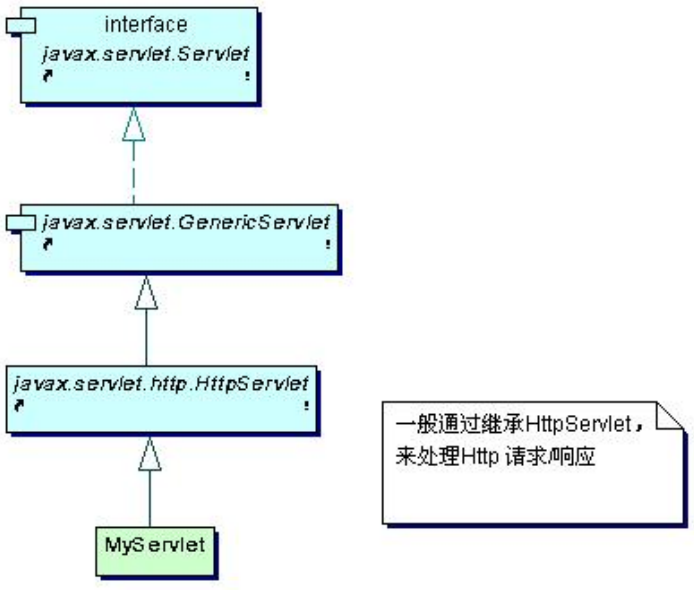
16.1.2.1 Http Servlets 概述

本章节介绍了HTTP Servlet的基本概念，概述了如何在Apusic应用服务器上进行HTTP Servlet的开发。

16.1.2.1.1 什么是SERVLET

Servlet是基于Java技术的Web组件，用来扩展以请求/响应为模型的服务器的能力。Servlets可以响应任何类型请求，但我们通常使用Http Servlets来处理HTTP请求（request）和提供HTTP响应（response）。

Servlet继承层次



Apusic HTTP Servlets通常用来创建交互式的应用：以标准Web浏览器作为客户端表示，Apusic应用服务器处理服务器端的商业逻辑。Apusic HTTP Servlets可访问数据库、EJB 、消息API 、HTTP Session和其它Apusic应用服务器提供的其他功能。

Apusic服务器完全支持最新的Servlet2.5规范。HTTP Servlets是JavaEE标准的组成部分。

16.1.2.1.2 使用SERVLET的目的

创建动态Web页面。使用HTML表单得到最终用户的输入，并提供HTML页面作为输出，响应用户的请求。例如在线购物车，金融服务和个性化内容等。

生成如在线会议等协作系统。

运行在Apusic应用服务器上的Servlets可以获得各种不同的API和系统服务的支持，如：

Session跟踪——使Web站点可以跨越多个页面跟踪用户的过程。这个功能支持Web站点提供类似于电子商务中的购物车。Apusic应用服务器在集群服务器之间提供Session迁移。

JDBC驱动程序——可以使用JDBC驱动程序提供的基本的数据库访问服务，同时也可以使用Apusic应用服务器提供的数据库连接池、服务器端的数据缓存和事务处理等多种服务。

可对Servlets使用不同的安全类型，如使用SSL来提供安全通讯。

可使用EJB来封装Sessions 、数据库中的数据和其他功能。

JMS——允许Servlet与其他Servlet和程序交换消息。

Servlets可使用标准JDK API 。

可传递一个请求（request）到其他Servlet或其他资源。

为任何其他J2EE兼容的Servlet引擎编写的Servlet可以轻松的部署到Apusic应用服务器上。

Servlets和JSP可以一起创建Web应用。

16.1.2.1.3 SERVLET开发概述

HTTP Servlets程序员使用标准的Java API（javax.servlet.http）来创建交互式应用程序。

HTTP Servlets可以读取HTTP头信息，并向客户端浏览器发送HTML代码。

在Apusic应用服务器中，Servlet作为web应用的一部分部署。Web应用是一个由诸如servlet类、JSP、静态Web页面、图像、安全描述等应用组件组成的。

16.1.2.1.4 SERVLET与JAVAEE

Servlet2.5 规范作为JavaEE的组成部分，定义了Servlet API的实现和Servlet如何部署在企业应用中。在兼容服务器如Apusic应用服务器上部署Servlet ，是通过将组成企业应用的Servlet和其他资源打包到一个"Web应用"中来完成的。Web应用通过特定的目录结构来容纳资源和部署描述。Web应用可使用后缀是".war" 的压缩文件包部署。

16.1.2.1.5 SERVLET API参考

Apusic应用服务器支持Servlet 2.5规范，可以从Sun Microsystems的网站获得这些文档：

API文档： javax.servlet

javax.servlet.http

Servlet2.5规范

16.1.2.2 Apusic Http Servlets变动及其升级

本章节主要介绍Apusic Http Servlets部分跟随规范变动和升级的内容

16.1.2.2.1 从SERVLET2.4到SERVLET2.5的主要变动

一. 依赖于J2SE5.0

Servlet 2.5规范现在把J2SE 5.0(JDK 1.5)作为最低要求。这个变动保证了J2SE5.0中所有新的语言特性都在Servlet 2.5中可用，包括：泛型支持、自动包装、增强的迭代循环、新的enum枚举类型、静态导入、可变参数表、元数据注解等等

二. 支持注解(Annotation)

注解提供了一套机制，通过元数据来装饰Java代码结构（类、方法、域等等）。注解并不象代码一样执行，而是对代码进行标记，使代码处理机可以根据这些元数据信息作出相应的行为。以前的版本提供了一些不同的小技巧来对类或方法进行注解，例如使用Serializable接口来标记序列化类，或使用 @deprecated的Javadoc注释来标记弃用的方法。新的元数据机制提供一套标准的方法来对代码进行注解，并提供了类库来创建自定义的注解类型。请参考

三. 更便利的web.xml的配置

Servlet 2.5中引入了一些对web.xml的小变动，使得部署配置更为方便。

(1). servlet名称通配符

当编写<filter-mapping>元素时，现在可以在<servlet-name>中使用*号来代表多个servlet（当然也包括了多个jsp）。例如：

```
<filter-mapping>
    <filter-name>Image Filter</filter-name>
    <servlet-name>*</servlet-name>
    <!-- 新特性 -->
</filter-mapping>
```

或

```
<filter-mapping>
    <filter-name>Dispatch Filter</filter-name>
    <servlet-name>*</servlet-name>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

(2). 在servlet映射中使用多个url-pattern标签

现在一个<servlet-mapping>元素中可以包含多个<url-pattern>元素，例如：

```
<servlet-mapping>
    <servlet-name>color</servlet-name>
    <url-pattern>/color/*</url-pattern>
    <url-pattern>/colour/*</url-pattern>
</servlet-mapping>
```

同样地，现在一个<filter-mapping>可以包含多个<url-pattern>与<servlet-name>元素，例如：

```
<filter-mapping>
    <filter-name>Multipe Mappings Filter</filter-name>
    <url-pattern>/foo/*</url-pattern>
    <servlet-name>Servlet1</servlet-name>
    <servlet-name>Servlet2</servlet-name>
    <url-pattern>/bar/*</url-pattern>
</filter-mapping>
```

(3). HTTP方法名称

以前的版本中，<http-method>元素只能接受7种标准HTTP/1.1方法之一：GET，POST，PUT，DELETE，HEAD，OPTIONS，与TRACE。在Servlet 2.5规范中，可接受所有合法的HTTP/1.1方法名，包括WebDAV方法如：LOCK,UNLOCK,COPY,MOVE等

16.1.2.2.2 从SERVLET2.3到SERVLET2.4的变动

一. Servlet 2.4版使用XML Schema作为部署描述文件定义

Servlet 2.4版使用XML Schema定义作为部署描述文件，这样使Web容器更容易校验web.xml语法。同时XML Schema提供了更好的扩充性。

```
<DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

"http://java.sun.com/dtd/web-app_2_3.dtd">
```

变为在可以根元素指定属性，当然以前的方式也兼容

```
version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
```

二. Servlet 2.4版在加入了ServletRequest监听器

Servlet 2.4在事件监听器中加入了ServletRequest监听器，包括新的API：ServletRequestListener，ServletRequestAttributeListener和其他相关类。这些类可以用来管理和控制与ServletRequest动作有关的事件。请参考Application Events

三. Servlet 2.4版增强了请求分发器的过滤功能

Servlet 2.4的Web程序增强了filter和request dispatcher的配合功能，这样过滤器可以根据请求分发器(request dispatcher)所使用的方法有条件地对Web请求进行过滤。请参考：“Servlet Filtering”

四. Servlet 2.4版可以定义网站的字符编码方式

Servlet 2.4增加了Web程序国际化功能，在web.xml中可以定义网站的字符编码方式。

```
<locale-encoding-mapping>
    <locale>zh</locale>
    <encoding>GB2312</encoding>
</locale-encoding-mapping>
```

当客户请求了特定语言的Web资源时，servlet程序通过ServletResponse接口的setLocale方法设置一个Web响应的语言属性。

16.1.2.3 Apusic Http Servlets编程初步

本章节介绍了基本的Http Servlet编程知识。

16.1.2.3.1 SERVLET生命周期

Servlet的生命周期是由运行Servlet的容器控制的。当一个请求（request）被映射到Servlet上时，容器会按照下面的步骤执行：

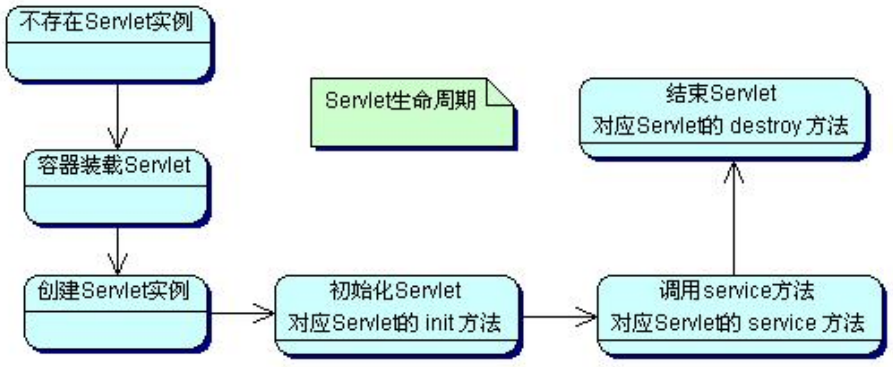
- 如果Servlet的实例不存在，那么容器

1. 装载Servlet类
2. 创建这个Servlet类的一个实例
3. 调用Servlet的init方法来初始化。相关的讨论在“初始化Servlet”

- 调用 Servlet 的service方法，传递request和response对象。相关的讨论在“编写Service方法”。

如果容器需要移去一个Servlet，它调用destroy方法来结束这个Servlet。相关的讨论在“结束一个Servlet”。

Servlet 的生命周期



16.1.2.3.2 编写一个基本HTTP SERVLET的步骤

这一节描述了编写一个简单的Http Servlet的基本步骤。

1. 引入相应的包和类，包括

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

2. 继承javax.servlet.http.HttpServlet

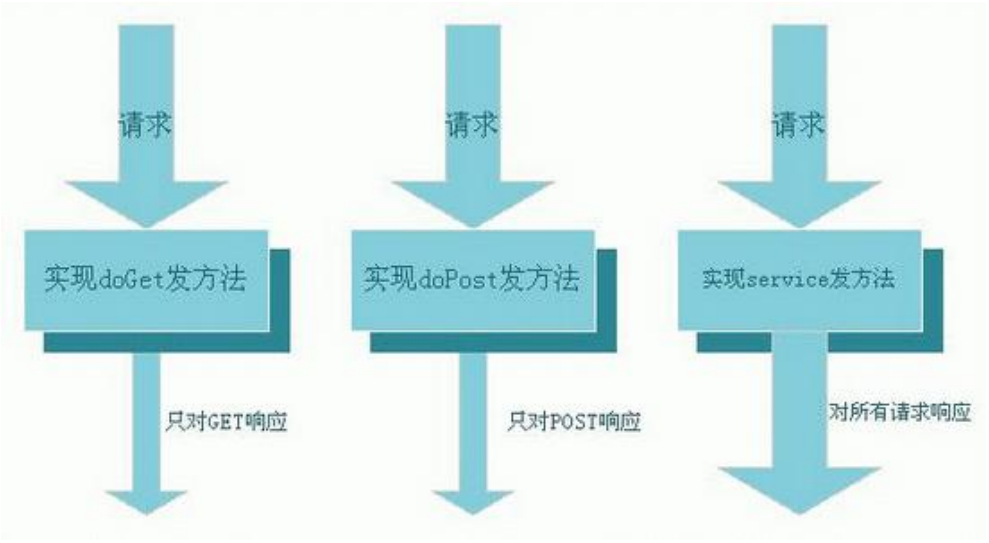
```
public class MyServlet extends HttpServlet{
```

3. 实现service方法

Servlet的主要功能是接受从浏览器发送过来的HTTP请求（request），并返回HTTP响应（response）。这个工作是在service方法中完成。service方法包括从request对象获得客户端数据和向response对象创建输出。

如果一个servlet从javax.servlet.http.HttpServlet继承，实现了doPost或doGet方法，那么这个servlet只能对POST或GET作出响应。如果开发人员想处理所有类型的请求（request），只要简单的实现service方法即可（但假如选择实现service方法，则不必实现doPost或doGet方法，除非在service方法的开始调用super.service()）。

Service方法对请求的处理



HTTP Servlet规范描述了用来处理其他请求（request）类型的方法，所有这些方法都可以归属于service方法。 所有的service 方法使用一样的参数。HttpServletRequest提供关于请求（request）的信息，servlet可以使用HttpServletResponse 对HTTP客户端作出响应。

```
public void service(HttpServletRequest req,HttpServletResponse res)

throws IOException{
```

4. 设置响应内容的类型：

```
res.setContentType("text/html");
```

5. 获得java.io.PrintWriter对象的引用，用来输出：

```
PrintWriter out = res.getWriter();
```

6. 使用PrintWriter对象的println()方法创建一些HTML代码，例如：

```
out.println("<html><head><title>Hello World!</title></head>");

out.println("<body><h1>Hello World!</h1></body></html>");

}

}
```

7. 编译Servlet：

从存放此Servlet源代码文件的目录编译此Servlet到包含此Servlet的应用中的WEB-INF/classes目录。如:

```
javac -d /your_application_dir/WEB-INF/classes your_servlet.java
```

8. 将此Servlet作为应用的一部分部署,请参看 “打包和部署Web模块”。

9. 从浏览器访问Servlet

一般说来，调用Servlet的URL取决于包含Servlet的Web应用的名字和Web应用部署描述中的Servlet映射的名字。请求（request）参数也可是调用Servlet的URL的一部分，一般Servlet的URL如以下模式：

```
http://APUSIC_ADDRESS:<port>/your_web_application_name/mapped_servlet_name?parameter
```

16.1.2.3.3 高级特征

通过前面的步骤创建了一个基本的Servlet，开发人员也许需要使用一些Servlet的高级特性：

- 处理HTML表单—— Http Servlets可以接收和处理从浏览器传送过来的HTML表单，请参考“提供HTTP响应”
- 应用的设计—— Http Servlets 提供了许多方法来设计开发人员的应用：
  - "取得客户端请求"
  - "处理多线程情况"
  - "调用其他Web资源"
- 初始化servlet——如果Servlet在初始化时需要初始化数据，获得初始化参数，或者执行一些其他的动作，那么就需要覆盖init()方法，请参考“初始化Servlet”
- 使用Session可以跟踪用户的信息，请参考“维护客户端状态”

16.1.2.4 Apusic Http Servlets编程进阶

本节介绍了如何在Apusic应用服务器的环境下编写Http Servlets。

16.1.2.4.1 初始化SERVLET

当Web容器装载和实例化一个Servlet类，在这个Servelt接收客户请求之前，Web容器需要初始化Servlet。开发人员可以覆盖Servlet接口的init()方法，允许Servlet读取配置信息、初始化资源和执行其他任何一次性的行为。Servlet初始化过程的失败将抛出UnavailableException异常。

- 启动Apusic应用服务器时初始化Servlet

通过设定Apusic Web应用的部署描述文件中Servlet类的<load-on-startup>标记，可使服务器在启动时初始化Servlet。同时，通过设定包含此Servlet的Web应用部署描述中的参数，可传递初始化参数到此Servlet 。

下例在Web应用部署描述文件中设定了一个名为"company_name"，值为"Apusic"的初始化参数，

```
<servlet>

...
<init-param>
        <param-name>company_name</paramname>
        <param-value>Apusic</param-value>
    </init-param>
</servlet>
```

通过调用`getInitParameter(String name)`方法可取得此参数。

- 重载init()方法

通过重载init()方法可以定制Servlet初始化的动作。下面代码重载了init()方法，并使用了前面定义的company_name初始化参数。

```
String company_name;

public void init(ServletConfig config) throws ServletException {
  company_name = getInitParameter("company_name");
}
```

16.1.2.4.2 使用注解

Servlet2.5 规范对某些注解如何在servlet环境中运作进行了规定。简单的servlet容器可以忽略这些规则，但JavaEE容器中的servlet则必须遵循这些规定。某些注解提供了部署信息，作为web.xml部署描述文件的替代品；某些注解请求容器进行某些动作，否则servlet就需要自行去实现这些任务。还有些注解则同时兼具以上两种功用。这些注解并不是由Servlet2.5规范定义的，规范仅仅解释了这些注解如何在servlet环境中运作。以下列出一些在JavaEE5中常见的注解以及它们的使用意图：

一. @Resource与@Resources

@Resource 用来标记一个类或变量，请求servlet容器对其进行"资源注入"。当容器遇到这个注解时，将在servlet进入服务状态之前对变量赋予合适的值。通过这个注解，避免了在代码中显式使用一个JNDI查找调用以及在web.xml中声明资源。服务器自动通过反射机制完成这两个任务。注入资源可以是数据源、JMS目的地、或环境变量。以下是个简单的例子：

```
@Resource
javax.sql.DataSource catalog;

public getData() {
  Connection con = catalog.getConnection();
}
```

在servlet进入服务状态之前，容器将自动通过JNDI找到名为catalog，类型为DataSource的变量并将其引用赋值给此处的catalog变量。

@Resources注解作用与@Resource相似，用来包含一系列的@Resource注解

二. @PostConstruct与@PreDestroy

这两个注解把一个或多个类方法标记为生命周期回调方法。标记为@PostConstruct的方法将在资源注入后被回调，提供机会对注入的资源进行初始化。标记为@PreDestroy的方法将在servlet停止服务前被回调，提供机会释放注入的资源。这些回调方法必须为无返回值(void)且不抛出任何checked异常的非静态方法。在servlet中，这些注解的作用基本上是可以让任意方法成为第二个init()或destroy()方法。

三. @EJB

与@Resource类似，但用于注入Enterprise JavaBeans引用。

四. @WebServiceRef

与@Resource和@EJB类似，但用于注入Web service引用。

五. @Persistence系列注解

@PersistenceContext，@PersistenceContexts，@PersistenceUnit，@PersistenceUnits：在EJB3.0规范中定义的用于持久对象的注解。

六. @DeclareRoles

定义了在应用中使用的安全角色。当用于servlet时，这个注解可作为web.xml文件中<security-role>标签的替代品。

七. @RunAs

声明一个类将以哪个角色的名义运行。当用于servlet时，这个注解可作为web.xml文件中<run-as>标签的替代品。

16.1.2.4.3 编写SERVICE方法

Servlet通过下面的方法来提供服务：

- 实现GenericServlet的service方法
- 实现HttpServlet的doMethod方法(doGet、doDelete、doOptions、doPost、doPut、doTrace)
- 实现Servlet接口并提供任何其他协议相关的方法。

通常，service方法用来从客户请求（request）中提取信息，访问扩展资源，并基于上面的信息提供响应（response）。

对于HTTP servlets，正确提供响应的过程是首先填写响应（response）的头信息，然后从响应（response）中得到输出流，最后向输出流中写入内容信息。响应（response）头信息必须最先设置。下面两节将描述从请求（request）中获得信息和产生HTTP响应（response）。

一. 取得客户端请求

客户端请求（request）包含了从客户端传递到Servlet的数据。所有的请求(request)都实现了ServletRequest接口。这个接口定义了一些方法来访问下面的信息：

类型描述	对应方法
参数，用来在客户端和Servlet之间传送信息	getAttribute(java.lang.String name) getAttributeNames() getInputStream() getParameter(java.lang.String name) getParameterMap() getParameterNames() getParameterValues(java.lang.String name)
对象值属性，用来在Servlet容器和Servlet之间，或者协作的Servlet之间传递信息	removeAttribute(java.lang.String name) setAttribute(java.lang.String name, java.lang.Object o)
有关请求使用的协议信息，客户端和服务器在请求中的调用	getContentLength() getContentType() getProtocol() getReader() getRealPath(java.lang.String path) getRemoteAddr() getRemoteHost() getRequestDispatcher(java.lang.String path) getScheme() getServerName() getServerPort() isSecure()
有关localization的信息	getCharacterEncoding() getLocale() getLocales() setCharacterEncoding(java.lang.String env)

下面的代码片断示范了如何使用request中的方法获得客户端信息。

```
Enumeration params = request.getParameterNames();
String paramName = null;
String[] paramValues = null;
while (params.hasMoreElements()) {
  paramName = (String) params.nextElement();
  paramValues = request.getParameterValues(paramName);
  System.out.println("\\nParameter name is " + paramName);
  for (int i = 0; i < paramValues.length; i++) {
    System.out.println(", value " + i + " is " + paramValues[i].toString());
  }
}
```

HTTP servlets使用HTTP request对象(HttpServletRequest)，它包含了request URL，HTTP头信息, 查询字符串等等。

HTTP request URL 包括几个部分：

http://<host>:<port><request path>?<query string>

一般情况下：

requestURI = contextPath + servletPath + pathInfo

Context path：通过getContextPath方法获得

Servlet Path：通过getServletPath方法获得

PathInfo：通过getPathInfo方法获得

例如：

Request Path	Path Elements
/catalog/help/feedback.jsp	• ContextPath: /catalog • ServletPath: /help/feedback.jsp • PathInfo: null

二. 提供HTTP响应

响应(response)包含了在服务器和客户端之间传递的数据。所有的响应（response）都实现了ServletResponse接口。这个接口定义了一些方法提供给开发人员使用：

类型描述	对应方法
获得向客户端发送数据的输出流	• 发送字符流:getWriter() • 发送字节流：getOutputStream()
指示响应返回的内容类型(例如：text/html)。已经注册的内容类型名称保存在IANA(Internet Assigned Numbers Authority)：	• setContentType(java.lang.String type)
指出是否是缓冲输出。缺省情况下写入输出的内容被立即发送到客户端。使用缓冲后写入输出的内容先不发送到客户端，这样Servlet有更多的时间设置相应的状态码和头信息，或者转移到其他的Web资源	• flushBuffer() • getBufferSize() • isCommitted() • reset() • resetBuffer() • setBufferSize(int size) • setContentLength(int len)
设置localization信息	• getCharacterEncoding() • getLocale() • setLocale(java.util.Locale loc)

HTTP response类（HttpServletResponse）有一些代表HTTP头信息的域：

- 状态码，用来指出响应（response）失败的原因
- Cookies，在客户端存储应用相关的信息，有时cookies用来维护和标识用户的session。

Servlet 首先设置响应（response）头信息：响应（response）的内容类别和缓冲区大小，然后在doGet方法中从响应(response)获得PrintWriter，最后向输出中写入HTML代码，调用close()方法提交这次对客户端的响应（response）。

```
public void doGet (HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

// 设置头信息

response.setContentType("text/html");

response.setBufferSize(8192);

PrintWriter out = response.getWriter();

// 向response中输出

out.println("<html>" +

"<head><title>+

messages.getString("TitleBookDescription")

+</title></head>");

...

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

// 关闭输出流

out.close();
```

```
}
```

16.1.2.4.4 处理多线程情况

HTTP servlets通常具有处理多个客户端并发访问的能力。如果在Servlet中的方法访问了共享资源，那么开发人员需要创建在某一时刻只接收一个客户端请求（request）的Servlet来进行并发控制。

如果Servlet不仅继承HttpServlet类，而且同时实现SingleThreadModel接口，那么Servlet就可以在某一时刻只处理一个客户端请求（request）。

实现了SingleThreadModel接口不需要写任何额外的方法。只要Servlet实现了SingleThreadModel接口，Apusic应用服务器就确保在某一时刻只运行一个service方法。

例如，ReceiptServlet接受用户名和信用卡号，然后处理用户的订单并更新数据库。数据库连接作为共享资源，servlet应该同步对数据库连接的访问，或者servlet实现SingleThreadModel接口。

```
public class ReceiptServlet extends HttpServlet implements SingleThreadModel
{
  public void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
...
  }
...
}
```

16.1.2.4.5 维护客户端状态

许多应用需要获得一系列互相关联的客户端请求，例如购物车功能。因为HTTP是无状态的协议，Web应用应该维护这些状态，这就叫做session。为了使应用具有session功能，Java Servlet技术提供了管理session的API和集中实现session的机制。

一. 访问Session

Session表现为HttpSession对象。调用request对象的getSession方法访问session。这个方法返回当前客户端请求（request）所关联的session，如果不存在，则为当前请求（request）创建一个session。由于getSession方法可能会改变响应（response）的头信息，所以需要在获得PrintWriter或ServletOutputStream之前被调用。

二. 使属性和Session相关联

可以通过名称使对象值属性和一个session相关联。例如把购物车作为属性存储在session中，在其他Servlet中可以通过session再获得购物车。

```
// 得到用户session和购物篮
HttpSession session = request.getSession();
```

```
ShoppingCart cart = (ShoppingCart) session.getAttribute("cart");
...
```

三. Session管理

由于没有办法知道HTTP客户端不再需要session，因此每个session都关联一个时间期限使它的资源可以被回收。通过session的setMaxInactiveInterval和getMaxInactiveInterval方法访问超时时间。

为了确保session的有效，不超时，开发人员应该在service方法中周期性的访问session。当客户端完成一个（组）完整的交互过程后，可以使用invalidate()方法使服务器端的session无效，并清除session数据。

```
// 得到用户session和购物篮
HttpSession session = request.getSession();
// 付款完成，使session无效
session.invalidate();
...
```

四. Session跟踪

Web容器使用了一些方法使用户和特定的session相关联，这些方法在客户端与服务器端之间传递session的标识。这个标识可以作为cookies在客户端被维护，或者Web组件把这个标识包含在每个URL中返回到客户端。

如果应用需要使用session对象，那么开发人员必须确保在用户关闭cookies的情况下，应用能够改写URL使session跟踪功能激活。在所有返回给用户URL之前都调用response的encodeURL(URL)方法，这样在用户关闭cookies的情况下URL中就会包含session ID，否则不改变URL。 例如：

```
out.println("<p>   <p><strong><a href=\\"" +

response.encodeURL(request.getContextPath() + "/catalog") +

"\\">" + messages.getString("ContinueShopping") +

"</a>      " +

"<a href=\\"" +

response.encodeURL(request.getContextPath() + "/cashier") +

"\\">" + messages.getString("Checkout") +

"</a>      " +
```

```
"<a href=\\"" +

response.encodeURL(request.getContextPath() +

"/showcart?Clear=clear") +

"\\">" + messages.getString("ClearCart") +

"</a></strong>");
```

上面的代码改写了三个URL，如果客户端关闭 cookies，则URL被改写为:

```
http://localhost:6888/bookstore1/cashier;jsessionid=wKgUUxroPN$HVmpTkhU6YPLTqyM
```

如果客户端cookies未关闭，则URL不作任何改变:

```
http://localhost:6888/bookstore1/cashier
```

16.1.2.4.6 访问SERVLET环境(SERVLET CONTEXT)

Web组件中的环境(context)是实现了ServletContext接口的对象。可以通过getServletContext方法获得Web环境（Web context）。通过一些方法可以访问Web环境（ServletContext），以得到如下一些信息或资源:

- 初始化参数
- 与Web环境相关的资源
- 对象值属性
- 记录日志

```
//获得ServletContext
ServletContext context = getServletContext();
Counter counter = (Counter) context.getAttribute("hitCounter");
...
writer.println("The number of hits is: " + counter.incCounter());
...
context.log(sw.getBuffer().toString());
```

16.1.2.4.7 使用APUSIC SERVICES

在Apusic应用服务器的编程模型中，业务逻辑（包括数据库访问、事务和复杂计算等）被包装在EJB组件中。作为表现层，Servlet能够通过Java Naming Directory Interface (JNDI)访问EJB组件。首先访问JNDI获得EJB组件的代理，然后可以象引用普通对象一样访问

EJB组件。所有这一切，都是由EJB容器进行管理。下面的代码展示了通过JNDI定位EJB：

```
InitialContext ic = new InitialContext();

Object objRef = ic.lookup("java:comp/env/ejb/BookDBEJB");

BookDBEJBHome home = (BookDBEJBHome)PortableRemoteObject.narrow(objRef,

database.BookDBEJBHome.class);

bookDBEJB = home.create();
```

当编写Http Servlet时，可以使用许多Apusic应用服务器提供的系统服务：

- 使用JDBC连接池
- 使用JMS

16.1.2.4.8 调用其他WEB资源

Web组件可以通过两种方式调用其他Web资源：间接方式和直接方式。间接方式调用是指在Web组件的内容中包含了指向其他Web资源的URL。

Web组件也能够在它执行的时候调用其他的Web资源。有两种可能性：

- 包含其他Web资源的内容
- 传递请求（request）到其他Web资源

调用方式	描述
间接方式	包含了指向其他Web资源的URL
直接方式	包含其他Web资源的内容传递请求(request)到其他Web资源

为了调用运行Web组件的服务器上的可用资源，必须首先通过getRequestDispatcher("URL")方法获得RequestDispatcher对象。

开发人员可以从request或Web Context获得RequestDispatcher对象，然而这两种方式有细微的区别。

- 从request获得的RequestDispatcher对象，可以接受相对路径作为参数
- 从Web Context获得的 RequestDispatcher对象，需要绝对路径作为参数

一. 包含其他Web资源

包含其他Web资源非常有用，例如从Web组件返回给客户端的响应(response)中可以包含banner或版权（copyright）信息。调用RequestDispatcher对象的相关方法来实现此功能：

- include(request, response)

如果资源是静态的，include方法能够实现服务器端包含。如果资源是Web组件，那么执行include()方法的效果等同于：

1. 向被包含的Web组件发送请求
2. 执行Web组件
3. 把执行的结果返回给包含Web组件的Servlet

被包含的Web组件可以访问request对象，当访问response对象有一些限制：

- 可以向response对象写入内容和提交这个response
- 不能设置头信息或调用任何可能影响头信息的方法

以下BannerServlet用来产生Banner信息：

```
public class BannerServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println(" ... ");
  }

  public void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println(" ... ");
  }
}
```

其他的servlet可以包含BannerServlet：

```
RequestDispatcher dispatcher =

getServletContext().getRequestDispatcher("/banner");

if (dispatcher != null)

dispatcher.include(request, response);

}
```

二. 传递控制给其他Web组件

在一些应用中，也许需要一个Web组件对请求（request）作预处理，然后利用另一个Web组件产生响应（response）。可以调用RequestDispatcher对象的forward()方法实现这个功能。当一个请求（request）被转移，request URL会被设置成新的页面。如果原先的URL还有用，可以把它作为request的属性保存。以下Dispatcher类，从request中获得RequestDispatcher对象，然后把控制权交给template.jsp：

```
public class Dispatcher extends HttpServlet {

public void doGet(HttpServletRequest request,

HttpServletResponse response) {

request.setAttribute("selectedScreen",request.getServletPath());

//从 request 中获得 RequestDispatcher 对象

RequestDispatcher dispatcher =
request.getRequestDispatcher("/template.jsp");

//传递控制到template.jsp
if (dispatcher != null)

dispatcher.forward(request, response);

}

public void doPost(HttpServletRequest request,

...

}
```

forward()方法使另外的资源来答复客户端的请求（request）。如果已经访问了ServletOutputStream或PrintWriter，那么将不能使用include()方法，否则会抛出IllegalStateException异常。

16.1.2.4.9 结束一个SERVLET

当Servlet容器决定结束一个Servlet时（例如，Servlet容器将要关闭，或Servlet容器回收内存资源），它会调用Servlet接口的destroy方法。在destroy方法中，Servlet释放所有使用的资源并保存所有持久性状态。例如下面的destroy()方法释放了初始化Servlet时分配的数据库辅助处理对象：

```
public void destroy() {
  bookDB = null;
}
```

当需要结束一个Servlet时，必须完成此Servlet所有的service()方法。只有在处理完所有的客户请求（request）或超过特定的时间期限，服务器才会调用destroy()方法。

16.1.2.5 Apusic Http Servlets高级开发

16.1.2.5.1 SERVLET FILTERING

过滤器（filter）是Java类，可以改变请求（request）和响应（response）的头信息与内容信息。过滤器不同于其他Web组件的地方是它本身并不创建响应（response），然而它可以依附在任何类型的Web资源上。过滤器截取请求（request），检查和改变request对象、response对象，并可以执行一些其他的任务。过滤器提供的主要功能是：

- 实现日志功能
- 实现用户定义的安全功能
- 调试功能
- 加密
- 数据压缩
- 改变发送给客户端的响应（response）

过滤器截获对特定命名的一个资源和一组资源的请求（request），然后执行过滤器中的代码。对于特定的资源，可以指定按照一定顺序调用的一个和多个过滤器，这就组成了链（chain）。使用过滤器主要包括：

- 编写过滤器类
- 定制请求（request）和响应（response）
- 为特定的Web资源指定过滤器链

一. **编写过滤器类**

编写过滤器的API是javax.servlet包中Filter、FilterChain和FilterConfig接口中定义的一些方法。定义一个过滤器就是实现Filter接口。Filter接口中最主要的方法是doFilter()方法，它接收三个参数：request对象、response对象、filterchain对象。

```
void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
```

这个方法能够执行的动作包括：

- 检查请求（request）的头信息
- 定制request对象，改变请求（request）的头信息或数据

- 定制response对象，改变响应（response）的头信息或数据
- 调用在过滤器链中的下一个实体。如果当前过滤器是链中的最后一个过滤器，那么下一个实体就是客户请求（request）的资源；否则，链中的下一个过滤器会被调用。通过chain对象的doFilter()方法调用下一个实体，并传递request对象和response对象作为参数。另外，也可以不调用doFilter()方法阻塞请求（request），这样，过滤器应该负责填充对客户的响应（response）。
- 检查响应的头信息
- 抛出异常显示处理过程中的错误

除了doFilter()方法，开发人员也必须实现init()和destroy()方法。当容器创建过滤器实例时调用init()方法，

```
void init(FilterConfig filterConfig)
```

可以从FilterConfig对象中获得初始化参数。

在doFilter()方法中，过滤器可以从FilterConfig对象获得ServletContext对象，那么就可以访问存储在ServletContext中的属性对象。当过滤器完成特定的处理过程后，调用chain对象的doFilter()方法。例如

```
public final class HitCounterFilter implements Filter {

private FilterConfig filterConfig = null;

// 初始化

public void init(FilterConfig filterConfig)

throws ServletException {

this.filterConfig = filterConfig;

}

// 结束

public void destroy() {

this.filterConfig = null;

}
```

```
public void doFilter(ServletRequest request,

ServletResponse response, FilterChain chain)

throws IOException, ServletException {

if (filterConfig == null)

return;

StringWriter sw = new StringWriter();

PrintWriter writer = new PrintWriter(sw);

writer.println(" ... ");

.

.

.

writer.flush();

// 输出日志

filterConfig.getServletContext().

log(sw.getBuffer().toString());

...

//调用在过滤器链中的下一个实体

chain.doFilter(request, wrapper);

...

}
```

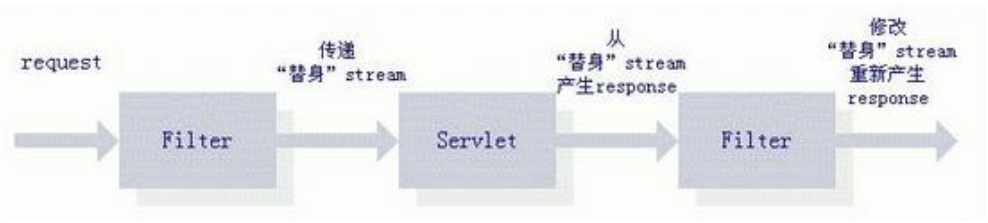
```
}
```

二. 定制请求和响应

有许多方法可以改变请求（request）或者响应（response）。例如过滤器能够给请求（request）增加属性或在响应（response）中插入数据。

过滤器如果需要改变响应（response）必须在响应（response）返回给客户端之前捕获它。要做到这一点需要传递"替身"流（stream）给Servlet，然后利用"替身"stream产生响应（response）。"替身"stream防止了当响应（response）结束后关闭了原始的响应（response）输出流，并且允许过滤器改变Servlet的响应（response）。

过滤器产生"替身"stream



为了给Servlet传递"替身"stream，过滤器需要创建response对象的包装类并且覆盖getWriter或getOutputStream 方法。包装类被FilterChain对象的doFilter方法传递。创建请求（request）的包装类继承ServletRequestWrapper或HttpServletRequestWrapper，创建响应（response）的包装类继承ServletResponseWrapper或HttpServletResponseWrapper。

以下CharResponseWrapper类包装了响应（response）：

```
public class CharResponseWrapper extends HttpServletResponseWrapper {
  private CharArrayWriter output;

  public String toString() {
    return output.toString();
  }

  public CharResponseWrapper(HttpServletResponse response) {
    super(response);
    output = new CharArrayWriter();
  }

  public PrintWriter getWriter() {
    return new PrintWriter(output);
  }
}
```

包装类被传递给BookStoreServlet，BookStoreServlet把响应写入"替身"stream，当chain.doFilter返回，HitCounterFilter重新找回response把它写入缓冲，过滤器插入计数器值到缓冲中然后重新设置response的头信息，最后把缓冲中的内容写入response。

```
PrintWriter out = response.getWriter();
//构造包装类
CharResponseWrapper wrapper = new CharResponseWrapper((HttpServletResponse)
response);
//向 doFilter 传递包装类
chain.doFilter(request, wrapper);
CharArrayWriter caw = new CharArrayWriter();
caw.write(wrapper.toString().substring(0, wrapper.toString().indexOf("
</body>") - 1));
...
caw.write("\\n</body></html>");
//重新设置响应长度
response.setContentLength(caw.toString().length());
out.write(caw.toString());
out.close();
```

三. **映射过滤器**

Web容器使用过滤器映射来决定是否过滤Web资源。在Web应用的部署描述文件中映射过滤器到Servlet或URL模板。

- 在部署描述文件中加入<filter>标记，此标记包括：
  - <filter-name>：过滤器名称
  - <filter-class>：过滤器的实现类
  - <init-params>：过滤器的初始参数

```
<filter>
    <filter-name>Compression Filter</filter-name>
    <filter-class>CompressionFilter</filter-class>
    <init-param>
        <param-name>compressionThreshold</param-name>
        <param-value>10</param-value>
    </init-param>
</filter>
```

- 在部署描述文件中加入<filter-mapping>标记，映射过滤器到Servlet：

```
<filter-mapping>
    <filter-name>Compression Filter</filter-name>
    <servlet-name>CompressionTest</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
<servlet>
    <servlet-name>CompressionTest</servlet-name>
    <servlet-class>CompressionTest</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>CompressionTest</servlet-name>
    <url-pattern>/CompressionTest</url-pattern>
</servlet-mapping>
```

映射过滤器到URL模板：

```
<filter>
    <filter-name>HitCounterFilter</filter-name>
    <filter-class>HitCounterFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>HitCounterFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

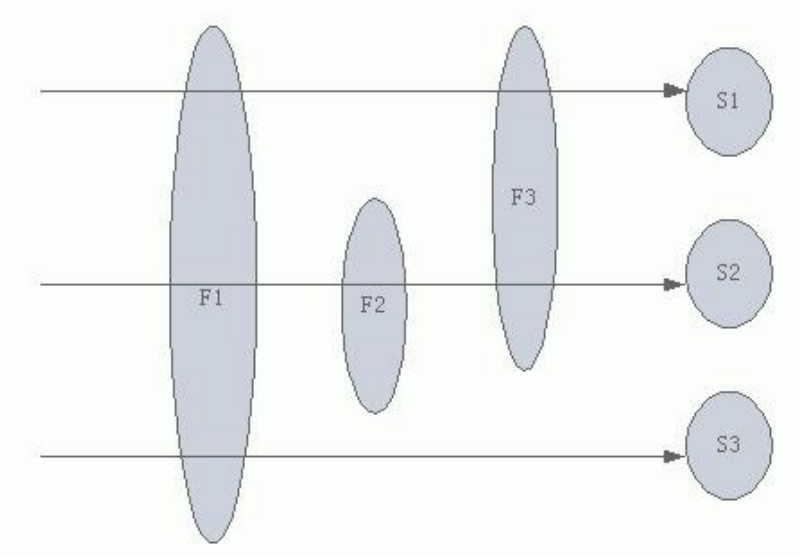
使用<url-pattern>/*<url-pattern>标记，此过滤器将使用于此web应用中的任何静态资源或Servlet内容，因为任何URL都可匹配"/"的URL模式。

- dispatcher元素的作用 ： Servlet 2.4版的Web程序增强了filter和request dispatcher的配合功能，这样过滤器可以根据请求分发器(request dispatcher)所使用的方法有条件地对Web请求进行过滤。
    - 只有当request直接来自客户，过滤器才生效，对应为REQUEST条件。
    - 只有当request被一个请求分发器使用forward()方法转到一个Web构件时，对应称为FORWARD条件。

- 只有当request被一个请求分发器使用include()方法转到一个Web构件时，对应称为INCLUDE条件。
- 只有当request被一个请求分发器使用"错误信息页"机制方法转到一个Web构件时，对应称为ERROR条件。
- 第五种过滤器作用的条件可以是上面四种条件的组合。
- 当不使用dispatcher元素时，客户的直接request会被用来过滤请求。如果请求是从一个request dispatcher转发过来的，这个过滤器不工作。

如下图，可以映射一个过滤器到一个或多个Servlet，或者可以映射一个Servlet到多个过滤器。

过滤器的映射



过滤器F1映射到servlet S1，S2和S3，过滤器F2映射到servlet S2，过滤器F3 映射到servlet S1和S2。

16.1.2.5.2 APPLICATION EVENTS

应用事件模型提供了当ServletContext，HttpSession，ServletRequest状态改变时的通知功能。可以编写事件监听类来响应这些状态的改变，并且可以配置和部署应用事件和监听类到Web应用。

对于ServletContext事件，当Web应用部署、卸载和对context增加属性时，事件监听类可以得到通知。下表列出了ServletContext的事件类型，对应特定事件的监听类必须实现的接口和当事件发生时调用的方法。

事件类型	接口	方法
Servlet context被创建	javax.servlet.ServletContextListener	contextInitialized()
Servlet context被注销	javax.servlet.ServletContextListener	contextDestroyed()
增加属性	javax.servlet. ServletContextAttributesListener	attributeAdded()
删除属性	javax.servlet. ServletContextAttributesListener	attributeRemoved()
属性被替换	javax.servlet. ServletContextAttributesListener	attributeReplaced()

对于HttpSession事件，当session激活、删除或者session属性的增加、删除和替换时，事件监听类得到通知。下表列出了HttpSession的事件类型，对应特定事件的监听类必须实现的接口和当事件发生时调用的方法。

事件类型	接口	方法
session激活	javax.servlet.http. HttpSessionListener	sessionCreated()
session删除	javax.servlet.http. HttpSessionListener	sessionDestroyed()
增加属性	javax.servlet.http. HttpSessionAttributesListener	attributeAdded()
删除属性	javax.servlet.http. HttpSessionAttributesListener	attributeRemoved()
属性被替换	javax.servlet.http. HttpSessionAttributesListener	attributeReplaced()

对于ServletRequest事件，当request初始化、销毁或者request属性的增加、删除和替换时，事件监听类得到通知。下表列出了ServletRequest的事件类型，对应特定事件的监听类必须实现的接口和当事件发生时调用的方法。

事件类型	接口	方法
session初始化	javax.servlet.ServletRequestListener	requestInitialized()
session销毁	javax.servlet.ServletRequestListener	requestDestroyed()
增加属性	javax.servlet.ServletRequestAttributeListener	attributeAdded()
删除属性	javax.servlet.ServletRequestAttributeListener	attributeRemoved()
属性被替换	javax.servlet.ServletRequestAttributeListener	attributeReplaced()

一. 配置事件监听类

配置事件监听类的步骤：

1. 打开Web应用的部署描述文件web.xml
2. 增加事件声明标记<listener>。事件声明定义的事件监听类在事件发生时被调用。<listener>标记必须在<filter>和<filter-mapping>标记之后和<servlet>标记之前。可以为每种事件定义多个事件监听类，Apusic应用服务器按照它们在部署描述文件声明的顺序调用。例如：

```
<listener>
    <listener-class>myApp.myContextListenerClass</listener-class>
</listener>
<listener>
    <listener-class>myApp.mySessionAttributeListenerClass</listener-class>
</listener>
```

3. 编写和部署监听类。

二. 编写事件监听类

编写事件监听类的步骤：

1. 创建新的类并实现事件对应的接口
2. 定义不接受参数、访问属性为public的构造函数
3. 实现接口的方法
4. 编译并拷贝到对应Web应用的WEB-INF/classes目录下，或者打包成jar文件拷贝到WEB-INF/lib目录下

三. 事件监听类模板

ServletContext 监听类例子：

```
import javax.servlet.*;

public final class myContextListenerClass implements ServletContextListener {
  public void contextInitialized(ServletContextEvent event) {
    /*
     *
     * 当 ServletContext 初始化时被调用，可以在这儿
     *
     * 初始化 ServletContext 的相关数据
     *
     */
  }

  public void contextDestroyed(ServletContextEvent event) {
    /*
     *
     * 当 Web 应用被卸载或 Apusic 服务器关闭时被调用
     *
     */
  }
}
```

HttpSession 属性监听类例子：

```
import javax.servlet.*;

public final class mySessionAttributeListenerClass implements
```

```
HttpSessionAttributesListener {
  public void attributeAdded(HttpSessionBindingEvent sbe) {
    /*
     *
     * 增加session属性时被调用
     *
     */
  }

  public void attributeRemoved(HttpSessionBindingEvent sbe) {
    /*
     *
     * 删除session属性时被调用
     *
     */
  }

  public void attributeReplaced(HttpSessionBindingEvent sbe) {
    /*
     *
     * 替换session属性时被调用
     *
     */
  }
}
```

ServletRequest 属性监听类例子：

```
import javax.servlet.*;

public final class myRequestAttributeListenerClass implements
ServletRequestAttributeListener {
  public void attributeAdded(HttpSessionBindingEvent sbe) {
    /*
     *
     * 增加request属性时被调用
     *
     */
  }
```

```
    public void attributeRemoved(HttpSessionBindingEvent sbe) {
        /*
         *
         * 删除request属性时被调用
         *
         */
    }

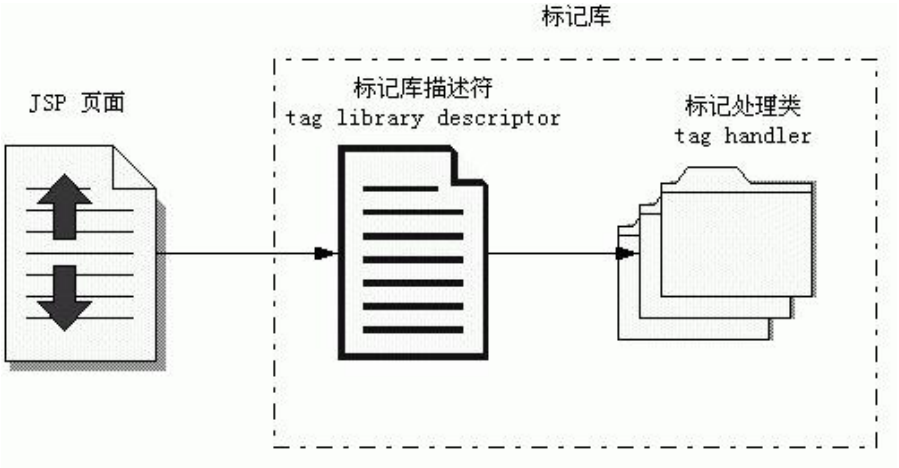
    public void attributeReplaced(HttpSessionBindingEvent sbe) {
        /*
         *
         * 替换request属性时被调用
         *
         */
    }
}
```

16.1.3 JSP扩展标记

16.1.3.1 JSP 扩展标记介绍

标准的JSP标记可以调用JavaBeans组件和执行请求（request）的分配，简化JSP页面的开发和维护。JSP技术同样支持封装其他类型的动态功能到客户定制的扩展标记中。扩展标记通常以标记库（tag library）的形式发布，包含一系列相关的标记和这些标记的实现对象。

标记库体系结构

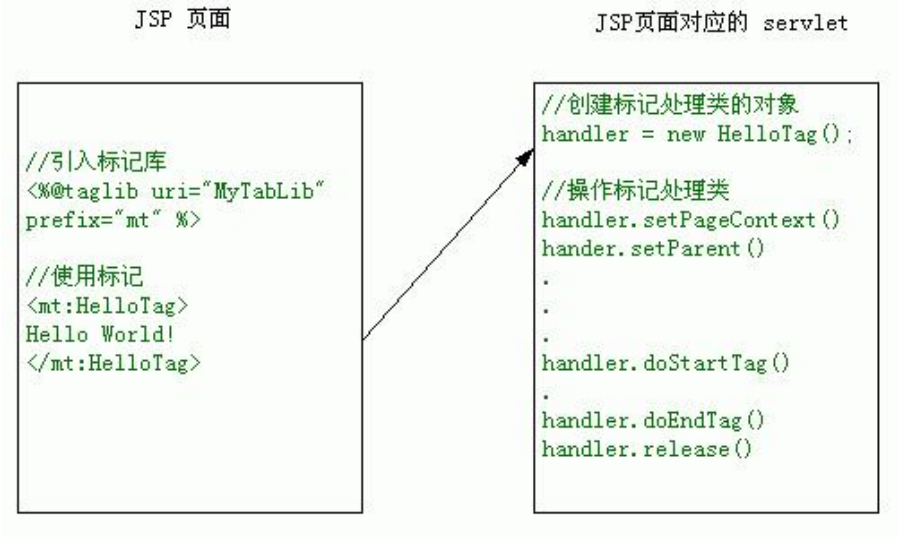


JSP扩展标记能够完成很多任务，例如操作内置对象、处理表单、访问数据库、访问其他企业服务和实现流程控制。JSP扩展标记由精通Java语言和数据库的开发人员创建，供Web应用的设计者使用，他们只关心页面的表现形式。

扩展标记是用户定义的JSP元素。当包含扩展标记的JSP页面转化成servlet时，标记被转换成对标记处理类（tag handler）的操作。Web 容器在执行JSP页面对应的servlet时会调用标记处理类（tag handler）的方法。扩展标记有很多特性：

- 可以通过传递属性定制
- 访问JSP页面中的可用对象
- 改变JSP页面的响应（response）
- 可以互相通讯
- 可以嵌套，实现复杂的交互

扩展标记实现原理



JSP2.0定义一种新的Tag 处理方式，直接handler.serJspContext()，然后直接执行doTag()方法。

16.1.3.2 使用扩展标记

这一节介绍了如何在JSP页面中使用扩展标记和扩展标记的类型。

16.1.3.2.1 声明扩展标记

在使用标记库中的标记之前，开发人员需要在JSP页面中使用taglib伪指令声明标记库：

```
<%@ taglib uri="myTagLib" prefix="mt" %>
```

uri 属性指向唯一标识标记库描述符的URI。JSP引擎通过匹配uri属性与Web引用部署描述(web.xml)中定义的<taglib-uri>元素来尝试查找标记库描述符。例如，上述taglib指令中的myTaglib将引用在如下Web应用部署描述中的标记库描述符：

```
<taglib>
    <taglib-uri>myTagLib</taglib-uri>
    <taglib-location>/WEB-INF/myTagLib.tld</taglib-location>
</taglib>
```

prefix属性定义了标记库的前缀，用来和其他标记库进行区分，当使用JSP扩展标记编写页面时使用此前缀引用标记库中的标记。例如，如上例中名为myTagLib的库定义了一个名为helloTag的标记，可在JSP页面中这样引用：

```
<mt:helloTag>
```

16.1.3.2.2 扩展标记的类型

JSP扩展标记使用XML语法，具有开始标记和结束标记，可能还包含标记体：

```
<tt:tag>

body
</tt:tag>
```

没有标记体的标记可以表示为：

```
<tt:tag />
```

简单标记

简单标记不包含标记体和属性：

```
<tt:simple />
```

带属性的标记

扩展标记可能带有属性。属性被列在开始标记中，语法形式为：attr="value"。属性值用来定制扩展标记的行为，就象参数定义方法的行为一样。开发人员可以在标记库描述符中指定标记属性的类型。属性值来自字符串（String）常量和表达式，由于这两种类型都是字符串（String），那么它们转换为正确属性类型的规则遵循在"设置JavaBeans的属性"中的描述。

带有标记体的标记

扩展标记在开始标记和结束标记之间可以包含扩展和标准标记、脚本元素、HTML代码等标记体。

定义脚本变量的标记

扩展标记可以定义在JSP页面中使用的脚本变量。下面的代码片断示例了如何定义和使用脚本变量，脚本变量包含了从JNDI返回的对象：

```
<mt:lookup id="tx" type="UserTransaction"

name="java:comp/UserTransaction" />
<% tx.begin(); %>
```

协作标记

扩展标记可以通过共享对象互相协作。下面的例子tag1创建了对象obj1，然后在tag2中使用。

```
<tt:tag1 attr1="obj1" value1="value" />
<tt:tag2 attr1="obj1" />
```

下一个例子，对象由一组嵌套标记中的封装标记创建，在所有内部标记中也可用。由于对象是没有命名的，可以减少潜在的命名冲突。

```
<tt:outerTag>
    <tt:innerTag />
</tt:outerTag>
```

16.1.3.3 定义扩展标记

定义扩展标记需要：

- 开发标记处理类（tag handler ）和辅助类
- 在标记库描述符（ tag library descriptor ）中声明标记

这一节描述了标记处理类和标记库描述符的属性，说明如何开发标记处理类和标记库描述符。

16.1.3.3.1 标记库描述符

标记库描述符（tag library descriptor，TLD）是描述标记库的XML文档，包含了关于标记库的整体信息和其中每一个标记的信息。Web容器使用标记库描述符校验标记。标记库描述符必须以.tld为文件扩展名，存放在WEB-INF目录或其子目录中。

标记库描述符必须以XML文档的序言开始，定义XML和DTD的版本信息。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag
Library 1.2//EN"

"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

也可在根元素指定(JSP2.0，jsptaglibrary 2.0起支持)

```
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-
jsptaglibrary_2_0.xsd" version="2.0"
```

标记库描述符的根元素是taglib ， taglib的子元素为：

元素	描述
tlib-version	标记库的版本
jsp-version	JSP 规范的版本
short-name	可选
uri	唯一标识标记库的 URI
display-name	可选
small-icon	可选，小图标
large-icon	可选，大图标
description	可选，描述信息
listener	应用事件监听类
tag	标记

标记库描述符中的每一个tag元素描述了标记的名称、处理类、标记创建的脚本变量和标记的属性。tag元素的子元素为：

元素	描述
name	标记名
tag-class	标记处理类
tei-class	可选，javax.servlet.jsp.tagext.TagExtraInfo 子类
body-content	标记体类型
display-name	可选
small-icon	可选，小图标
large-icon	可选，大图标
description	可选，描述信息
variable	可选，脚本变量信息
attribute	标记属性信息

例如，下面定义了helloTag标记，包含parameter属性。

```
<tag>
    <name>helloTag</name>
    <tag-class>com.apusic.test.MyTagHandler</tag-class>
```

```
    <body-content>JSP</body-content>

...
<attribute>
        <name>parameter</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

...
</tag>
```

16.1.3.3.2 标记处理类（TAG HANDLERS ）

标记处理类（Tag Handlers ）是在处理包含扩展标记的JSP页面时由Web容器调用。标记处理类必须实现Tag或BodyTag接口，开发人员可以使用TagSupport和BodyTagSupport作为基类，这些类和接口都包含在javax.servlet.jsp.tagext包中。

标记处理类的方法定义在Tag和BodyTag接口中，被JSP页面的实现servlet在不同的地方调用。当遇到开始标记时，JSP页面的实现servlet调用标记处理类适当的方法初始化，然后调用doStartTag方法。当遇到结束标记时，标记处理类的doEndTag方法被调用。如果标记处理类需要和标记体交互，那么附加的方法将在开始和结束标记之间被调用。请参考"标记处理类如何被调用"。为了提供标记处理类的实现，开发人员必须针对不同的情况实现接口的方法，下面总结了标记处理类的实现方法。

标记类型	实现方法
简单标记	doStartTag, doEndTag, release
带属性的标记	doStartTag, doEndTag, set/getAttribute1...N, release
带标记体，需要执行，但不交互	doStartTag, doEndTag, release
带标记体，需要反复执行	doStartTag, doAfterBody, doEndTag, release
带标记体，需要交互	doStartTag, doEndTag, release, doInitBody, doAfterBody, release

标记处理类能够访问与JSP页面通讯的API。访问这些API的入口点是page context对象（javax.servlet.jsp.PageContext），通过PageContext标记处理类能够访问其他JSP页面可用的内置对象：request、session和application。

如果标记是嵌套的，那么内部标记的处理类可以访问封装标记的处理类。

16.1.3.3.3 简单标记

简单标记的处理类必须实现Tag接口的doStartTag和doEndTag方法。当遇到开始标记时，doStartTag方法被调用。因为简单标记没有标记体，所以doStartTag方法返回SKIP_BODY。当遇到结束标记时，doEndTag方法被调用。如果需要执行页面后面的内容，doEndTag返回 EVAL_PAGE，否则返回SKIP_PAGE。例如一个简单标记

```
<tt:simple />
```

该标记的处理类为：

```
public SimpleTag extends TagSupport {

//遇到开始标记时调用

public int doStartTag() throws JspException {

try {

pageContext.getOut().print("Hello.");

} catch (Exception ex) {

throw new JspTagException("SimpleTag: " +

ex.getMessage());

}

return SKIP_BODY;

}

//遇到结束标记时调用

public int doEndTag() {

return EVAL_PAGE;

}

}
```

简单标记不包含标记体，在标记库描述符中body-content必须声明为empty：

```
<body-content>empty</body-content>
```

16.1.3.3.4 带属性的标记

对标记的每一个属性，开发人员必须遵循JavaBeans体系结构的设计惯例定义get和set方法。例如标记present，

```
<logic:present parameter="Clear">
```

包含属性parameter， 必须定义get和set方法，

```
protected String parameter = null;

public String getParameter() {
  return (this.parameter);
}

public void setParameter(String parameter) {
  this.parameter = parameter;
}
```

	**注意**
	如果属性命名为id，且标记处理类从TagSupport类继承，那么不需要定义get和set方法。TagSupport类中已经定义了getId和setId。

对标记的每一个属性，开发人员必须确定属性是否是必须的，属性值是否可以通过动态的表达式的值得到。通过attribute元素指定属性的类型。对于静态的属性值，属性的类型总是java.lang.String。如果rtexprvalue元素是true或yes，属性类型可以定义为期待表达式返回的类型。

```
<attribute>
    <name>attr1</name>
    <required>true|false|yes|no</required>
    <rtexprvalue>true|false|yes|no</rtexprvalue>
    <type>fully_qualified_type</type>
</attribute>
```

如果标记属性不是必须的，标记的处理类应该提供缺省值。

传递给标记的属性可以在JSP页面转换的时候被TagExtraInfo类的isValid方法验证。isValid方法的参数为TagData对象，包含了每个标记的属性 - 值信息。由于验证发生在JSP页面的转换阶段，需要在请求（request）阶段计算的属性值将被设置为TagData.REQUEST_TIME_VALUE。

标记 <tt:twa attr1="value1"/> 在标记库包含下面的属性声明：

```
<attribute>
    <name>attr1</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
</attribute>
```

这段声明指出了属性attr1的值可以在运行时刻决定。下面的isValid()方法检查了属性attr1的值是否为Boolean类型。由于属性attr1的值可以在运行时刻决定，isValid()必须检查标记的使用者是否提供了运行时刻的值。

```
public class TwaTEI extends TagExtraInfo {
  // 检验属性值
  public boolean isValid(Tagdata data) {
    Object o = data.getAttribute("attr1");
    if (o != null && o != TagData.REQUEST_TIME_VALUE) {
      if (o.toLowerCase().equals("true") || o.toLowerCase().equals("false"))
        return true;
      else
        return false;
    } else
      return true;
  }
}
```

16.1.3.3.5 带标记体的标记

处理带有标记体的标记处理类有两种不同的实现方式，实现方式的选择取决于是否需要和标记体交互。需要交互意味着标记处理类可以读取和更改标记体的内容。

标记处理类不与标记体交互

如果不需要和标记体交互，标记处理类应该实现Tag接口或继承TagSupport类。doStartTag返回EVAL_BODY_INCLUDE，代表标记体需要执行，否则返回SKIP_BODY。

如果标记处理类需要反复的执行标记体，应该实现IterationTag接口或继承TagSupport类。doStartTag和doAfterBody方法返回EVAL_BODY_AGAIN，确定标记体需要再次执行。

标记处理类和标记体交互

如果需要和标记体交互，标记处理类必须实现BodyTag接口或继承BodyTagSupport类。标记处理类一般通过实现doInitBody和doAfterBody方法来与JSP页面的实现servlet传递过来的标记体内容交互。

BodyContent支持几种读取和写入的方法。标记处理类使用BodyContent的getString或getReader方法读取标记体的内容信息，使用writeOut(out)方法向输出流写入标记体内容。

如果标记体需要执行，doStartTag方法返回EVAL_BODY_BUFFERED，否则返回SKIP_BODY。

doInitBody方法在BodyContent设置后、标记体执行之前被调用。通常用来完成依赖于标记体内容的初始化操作。

doAfterBody方法在标记体执行后被调用。象doStartTag一样，doAfterBody必须返回指示，确定是否需要继续执行标记体。如果需要再次执行标记体，返回EVAL_BODY_BUFFERED，否则doAfterBody返回SKIP_BODY。

标记处理类在release方法中重新设置状态，释放所有资源。

类型	概览
处理类不与标记体交互	• 实现Tag或继承TagSupport • doStartTag返回EVAL_BODY_INCLUDE或SKIP_BODY • doAfterBody返回 EVAL_BODY_AGAIN或SKIP_BODY
处理类和标记体交互	• 实现 BodyTag或继承BodyTagSupport • doInitBodydoAfterBody返回EVAL_BODY_BUFFERED或SKIP_BODY • release

下面的例子读取了标记体的内容，然后传递给对象作为SQL语句执行。因为标记体不需要再次执行，doAfterBody返回SKIP_BODY。

```
public class QueryTag extends BodyTagSupport {
  public int doAfterBody() throws JspTagException {
    BodyContent bc = getBodyContent();
    // 得到标记体内容
    String query = bc.getString();
    // 清除
    bc.clearBody();
    try {
      // 作为 SQL 语句执行
      Statement stmt = connection.createStatement();
      result = stmt.executeQuery(query);
    } catch (SQLException e) {
      throw new JspTagException("QueryTag: " + e.getMessage());
    }
    return SKIP_BODY;
  }
}
```

对于包含标记体的标记，开发人员必须在body-content指定标记体的类型。

```
<body-content>JSP|tagdependent</body-content>
```

包含扩展和标准标记、脚本元素、HTML文本的标记体归类为JSP，所有的其他类型归类为tagdependent 。

16.1.3.3.6 定义脚本变量的标记

标记处理类能够创建和设置对象，然后通过页面运行环境（context）关联到脚本变量。使用pageContext.setAttribute(name,value,scope)或pageContext.setAttribute(name,value)来完成这种关联。一般通过扩展标记的属性传递脚本变量的名称，然后可以调用属性的get方法得到这个名称。通常的处理过程是标记处理类得到脚本变量，执行一些处理，然后使用pageContext.setAttribute(name,object)设置脚本变量值。

JSP页面包含了能够定义脚本变量的标记，在它转换成servlet的阶段，Web容器产生定义脚本变量、设置脚本变量到对象引用的代码。为了产生这些代码，Web容器需要确定脚本变量的信息：

- 变量名
- 变量类型
- 变量引用新的对象或已存在的对象
- 变量的可用范围

有两种方式可以提供这些信息：

- 在标记库描述符中定义variable元素
- 定义标记扩展信息类（TagExtraInfo ）并在标记库描述符中定义tei-class元素

variable元素使用简单，但缺少一些弹性。variable元素包含下面的子元素：

- name-given：给定常量作为变量名
- name-from-attribute：属性的名称，属性值作为变量名

name-given和name-from-attribute之中一个是必须的。其他子元素是可选的：

- variable-class：变量的类型，缺省是java.lang.String
- declare：变量是否关联到新的对象，缺省是true
- scope：变量的作用范围，缺省是NESTED

下面列出所有的变量作用范围。

值	可用性	方法
NESTED	开始和结束标记之间	doInitBody 、doAfterBody 、doStartTag
AT_BEGIN	从开始标记到页面结束	doInitBody、doAfterBody、doEndTag 、doStartTag
AT_END	从结束标记到页面结束	doEndTag

开发人员可以继承javax.servlet.jsp.TagExtraInfo定义标记的扩展信息类。TagExtraInfo必须实现getVariableInfo方法，返回VariableInfo对象数组，包含了确定脚本变量的所有信息。例如下面的代码：

```
public class DefineTei extends TagExtraInfo {
  public VariableInfo[] getVariableInfo(TagData data) {
    String type = data.getAttributeString("type");
    if (type == null) type = "java.lang.Object";
    return new VariableInfo[] {new
VariableInfo(data.getAttributeString("id"), type, true,
VariableInfo.AT_BEGIN)};
  }
}
```

VariableInfo对象构造函数的参数为：脚本变量名称、类型、是否为新的变量和变量的作用范围。另外，需要在标记库描述符中定义tei-class元素指向扩展信息类：

```
<tei-class>com.apusic.test.DefineTagTei</tei-class>
```

16.1.3.3.7 协作标记

标记可以通过共享对象进行协作。JSP技术支持两种形式的对象共享。

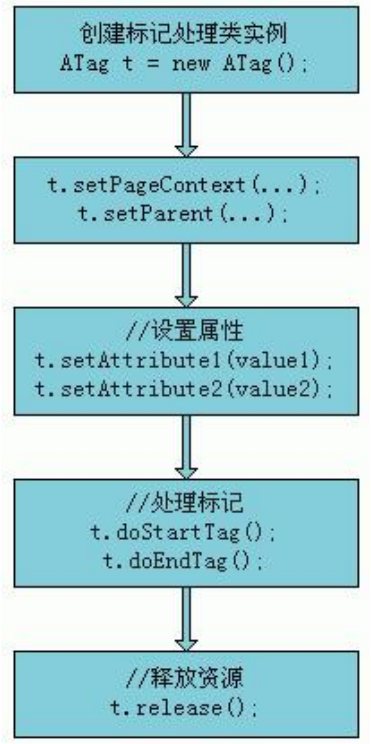
- 使用page context命名和保存对象。page context在JSP页面和标记处理类中都是内置对象。
- 对象由一组嵌套标记中的封装标记创建，在所有内部标记中也可用。这种方式的好处是使用了私有的对象命名空间，可以减少潜在的命名冲突。

访问由封装标记创建的对象，标记处理类必须首先通过静态方法TagSupport.findAncestorWithClass(from,class)或TagSupport.getParent方法获得封装标记。

16.1.3.4 标记处理类如何被调用

Tag接口定义了标记处理类和JSP页面的实现servlet之间的基础协议。JSP页面的实现servlet会按照一定的顺序去调用标记处理类的方法，典型的调用顺序为：

调用tag handler方法的典型顺序

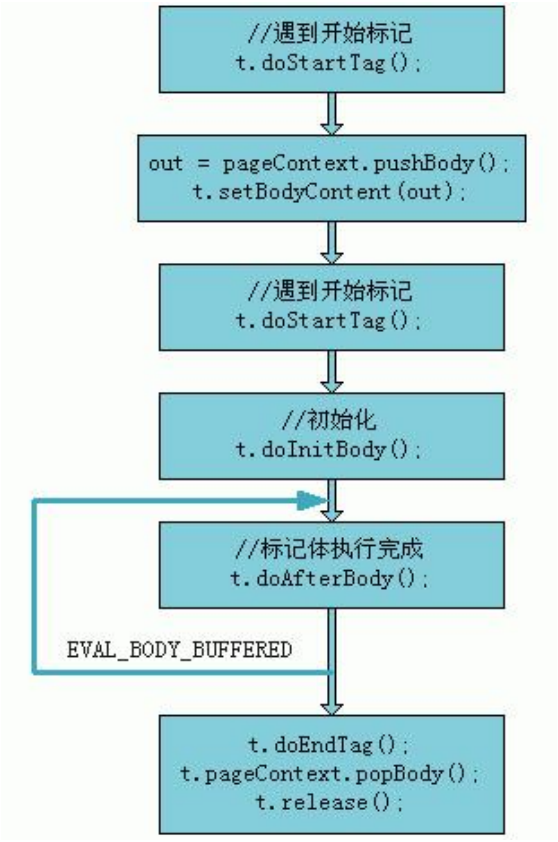


BodyTag接口扩展了Tag，定义了处理标记体的附加方法。

- setBodyContent：创建BodyContent，加入tag handler
- doInitBody：执行标记体之前调用
- doAfterBody：执行标记体之后调用

JSP页面的实现servlet调用标记处理类的方法的典型顺序为：

调用 tag handler 方法的典型顺序



16.1.4 Apusic JSF

16.1.4.1 什么是JavaServer Faces?

基本上来说JSF应用就像普通的web应用一样运行在Servlet容器中。一个典型的JSF应用通常包括：

Java Bean组件，完成业务于逻辑事件监听器页面服务器端帮助类，比如DAO类

显示UI组件的taglib 代表事件处理器（event handler），校验器（validator）和其他动作的taglib代表服务器上状态的UI组件Backing beans定义UI组件的属性和函数校验器（validator），转换器（converter），事件监听器，事件处理器配置的资源文件JSF的页面用代表UI组件的自定义标签来显示HTML，一般还可能有其他核心动作的自定义标签如校验器，事件处理器。这些标签由JSF提供

角色与分工

- 页面开发：通常是美工，用HTML脚本等，在JSF中用JSF的自定义标签
- 应用开发：写应用逻辑，事件处理，对象转换，校验器等
- 组件开发：写自定义的UI组件，有UI经验，或扩展JSF的标准组件
- 应用架构师：设计web应用，确保应用伸缩性，定义页面流转，配置beans，注册应用的对象
- 工具提供商：提供开发工具

16.1.4.2 JSF页面的生命周期

JSF页面的生命周期与JSP页面的生命周期相类似：客户相页面发出HTTP请求，服务器返回翻译成HTML的响应，但JSF提供了更多处理页面的服务。生命周期与页面编写者无关，这个概念主要面对开发人员。

16.1.4.2.1 重建视图阶段（RESTORE VIEW PHASE)

当请求JSF页面时，如点击按钮或链接，JSF开始重建视图阶段。

在这个阶段JSF建立页面的视图，给视图中的组件设置事件处理器、校验器，在FacesContext中保存视图。FacesContext含有所有处理请求的信息，所以页面元素包括组件标签、事件处理器、转换器、校验器都要接触FacesContext。如果请求是第一次的请求，JSF在这个阶段产生一个空的视图，生命周期进入显示应答阶段，这个空的视图会在页面返回的时候用到。如果请求是返回的请求，对应于这个页面的视图已经存在，JSF用存在客户端或服务器端的信息重建视图。

16.1.4.2.2 应用视图值阶段（APPLY REQUEST VALUES PHASE)

在组件树重建后，每一个树上的组件用decode方法从请求中解出其新的值，这个值保存在组件中。如果值数据转换失败，产生与此组件相联系的错误，并入FacesContext的上下文，错误信息在其后的显示应答阶段显示。

如果任何decode方法或事件监听器调用了当前FacesContext的renderResponse方法，则JSF直接跳到显示应答阶段。如果在这个阶段有事件产生，JSF广播事件到感兴趣的监听器。

如果此时应用转到另一个web应用或应答不含有JSF组件，则调用FacesContext.renderComplete方法。在此阶段结束时，所有组件已得到新值，错误信息和事件已入队列。

16.1.4.2.3 处理校验阶段（PROCESS VALIDATIONS PHASE)

此阶段，JSF处理所有组件树上注册的校验器，检查设置了校验的组件属性，如果值不合法JSF在上下文（FacesContext）中加入错误信息，生命周期直接进入显示应答阶段，显示错误信息，如果有转换错误也会显示。

如果任何validate方法或事件监听器调用上下文的renderResponse方法，JSF直接跳到显示应答阶段。

16.1.4.2.4 更新模型值阶段（UPDATE MODEL VALUES PHASE)

在JSF确定数据合法之后，遍历组件树，从组件中取得相应值设置到服务器对象上。

如果任何updateModels或监听器调用renderResponse方法，JSF直接跳到显示应答阶段。

16.1.4.2.5 调用应用阶段（INVOKE APPLICATION PHASE)

此阶段，JSF除了应用级别事件，如：表格提交或到其它页面的链接等；重建视图时产生的事件广播到感兴趣的监听器上，JSF计算应答到新的页面。

16.1.4.2.6 显示应答阶段（RENDER RESPONSE PHASE)

此阶段，如果应用是JSP页面，JSF将控制转到JSP容器。

如果是第一次请求，执行JSP页面是会把页面上显示的组件加到组件树中。当JSP容器遍历页面的标签时组件会将自己显示出来。如果是返回的请求且在其它阶段产生了错误，则显示原始页面并显示错误信息。

16.1.4.3 UI组件模型

JSF的UI组件是可配置可重用的，组件可以是最简单的如一个按钮，也可以组合如包括若干组件的表格

JSF提供：

一套确定UI组件状态和行为的UIComponent类

- 定义了怎样显示组件的显示模型
- 定义了怎样处理组件事件的事件与监听模型

- 定义了怎样注册数据转换器到组件的数据转换模型
- 定义了怎样注册校验器到组件的校验模型

16.1.4.3.1 UI组件类

SF提供一套UI组件类和与其相关联的行为接口，如：保持组件状态，保持对象引用，驱动事件处理，显示标准组件等。

所有JSF UI组件扩展UIComponentBase类，包含以下组件类：

- UIComponent：表示UIData的一个数据列
- UICommmand：激活产生行为
- UIData：一个由DataModel实例表示数据集合的绑定
- UIForm：提交到应用的一组数据，HTML里就是form标记
- UIGraphic：显示一个图片
- UIInput：接受用户数据输入，UIOutput的子类
- UIMessage：显示本地化信息
- UIMessages：显示本地化信息集合
- UIOutput：显示数据输出
- UIPanel：管理子组件的摆放
- UISelectBoolean：用户选择或不选择来获得一个布尔值，UIInput的子类
- UISelectItem：选择单一项
- UISelectItems：选择多项
- UISelectMany：允许用户选择多项，UIInput子类
- UISelectOne：允许用户选择单项，UIInput子类
- UIViewRoot：组件树的根

除了扩展UIComponentBase类，组件还可以实现一个或多个行为接口，这些接口定义了特定的行为：

- ActionSouce：表示组件可以发生行为事件
- EditableValueHolder：扩展ValueHolder类，定义可编辑组件的额外特性如校验、发出值变化事件
- NamingContainer：强制以此组件为根的组件必须有唯一ID
- StateHolder：表示组件在请求之间保存状态
- ValueHolder：组件保存局部状态

- UICommand 实现ActionSouce和StateHolder。UIOutput实现StateHolder和ValueHolder；UIInput实现EditableValueHolder，StateHolder，ValueHolder；UIComponentBase实现StateHolder。可以参考JSF的JavaDoc（http://java.sun.com/j2ee/javaserverfaces/1.1 /docs/api/index.html），当然这些类和接口是面向组件编写者的，页面开发和应用开发者只需要使用组件就可以了。

16.1.4.3.2 组件显示模型

JSF 的组件架构设计成组件的功能由组件类决定，而它显示出来是什么样子由另外的显示器（renderer）决定。这样的设计包含了这样的考虑：组件编写者写一次组件的行为却可以定义多个显示器，由显示器决定呈现给客户组件的样子，可以是同一个客户或不同的客户；页面编写者或应用开发者可以通过更换不同组件与显示器结合的tag来改变组件在页面上的样子。

显示包（render kit）定义了对特定客户的组件与标记之间的映射。我们对JSF的支持包含了HTML客户的标准显示包。显示包对其支持的每一种UI组件都定义了一套显示器类，每一个Renderer类不同的显示组件的方式，比如：UISelectOne有三个不同的显示器，或者显示成圆按钮，或者显示成多选框，或者显示成下拉框。

标准显示包里的每一个自定义标签由组件功能和显示属性组成。例如，UICommand组件就可以显示成按钮或一个链接，功能都是发生一个动作。

Apusic服务器中提供HTML显示的自定义标签，如下表：

标签	功能	显示	样子
column	UIData组件的一列数据	HTML表的一列	表格的一列
commandButton	提交表格	HTML<input type=type>元素，type值为submit，image，reset	按钮
commandLink	链接到其他页或页面的某一位置	HTML<a href>元素	链接
dataTable	数据包装	HTML<table>元素	动态更新的表
form	输入表，表内的标签接收的数据和表一起提交	HTML<form>元素	无
graphicImage	显示图片	HTML<image>元素	图片
inputHidden	页面中包含隐藏变量	HTML<input type=hidden>元素	无
inputSecret	加密输入	HTML<intput type=password>元素	密码输入
inputText	输入字符串	HTML<input type=text>元素	字符串输入
inputTextarea	多行字符串	HTML<textarea>元素	多行字符串输入
message	显示本地化信息	如果用了style是HTML的元素	文本字符串
messages	显示多个本地化信息	如果用了style，是HTML的多个	同上

outputLabel	输入项的标记	HTML的<label>元素	文本
outputLink	链接到其它页或其它位置但每有动作事件	HTML<a>元素	链接
outputFormat	显示本地化信息	文本	文本
outputText	显示一行文字	文本	文本
panelGrid	显示表	HTML包含<td>和<tr>的<table>元素	表
panelGroup	同一父组件下的一套组件组	无	表的一行
selectBooleanCheckbox	用户可以改变布尔选择的值	HTML<input type=checkbox>元素	选择框
selectItem	UISelectOne组件的一项	HTML<option>元素	无
selectItems	UISelectOne组件选项列表	HTML<option>元素的列表	无
slectManyCheckbox	复选框	HTML<select>元素	列表框
selectManyListbox	复选列表	HTML<select>元素	列表框
selectManyMenu	多个项目中可多选	同上	可滚动多选框
selectOneListbox	所有选择中选一个	同上	列表框
selectOneMenu	多个项目中选一	同上	可滚动选择框
selectRadio	选一项	HTML<input type=radio>元素	圆按钮

16.1.4.3.3 转换模型

JSF 应用可以将组件联系在服务器对象数据上，如Java Bean称为backing bean，应用存取组件的数据通过调用组件的对象属性。当组件绑定对象，应用有两个组件数据的视图：模型视图，数据是由数据类型表示的，如int或long；表示视图，这时数据可由客户读写，如：java.util.Date可以表示为mm/dd/yy格式的字符串。Apusic服务器JSF实现自动转换转换这两个视图间的数据，但要求与组件相联系的bean属性是组件数据支持的类型，例如：与UISelectBoolean组件相联系的bean属性是java.lang.Boolean，JSF自动将组件数据从String转换为Boolean，当然组件数据必须与合适的数据类型相绑定，即UISelectBoolean必须与类型boolean或java.lang.Boolean相绑定。

如果是非标准数据，你可以在UIOutput或其子类组件上注册转换器（Conveter），到时转换器将完成两个视图间数据的转换。

转换器可以自行编写，有三个步骤：完成Conveter接口、注册转换器、页面编写者在需要转换数据的组件标签上应用这个转换器。

16.1.4.3.4 事件和监听器模型

JSF的事件监听模型与JavaBean的事件机制类似，但要求强事件类型且监听器接口由UI组件生成。

Event对象确定产生事件和存储事件信息的组件。如果要处理事件提示，应用要完成监听器接口，并注册在产生事件的组件上。当用户激活某个组件，如单击某个按钮，一个事件产生，这时JSF调用监听器方法处理事件。

JSF支持三类事件：值改变、动作事件、数据模型事件。

动作事件发生在用户激活了完成ActionSource接口的组件，包括按钮和链接。

值改变事件发生在用户改变了UIInput及其子类组件所代表的值，例如用户选择了一个单选框，则产生这个组件值变成true的事件，这种类型的事件可以由如UIInput、UISelectOne、UISelectMany、UISelectBoolean等组件产生，不过这种事件的产生是以没有校验错产生为前提的。由这种组件的immediate属性来决定动作事件的处理是在应用调用阶段（invoke application phase）或应用请求值阶段（apply request phase）1；值变化事件是在校验阶段（validation phase）或应用请求值阶段。

数据模型事件UIData组件的一行被选中的情况下，具体见高级功能。

有两个方法使你的应用响应动作事件或值变化事件：完成处理事件的监听器类，并在组件标签内使用valueChangeListener标签或actionListener标签以注册监听器到这个组件；完成backing bean的方法来处理事件，并从标签的合适属性上使用方法绑定表达式（method - binding expression）来引用它。

如果你自己编写组件产生事件，除了上述步骤还要编写事件类，然后手动将事件入组件的事件队列。

16.1.4.3.5 校验模型

JSF 支持校验可编辑组件的本地数据，如文本输入组件，校验发生在本地值更新之前。JSF定义了一系列标准校验器做常规数据校验，在核心标签里定义了一套完成Validator接口的标签，大多数标签都有多个属性用来配置校验器，如允许组件数据的最大最小值。页面编写者使用时在组件标签内部使用校验器标签即注册了这个校验器。

你也可以自己编写校验器来达到完成自定义的校验，有两种方式：完成Validator接口或完成backing bean的方法，如果是完成Validator接口就必须注册校验器，然后再写一个自定义标签；如果是完成backing bean的方法就需要从组件标签的validator属性引用此校验器

16.1.4.4 导航（Navigation）模型

基本上来说所有web应用都是一系列的页面组成，这就有一个基本任务是如何管理这些页面的导航，JSF的导航模型管理页面的流转非常容易。JSF定义导航 （navigation）是指当点击一个按钮或链接时系统选择下一个要显示的页面的规则，这些规则定义在JSF的配置文件中。

简单的导航规则先在配置文件中定义规则，然后在按钮或链接的action属性中指定得出的字符串结果，这个字符串就是系统选定的导航规则。复杂的应用则需要开发人员写动作（action）方法，在一定的计算后得到下一个要显示的页面，而触发的组件应用此方法。

当点击一个按钮或链接时，与此相关联的组件产生动作事件，动作事件由缺省动作监听器（ActionListener）处理，触发组件的动作方法；动作方法由应用开发人员在backing bean中定义，完成一定的计算输出逻辑结果的字符串代表处理结果；监听器输出处理结果并传给缺省导航处理器（Navigation Handler），导航处理器通过匹配在配置文件中定义的输出字符串或动作方法来选择下一个页面。

导航规则定义了怎样从一个页面浏览到应用中的其它页面的规则，每一个导航项（navigation case）定义了一个目标页和或者是逻辑输出字符串或者是动作方法的引用，或者二者都有。例如：

```
<navigation-rule>
    <from-view-id>/greeting.jsp</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/response.jsp</to-view-id>
```

```
    </navigation-case>
</navigation-rule>
```

这个规则说明当greeting.jsp页面上的按钮或链接被激活并且逻辑输出字符串是success时应用从greeting.jsp流转到response.jsp。

16.1.4.5 Backing Bean管理

Web 应用的另一个重要功能就是资源的管理，包括分离UI组件与商业逻辑对象，数据对象；还包括在合理可见范围内管理与存储这些对象。JSF应用通常包括一个或多个与UI组件相联系的backing bean。Backing bean定义了绑定在组件值或组件实例上的UI组件属性。Backing bean也可以定义与组件相联系的功能方法，如校验、事件处理、导航处理等。页面编写者用EL（Expression Language）将组件值与实例和backing bean属性绑定，或者从UI组件的标签里引用backing bean的方法，EL用#｛｝作为分隔符。JSF表达式可以是值绑定或方法绑定，可以混合字符串、求值、JSP2.0EL定义的操作符。

例如：绑定userNo组件的值到backing bean UserNumberBean的userNumber属性，同时用backing bean的validate方法完成组件值的校验，以确保用户输入的是有效值。

```
<h:inputText id="userNo" value="#{UserNumberBean.userNumber}" validator="#
{UserNumberBean.validate}" />
```

绑定到组件属性的值必须是组件支持的类型，如：UserNumberBean的userNumber属性是整数类型，而UIInput组件支持此类型。UIInput组件除了validator属性还有一个valueChangeListener属性响应UIInput组件产生的ValueChangeEvent事件。完成ActionSource接口的组件可以用actonListener和action属性来对应backing bean的方法，actionListener对应处理动作事件的方法，action对应与导航输出逻辑串的计算相关的方法。也可以用标签（tag）来绑定backing bean的属性和组件实例，如：<inputText binding="#{UserNumberBean.userNoComponent}"/>，绑定的属性必须与组件的接受的类型一致，如上面的inputText标签必须是这样编写：

```
UIInput userNoComponent = null;

...
public void setUserNoComponent(UIInput userNoComponent) {
  this.userNoComponent = userNoComponent;
}

public UIInput getUserNoComponent() {
  return userNoComponent;
}
```

当组件实例与backing bean相绑定，则backing bean的属性保持组件的当前状态；而如果将组件值与backing bean绑定，JSF在update model生命周期时backing bean会更新组件的状态。绑定组件实例的好处是backing bean可以在程序里修改组件属性，backing bean可以自己实例化组件而不需要页面编写者来做；绑定组件值的好处是页面编写者可以控制组件属性，backing bean不需要依赖JSF API，有利于分离表示层与模型层，JSF可以自动完成bean属性的数据转换，不需要程序员来做。

大多数情况下可以使用组件值绑定，只有在你需要动态改变组件的某一属性时才需要绑定组件实例，例如应用在某些条件下才显示某个组件，你可以依据不同的条件设定组件的rendered属性。

Backing bean的创建与存储由在配置文件中指定，由managed bean机制来管理。

除了可以用binding、value属性来引用bean，还可以创建ValueBinding来引用bean的更多资源。

16.1.4.6 更多参考：

• JavaServer Faces 1.1 TLD 文档:

http://java.sun.com/j2ee/javaserverfaces/1.1/docs/tlddocs/index.html • JavaServer Faces 1.1 标准 RenderKit 文档:

http://java.sun.com/j2ee/javaserverfaces/1.1/docs/renderkitdocs/index.html • JavaServer Faces 1.1 API 规范:

http://java.sun.com/j2ee/javaserverfaces/1.1/docs/api/index.html • JavaServer Faces 1.1 规范:

http://java.sun.com/j2ee/javaserverfaces/download.html • The JavaServer Faces 主页:

http://java.sun.com/j2ee/javaserverfaces

16.2 EJB开发手册

16.2.1 简介

摘要

Enterprise JavaBeans™(EJB™)是Java EE™平台规范中定义的一种组件模型，用于对分布式企业应用中的业务逻辑进行封装。使用EJB技术编写的组件需要部署到EJB容器中才能运行。通过容器，应用服务器向EJB组件提供了多种底层系统服务，诸如事务控制、分布式服务、安全控制等，因此，使用EJB技术开发业务组件，可以提高应用的开发、维护的效率，同时，也将提高应用的可移植性及可靠性。应用开发人员可以更专注于特定的业务需求与模型。

金蝶Apusic应用服务器提供了对EJB3.0规范的完整支持，在其中提供了高效而可靠的EJB容器。

16.2.1.1 Enterprise JavaBean简介

实际上，EJB是由Sun公司定义的一种组件模型的技术规范，EJB组件必须符合EJB组件模型规范中关于相关的接口、实现以及部署描述等等方面的规定。根据规范，EJB组件必须使用Java语言进行编写，用于表示多层分布式企业应用中的可重用业务逻辑组件，以满足应用中特定的业务需求。EJB组件必须部署到EJB容器中运行，并可获得EJB容器提供的种种服务，如持久性、安全性、事务控制、并发访问控制等等。

同样，应用服务器的提供者也按照EJB组件模型规范，提供标准EJB组件的运行时环境，即EJB容器，同时，向运行于容器中的EJB组件提供底层系统服务。这样，符合EJB组件模型规范的标准EJB组件，可以不需要进行任何改动，运行于不同的应用服务器上。

在提供了一个严格遵循规范的EJB容器的基础上，Apusic应用服务器还提供了很多覆盖EJB组件开发、调试、部署和运行方面的增值特性。如CMP Entity Bean自动升迁技术：对于一个按照EJB1.1规范编写的CMP Entity Bean，应用服务器在运行时将其自动升级到EJB2.1，使按照EJB1.1规范编写的EJB可以使用EJB2.1规范中提供的Lazy Loading和Smart Update技术以提高应用执行的效率，同时降低维护已有应用成本。向EJB组件提供优越的服务质量和其他增值特性，这也正是Apusic应用服务器的优势所在。

16.2.1.1.1 作为组件模型的EJB

EJB是基于组件的分布式计算的架构，是面向事务、分布式的企业应用中的组件。

一. 特征

- 包含处理企业数据的业务逻辑；

- EJB实例由容器在运行时创建及管理；
- 可在部署时通过编辑环境项（environment entry）定制EJB的行为；
- EJB的各种服务设置信息，如事务及安全属性，从EJB的类文件中分离出来。在部署和运行时，可通过工具对EJB的服务设置信息进行管理；
- EJB部署到EJB容器后，客户端才可通过EJB容器对EJB进行间接访问；
- EJB可使用任何EJB规范中指定可以使用的服务；
- EJB可以不经改动代码或重新编译，即可直接装配到一个新的应用中；
- 当EJB被部署到不同的容器或服务器时，EJB开发者定义的客户视图(Client View)不会发生改变。

二. 使用EJB组件模型的优势

- 服务器完成底层的复杂工作：应用开发人员关注于业务逻辑的实现而不是底层系统的实现机制；
- 事务处理：服务器提供对EJB组件的事务控制服务，可以通过代码外的部署描述来设置组件事务处理级别；
- 可扩展性：EJB组件可以根据应用的增长进行扩展；Apusic应用服务器还提供了负载均衡和失败恢复的特性，在保证系统可靠性的同时，增加系统的可扩展性；
- 安全性：由应用服务器提供资源的访问权限控制；

16.2.1.1.2 EJB组件模型

一. 客户端类型

EJB的客户端有以下几种类型：

- 运行于相同容器或其他容器中的EJB；
- 一般的Java类，如Java应用程序、applet、servlet；
- 非Java的客户环境，如非Java语言编写的CORBA客户。

二. EJB组件模型的组成部分

一般，EJB组件由以下几个部分组成，即组件接口、Home接口、Enterprise Bean类和部署描述文件(Message-driven Bean不具有组件接口和Home接口）。下面分别描述这些组成部分。

(1). EJB的组件接口

客户端通过EJB的组件接口访问EJB对象，组件接口中定义了可被客户端访问的业务方法(Message-driven Bean不具有组件接口）。组件接口分为远程接口和本地接口。

EJB对象通过远程或本地接口，提供远程客户访问或本地客户访问的支持。

提供远程接口的EJB拥有可被远程客户访问或本地客户访问的能力。对于提供了远程接口的对象，客户可通过标准的Java RMI（Remote Method Invocation）进行远程对象调用。

提供本地接口的EJB只可被本地组件通过本地接口进行调用。所谓本地组件，即是运行于相同Java虚拟机中的本地EJB对象。本地调用通过一般的标准Java编程语言接口进行。

EJB可以同时提供本地接口和远程接口，但一般只提供二者之一。

(2). HOME接口

EJB2.0 规定了通过Home接口来提供客户端创建、清除和在同种类型的EJB中查找特定EJB对象的方法(Message-driven Bean不包含Home接口）。对于提供远程接口的EJB，需要提供远程Home接口；提供本地接口的EJB，需要提供本地Home接口。

Home接口由EJB开发人员编写，远程Home接口必须扩展（extend）javax.ejb.EJBHome接口；本地Home接口必须扩展（extend）javax.ejb.EJBLocalHome接口。

EJB客户端通过标准的JNDI（Java Naming and Directory Interface™）API定位Home接口。

EJB3.0中，Home接口的功能由依赖注入以及可选的生命周期回调方法实现。EJB组件不再需要提供Home接口

(3). Enterprise Bean类

Enterprise Bean类包含了组件的实现细节。

Enterprise Bean类由EJB开发人员编写，EJB2.0规范中的Enterprise Bean组件，必须分别声明实现如下接口，javax.ejb.SessionBean、javax.ejb.EntityBean和javax.ejb.MessageDrivenBean。符合EJB3.0规范的Enterprise Bean组件则不须实现以上接口，而使用@Stateful、@Stateless、@MessageDriven注解标记EJB类。

(4). 部署描述文件

部署描述文件是用于包含Enterprise Bean的运行时属性(安全性，事务性等等)信息的文件，与以上部分一起形成完整的EJB组件，通常部署描述文件使用图形化的部署工具进行处理。在EJB3.0规范中，EJB部署描述文件并不是必须的，可以在代码中使用注解来为EJB类附加部署信息。但部署描述文件可以令管理人员在部署时更为灵活。 EJB3.0规范规定，当部署描述文件的信息与程序代码中的注解信息出现冲突时，以部署描述文件为准。这样管理人员就可以通过修改部署描述文件来改变部署信息，而不须重新编译应用程序。

16.2.1.1.3 EJB的类型

EJB组件模型包含三种类型的EJB：

- Session Bean
- Entity Bean
- Message-driven Bean

下面将介绍这三种EJB组件类型。

16.2.1.2 Session Bean

16.2.1.2.1 什么是SESSION BEAN

Session Bean可被视为客户端程序在服务器上的部分逻辑延伸，每个Session Bean对象对应于特定的客户端，不能在多个客户端间共享。换句话说，Session Bean用于表示运行于服务器端的部分业务过程，作为客户端的代理，管理业务过程或任务，如客户对账户的借贷操作、汇率的计算，等等这些涉及逻辑、算法和工作流的种种任务。这些过程都是特定的客户行为，EJB根据这些过程在运行时创建过程实例、执行计算或者清除实例。

16.2.1.2.2 SESSION BEAN的生存时间

相对于表示业务实体的Entity Bean，Session Bean的生存时间要短，大致等于于一个客户端会话的延续时间。例如，客户通过浏览器，访问与Session Bean表示的相关业务过程，或通过一个Java应用程序或者是Applet访问的相关业务过程的持续时间，或其它Enterprise Bean访问此业务过程的持续时间。

Session Bean是非持久的，其状态不被保存到持久存储机制（如数据库、文件系统）中，尽管Session Bean本身可以执行对数据库的操作，但它并不是一种持久对象的表示。

Enterprise Bean类必须实现的接口中，定义了Enterprise Bean类的一些容器用于管理实例的回调方法。这些回调方法被容器用于与组件进行交互，当容器执行与此Bean相关的一些重要操作时，通过调用这些方法通知组件。例如，用于容器在初始化或清除组件实例时，将调用这些方法中对应的管理回调方法。由于这些方法是由容器使用的，所以，组件中不应去调用这些方法(有状态会话bean的remove方法除外)。

16.2.1.2.3 会话状态

Session Bean表示客户端与应用之间的会话。会话由客户端与组件之间的交互组成，一般表现为一系列的方法调用。

根据Session Bean有保持会话状态的方式，可分为有状态的和无状态的Session Bean。

一. 无状态的Session Bean

无状态的Session Bean，会话状态只会在单个方法调用中被保持，一旦方法调用结束，组件将丢弃方法调用过程中保持的状态，不保持跨越方法调用的会话状态。

由于无状态Session Bean不提供跨越方法调用的状态保持，因此，对于任何客户端调用，相同类别的Session Bean实例之间没有区别。例如，某个特定客户端对某个特定的无状态Session Bean实例的方法进行了调用后，此方法调用中的状态在调用完成后即被清除，对于后续的其他客户端方法调用而言，前一个是没有影响的。因此，无状态Session Bean通常被用于一些不需要保存跨越方法调用的会话状态的计算，如汇率的换算等较为简单的操作。

由于具有这种会话状态无关的特性，无状态Session Bean通常可以在服务器启动时在实例池中创建并保持一些实例，为不同的客户端调用从实例池中分配已有的实例，免去频繁的创建、初始化和清除实例的动作。

二. 有状态的Session Bean

对于有状态的Session Bean，在整个会话期间，特定组件实例将保持与某个特定客户端之间跨越方法调用的会话状态。不同于无状态的Session Bean，有状态Session Bean的实例只能对应于一个特定的客户端会话，不可在不同的客户端会话间共享，当新的会话开始，有状态的Session Bean实例即被创建和初始化，当会话结束，实例即被清除。

因此，一般情况下，有状态的Session Bean所能提供的功能性，要强于无状态的Session Bean。

16.2.1.3 Entity Bean

16.2.1.3.1 什么是ENTITY BEAN

Entity Bean用于表示保存在持久数据存储机制中的实体，为这些实体提供面向对象的视图，如关系型数据库中的业务实体数据，或传统业务系统中的业务实体。

	**注意**

	在Java EE 5规范中，EJB3.0组件模型的持久机制由独立的Java Persistence API(JPA)实现。同时，EJB3.0规定了对EJB2.1兼容，因此在Apusic服务器中，仍可按照EJB2.1的习惯使用Entity Bean。

一. Session Bean与Entity Bean

Entity Bean与Session Bean的不同表现在如下几个方面：

(1). 持久性

在应用或应用服务器的运行时间外，Entity Bean状态数据仍存在于持久数据存储之中。

Entity Bean有两种持久类型：Bean管理持久类型（Bean-managed Persistence，BMP）和容器管理的持久类型（Container-managed Persistence，CMP）。对于Bean管理的持久类型，由组件模型中Enterprise Bean类的代码控制对持久存储中状态数据的访问；容器管理持久性类型的Entity Bean，则由容器生成和管理对应持久存储中状态数据的访问方法。开发者可以不用编写这些方法。

(2). 共享访问

相对于Session Bean对应于特定的客户端会话而言，Entity Bean可以在客户端间被共享访问，对于多个并发的客户端访问，由容器提供了完整的事务（transaction）管理机制，保证状态数据的完整性。并且提供了通过修改部署描述，指定事务属性的能力，无需在代码中对事务边界进行控制。

(3). Primary Key

每个Entity Bean实例拥有一个唯一的标识对象，客户端可以使用此标识定位特定的Entity Bean实例。

(4). 关系

如存在于关系型数据库中的业务实体数据，Entity Bean之间也可以具有类似的关系。对于Bean管理持久性的Entity Bean，需要在Entity Bean的组件类中对关系进行编码；而对于容器管理持久性的Entity Bean，则由容器对关系进行控制。因此，容器管理持久性的Entity Bean之间的关系，被称为容器管理关系(Container-managed relationship，CMR）。

二. 容器管理持久性（Container-managed Persistence，CMP）

容器管理持久性指由EJB容器处理Entity Bean需要的对数据库的访问。使Entity Bean可以脱离特定类型的数据库，从而具有更高的灵活性。

为使Entity Bean可以由EJB容器对持久性进行管理，需要提供Entity Bean的抽象模式（abstract schema）信息。

(1). 抽象模式（Abstract Schema）

抽象模式作为容器管理持久性的Entity Bean部署描述的一部分，提供对Entity Bean持久域和关系的定义。之所以称其为"抽象"模式，是为了与具体的底层数据存储的物理模式区分开来。例如，在关系型数据库中，物理模式由诸如表或列等结构构成。

对于容器管理持久性的Entity Bean，需要在部署描述中定义抽象模式的名字。这些名字将可以通过使用Enterprise JavaBeas™ Query Language（EJB™QL）编写的查询语句进行引用。例如，必须为每一个finder方法定义一个EJB QL查询语句，通过此语句定义当此finder方法被调用时，容器执行的查询。

- 持久域（persistent fields）

被定义的Entity Bean的持久域保存在底层的数据存储中，所有这些域构成整个Entity Bean的状态。在运行时，容器自动对这些状态与数据库中的数据进行同步。一般，容器在部署时将Entity Bean与特定的数据库表进行映射，将持久域与数据库表中的列进行映射。

- 关系域（relationship fields)

关系域可视为数据库表中的外键（foreign key），用于标识Bean之间的关系。

三. **容器管理关系（Container-managed Relationship，CMR）**

容器管理持久性的Entity Bean有以下四种容器管理关系：

- one-to-one，一个Bean的单个实例关联另一个Bean的单个实例；
- one-to-many，一个Bean的单个实例关联另一个Bean的多个实例；
- many-to-one，一个Bean的多个实例关联另一个Bean的单个实例；
- many-to-many，一个Bean的多个实例关联另一个Bean的多个实例；

四. **容器管理关系的方向**

容器管理关系的方向可以是双向或单向。在一个双向的关系中，涉及的Bean都有一个关系域与另外的Bean关联，通过关系域，可以从一个Bean的实例中访问关联的Bean对象，反之亦可。在一个单向的关系中，只有一个Bean拥有关联其他Bean的关系域，只能从这个Bean的实例访问被关联的Bean对象，而不可从被关联的Bean对象访问到这个实例。

EJB QL查询可以通过关系进行定位。关系的方向决定了是否可以从一个Bean定位另一个Bean。如订单作为一个Bean，订单中的项目作为一个Bean，二者拥有双向关系，即订单知道拥有哪些项目，而项目知道自己属于某个订单，因此，通过关系，可以从订单定位订单中的项目，也可以从订单中的项目定位到订单；

16.2.1.3.2 ENTITY BEAN的特征

一. **持久性**

由于Entity Bean的状态数据保持在持久存储中，因此，即使容器或服务器失效，Entity Bean的状态数据仍然存在。

二. **持久业务数据的对象表示**

Entity Bean是持久业务数据的对象表示，更改Entity Bean的实例的状态，数据库中的对应信息也将自动更新。实例状态是数据库数据的视图，而不是两个分离的数据。

三. **表示相同的底层数据的多个Entity Bean实例**

在企业应用中，通常出现多个客户对某些业务实体的并发访问，一般，EJB容器通过维护同一个EJB实例的多个拷贝，提供给不同的客户，以此提高对客户请求的响应速度，EJB容器提供保持实例状态的一致性的机制。而这种特性对于客户来说是透明的，因此，客户可以认为操作中的EJB实例是唯一的实例，可以不用考虑EJB实例状态在多个并发客户间的一致性。

四. **可查找的Entity Bean**

因为Entity Bean实例拥有其独有的标识，因此，除了可以创建Entity Bean实例外，如使用SQL中的SELECT语句，还可对在数据库中已有的业务数据进行查找。对Entity Bean实例的查找一般通过在Entity Bean的Home接口中定义finder方法实现。

16.2.1.4 Message-driven Bean

Message-driven Bean是EJB2.0 规范中提出的的Enterprise Bean。金蝶Apusic应用服务器提供了对规范中Message-driven Bean相关内容的完整实现。Message-Driven Bean 作为EJB2.0规范中新增的一个enterprise bean 类型，除得到应用服务器管理的事务，安全

和资源访问的服务之外，同时作为JMS消息系统中的消息使用者 （Message Consumer）接收并处理应用消息。有关JMS消息系统，请参阅JMS开发。

16.2.1.4.1 MESSAGE-DRIVEN BEAN 的产生原因

- 效率原因

在Java EE™平台中，客户端对Session Bean 和Entity Bean 的方法调用通过RMI或RMI-IIOP协议进行，这是传统的通过网络进行远程调用的方法，当调用请求通过网络传播到容器，容器则将客户端请求变成一序列的方法调用依次进行。客户端只有在容器处理完请求并返回结果后方可继续进行。

- 可靠性的原因

当客户端对Session Bean或Entity Bean进行调用时，必须保证服务器容器处于运行状态，如容器或网络出现错误，客户端调用将无法进行。

- 事件的广播

传统的RMI 或RMI-IIOP机制中，客户端在某一时刻只能与某一具体的服务器通讯，没有任何内置的机制来将事件广播到多个服务器。

Message-driven Bean作为远程方法调用的一种替代方法，在客户端和服务器的直接方法调用之间放置了一个中间层，接收一个或多个客户端的消息，并将消息转发给一个或多个消息的使用者(Message Consumer)。

通过消息机制而非直接的方法调用，客户端可以继续执行而不必等待服务器的运行结果，服务器可以选择在方法调用完成后通知客户，而消息机制本身保证了信息传输的可靠性，同时使用消息域(Message Domain)中的消息类型模型以达到事件广播的机制。

但是，对Message-Driven Bean 的使用也有一定的限制，如不适用于依赖于方法调用、要求具有明确的返回值才能继续的客户端程序。另外，如果在一个应用中过多的使用了Message-Driven Bean，对应用的执行效率将会产生影响，所以不适用于对时间因素敏感的客户端程序(如在下午两点定购下午四点的机票，而在四点后才得到订购是否成功的结果，这时结果已毫无用处)。

16.2.1.4.2 MESSAGE-DRIVEN BEAN 作为一般的JMS 使用者(CONSUMER)

作为一种具有JMS使用者(consumer)功能的Enterprise Bean组件模型，Message-Driven Bean由EJB容器进行管理，具有一般的JMS使用者(consumer)所不具有的优点，如对于一个Message-driven Bean，容器可创建多个实例来处理大量的并发消息，而一般的JMS使用者(consumer)开发时则必须对此进行处理才能获得类似的功能。同时Message-Driven Bean可取得EJB所能获得的标准服务，如容器管理事务等服务。

由于与Message-driven Bean相关的主题(Topic)或队列(Queue)可以在部署时配置，因此，Message-driven Bean具有更多的灵活性。

但注意，一个Message-driven Bean在部署时只可与一个具体的主题(Topic)或队列(Queue)建立关联。如有多个主题(Topic)或队列（Queue）需要与一个Message-driven Bean关联，则可以在部署时部署多个Message-driven Bean类,或使用一般的JMS使用者(consumer) 。

16.2.1.4.3 MESSAGE-DRIVEN BEAN 与其他ENTERPRISE BEAN

作为Enterprise Bean组件模型之一，Message-driven Bean，具有一些与Session Bean 和Entity Bean相同的方法，但由于Message-driven Bean本身不处理客户端调用，也无会话状态，客户只能通过向与Message Driven Bean关联的队列或主题发送消息从而与Message-driven Bean 进行交互，因此，Message-driven Bean 与Session Bean 和Entity Bean之间最大的不同之处在于Message-Driven Bean不具有组件接口及Home接口。

另外，Message-driven Bean异步地处理队列（Queue）或主题(Topic)中的消息，而非方法调用。

16.2.1.5 使用接口定义客户访问

	注意
	本节仅适用于Entity Bean与Session Bean，不适用于Message-driven Bean。因Message-driven Bean组件模型不同于Entity Bean与Session Bean，不具有定义客户端视图的组件接口。

对于Session Bean和Entity Bean，客户端只可通过在组件接口中定义的方法对Bean进行访问。组件接口定义了Bean的客户端视图。有关Bean的方法实现、部署描述设置、抽象模式以及数据库访问等等，对于客户而言是不可见的。

通过定义组件接口，可以更好地简化应用的开发和维护工作。清晰的组件接口可以将客户端与EJB层的复杂性进行分离，而且在改变Bean内部实现而不影响客户端。

组件接口被划分为本地（local）接口和远程（remote）接口，用于定义客户端的访问方式。

16.2.1.5.1 远程访问

Enterprise Bean的远程客户端具有如下特征：

- 对于要访问的Enterprise Bean，远程客户可以运行在不同的计算机或Java虚拟机中，也可以运行在相同的计算机或Java虚拟机中；
- 可以是Web组件、Java EE应用客户端或者是另一个Enterprise Bean；
- 对于远程客户，访问Enterprise Bean与Enterprise Bean的位置无关。

在EJB2.1规范中，如要编写一个远程客户端可访问的Enterprise Bean，必须为组件提供一个远程接口和一个Home接口，并在远程接口中定义远程客户端可访问的业务方法。在EJB3.0规范中，可以使用@Remote注解定义远程接口。调用EJB3.0规范中的远程业务接口与客户端位置无关，无论是与会话Bean运行于不同Java虚拟机中的远程客户端，或运行于同一JVM中的本地客户端，均可通过远程接口访问会话Bean。容器为远程接口提供了位置透明性。

16.2.1.5.2 本地访问

Enterprised Bean的本地客户端具有如下特征：

- 对于要访问的Enterprise Bean，本地客户必须运行在同一个Java虚拟机中；
- 可以是Web组件或另一个Enterprise Bean;
- 对本地客户而言，对Enterprise Bean的访问与Enterprise Bean所处的位置有关；
- 通常是某个与其他Entity Bean具有容器管理关系的Entity Bean。

在 EJB2.1规范中，如要编写一个本地客户端可访问的Enterprise Bean，必须为组件提供一个本地接口和一个本地Home接口，并在本地接口中定义本地客户端可访问的业务方法。在EJB3.0规范中，可以使用@Local 注解进行标识。调用本地接口的客户端必须与会话BeanJava运行于同一Java虚拟机中。

16.2.1.5.3 本地接口与容器管理关系

如在容器管理关系中，某个Entity Bean可以被其他Entity Bean所关联，则它必须提供可本地访问的接口类型。Entity Bean之间关系的方向决定Entity Bean是否被其他Entity Bean所关联，例如，表示账户的Entity Bean与表示在账户上发生的历史操作之间，账户知道自己发生的所有历史纪录，可以从账户定位相关的历史纪录的集合，可以称为表示历史纪录的Bean被表示账户的Bean所关联，因此，

表示账户历史纪录的Bean必须提供可本地访问的接口类型，即提供本地接口与本地Home接口。如账户与账户历史纪录之间存在双向的关系，即可从表示历史纪录的Bean定位到表示账户的Bean，则表示账户的Bean也必须提供可本地访问的接口类型。

因为本地访问的关系，参与同一个容器管理关系的Entity Bean，必须被打包到同一个EJB JAR文件中。本地访问的最大好处是效率上的提高，因为，通常本地访问要快于远程访问。

16.2.1.5.4 方法参数和返回值

组件接口类型影响组件方法被客户端调用时的参数传递方式。

远程调用中的参数是按值传递的，传递的是对象的拷贝，但本地调用中参数的传递则是按引用传递的，与Java编程语言中的正常的方法调用相同。

远程调用中，客户端与组件相对比较独立。因为客户端与组件操作不同的参数对象拷贝，任何一方对数据的修改，不会影响另一方的数据。

在本地调用中，客户端和组件修改的是同一个对象。在编写Enterprise Bean时，应避免使用由这种传递方式带来的副作用，因为，当应用规模增长，客户端被分布到不同的物理服务器，参数的传递方式亦随之改变。

16.2.1.5.5 远程还是本地

决定一个组件应该提供远程访问还是本地访问，一般有以下一些因素。

- 容器管理关系

如果某Entity Bean被其他Entity Bean所关联，则此Entity Bean必须提供本地访问接口。

- 相关Bean之间的耦合程度

耦合程度高的Bean之间存在互相依存的关系。因此，这一类Bean一般需要提供本地访问接口，以形成一个逻辑上的单元，同时提供整个单元内交互的效率。

- 客户端类型

如Bean将被Java EE应用客户端所访问，则应该提供远程访问接口。如果客户端是Web组件或其他Bean，提供何种类型的访问接口决定于这些组件的分布状况。

- 组件分布

Java EE应用可以分布在多个计算机，为提供这种分布上的可伸缩性，将被分布到不同位置的组件所访问的Bean，应该提供远程访问接口。

尽管一般Bean只提供一种类型的客户访问接口，但事实上，有一些Bean仍需要同时提供两种类型的接口。

16.2.1.5.6 执行效率与组件接口类型

因为网络延迟等因素，本地访问通常要快于远程访问；另一方面，在不同的服务器上分布应用，可以提高应用的整体运行效率。而且，不同的操作环境中，执行效率也会有不同。因此，设计应用时应考虑执行效率的影响。

16.2.1.6 Enterprise Bean的内容

一个合法的金蝶Apusic应用服务器的EJB，应包含如下几个部分：

- 部署描述：定义Bean部署信息，如包含持久类型、事务属性等信息的XML文件。通常通过使用图形化的部署工具生成。对于Apusic应用服务器，部署描述文件包括一个EJB规范中规定的ejb-jar.xml文件和一个Apusic规定的apsuic-ejb-jar.xml文件。在

EJB3.0 中，ejb-jar.xml中的部署描述信息可通过注解在源代码中进行标记。

- 组件接口：在EJB2.1提供远程客户访问能力的Bean必须提供远程接口文件和远程Home接口文件；提供本地客户访问能力的Bean必须提供本地接口文件和本地Home接口文件（不包含Message-driven Bean）。
- 组件类：组件接口中定义的方法，在此类文件中提供实现。

将以上文件打包成为EJB JAR文件之后，形成一个EJB模块，可连同其他模块装配到不同的表示Java EE应用的EAR文件中。

16.2.1.7 EJB组件模型的灵活性

EJB组件模型的灵活性表现在:

- EJB可作为表示无状态服务的对象;
- EJB可作为表示无状态服务的对象，可通过向指定的消息队列或主题发送JMS消息，以实现对此对象的异步调用;
- EJB可作为表示与特定客户的会话对象。此类对象在客户进行跨越方法的调用时，自动维持会话状态;
- EJB可作为表示业务对象的实体对象，在多个客户间共享;

EJB可作为一个细粒度的持久对象，包含在一个粗粒度业务对象的持久状态中。

通常，被远程访问的组件往往是粗粒度的业务对象，如订单、雇员纪录；细粒度的业务对象往往不会采用可远程访问的EJB组件模型，如订单中的采购项、雇员纪录中的地址，而是采用可本地访问的EJB组件模型或是作为EJB的附属类出现。

16.2.1.8 何时使用EJB组件

下面是在使用EJB组件模型构建企业应用时，一些判断EJB组件模型是否适用的标准。

- EJB组件是构建分布式企业应用的组件模型技术;

EJB组件规范是针对分布式企业应用制定的，是基于分布式对象技术的Java组件；EJB组件不涉及表示层的内容，因此，必须与其他表示层技术一起使用；应用服务器提供了可以解决安全性、资源共享、持续运行、并行处理、事务完整性等复杂问题的服务，从而简化了商业应用系统。

- 应用客户端类型的考虑;

一般，企业应用开发都会有多种类型的客户端的需求，访问相同的数据或业务逻辑。如使用Web客户提供对应用的基于Internet的访问，使用应用客户端提应用基于Intranet的访问。EJB组件模型将业务逻辑与数据封装到EJB组件中，提供对多种客户端的支持。

- 应用数据与业务逻辑的并发访问控制的考虑;

企业应用通常需要提供数据或业务逻辑的并发访问能力，以此保证数据的完整性，由于EJB 组件控制对后台数据的访问，并管理当前事务和数据库的内部锁定。节省了编写数据库控制逻辑的工作量，同时也保证了数据的一致性与正确性，从而降低了总编程量。

- 全局事务控制的考虑;

企业应用通常需要对不同的资源进行事务性的操作，如某个操作需要对数据库进行访问，同时可能需要通过JMS消息服务发送消息，或者，需要访问两个位于不同物理位置的异种数据库，这些操作必须在相同的事务环境中完成。

Apusic应用服务器提供了全局事务控制，使得对数据库的操作和对消息服务的操作可以加入到一个事务环境中，而这个特性对开发者而言是透明的，由应用服务器内部的事务管理器对事务环境中的操作进行管理。

- 基于访问控制的考虑；

企业应用中往往需要对某些资源进行访问控制，如需要针对不同用户对组件方法调用设置访问控制策略，对访问特定Web资源的用户设置访问控制策略等，Apusic应用服务器提供了基于用户和组的身份鉴定和授权机制，可以通过部署描述和图形工具实现对Web资源的访问控制和对EJB组件的方法级别的访问控制。

- 基于伸缩性的考虑；

企业应用的用户规模往往成为影响企业应用开发难度的主要因素，随着企业经营规模的扩大，用户规模也会随之增长，Apusic应用服务器提供了负载均衡与实效恢复的能力，可以在降低开发难度的情况下，通过增加服务器节点即可适应应用规模的快速增长。

- 基于重用性的考虑；

企业应用往往涉及很多重复的业务逻辑和数据模型，使用EJB组件模型，可以通过修改部署描述文件在不同的应用中方便地重用这些组件。

16.2.2 会话Bean

摘要

本节将描述Session Bean的基本原理；开发过程中涉及的普遍过程；组件模型组成部分的开发规则及注意事项。

16.2.2.1 会话Bean

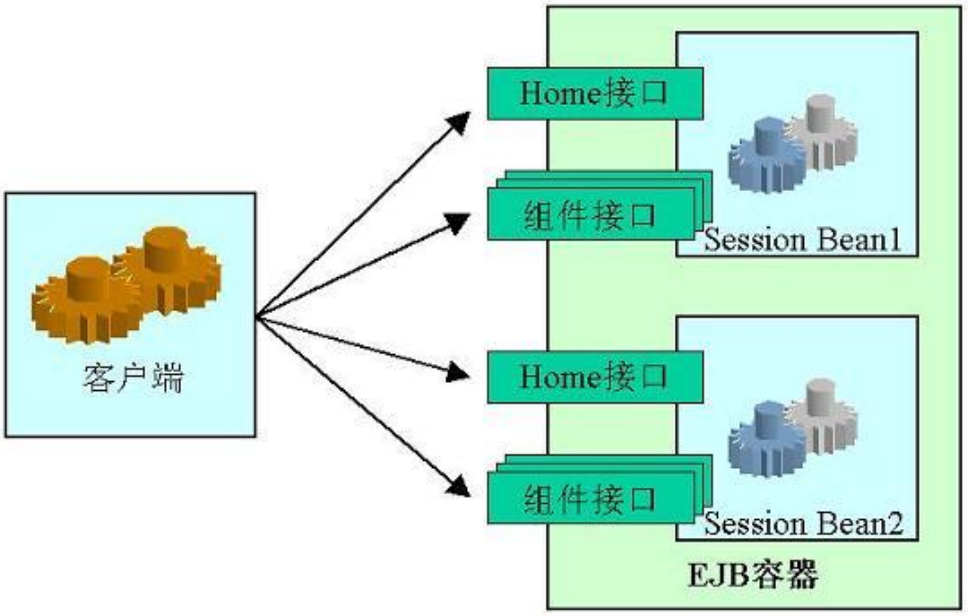
16.2.2.1.1 容器与会话BEAN

容器是为EJB组件提供运行时环境的系统。同一容器中可以部署多个EJB组件。容器提供客户端通过JNDI对已部署组件进行依赖注入（EJB3.0）或访问Home接口（EJB2.1）的能力，即客户端可请求容器注入会话Bean远程接口或通过JNDI查找指定EJB组件的Home接口。

客户端不能直接访问EJB组件的实例，只能通过JNDI查找指定组件的Remote接口，通过Remote接口取得对组件接口的引用。客户端对组件接口的方法调用，通过组件接口传播到容器中对应的EJB组件实例。

下图表示EJB组件及其接口与容器、客户端间的关系：

EJB组件及其接口与容器、客户端间的关系



16.2.2.1.2 会话BEAN的会话状态

容器为会话Bean提供运行时环境，并对会话Bean实例进行管理，因此，会话Bean实例的状态也由容器进行维护。根据状态管理模式，可以将会话Bean 划分为有状态（Stateful）会话Bean和无状态（Stateless）会话Bean，有状态会话Bean的状态由容器负责维护，而无状态会话Bean则不需要容器进行状态管理。

一. 有状态Session Bean实例的钝化与激活

对于当前容器中存在的有状态会话Bean实例，为提供更有效的管理，容器需要将某些空闲的实例状态从内存临时转移到存储机制中。这种转移称为实例的钝化（Passivation），相反地，将实例状态从存储机制恢复，则称为激活。（Activation）。

当实例被包含于某个事务中时，容器不能对实例进行钝化操作。

二. 会话状态

有状态会话Bean的会话状态指会话Bean实例中的域值，连同实例中的域通过Java对象引用指向的对象的传递闭包（transitive closure）。

	注意
	对象的传递闭包按照Java编程语言中的串行化协议定义，即是通过串行化对象实例，对实例的域进行保存。参考Sun的。

某些情况下，会话对象的会话状态可能会包含打开的资源，如网络端口和数据库连接等。当会话对象被钝化时，容器不能钝化这些资源，因此，需要开发者在容器对实例发出钝化事件通知时（容器调用实例中注解为PrePassivate的方法(EJB3.0)或ejbPassivate方法），对上述资源进行关闭；在容器对实例发出激活事件通知时（容器调用实例中注解为PostActivate的方法或ejbActivate方法），重新打开上述资源。

因此在编写有状态的Session Bean时，开发者必须注意，保证实例的非transient域是下列类型之一，使容器可以在钝化实例时，可以完整保存对象的会话状态：

- 可串行化的对象；
- null值；
- EJB的业务接口引用；
- EJB的远程接口引用，即使stub class为非序列化亦可
- EJB的远程Home接口引用，即使stub class为非序列化亦可；
- Entity Bean的本地接口引用，即使其为非序列化亦可；
- EJB的本地Home接口引用，即使其为非序列化亦可；
- 对SessionContext对象的引用，即使其为非序列化亦可；
- 环境命名上下文（指"java:comp/env"JNDI上下文）及其任何子上下文；
- UserTransaction的引用；
- Resource manager连接工厂的引用；
- 对容器管理的EntityManager对象的引用，即使其为非序列化亦可；
- 通过依赖注入或JNDI查找获得的EntityManagerFactory引用，即使其为非序列化亦可；
- 对javax.ejb.Timer对象的引用
- 不可直接串行化的对象，但是通过在串行化对象期间，将对象的引用更改为对象的业务接口，Home接口和组件接口的引用、对象对SessionContext 对象的引用、对象对"java:comp/env"JNDI上下文及其子上下文的引用、对象对UserTransaction接口的引用、对象对EntityManager与EntityManagerFactory的引用进行串行化，可将对象视为可串行化的对象。

三. **事务操作与状态域**

会话Bean的会话状态是非事务性的。如对象在参与的事务中改变了状态，之后事务回滚，对象是不能自动回到参加事务前的状态的。因此，在不能保证会话Bean的会话状态与事务状态不一致的情况下，可以通过实现SessionSynchronization接口，在afterCompletion方法发出的通知中，根据事务提交的情况，决定是否重新设置对象的会话状态。

16.2.2.2 组件模型单元

对于会话Bean，其组件模型包含以下几部分：

- Home接口，定义客户端创建、清除EJB实例的方法；(注：在EJB3.0组件模型中并不需要此接口)
- 组件接口（component interface)，定义客户端可访问的组件的业务方法，在EJB3.0组件模型中称为业务接口(business interface)；
- 组件类，提供对定义在组件接口中的方法的实现；
- 部署描述，包含此EJB的部署信息，如事务属性等，在EJB3.0中可使用注解来描述部署信息。

下面分别对开发这些单元时，涉及的普遍过程、规则及注意事项进行描述。

16.2.2.2.1 HOME接口

EJB2.1规范通过Home接口定义客户端创建、清除EJB对象的方法。Home接口有两种类型，本地Home接口和远程Home接口，分别提供给本地和远程客户端使用。在EJB3.0中，可通过依赖注入，JNDI查找，可选的生命周期回调方法注解来实现类似的功能。

一. 远程Home接口

远程Home接口使远程客户端可以：

- 创建新的Session对象；
- 清除一个Session对象；
- 取得此Session Bean的javax.ejb.EJBMetaData接口。javax.ejb.EJBMetaData接口用于表示此Session Bean的信息，降低客户端与服务器的绑定程度和用于支持客户端脚本；
- 取得远程Home接口的句柄（Handler），此句柄可串行化（serialization）到持久存储中，然后，可在其他虚拟机中，从持久存储中，对此串行化对象进行解串行化（deserialization），取得对此远程Home接口句柄的引用。

(1). 编写远程Home接口的规则

编写EJB的远程Home接口时，开发人员必须遵循如下规则：

- 远程Home接口必须扩展（extend）javax.ejb.EJBHome接口；
- 定义在接口中的方法的参数和返回值必须是合法的RMI类型，并且必须显式声明抛出java.rmi.RemoteException异常；
- 在继承关系上，远程接口可以扩展已有的接口，但继承的方法必须依从上一规则；
- 远程接口方法必须定义一个或多个create方法。无状态的Session Bean必须提供一个无参数的create方法，并且只能命名为“create()”的形式；每个create方法必须匹配一个定义在组件类中的ejbCreate方法，此组件类中的ejbCreate方法必须具有相同个数和类型的参数。无状态Session Bean的create方法在组件类中的匹配方法只能命名为“ejbCreate()”的形式。；
- 每个create方法名称开头必须是“create” 。
- create方法的返回值类型必须是Session Bean的远程接口；
- create方法必须声明抛出javax.ejb.CreateException异常；

(2). 代码范例

Home接口通过定义一个或多个create方法，提供一种或多种创建session对象的方式。create方法的参数一般用于初始化被创建session对象的状态。

create方法的返回值类型为Bean的远程接口类型。

下面是一个范例Session Bean的Home接口的代码，定义了两个create方法：

```
public interface CartHome extends javax.ejb.EJBHome {
  Cart create(String customerName, String account) throws RemoteException,
BadAccountException, CreateException;
```

```
  Cart createLargeCart(String customerName, String account) throws
RemoteException, BadAccountException, CreateException;
}
```

客户端可通过如下代码，对已部署在应用服务器上的Session Bean的Home接口进行查找，并使用CartHome接口中定义的create方法创建session对象：

```
Context initialContext = new InitialContext();

CartHome cartHome = (CartHome)

javax.rmi.PortableRemoteObject.narrow(

initialContext.lookup(“java:comp/env/ejb/cart”),

CartHome.class);

Cart shoppingCart = home.create("Duke DeEarl","123");
```

二. 本地Home接口

本地Home接口使本地客户端可以：

- 创建新的Session对象；
- 清除一个Session对象；

(1). 编写本地Home接口的规则

如果EJB需要提供本地客户端的访问，必须提供EJB的本地Home接口，开发人员必须遵循如下规则进行编写：

- 本地Home接口必须扩展（extend）javax.ejb.EJBLocalHome接口；
- 定义在接口中的方法不能抛出java.rmi.RemoteException异常；
- 在继承关系上，本地接口可以扩展已有的接口；
- 本地接口方法必须定义一个或多个create方法。无状态的Session Bean必须提供一个无参数的create方法，并且只能命名为“create()”的形式；每个create方法必须匹配一个定义在组件类中的ejbCreate方法，此组件类中的ejbCreate方法必须具有相同个数和类型的参数。无状态Session Bean的create方法在组件类中的匹配方法只能命名为“ejbCreate()”的形式。；
- 每个create方法名称开头必须是“create” 。
- create方法的返回值类型必须是Session Bean的本地接口；

- create方法必须声明抛出javax.ejb.CreateException异常；

(2). **代码范例**

客户端可通过类似如下代码，对已部署在应用服务器上的Session Bean的本地Home接口进行查找：

```
Context initialContext = new InitialContext();

CartHome cartHome = (CartHome)

initialContext.lookup("java:comp/env/ejb/cart");
```

16.2.2.2.2 业务（组件）接口

会话Bean和实体Bean的客户端不能直接访问EJB组件类的实例，客户端程序通过业务接口访问EJB组件，开发者在业务接口中定义可供客户端访问的业务方法。业务接口分为两种类型，本地接口和远程接口。

一. **远程接口**

如果EJB需要被远程客户端访问，必须提供EJB的远程接口，远程接口为组件提供以下支持：

- 定义EJB对象的业务逻辑方法， 远程接口把对业务方法的调用传播到Session Bean实例；
- 提供允许客户端取得远程Home接口实例引用的方法；
- 提供使客户端取得Session对象句柄的方法；
- 提供比较两个EJB实例是否相等的方法；
- 移除Session Bean实例的方法；

(1). **必须遵守的的规则**

开发人员必须遵循如下规则编写远程接口：

- 远程接口必须使用@Remote注解标记（EJB3.0），或扩展（extend）javax.ejb.EJBObject接口（EJB2.1）；
- 若使用EJB2.1规范，定义在接口中的方法的参数和返回值必须是合法的RMI类型，并且必须显式声明抛出java.rmi.RemoteException异常；
- 在继承关系上，远程接口可以扩展已有的接口，但继承的方法必须依从上一规则；
- 每个在远程接口中定义的方法，必须在组件类中有一个匹配的方法，组件类中的匹配方法必须与接口中定义的方法具有相同的名字、相同的参数个数、参数类型和相同的返回值，而且所有在远程接口中定义的抛出的异常也必须在匹配方法中定义；
- 远程接口方法不可暴露本地接口类型、本地Home接口类型，而且，对于容器管理持久性Entity Bean，也不可暴露作为Entity Bean中方法参数和返回值的受管理的集合类（collection）。

(2). **代码范例**

下面是一个EJB2.1的会话Bean的远程接口的代码，其中定义了三个业务方法：

```
public interface Cart extends EJBObject {
  public void addBook(String title) throws RemoteException;

  public void removeBook(String title) throws BookException, RemoteException;

  public Vector getContents() throws RemoteException;
}
```

在EJB3.0中，可使用以下形式达到同样效果：

```
@Remote
public interface Cart extends EJBObject {
  public void addBook(String title) throws RemoteException;

  public void removeBook(String title) throws BookException, RemoteException;

  public Vector getContents() throws RemoteException;
}
```

二. **本地接口**

在EJB2.1规范中，如果EJB需要被本地客户端访问，必须提供EJB的本地接口，本地接口为组件提供以下支持：

- 定义EJB对象的业务逻辑方法， 本地接口把对业务方法的调用传播到Session Bean实例；
- 提供允许客户端取得本地Home接口实例引用的方法；
- 提供比较两个EJB实例是否相等的方法；
- 移除Session Bean实例的方法；

在EJB3.0规范中，容器为远程接口提供了位置透明性，本地客户端亦可通过远程接口访问业务方法。但使用本地接口可以为EJB提供细粒度的业务方法，获得更好的可重用、与可维护性与运行效率。EJB3.0中的本地接口可以使用@Local注解来标记。

(1). **必须遵守的规则**

开发人员必须遵循如下规则编写本地接口：

- 本地接口必须使用@Local注解进行标记，或扩展（extend）javax.ejb.EJBLocalObject接口；
- 本地接口中定义的方法不能声明抛出java.rmi.RemoteException异常；
- 在继承关系上，本地接口可以扩展已有的接口；

- 每个在本地接口中定义的方法，必须在组件类中有一个匹配的方法，组件类中的匹配方法必须与接口中定义的方法具有相同的名字、相同的参数个数、参数类型和相同的返回值，而且所有在远程接口中定义的抛出的异常也必须在匹配方法中定义；

16.2.2.2.3 组件类

对于一个Session Bean组件，开发者在业务（组件）接口中定义的的业务方法和在Home接口中定义的create方法，需要在组件类中对这些方法提供实现。EJB2.1 中的组件类必须实现javax.ejb.SessionBean接口，容器通过调用从javax.ejb.SessionBean中继承的管理方法，提供组件访问容器提供的服务的能力，并通过一些状态管理回调方法，向组件实例发送其生存周期中的关键事件的信息。在EJB3.0中的组件类则不必实现javax.ejb.SessionBean接口，而是通过依赖注入、JNDI查找、生存周期回调方法注解达到同样的功能。

一. 在组件类中使用注解

EJB3.0的组件类不再强制实现javax.ejb.SessionBean接口，而是通过注解实现同样的功能，从而避免了严格的隐式命名约定。在组件中常用的注解有以下几个：

(1). @Resource注解

在组件类中可以使用@Resource注解通知容器对上下文资源进行依赖注入，特别地，可以注入SessionContext实例：

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
  @Resource
  SessionContext ctx;
...
}
```

(2). @PostCreate注解

在组件类中可以使用@PostCreate注解标记初始化方法按照EJB3.0规范，容器将会在完成对会话Bean依赖注入之后，会话Bean的第一次方法被调用之前回调该方法。可以在该方法中对依赖注入后的会话Bean状态进行初试化。

(3). @Remove与@PreDestroy注解

在会话Bean的业务接口中使用@Remove注解标记客户端请求清除会话Bean的方法。该方法由客户端调用，若成功完成，容器将清除会话Bean。此注解适用于有状态会话Bean，无状态会话Bean的生存周期完全由容器管理，毋须在客户端显式删除。

在会话Bean组件类中可使用@PreDestroy注解标记容器清除会话Bean前的回调方法。容器将在会话Bean被清除前调用该方法。一般，实例可以在此方法中对实例占用的资源进行释放。

(4). @PrePassivate与@PostActivate注解

在实例将被容器钝化（Passivate）时，容器会调用被标记为PrePassivate的方法。同样，当实例将被再次激活（Activate）时，容器会调用被标记为PostActivate的方法。在实例钝化时，容器将会自动保存实例的状态信息；在激活实例时，容器也将恢复被保存的实例状态信息。一般的Session Bean都会忽略此事件，但是，对于使用某些不可串行化的资源，如与特定数据库的连接等，作为实例状态一部分的Session Bean，通常需要在ejbPassivate方法中释放资源，在ejbActivate方法中重新获取资源。请参考：“有状态Session Bean实例的钝化与激活”

二. javax.ejb.SessionBean接口

在EJB2.1规范中，javax.ejb.SessionBean是容器与组件实例之间的协议。通过实现此接口，组件提供容器访问组件的能力，实际上，容器不需要组件提供任何服务，容器通过javax.ejb.SessionBean中定义的接口方法访问组件实例，主要是为了给组件提供访问容器提供的服务的能力，并向组件实例发送通知其生存周期中重要事件的发生信息。

javax.ejb.SessionBean中定义了以下方法。

(1). setSessionContext

容器通过调用setSessionContext方法，将由容器维护的Bean实例的上下文（context）与Bean实例进行关联。一般，Session Bean实例的上下文被作为实例会话状态的一部分，被实例所保存。

(2). ejbRemove

在实例将被容器清除时，容器会调用此方法。一般，实例可以在此方法中对实例占用的资源进行释放。

(3). ejbPassivate与ejbActivate

在实例将被容器钝化（Passivate）时，容器会调用ejbPassivate方法。同样，当实例将被再次激活（Activate）时，容器会调用ejbActivate方法。对于使用某些不可串行化的资源，如与特定数据库的连接等，作为实例状态一部分的会话Bean，通常需要在ejbPassivate方法中释放资源，在ejbActivate方法中重新获取资源。请参考 “有状态Session Bean实例的钝化与激活”

三. SessionContext接口

对于会话Bean实例，容器将提供一个SessionContext对象，使实例可以访问由容器维护的实例的上下文环境。在SessionContext接口中，定义了如下方法：

- getEJBObject方法，返回Session Bean的远程接口；
- getEJBHome方法，返回Session Bean的远程Home接口；
- getEJBLocalObject，返回Session Bean的本地接口；
- getEJBLocalHome，返回Sesson Bean的本地Home接口；
- getCallerPrincipal方法，返回标识调用此实例的调用者的java.security.Principal对象；
- isCallerInRole方法，检查此Session Bean的调用者是否是某个特定的角色；
- setRollbackOnly方法，当前事务将被永久标记为回滚，不会被提交。只有容器管理事务的Session Bean可被允许使用此方法；
- getRollbackOnly方法，检查当前事务是否已被标记为回滚。例如，EJB实例可以通过此方法，判断是否继续在当前事务边界内进行计算。只有容器管理事务的Session Bean可被允许使用此方法；
- getUserTransaction方法，返回javax.transaction.UserTransaction接口。EJB实例可通过此接口对事务边界进行划分，并取得事务的状态。只有容器管理事务的Session Bean可被允许使用此方法；

四. 可选的SessionSynchronization接口

会话Bean的组件类可以选择是否实现javax.ejb.SessionSynchronization接口。此接口对会话Bean提供事务的同步通知。

- afterBegin通知，标志会话Bean实例一个新事务的开始。当包含在事务中第一个的业务方法被调用之前，容器将调用此方法，后续的业务方法调用将会存在于此事务上下文中；
- afterCompletion通知，标志一个会话Bean事务的提交操作结束，并通知实例提交操作是成功还是回滚；
- beforeCompletion通知，通知会话Bean实例事务将进行提交操作；

EJB3.0规范规定，如果会话Bean直接或间接地实现了SessionSynchronization接口，容器必须回调afterBegin,beforeCompletion与afterCompletion方法。若会话Bean没有实现SessionSynchronization接口，则容器不会调用这些方法。

五. **串行化的会话Bean方法调用**

容器将对每个会话Bean实例的方法调用进行串行化。Apusic应用服务器中的EJB容器支持会话Bean的多个实例并发执行；但是，每个会话Bean实例只会看到依次进行的方法调用，因此开发Session Bean时，不需要将其以可重入（reentrant）的方式进行编写。

客户端不能对有状态的会话Bean对象进行并发调用。当某个客户端对某特定的会话Bean实例的业务方法调用正在执行中，从相同或不同的客户端发出了另一个客户端调用，容器将对第二个客户端调用抛出java.rmi.RemoteException或javax.ejb.EJBException，这取决于第二个客户端调用是通过远程或本地组件接口进行调用。

六. **业务方法必须遵守的规则**

会话Bean的组件类中可以定义零到多个业务方法，其方法签名和命名必须遵守如下规则：

- 方法名是任意的，但不能使用"ejb"开头，以免与EJB组件架构中的容器管理回调方法发生冲突，如ejbPassivate方法;
- 业务方法必须被声明为public方法；
- 业务方法不能被声明为final或static；
- 如业务方法是对应于会话Bean远程接口中定义的业务方法，则方法参数和返回值必须是合法的RMI/IIOP类型；
- 可以抛出任意应用级异常；

💡	**注意**
	EJB1.0允许业务方法抛出java.rmi.RemoteException，以指出非应用级的异常。在EJB1.1和EJB2.1兼容的Enterprise Bean开发中，这种方式不建议使用（deprecated），EJB2.1兼容的Enterprise Bean开发中，不应从业务方法抛出java.rmi.RemoteException。

七. **代码范例**

以下是一个会话Bean组件类的范例代码，其对应的组件接口和Home接口，请参考本章中关于组件接口和Home接口中的范例部分：

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

public class CartBean implements SessionBean {
  String customerName;
```

```
String customerId;
Vector contents;

public void ejbCreate(String person) throws CreateException {
  if (person == null) {
    throw new CreateException("Null person not allowed.");
  } else {
    customerName = person;
  }
  customerId = "0";
  contents = new Vector();
}

public void ejbCreate(String person, String id) throws CreateException {
  if (person == null) {
    throw new CreateException("Null person not allowed.");
  } else {
    customerName = person;
  }
  IdVerifier idChecker = new IdVerifier();
  if (idChecker.validate(id)) {
    customerId = id;
  } else {
    throw new CreateException("Invalid id: " + id);
  }
  contents = new Vector();
}

public void addBook(String title) {
  contents.addElement(title);
}

public void removeBook(String title) throws BookException {
  boolean result = contents.removeElement(title);
  if (result == false) {
    throw new BookException(title + " not in cart.");
  }
}

public Vector getContents() {
```

```
    return contents;
  }

  public CartBean() {}

  public void ejbRemove() {}

  public void ejbActivate() {}

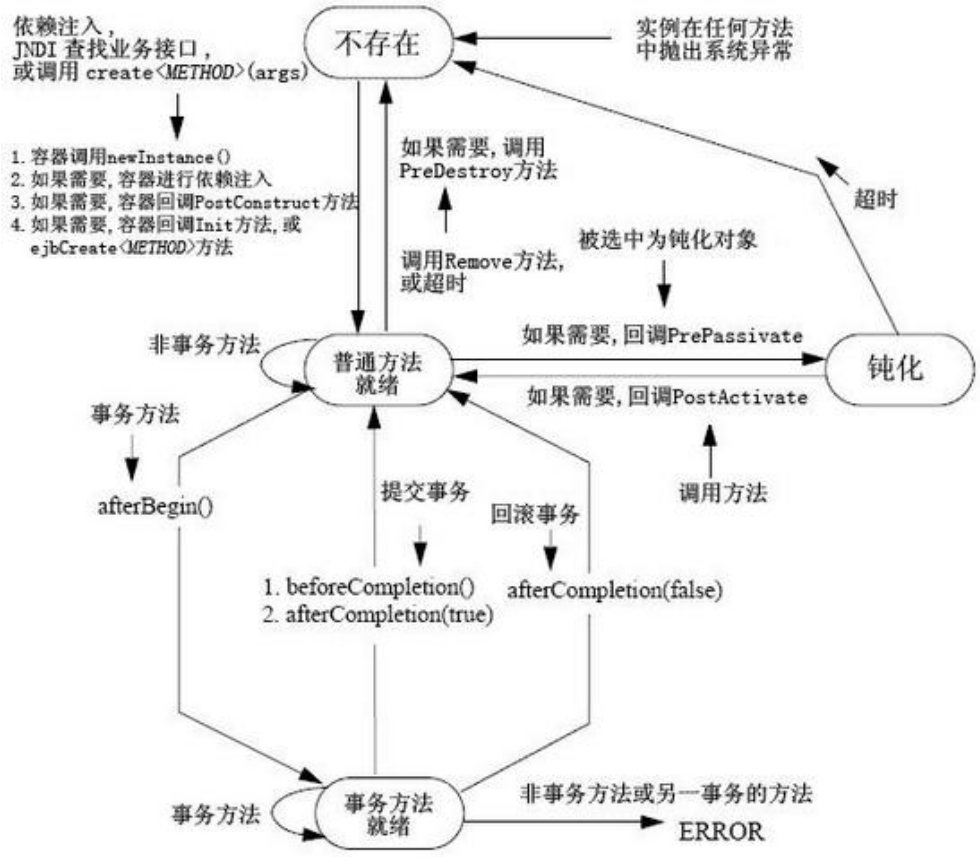
  public void ejbPassivate() {}

  public void setSessionContext(SessionContext sc) {}
}
```

16.2.2.3 生存周期

16.2.2.3.1 有状态SESSION BEAN的生存周期

有状态Session Bean的生存周期



上图表示有状态Session Bean的生存周期。

有状态会话Bean实例的生命周期开始于用户端通过依赖注入或JNDI查找获得一个有状态会话Bean实例的引用，或者用户端调用了会话Bean的Home 接口中的create<METHOD>方法。这将通知容器调用会话Bean类的newInstance方法创建一个新的会话Bean实例。

然后容器将注入SessionContext实例（若可用），并执行其它由元数据注解标记的或在部署描述符中设定的依赖注入。然后容器将调用会话Bean的PostConstruct生存周期回调方法（若已定义）。如果会话Bean遵照EJB2.1规范，则容器调用实例中匹配的ejbCreate<METHOD>或Init方法。最后，容器返回会话Bean的对象引用。现在实例进入就绪状态，客户端可以调用其业务方法。

根据会话bean元数据注解中的事务属性或部署描述文件中的设定，以及客户端调用所关联的事务上下文，业务方法将会在一个特定的事务上下文环境或一个未指定的业务上下文环境中执行（在图中分别以"事务方法"与"非事务方法"表示）。

容器的缓冲算法决定会话Bean实例是否需要移出内存（例如采取最近最久未使用算法）。容器先回调Bean实例的PrePassivate生存周期回调方法（若已定义）。然后，容器把实例的状态信息保存到二级存储设备中，会话Bean进入钝化状态。在事务之中的会话Bean不能被钝化。

当会话Bean处于钝化状态，容器可能会在会话Bean超时后清除该实例，超时信息在部署描述文件中设置。此时所有对于该实例的引用都将失效。如果客户端试图通过业务接口调用任一方法，容器将抛出javax.ejb.NoSuchEJBException异常。如果使用EJB2.1规范，则容器分别为远程客户端抛出java.rmi.NoSuchObjectException异常，为本地客户端抛出javax.ejb.NoSuchObjectLocalException异常。

如果客户端调用了被标记为Remove的方法，或者home接口或组件接口中的remove方法，容器将在该方法成功完成后回调被标记为PreDestroy的生存周期回调方法（若已定义）。这将终结会话Bean实例的生存周期。此时所有对于该实例的引用都将失效。如果客户端试图通过业务接口调用任一方法，容器将抛出javax.ejb.NoSuchEJBException异常。如果使用EJB2.1规范，则容器分别为远程客户端抛出java.rmi.NoSuchObjectException异常，为本地客户端抛出javax.ejb.NoSuchObjectLocalException异常。要注意即使客户端没有显式调用Remove方法，容器也会在EJB对象实例超时后主动调用其PreDestroy方法并清除该实例。

如果客户端调用了钝化状态的会话Bean的方法，容器将激活该实例。此时容器将先从二级存储设备中恢复实例的状态信息，并调用实例的PostActivte方法（若已定义）。

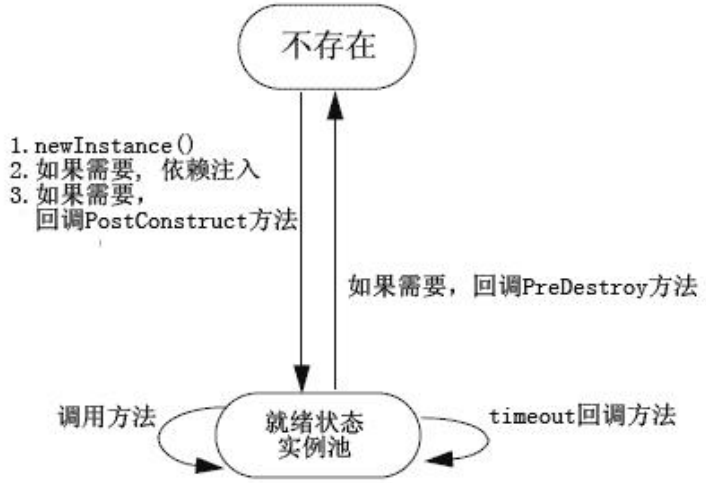
如果Remove方法成功完成，或Remove方法中抛出了应用异常而retainIfException为假，或抛出了系统异常，SessionSynchronization方法将不会被调用。如果抛出了应用异常而retainIfException为真，则该会话bean实例既不会被清除也不会被丢弃，SessionSynchronization方法（若已定义）将会在事务结束时被调用。

客户端代码对有状态Session Bean生存周期的控制只能创建（使用依赖注入、JNDI查找或create<METHOD>方法）和清除（使用Remove方法）。其他方法由EJB容器进行调用。

16.2.2.3.2 无状态SESSION BEAN的生存周期

因无状态Session Bean不会进行被钝化操作，因此，其生存周期只有两个阶段，不存在和就绪状态。下图表示无状态Session Bean的生存周期：

无状态Session Bean的生存周期



16.2.2.4 Message-driven Bean

16.2.2.5 Message-driven Bean

Message-driven Bean是JMS消息驱动的Java EE™平台服务器端组件，具备无状态、支持事务的特点。当从JMS队列（Queue）或主题（Topic）中接收到JMS消息后，由容器对组件进行调用。一般，可理解为消息的监听器（Listener）及接收者（Consumer）。

有关JMS消息系统，请参阅"消息服务开发"。

Message-driven Bean组件对于客户端是不可见的。客户端如希望调用封装在组件中的业务逻辑，只能通过向组件监听的JMS队列（Queue）或主题（Topic）发送消息，然后容器以事件的形式向组件实例发送消息，组件实例根据消息的内容调用相应的业务逻辑或其他组件。

因此，任何向组件监听的特定JMS队列（Queue）或主题（Topic）发送JMS消息的客户端，即可视为Message-driven Bean的客户端。

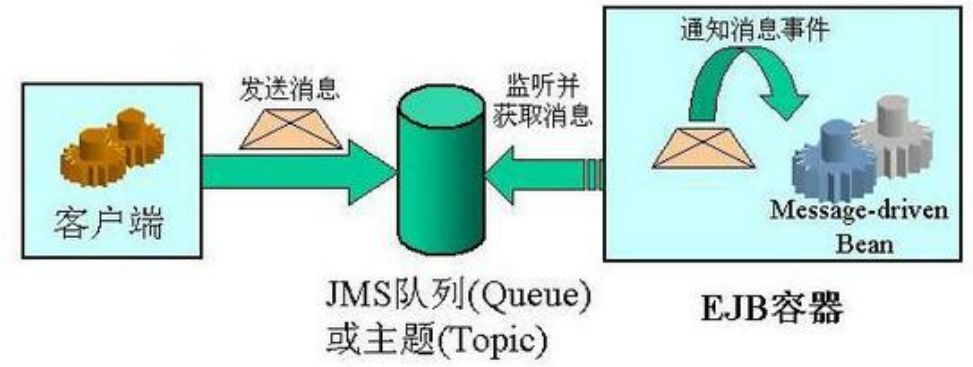
Message-driven Bean组件模型不具备会话状态，也就是说，当组件实例在没有对客户端的JMS消息提供处理的时候，所有的实例间没有差别。

Message-driven Bean的运行和客户端的运行是异步的。同时，对于客户端不可见。其实例由容器创建，其生存周期由容器控制。

16.2.2.5.1 MESSAGE-DRIVEN BEAN与EJB容器、客户端、消息系统

下图是EJB容器、客户端、消息系统与EJB之间的关系：

EJB容器、客户端、消息系统与EJB之间的关系



客户端发送消息到JMS消息系统中的队列或主题。Message-driven Bean在部署到容器中时，指定的队列或主题，容器对其进行监听；当容器从队列或主题中接收到消息之后，将消息作为事件的一部分通知容器中相应的Message-driven Bean实例。

16.2.2.6 组件模型单元

Message-driven Bean由容器控制其生存周期，容器提供安全、并发、事务等等服务，对于客户端来说，Message-driven Bean是不可见的。因此，Message-driven Bean不同于Session Bean和Entity Bean，不具有组件接口和Home接口。

一般，在Apusic应用服务器启动之后，通过开发者在部署描述中指定的消息或队列，容器即创建Message-driven Bean的实例，对队列或主题进行监听并接受消息。

Message-driven Bean组件模型包含两个单元，即组件类和部署描述。下面分别对开发这些单元时，涉及的普遍过程、规则及注意事项进行描述。

16.2.2.6.1 组件类

一. javax.ejb.MessageDrivenBean接口

EJB2.1 规范中的Message-driven Bean组件中的组件类必须实现MessageDrivenBean接口。EJB3.0规范不强制Message-driven Bean实现该接口，而通过@MessageDriven注解进行标记并通过依赖注入与注解实现类似功能。

MessageDrivenBean接口中定义了两个容器管理回调的方法：

- setMessageDrivenContext 方法，容器创建Bean实例后，容器将调用该方法将由容器维护的Bean实例的上下文（context）与Bean实例进行关联。在EJB3.0规范中，可使用@Resource注解通知容器注入MessageDrivenContext实例。
- ejbRemove方法，在实例被容器清除时，容器将调用此方法。一般，实例会在此方法中对实例占用的资源进行释放。在EJB3.0规范中，可使用@PreDestroy注解标记此方法。

二. javax.jms.MessageListener接口

Message-driven Bean组件中的组件类必须实现MessageListener接口。

在消息到达Message-driven Bean指定的监听队列或主题时，容器将调用javax.jms.MessageListener接口中定义的onMessage方法。开发者在此方法中提供对消息进行处理的业务逻辑。

Session Bean和Entity Bean不可实现javax.jms.MessageListener接口。

三. javax.ejb.MessageDrivenContext接口

容器将提供一个MessageDrivenContext对象，使实例可以访问由容器维护的实例的上下文环境。在此接口中，定义了如下方法：

- getEJBHome、getEJBLocalHome方法，从EJBContext接口继承的方法，Message-driven Bean实例不可调用此方法；
- getCallerPrincipal方法，从EJBContext接口继承的方法，Message-driven Bean实例不可调用此方法；
- isCallerInRole方法，从EJBContext接口继承的方法，Message-driven Bean实例不可调用此方法；
- setRollbackOnly方法，当前事务将被永久标记为回滚，不会被提交。只有容器管理事务的Message-driven Bean可被允许使用此方法；
- getRollbackOnly方法，检查当前事务是否已被标记为回滚。例如，EJB实例可以通过此方法，决定是否继续在当前事务边界内继续进行计算。只有容器管理事务的Message-driven Bean可被允许使用此方法；
- getUserTransaction方法，返回javax.transaction.UserTransaction接口。EJB实例可通过此接口对事务边界进行划分，并取得事务的状态。只有容器管理事务的Message-driven Bean可被允许使用此方法；

四. 串行化的调用

Apusic应用服务器中的EJB容器支持Message-driven Bean的多个实例的并发运行，但是每个实例只“看到”一个串行的方法调用过程，因此开发Message-driven Bean时，不需要将其以可重入（reentrant）的方式进行编写。

五. 消息处理的并发

Apusic应用服务器允许Message-driven Bean的多个实例并发执行，提供对流（Stream）消息并发处理。

六. Message-driven Bean方法的事务上下文

onMessage方法在何种事务范围内被调用，取决于部署描述中指定的事物属性，如Bean被指定使用容器管理的事务的方式，则必须将事务属性设置为“Required”或“NotSupported”。

当Bean采用Bean管理的事务的方式，即使用javax.transaction.UserTransaction接口进行事务划分时，导致Bean实例被调用的消息接收操作并非是事务中的一部分。如果希望消息接收操作是事务中的一部分，则Bean必须使用容器管理事务的方式，并且设置事务属性为 “Required”。

七. 消息接收确认（Message Acknowledgement）

Message-driven Bean不能使用JMS API中提供的消息接收确认操作。消息接收确认操作由容器自动完成。如Bean采用了容器管理事务的方式，则消息接收确认操作作为事务提交的一部分自动进行。如使用了Bean管理事务的方式，消息接收确认操作不能作为事务提交的一部分，开发者可通过在部署描述中的acknowledge-mode元素指定消息接收确认操作的方式为AUTO_ACKNOWLEDGE或DUPS_OK_ACKNOWLEDGE，如未指定acknowledge-mode元素，容器将使用AUTO_ACKNOWLEDGE方式进行消息接收确认操作。

八. 指定队列（Queue）或主题（Topic）

当Message-driven Bean被部署到容器时，必须关联到某个消息队列（Queue）或主题（Topic），以便容器对此队列或主题进行监听。

开发者可通过@MessageDriven注解的mappedName属性或部署描述文件中的message-driven-destination元素指定关联的队列或主题。

如Bean关联的是一个消息主题，则通过部署描述中的subscription-durability元素指定对队列进行的是持久还是非持久订阅，如此元素未指定，则使用非持久订阅的方式。

九. 异常处理

Message-driven Bean中的onMessage方法不能声明抛出java.rmi.RemoteException异常。

一般来说，Message-driven Bean在运行期间不应向容器抛出RuntimeException异常。RuntimeException异常是指会导致Message-driven Bean进入"不存在"状态的非应用级异常。如果Bean使用了Bean-managed事务并抛出了RuntimeException异常，容器不应确认收到了这条消息。

从发信端看来，收信端一直存在，若发信端继续对该目的地发送消息，容器会自动把消息转向到其他Message-driven Bean实例。

十. 遗漏的PreDestroy调用

在系统发生异常的情况下，不能保证容器总会调用Bean的PreDestroy方法，因此，如果Bean在PostConstruct方法中打开了一些资源，并在PreDestroy方法中释放这些资源，在这种情况下，则这些资源不能被释放。

鉴于以上原因，使用Message-driven Bean的应用需要提供一种机制，以便周期性的清除这些未释放的资源占用。

16.2.2.6.2 必须遵守的规则

在开发Message-driven Bean时，开发者必须遵守如下规则：

一. 组件类

- 使用EJB2.1规范时，必须间接或直接实现javax.ejb.MessageDrivenBean接口；使用EJB3.0规范时，可改为使用@MessageDriven注解对组件类进行标记。
- 必须间接或直接实现javax.jms.MessageListener接口；
- 类必须声明为public，不可被声明为final或abstract类；
- 必须拥有一个无参数的public构造函数（constructor）；
- 类不能定义finalize()方法；
- 在原EJB2.1规范中，类必须实现ejbCreate()方法用来创建组件实例，在EJB3.0中，这一要求已被移除了。EJB3.0的兼容规则规定，如果Message-driven Bean类实现了ejbCreate()方法，将看作被@PostConstruct注解标记的方法处理。此时若同时使用@PostConstruct注解，则只能标记ejbCreate()方法。

二. onMessage方法

- 方法必须被声明为public；
- 方法不能被声明为final或static；
- 返回值必须为void；
- 方法只能有一个javax.jms.Message类型的参数；
- 不能抛出java.rmi.RemoteException异常

- 运行期间一般来说不应抛出RuntimeException。请参考："异常处理"

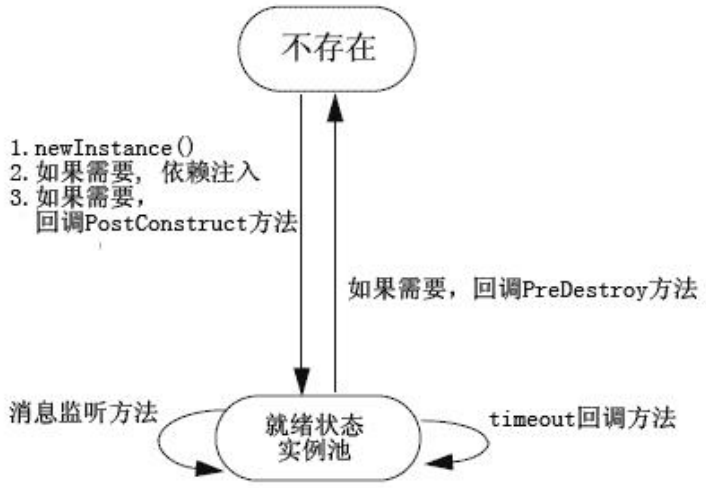
三. ejbRemove方法

- 方法名必须是ejbRemove;
- 方法必须被声明为public;
- 方法不能被声明为final或static;
- 返回值必须为void;
- 方法不能有参数;
- 不能抛出java.rmi.RemoteException异常。
- 在EJB3.0规范中，可使用@PreDestroy注解实现同样效果。若实现javax.ejb.MessageDrivenBean接口同时使用注解，则只能把ejbRemove()方法注解为@PreDestroy

16.2.2.7 生存周期

下图表示Message-driven Bean的生存周期。

Message-driven Bean的生存周期



16.2.3 Entity Bean

摘要

本节描述符合EJB2.1规范的Entity Bean组件模型。EJB3.0规范对Java持久框架作出了重大改动，形成了相对独立的Java Persistence API（JPA），新的EJB3.0组件模型不再包含Entity Bean，由轻量级的Entity Class取而代之。EJB3.0规范同时规定了对EJB2.1组件模型兼容，因此本节及后续两节所描述的Entity Bean在Apusic应用服务器中仍然可用。但应注意EJB2.1的Entity Bean并不支持依赖注入、拦截器、元数据注解等新特性。对于新的JPA将另文阐述。

16.2.3.1 概览

EJB 规范中定义了Entity Bean编程模型和EJB容器，使开发者可以摆脱种种底层细节，集中处理业务逻辑，在EJB规范中定义的许多特性，如容器管理关系、EJB QL等，更加简化了应用业务对象的开发，同时，数据库与应用程序之间的耦合层度也因此降到了更低的水平。Apusic应用服务器提供了对EJB2.1规范的完全支持，同时，在不影响与J2EE™平台相关规范兼容的情况下，提供了很多增进EJB组件开发、部署、维护、移植等等方面的增值特性。

在本章中，我们将讨论Entity Bean这种编程模型所提供的特性，以及Apusic应用服务器在这方面所具有的优势。

16.2.3.1.1 数据的对象视图

多层企业应用中，通常有逻辑层和数据层之分，逻辑层中的逻辑组件用于表示特定的业务逻辑或满足特定的业务需求，逻辑组件中一般只封装逻辑，而业务数据则通过数据层中的数据组件提供表示。

业务数据通常以数据库表的方式保存在关系型数据库中，在EJB编程模型成为企业应用开发的标准之前，开发者一般通过JDBC或其他数据库访问机制，将数据映射成为内存中的业务对象，并提供保证这两个数据拷贝之间的一致性的方法，即业务对象知道如何从数据库中获得自身的状态，当自身状态改变时，同步数据库中的对应数据，对于关系型数据，往往需要更复杂的控制。

这样，数据层中的数据组件往往显得非常臃肿，代码中充斥大量的底层服务处理，不能提供业务实体清晰的面向对象表示，即由紧凑的状态和方法构成的对象，而且给开发和维护带来极大的不便。

通过Entity Bean组件模型，开发者可以快速地建立起紧凑、灵活而且具有分布式特性的业务数据的对象视图，通过类似于标准的Java语言中创建、清除对象实例、调用实例的方法，对组件进行操作，组件中只需提供应用的业务逻辑。

16.2.3.1.2 组件结构

Entity Bean由以下几部分构成：

- Home接口，客户端通过此接口对Entity Bean对象进行创建、清除、查找等操作；
- 组件接口，客户端通过此接口访问Entity Bean对象，访问对象状态及执行对象业务方法；
- 组件类，开发人员在此类中提供对以上两个接口中定义的方法的实现；
- 部署描述，提供Entity Bean的事务属性、安全等配置信息。

16.2.3.1.3 PRIMARY KEY

与关系型数据库中的数据类似，每个Entity Bean实例也具有一个标识数据的唯一性的Primary Key对象，Primary Key对象可以具有多个属性。属性可以是任意用于说明此Entity Bean实例的的数据。某些情况下，用于表示复杂关系的Entity Bean，Primary Key甚至可以是整个Entity Bean。

可以通过定义Entity Bean组件中的Primary Key类，来为组件提供可以任何满足要求的Primary Key，但必须保证此Primary Key符合Java编程语言中的串行化要求，即Primary Key必须是合法的RMI类型。

16.2.3.2 客户端

Entity Bean的客户端按照客户端与Entity Bean所处的JVM是否相同，分为远程客户和本地客户。

16.2.3.2.1 JNDI

由于Entity Bean运行于EJB容器中，由容器提供对Entity Bean实例的生存周期管理，客户端不能直接访问容器中的Entity Bean，容器使客户端可以通过JNDI访问Entity Bean的Home接口，客户端通过Home接口取得对Entity Bean对象的引用，通过引用对容器中的Entity Bean实例进行访问。

16.2.3.2.2 远程客户

Entity Bean的远程客户端通过Entity Bean的远程Home接口和远程接口访问Entity Bean对象。

Entity Bean的远程客户端与运行客户端的JVM是否与Entity Bean所处JVM是否相同无关，相同或不同JVM中的客户端都可通过远程Home接口和远程接口访问Entity Bean对象。只是远程客户端对Entity Bean进行调用时，其调用通过网络传播，参数与返回值必须是合法的RMI类型，并且都按值传递。

由于网络延迟的原因，客户端通过远程访问比本地访问效率上要差一些，但是，由于远程访问与客户端的位置无关，因此，可以提供Entity Bean的可分布特性，适合粗粒度的对象表示。

Entity Bean的远程客户端可以是用Java语言编写的任意程序类型，如Java应用、applet、Servlet等等，也可以是非Java的客户端。

16.2.3.2.3 本地客户

Entity Bean的本地客户通过本地Home接口与本地接口访问Entity Bean对象。

Entity Bean的本地客户端必须位于同一个JVM中。

Entity Bean的本地客户端在对Entity Bean进行调用时，按照标准Java语言中的对象调用方式，其参数和返回值按引用传递。

因为不涉及网络延迟的因素，一般，通过本地访问，效率要高一些，不过，组件的分布式特性就受到了影响。通过定义本地和远程的访问接口，Entity Bean可以同时提供两种访问方式，不过，通常只提供二者之一。

具有容器管理关系的EJB，必须提供本地访问接口。

16.2.3.3 容器与Entity Bean

从Entity Bean实例的产生到清除，它都运行于EJB容器中，容器中可以部署多个Entity Bean，容器提供客户端对Entity Bean实例的访问机制，容器为部署的在其中的Entity Bean提供安全、并发、持久等等服务。对于Entity Bean的客户端而言，容器是透明的。

Entity Bean作为业务数据在应用系统中的对象表示，多个客户通常会并发地访问Entity Bean对象，开发者不必提供对并发访问的支持，通过实例池与事务管理器，EJB容器会保证实例对客户端请求进行进行高效的响应，并保证实例状态与数据库的同步。如果开发者使用了容器管理持久性的Entity Bean，则如何取得与数据库的连接、如何使用SQL语句操作数据、如何利用资源等等，都可以完全由容器进行处理。开发者只需通过部署描述，对组件的事务、安全等进行声明即可。

16.2.3.3.1 通过JNDI定位远程HOME接口

客户端可通过JNDI定位Entity Bean的远程Home接口，JNDI命名空间中可以包含多个部署在容器中的Entity Bean的远程Home接口。

如下代码段示意如何通过JNDI定位Entity Bean的远程Home接口：

```
Context initialContext = new InitialContext();

AccountHome accountHome = (AccountHome)
javax.rmi.PortableRemoteObject.narrow(

initialContext.lookup(“java:comp/env/ejb/accounts”),

AccountHome.class
);
```

16.2.3.3.2 通过JNDI定位本地HOME接口

客户端通过JNDI定位Entity Bean的本地Home接口与定位远程Home接口的方法非常类似，只是没有牵涉到远程访问的API。

如下代码段示意如何通过JNDI定位Entity Bean的本地Home接口：

```
Context initialContext = new InitialContext();

AccountHome accountHome = (AccountHome) initialContext.lookup(

“java:comp/env/ejb/accounts”

);
```

16.2.3.4 组件模型

16.2.3.4.1 HOME接口

根据Entity Bean是否提供远程或本地访问，Entity Bean组件必须提供本地或远程的Home接口。通过JNDI，容器使Home接口可被客户端访问。

一. 远程Home接口

通过Entity Bean的远程Home接口，开发者向Entity Bean的客户端提供进行如下操作的方法：

- 创建新的Entity Bean实例；
- 查找已存在的Entity Bean实例；
- 清除Entity Bean实例；
- 执行Home接口中的业务方法；
- 取得Entity Bean的javax.ejb.EJBMetaData接口，javax.ejb.EJBMetaData接口用于表示此Entity Bean的信息，降低客户端和服务器的绑定程度和用于支持客户端脚本；
- 取得Home接口句柄（Handler）,此句柄可串行化（serialization）到持久存储中，然后，可在其他虚拟机中，从持久存储中，对此串行化对象进行解串行化（deserialization），取得对此远程Home接口句柄的引用;

Entity Bean的远程Home接口必须扩展（extends）javax.ejb.EJBHome接口，并遵循Java语言的远程接口标准。

(1). create方法

Entity Bean的远程Home接口可定义零个或多个create方法，每个create方法提供一种创建Entity Bean实例的方式，create方法中的参数一般被用于初始化Entity Bean实例。每个create方法必须以“create”作为前缀；

远程Home接口中定义的create方法的返回值必须是Entity Bean的远程接口，抛出语句必须包含java.rmi.RemoteException和javax.ejb.CreateException异常，同时可包含任意的应用级的异常。

以下代码段是远程Home接口中三个create方法的范例：

```
public interface AccountHome extends javax.ejb.EJBHome {
  public Account create(String firstName, String lastName, double
initialBalance) throws RemoteException, CreateException;

  public Account create(String accountNumber, double initialBalance) throws
RemoteException, CreateException, LowInitialBalanceException;

  public Account createLargeAccount(String firstname, String lastname, double
initialBalance) throws RemoteException, CreateException;
...
}
```

以下是客户端如何通过create方法创建Entity Bean实例的代码范例：

```
AccountHome accountHome = ...;

Account account = accountHome.create("John", "Smith", 500.00);
```

(2). finder方法

通过Entity Bean的远程Home接口，可以定义一到多个finder方法。所谓finder方法，即是开发者提供给客户端的方法，用于查找某个Entity Bean对象或包含多个Entity Bean对象的集合。远程Home接口中定义的每个finder方法对应一种查找的方式。finder方法的名称必须以"find"前缀开始，如：findLargeAccounts(...)。方法中定义的参数，被Entity Bean组件类中的对应实现方法用于定位需要查找的Entity Bean对象或对象集合。

finder方法的返回值类型必须是Entity Bean的远程接口对象，或是只包含远程接口对象的集合（Collection）。

每个远程Home接口中定义的finder方法的抛出语句必须包含对java.rmi.RemoteException异常和javax.ejb.FinderException异常的抛出。

远程Home接口必须包含一个findByPrimaryKey(primaryKey)方法，客户端可通过此方法使用Entity Bean的Primary Key对Entity Bean对象进行查找。此方法的唯一参数必须与Entity Bean的Primary Key为同一类型，其返回值必须是Entity Bean的远程接口类型。

以下代码范例定义了一个findByPrimaryKey方法：

```
public interface AccountHome extends javax.ejb.EJBHome {
...
  public Account findByPrimaryKey(String AccountNumber) throws
RemoteException, FinderException;
}
```

以下代码范例是客户端调用finder方法的方式：

```
AccountHome = ...;

Account account = accountHome.findByPrimaryKey("100-3450-3333");
```

(3). Remove方法

javax.ejb.EJBHome 接口中定义的方法，由于Entity Bean组件的远程Home接口扩展（extends）javax.ejb.EJBHome接口，因此，客户端可通过调用Entity Bean的远程Home接口中的remove方法清除容器Entity Bean实例。

(4). Home方法

作为应用中业务数据的对象表示，Entity Bean通常表示一类业务实体，如雇员信息、订单、账户等等，通常这些实体都会涉及针对所有或部分已存在数据的操作，如对所有雇员的某些信息进行统计，或者对所有雇员增加花红等等，这些操作不涉及单个的Entity Bean实例。

Entity Bean组件模型中，可以通过在远程Home接口中定义Home方法，提供对不涉及单个特定的Entity Bean实例的操作。

在远程Home接口中定义的Home方法必须抛出java.rmi.RemoteException异常，可抛出任意应用级异常。同时，其参数类型和返回值类型必须是合法的RMI类型。

下面是两个Home方法的范例代码：

```
public interface EmployeeHome extends javax.ejb.EJBHome {
...
  // this method returns a living index depending on
  // the state and the base salary of an employee:
  // the method is not specific to an instance
  public float livingIndex(String state, float Salary) throws
RemoteException;

  // this method adds a bonus to all of the employees
  // based on a company profit-sharing index
  public void addBonus(float company_share_index) throws RemoteException,
ShareIndexOutOfRangeException;
...
}
```

二. 本地Home接口

通过Entity Bean的本地Home接口，开发者向Entity Bean的客户端提供进行如下操作的方法：

- 创建新的Entity Bean实例；
- 查找已存在的Entity Bean实例；

- 清除Entity Bean实例；
- 执行Home接口中的业务方法；

Entity Bean的本地Home接口必须扩展（extends）javax.ejb.EJBLocalHome接口

(1). create方法

Entity Bean的本地Home接口可定义零个或多个create方法，每个create方法提供一种创建Entity Bean实例的方式，create方法中的参数一般被用于初始化Entity Bean实例。每个create方法以“create”作为前缀；

本地Home接口中定义的create方法的返回值必须是Entity Bean的远程接口，抛出语句不可包含java.rmi.RemoteException异常，可包含任意的应用级的异常，必须包含javax.ejb.CreateException异常。

以下代码段是本地Home接口中三个create方法的范例：

```
public interface AccountHome extends javax.ejb.EJBLocalHome {
  public Account create(String firstName, String lastName, double
initialBalance) throws CreateException;

  public Account create(String accountNumber, double initialBalance) throws
CreateException, LowInitialBalanceException;

  public Account createLargeAccount(String firstname, String lastname, double
initialBalance) throws CreateException;
...
}
```

以下是客户端如何通过create方法创建Entity Bean实例的代码范例：

```
AccountHome accountHome = ...;

Account account = accountHome.create(“John”, “Smith”, 500.00);
```

(2). finder方法

通过Entity Bean的本地Home接口，可以定义一到多个finder方法。所谓finder方法，即是开发者提供给客户端的方法，用于查找某个Entity Bean对象或包含多个Entity Bean对象的集合。本地Home接口中定义的每个finder方法对应一种查找的方式。finder方法的名称必须以“find”前缀开始，如：findLargeAccounts(...)。方法中定义的参数，被Entity Bean组件类中的对应实现方法用于定位需要查找的Entity Bean对象或对象集合。

finder方法的返回值类型必须是Entity Bean的本地接口对象，或是只包含本地接口对象的集合（Collection）。

每个本地Home接口中定义的finder方法的抛出语句必须包含javax.ejb.FinderException异常的抛出，不可包含java.rmi.RemoteException异常。

本地Home接口必须包含一个findByPrimaryKey(primaryKey)方法，客户端可通过此方法使用Entity Bean的Primary Key对Entity Bean对象进行查找。此方法的唯一参数必须与Entity Bean的Primary Key为同一类型，其返回值必须是Entity Bean的本地接口类型。

以下代码范例定义了一个findByPrimaryKey方法：

```
public interface AccountHome extends javax.ejb.EJBLocalHome {
...
  public Account findByPrimaryKey(String AccountNumber) throws
FinderException;
}
```

以下代码范例是客户端调用finder的方式：

```
AccountHome = ...;

Account account = accountHome.findByPrimaryKey("100-3450-3333");
```

(3). Remove方法

javax.ejb.EJBLocalHome 接口中定义的方法，由于Entity Bean组件的本地Home接口扩展（extends）javax.ejb.EJBLocalHome接口，因此，客户端可通过调用Entity Bean的本地Home接口中的remove方法清除容器Entity Bean实例。

三. Home方法

作为应用中业务数据的对象表示，Entity Bean通常表示一类业务实体，如雇员信息、订单、账户等等，通常这些实体都会涉及针对所有或部分已存在数据的操作，如对所有雇员的某些信息进行统计，或者对所有雇员增加花红等等，这些操作不涉及单个的Entity Bean实例。

Entity Bean组件模型中，可以通过在本地Home接口中定义Home方法，提供对不涉及单个特定的Entity Bean实例的操作。

每个Home方法的抛出语句可以包含任意应用级异常的抛出，不能抛出java.rmi.RemoteException异常。

下面是两个Home方法的范例代码：

```
public interface EmployeeHome extends javax.ejb.EJBLocalHome {
...
  // this method returns a living index depending on
  // the state and the base salary of an employee:
  // the method is not specific to an instance
  public float livingIndex(String state, float Salary);

  // this method adds a bonus to all of the employees
  // based on a company profit-sharing index
  public void addBonus(float company_share_index) throws
```

```
ShareIndexOutOfRangeException;
...
}
```

16.2.3.4.2 PRIMARY KEY

同一个Entity Bean组件的每个实例，都拥有一个用于标识实例的同一类型的Primary Key，如果两个Entity Bean对象拥有相同的Home接口，拥有相同的Primary Key，即可认为这两个Entity Bean对象相等。

Entity Bean的Primary Key可以是任何合法的RMI类型。每个Entity Bean都必须指定其唯一的Primary Key的类型，不同的Entity Bean可以使用相同的primary key类型。客户端可通过Entity Bean实例的组件接口引用，调用getPrimaryKey()方法，取得实例关联的Primary Key对象。

在整个Entity Bean实例的生存周期中，其关联的Primary Key对象不会发生改变，即在生存周期内，客户端调用实例的getPrimaryKey()方法返回的Primary Key对象是相同的。如果实例同时拥有远程接口和本地接口，则这两个接口的getPrimaryKey()方法返回的Primary Key对象也是相同的。

因此，客户端除了可以使用组件接口中的isIdentical方法判断两个Entity Bean实例是否相等之外，也可使用实例关联的Primary Key对象的equals方法进行判断。

下面是使用isIdentical方法判断两个对Entity Bean实例的引用是否相等的代码范例：

```
Account acc1 = ...;

Account acc2 = ...;

if (acc1.isIdentical(acc2)) {

//acc1 and acc2 are the same entity object
} else {

//acc2 and acc2 are different entity objects
}
```

知道Entity Bean实例的Primary Key的客户端，可以使用Home接口中定义的findByPrimaryKey方法，取得对Entity Bean实例组件接口的引用。

	**注意**
	当判断两个Entity Bean实例引用是否相等，不可使用“==”运算符，只可使用Primary Key对象的equals方法或组件接口的isIdentical方法进行判断。

16.2.3.4.3 组件接口

客户端通过Entity Bean提供的组件接口访问Entity Bean对象。开发者必须为组件提供组件接口，并在组件接口中对客户所能调用的业务方法进行定义。

根据Entity Bean是否提供远程或本地访问，组件接口分为远程接口和本地接口。

一. 远程接口

Entity Bean组件中的远程接口必须扩展（extends）javax.ejb.EJBObject接口。并在其中定义可供客户端调用业务方法。业务方法的抛出语句必须包含java.rmi.RemoteException异常，可包含任意的应用级异常。

下面是Entity Bean组件远程接口的范例代码：

```
public interface Account extends javax.ejb.EJBObject {
  void debit(double amount) throws java.rmi.RemoteException,
InsufficientBalanceException;

  void credit(double amount) throws java.rmi.RemoteException;

  double getBalance() throws java.rmi.RemoteException;
}
```

javax.ejb.EJBObject提供为客户端提供对Entity Bean对象引用进行如下操作的方法：

- 取得Entity Bean对象的远程Home接口；
- 清除Entity Bean对象；
- 取得Entity Bean对象的句柄；
- 取得Entity Bean对象的Primary Key。

容器为以上操作提供实现，开发者只需在Entity Bean的组件类中提供对远程接口中定义的业务方法的实现。

二. 本地接口

Entity Bean组件中的本地接口必须扩展（extends）javax.ejb.EJBLocalObject接口。并在其中定义可供客户端调用的业务方法。业务方法的抛出语句不能包含java.rmi.RemoteException异常，可包含任意的应用级异常。

下面是Entity Bean组件本地接口的范例代码：

```
public interface Account extends javax.ejb.EJBLocalObject {
  void debit(double amount) throws InsufficientBalanceException;

  void credit(double amount);
```

```
    double getBalance();
}
```

javax.ejb.EJBLocalObject提供为客户端提供对Entity Bean对象引用进行如下操作的方法：

- 取得Entity Bean对象的本地Home接口；
- 清除Entity Bean对象；
- 取得Entity Bean对象的Primary Key。

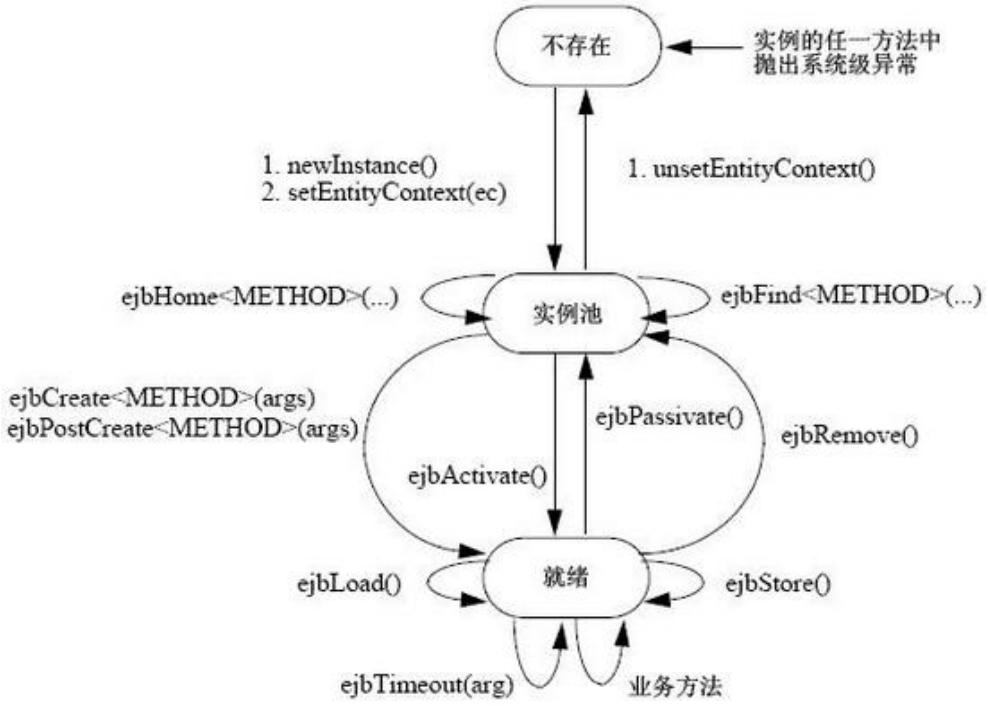
容器为以上操作提供实现，开发者只需在Entity Bean的组件类中提供对本地接口中定义的业务方法的实现。

16.2.3.4.4 组件类

由于Entity Bean具有两种不同的持久类型，Bean管理持久性（Bean-managed Persistence，BMP）和容器管理持久性（Container-managed Persistence，CMP），两种持久类型的组件类的开发方式具有较多的不同。因此，将在以下两章中分别对这两种持久性类型的Entity Bean进行介绍，并对不同持久类型的组件类的开发进行描述。

16.2.3.5 生存周期

Entity Bean对象的生存周期



上图是Entity Bean生存周期内的状态图。

在容器创建了Entity Bean实例之后，调用Entity Bean组件类的setEntityContext方法，将Entity Bean的上下文传递到实例。

实例化之后，Entity Bean被转移到实例池成为可用实例。实例池中的实例没有任何标识，只有在实例进入就绪状态时，容器将为实例分配标识。

实例有两种方式从实例池进入就绪状态。一种方式是客户端调用了Entity Bean的Home接口的create方法，使容器调用Entity Bean的ejbCreate和ejbPostCreate方法；另一种方式是容器调用Entity Bean的ejbActivate方法。

当实例进入就绪状态之后，即可对实例的业务方法进行调用。

同样，实例也有两种方式从就绪状态进入实例池。一种方式是客户端调用了remove方法，使容器调用Entity Bean的ejbRemove方法；另一种方式是容器调用Entity Bean的ejbPassivate方法。

当Entity Bean实例的生命周期结束，容器将Entity Bean从实例池中移出，并调用Entity Bean的unsetEntityContext方法

16.2.4 Bean管理持久性的Entity Bean

16.2.4.1 概述

对存储于数据库或者传统系统中的业务实体，在基于J2EE™平台的企业应用开发中，通常使用Entity Bean来提供业务实体的对象视图，持久性指Entity Bean实例和底层数据存储之间进行状态交换的数据访问协议。

在EJB2.1规范中，按照持久性的管理机制，划分了两种类型的Entity Bean。即容器管理持久性的Entity Bean和Bean管理持久性的Entity Bean。

Bean管理持久性的Entity Bean中，可以在组件类或其他辅助类对Bean的持久性进行编码。对于客户端而言，表示业务实体的Entity Bean不是底层数据存储中数据的简单表示，而是提供了面向对象特征的，具有访问接口、内部状态和封装机制的对象。

16.2.4.2 Bean管理持久性的Entity Bean

Bean 管理持久性的Entity Bean具有Enterprise JavaBean的共有特征，如包含组件类、组件接口、Home接口等。不同的是，由于这种组件模型必须依靠开发者提供对持久性管理的实现，因此，Bean管理持久性的Entity Bean的组件模型有一些特别的方面。

从前面的中，可以了解到，组件接口是开发者暴露给客户端的、对实例的访问方法，Home接口提供客户端创建、清除实例的方法，组件类中提供对定义在以上两个接口中方法的实现。对于Bean管理持久性的Entity Bean，这些定义在接口中的方法决定了开发者需要实现的数据访问和同步等等持久性方面的能力。

下面，我们从Bean实例的创建、状态装载与保存、实例的清除等方面，来了解Bean管理持久性的Entity Bean的特点。

16.2.4.2.1 打开与释放资源

由前面的中可知，Entity Bean实例从不存在状态到实例池状态时，容器将调用组件类的setEntityContext方法，在实例将容器被清除，由实例池返回到不存在的状态时，容器将调用组件类的unsetEntityContext方法。由于Bean管理持久性的Entity Bean通常需要对持久存储进行访问，需要占用一些资源连接，因此，开发者通常需要在这两个方法中对资源（如数据库连接）进行打开和释放的操作。

16.2.4.2.2 实例的创建

Entity Bean既然表示底层数据存储中的业务实体，Bean实例的创建意味着底层数据存储中数据的创建。从EJB的组件模型中，可以了解到，实例的创建通常是客户端调用组件Home接口中的create方法，此方法被传播到容器，容器随之调用组件类中对应的ejbCreate方法，客户端调用create方法使用的参数被容器用于调用组件的ejbCreate方法，因此，开发者在使用ejbCreate方法参数初始化实例时，也需要在底层数据存储中创建数据，如通过JDBC、连接器（Connector）等方式。

因此，开发者需要在ejbCreate方法中直接在数据存储中创建数据或者通过其他的辅助类进行，在数据创建之后，开发者需要编码返回实例的Primary Key。然后，容器返回实例的组件接口引用到客户端。

16.2.4.2.3 实例状态与持久存储

在实例的生存周期之内，客户端可通过组件接口改变实例的状态，因此，开发者需要提供同步实例状态与持久存储的实现。

从前面Entity Bean的介绍中，对于Bean管理持久性的Entity Bean组件类，实例的状态与持久存储数据的同步，是通过容器对定义在组件类接口中的方法进行调用完成的，即javax.ejb.EntityBean接口中的ejbStore和ejbLoad方法，因此，开发者必须在这两个方法中提供对实例状态数据的装载和保存的实现。

16.2.4.2.4 实例的清除

对于Bean管理持久性的Entity Bean，如同其他类型的EJB组件，客户端可以通过调用组件接口中和组件Home接口中的remove方法，清除实例。当客户端调用remove方法，容器调用组件类中的ejbRemove方法。开发者需要在ejbRemove方法中对持久存储进行访问，删除实例的状态。

16.2.4.3 必须遵守的规则与范例

本节以一个账户的例子演示Bean管理持久性的Entity Bean的编写。使用如下的命名约定：

- 组件类以"<实体名>EJB" 的方式命名，如SavingsAccountEJB；
- 组件的Home接口以"<实体名>Home"的方式命名，如SavingsAccountHome；
- 组件的组件接口以"<实体名>"的方式命名，如SavingsAccount；

本例中的业务实体对应的表模型根据如下SQL语句创建：

```
CREATE TABLE savingsaccount
(id VARCHAR(3)

CONSTRAINT pk_savingsaccount PRIMARY KEY,

firstname VARCHAR(24),

lastname VARCHAR(24),

balance NUMERIC(10,2));
```

	**注意**
	如采用不同类型的数据库，上面的SQL语句可能会不同。上面语句经验证可用于SQL Server2000，Oracle8.16 for windows。

16.2.4.3.1 组件类

开发者必须遵循如下规则：

- 实现javax.ejb.EntityBean接口；
- 零个到多个ejbCreate和对应的ejbPostCreate方法；

- 编写finder方法；
- 编写业务方法；
- 编写Home方法；
- 组件类必须声明为公有的（public）；
- 组件类不能被定义为抽象的（abstract）和静态的（static）；
- 必须包含一个无参数的公有构建器（constructor）；
- 不能实现finalize方法；

一. ejbCreate方法

ejbCreate方法中，开发者需要进行插入实体状态到数据库、初始化实例变量以及返回Primary Key等操作。

本例中的ejbCreate方法调用了类中的私有insertRow方法，此方法执行一个SQL中的INSERT语句，将实例状态插入到数据库，如下：

```
...
public String ejbCreate(String id, String firstName, String lastName,
BigDecimal balance) throws CreateException {
  if (balance.signum() == -1) {
    throw new CreateException("A negative initial balance is not allowed.");
  }
  try {
    insertRow(id, firstName, lastName, balance);
  } catch (Exception ex) {
    throw new EJBException("ejbCreate: " + ex.getMessage());
  }
  this.id = id;
  this.firstName = firstName;
  this.lastName = lastName;
  this.balance = balance;
  return id;
}
...
```

ejbCreate方法必须遵循如下规则：

- 方法签名必须为public；
- 返回值类型必须是Primary Key类型；
- 参数必须是合法的RMI类型；

- 方法不能声明为final或static；
- 方法必须声明抛出javax.ejb.CreateException异常；

二. ejbPostCreate方法

对于每个在组件类中定义的ejbCreate方法，必须有一个对应的ejbPostCreate方法。容器将在ejbCreate方法调用完成后，立即调用ejbPostCreate方法，由于在ejbPostCreate方法内部，可以调用EntityContext接口的getPrimaryKey和getEJBObject方法。

ejbPostCreate的方法签名必须遵循如下规则：

- 参数数量与类型必须与对应的ejbCreate方法一致；
- 访问控制签名必须是公有的（public）；
- 方法签名不能是final或static;
- 返回值类型必须为void；

通常情况下，ejbPostCreate方法为空。

三. ejbRemove方法

容器在清除组件实例之前会调用ejbRemove方法。本例中的ejbRemove方法调用了组件类中的deleteRow方法，从数据库中清除组件实例得状态，如下：

```
...
public void ejbRemove() {
try {
deleteRow(id);
catch (Exception ex) {
throw new EJBException("ejbRemove: " +
ex.getMessage());
}
}
...
```

四. finder方法

组件类中必须提供对Home接口中定义的每个finder方法的实现，必须遵循如下规则：

- 组件类中必须定义ejbFindByPrimaryKey方法；
- finder方法必须以“ejbFind”前缀开始；
- 方法必须被定义为public；
- 方法不能被声明为static或final;
- 对于提供远程Home接口的Entity Bean，方法的参数类型和返回值必须是合法的RMI类型；
- 返回值必须是Primary Key类型，或是包含Primary Key的集合（Collection）；
- 抛出语句必须包含javax.ejb.FinderException异常的抛出；

下面是范例中ejbFindByPrimaryKey方法的例子：

```
...
public String ejbFindByPrimaryKey(String primaryKey) throws FinderException {
  boolean result;
  try {
    result = selectByPrimaryKey(primaryKey);
  } catch (Exception ex) {
    throw new EJBException("ejbFindByPrimaryKey: " + ex.getMessage());
  }
  if (result) {
    return primaryKey;
  } else {
    throw new ObjectNotFoundException("Row for id " + primaryKey + " not
found.");
  }
}
...
```

五. 业务方法

通常情况下，业务方法不能直接对数据库进行访问，只包含必需的业务逻辑。业务方法的编写必须遵循如下规则：

- 不能与组件类中定义的其他方法的命名方式冲突，如“ejbCreate” 等；
- 方法签名必需被声明为public；
- 方法不能被声明为static或final；

- 对于提供远程访问接口的Entity Bean，方法的参数类型和返回值必须是合法的RMI类型

以下代码片断是一个业务方法的范例：

```
...
public void debit(BigDecimal amount) throws InsufficientBalanceException {
  if (balance.compareTo(amount) == -1) {
    throw new InsufficientBalanceException();
  }
  balance = balance.subtract(amount);
}
...
```

六. Home方法

Home方法用于对一个Entity Bean的所有或部分实例进行操作的方法。一般，Home方法对Entity Bean实例的集合进行定位，然后依次调用实例的业务方法，对之进行操作。如下例：

```
...
public void ejbHomeChargeForLowBalance(BigDecimal minimumBalance, BigDecimal
charge) throws InsufficientBalanceException {
  try {
    SavingsAccountHome home = (SavingsAccountHome) context.getEJBHome();
    Collection c = home.findInRange(new BigDecimal("0.00"),
minimumBalance.subtract(new BigDecimal("0.01")));
    Iterator i = c.iterator();
    while (i.hasNext()) {
      SavingsAccount account = (SavingsAccount) i.next();
      if (account.getBalance().compareTo(charge) == 1) {
        account.debit(charge);
      }
    }
  } catch (Exception ex) {
    throw new EJBException("ejbHomeChargeForLowBalance: " + ex.getMessage());
  }
}
...
```

Home方法必须遵循如下规则进行编写：

- 方法必须以“ejbHome ”前缀开头；

- 方法必须被声明为public；
- 方法不能被声明为static；
- 不能声明抛出java.rmi.RemoteException异常；

16.2.4.3.2 HOME接口

Home接口的编写，其规则参考Home接口中的描述，本节中的范例代码如下：

```
import java.util.Collection;
import java.math.BigDecimal;
import java.rmi.RemoteException;
import javax.ejb.*;

public interface SavingsAccountHome extends EJBHome {
  public SavingsAccount create(String id, String firstName, String lastName,
BigDecimal balance) throws RemoteException, CreateException;

  public SavingsAccount findByPrimaryKey(String id) throws FinderException,
RemoteException;

  public Collection findByLastName(String lastName) throws FinderException,
RemoteException;

  public Collection findInRange(BigDecimal low, BigDecimal high) throws
FinderException, RemoteException;

  public void chargeForLowBalance(BigDecimal minimumBalance, BigDecimal
charge) throws InsufficientBalanceException, RemoteException;
}
```

16.2.4.3.3 组件接口

组件接口编写的规则，参考"组件接口"中的描述，本节中的范例代码如下：

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

public interface SavingsAccount extends EJBObject {
  public void debit(BigDecimal amount) throws InsufficientBalanceException,
RemoteException;
```

```
  public void credit(BigDecimal amount) throws RemoteException;

  public String getFirstName() throws RemoteException;

  public String getLastName() throws RemoteException;

  public BigDecimal getBalance() throws RemoteException;
}
```

16.2.4.4 异常处理

通常把Entity Bean抛出的异常分为两个类别：系统级异常和应用级异常。

16.2.4.4.1 系统级异常

系统级异常的抛出通常用于指明对应用提供支持的服务产生的问题，如：不能获得数据库连接、由于数据库原因不能插入数据或lookup方法查找不到对象等等。对于此类问题，开发者应该抛出javax.ejb.EJBException异常，容器将封装此异常到一个java.rmi.RemoteException并返回到客户端。因javax.ejb.EJBException是java.lang.RuntimeException的子类，因此，在方法声明中，不需要在throws子句中进行声明。而且，如果一个Entity Bean实例抛出了一个系统级的异常，客户端并不具有对异常的处理能力，只能由系统管理员对应用进行调整。

16.2.4.4.2 应用级异常

应用级异常指应用的业务逻辑发生错误。包含两种类型，定制的应用级异常和预定义的应用级异常。定制的应用级异常通常由开发者定义，与业务逻辑紧密关联，如某个本节中的SavingsAccountEJB中的debit方法抛出的InsufficientBalanceException异常。另外，EJB的编程模型中还包括一些预定义的异常类型，如javax.ejb.CreateException、javax.ejb.FinderException异常等等。

16.2.4.4.3 组件异常类型

	**注意**
	如果某个事务边界中的Entity Bean实例的方法抛出了系统级异常，容器将回滚事务，如抛出应用级别的异常，容器将不会回滚事务。

下表包含Entity Bean的组件模型API中定义的异常类型的简单介绍，除javax.ejb.EJBException和javax.ejb.NoSuchEntityException之外，都属应用级异常：

方法名	抛出异常	原因
ejbCreate	javax.ejb.CreateException	输入参数无效
ejbFindByPrimaryKey	javax.ejb.FinderException	数据库中不存在需要查找的数据
ejbRemove	javax.ejb.RemoveException	不能从数据库中删除数据
ejbLoad	javax.ejb.NoSuchEntityException	不能在数据库中找到需要装载的数据

ejbStore	javax.ejb.NoSuchEntityException	不能更新数据库中的数据
所有方法	EJBException	系统问题发生

16.2.5 容器管理持久性的Entity Bean

16.2.5.1 概述

Entity Bean组件用于表示保存于持久存储中的业务实体，而这些业务实体通常是关系型数据库中的某个表。Entity Bean作为业务实体的面向对象表示，提供用于表示对象状态的域和可操作的方法，Entity Bean的域通常映射为业务实体表中的某一列，而一个Entity Bean实例通常用于表示表中的一行数据。

在Bean管理持久性的情况下，开发者必须在每个Bean中提供如何从数据库中装入数据到EJB实例和将实例状态存储到数据库中的代码，在更复杂的情况下，开发者还需要提供更多的编码，以方便地对持久存储中的业务数据进行创建、删除、同步等等操作。

通过容器对Entity Bean的持久性进行管理，开发者通过部署描述定义Entity Bean中需要容器进行管理的域，并指定域与数据库中列的映射，在Bean中定义访问这些域的抽象方法（Java Bean模型中的getter、setter方法）即可。对于被指定由容器管理持久性的域，必需的对数据库中数据的访问机制则可以完全由容器进行实现和管理，开发者不需要再编写访问、同步数据的代码，可以直接通过访问方法，即getter或setter方法，对域的状态进行访问。

由于Entity Bean都会用于表示数据库中的保存业务实体的表，而关系型数据库通常使用表间关系对数据的引用完整性进行约束。当使用容器管理持久性的Entity Bean时，可以对实体间的关系进行声明，由容器实现关系语义的面向对象的表示。

容器管理持久性给Entity Bean带来了更大的可移植特性和更大的独立性，当组件的内部实现发生更改，或其对应的数据存储结构发生更改，客户端不需更改访问Bean的代码并重新编译，同样，当Entity Bean映射的数据存储发生更改，组件也不存在需要改变的访问具体数据的代码，只需重新定义数据存储与Bean的域之间的映射。

不同于Bean管理持久性的Entity Bean，容器管理持久性给Entity Bean的开发者带来了更大的便利，和更少的需要调试的代码。开发者通过部署描述指定容器需要管理的数据访问的域和关系，在开发时即可直接通过方法对持久数据进行访问。

容器管理持久性的Entity Bean带来的好处是，通过使用部署描述文件和简单的组件代码，即可将业务实体的逻辑表示与具体的数据存储分离，并且消除组件中复杂且易出错的数据访问的代码。

16.2.5.2 容器管理持久性的Entity Bean

Entity Bean是数据库中或已有企业应用中存储的一个业务实体或一组相互紧密关联业务实体的对象表示。

容器管理持久性主要包含两个方面的特征，即对Entity Bean状态数据的持久性管理和Entity Bean间关系的管理。

16.2.5.2.1 CMP模型

在EJB1.1规范中的CMP模型，开发者通过部署描述将Entity Bean实例中的成员变量映射到某个数据库表，容器负责这二者之间的数据同步，但是，EJB1.1规范中的CMP模型缺乏控制业务实体间的复杂关系的机制。

保存在数据库中的业务实体以表的形式存在，表与表之间处于对等的地位，没有具体的从属关系，使用数据库表所表示的业务实体之间的逻辑关系及数据的引用完整性通常通过主键与外键来进行约束，而通过EJB1.1规范中的CMP模型进行数据的对象表示，对于单个的Entity Bean和对应的数据库表而言，可以提供表示，但是企业应用中使用数据库表所表示的业务实体之间往往存在很复杂的逻辑关系，虽然针对这一点，有经验的开发者提出了一些解决方案，但是，EJB1.1规范中的CMP模型远未完善。

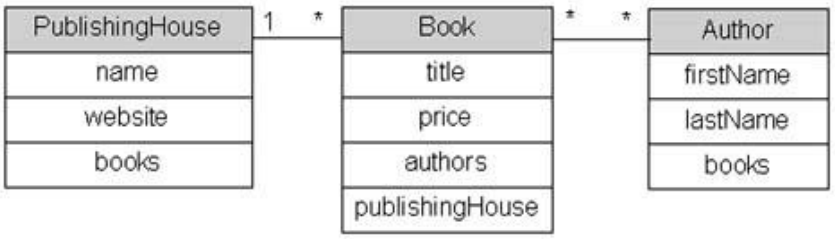
在EJB2.1规范中，对CMP模型进行了比较大的改进，除了完善Entity Bean与数据库表间的数据映射，还增加了新的Entity Bean间的关系模型，以此提供数据库表从数据到逻辑关系在Entity Bean编程模型的完整映射，同时对于EJB1.1规范中难于解决的从属对象

（dependent object），提供了解决办法。同时，为使用新的CMP模型对Entity Bean中的数据查询进行编写，定义了一套新的查询语言EJB QL。通过按照EJB QL规范编写的查询语句，可以对Entity Bean中的相关查询方法的语义进行定义，如EJB1.1规范中已出现的finder方法和EJB规范2.0版本中的select方法，并可在相互关联的Entity Bean的关系中进行定位，容器将使用EJB QL语句编写的查询，对关联的查询方法进行定义。

16.2.5.2.2 例子说明

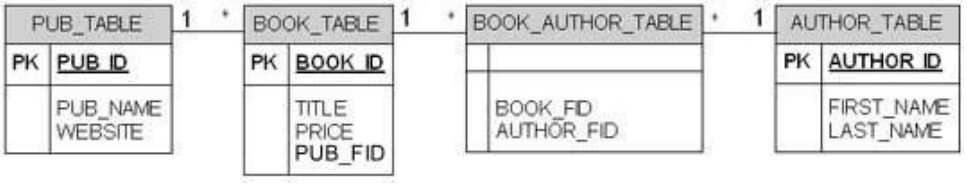
本节使用如下的范例对容器管理持久性的Entity Bean加以说明。通过实例介绍Apusic应用服务器中，容器管理持久性的Entity Bean在应用程序开发效率方面带来的提高和成本方面的降低。本节中将采用如下图所示的三个业务实体进行说明：

范例



通常，从上面描述的业务实体模型，可以得到如下的关系型数据库的表模型：

范例表模型



可以看出，在业务实体转换为关系型数据库中的表模型时，通常会使用主键（Primary Key）、外键（Foreign Key）或使用一个用于描述关系的辅助表描述表间的关系，以达到引用完整性的目的。

根据此范例关系模型和数据库表模型，按照如下命名规则，为其定义本章中使用的范例的数据库表名、列名和Entity Bean组件名、域名。

- 表名按照"<实体名>_TABLE"的方式命名：BOOK_TABLE，PUB_TABLE，AUTHOR_TABLE，BOOK_AUTHOR_TABLE；
- 组件类名按照"<实体名>EJB"的方式命名:BookEJB，AuthorEJB，PublishingHouseEJB;
- 组件接口（本例中的EJB组件都使用本地接口）按照"<实体名>"命名，如Book，Author，PublishingHouse；
- 组件Home接口（本例中的EJB组件都使用本地Home接口）按照"<实体名>Home"命名；
- 组件的抽象持久性模式名称，按照"<实体名>"命名，如Book，Author，PublishingHouse；

16.2.5.2.3 抽象持久性模式（ABSTRACT PERSISTENCE SCHEMA）

抽象持久性模式的概念，对于Apusic应用服务器中容器管理持久性的Entity Bean极为关键。因为，Apusic应用服务器中的持久性管理器为Entity Bean生成数据访问逻辑，主要是通过开发者在Entity Bean类的代码和部署描述文件定义的抽象持久性模式。现阶段的

Apusic应用服务器中的持久性管理器除了提供对关系型数据库和Entity Bean之间的持久性管理外，将来还可根据需要实现对传统系统（如ERP等）的持久性管理。抽象持久性模式的定义，主要涉及两个方面的要素：即容器管理持久性域和容器管理关系。

一. 容器管理持久性域（cmp-field）

为提供业务实体的面向对象表示，通常需要O/R mapping方面的技术，将对象的域与数据库中的表列进行映射，在Apusic应用服务器中的容器管理持久性的Entity Bean中，开发者需要在部署描述中将组件中的一个域映射到数据库表中的一个列，并在组件代码中，按照Java Bean规范中的setter或getter方法定义这些容器管理持久性域的访问方法，以完成映射和抽象持久性模式中容器管理持久性域的声明。容器根据开发者上面的声明和定义，在运行时负责装载、保存和同步实例变量与表中的数据，开发者或客户端可通过组件代码中定义的数据访问方法访问业务实体数据。

容器管理持久性的Entity Bean中，容器在运行时为Entity Bean生成数据访问的代码，在实例变量和底层数据存储间交换数据，这些代码包含创建、删除和在底层数据存储中查找实体数据等操作的方法。

如果使用容器管理持久性的Entity Bean来表示业务实体，则开发者不能对数据访问进行编码，所有的数据访问应由容器完成。

开发者通过部署描述中的cmp-field元素来声明容器必须为组件装入和存储的的实例变量，这些变量被称为容器管理持久性域（cmp-field）。在组件代码中，这些域必须被定义为公有的（public），且不能被定义为临时的（transient）。容器负责在执行ejbCreate、ejbRemove、ejbStore或ejbLoad等方法之前在Bean的实例变量和底层数据存储间进行数据交换，同时容器需要实现在Entity Bean的Home接口中定义的finder方法。

在Apusic应用服务器2.0版本以上，对抽象持久性模式中包含的容器管理持久性域的定义，通常如下进行：

- 在部署描述文件ejb-jar.xml中提供组件的抽象持久性模式的名称和需要容器进行持久性管理的域名称，还需要指定实体的Primary Key对应的域。即在entity元素声明中使用abstract-schema-name定义组件的抽象持久性模式名称，并使用cmp-field定义需要容器进行持久性管理的域，最后使用primkey-field定义Primary Key域，此域必须存在于使用cmp-field定义的容器管理持久性域中。需要注意的是，对于处理实体之间的关系而定义在实体中的域，需要以容器管理关系域出现，如Book实体中的publishingHouse域，将在"容器管理关系（cmr）与容器管理关系域（cmr-field）"中介绍。

以本节范例实体中的Book为例，因publishingHouse和authors域用于定义与PublishingHouse、Author等实体间的关系，不属于容器管理持久性域的范围，因此，Book只有三个容器管理持久性域：id、title和price域，其中，id域表示Primary Key域，不代表实体数据，因此，未表示在实体图形中 。按照前面的规则，有如下ejb-jar.xml片断：

```
...
<entity>

...
<abstract-schema-name>Book</abstract-schema-name>
    <cmp-field>
        <field-name>id</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>title</field-name>
    </cmp-field>
    <cmp-field>
```

```
        <field-name>price</field-name>
    </cmp-field>
    <primkey-field>id</primkey-field>

...
</entity>

...
```

- 对于每个在ejb-jar.xml文件中声明的容器管理持久性的Entity Bean，还需要在apusic-application.xml中，使用table-name元素定义组件映射的表名称，使用field-mapping元素对容器管理持久性域与数据库表中的列进行映射，并且使用cmp-resource元素声明需要使用的数据源名称，这一数据源由ejb-jar中所有的容器管理持久性的Entity Bean以及关系辅助表共同使用。以Book组件为例，有如下apusic-application.xml片断：

```
...
<ejb>
...
<entity ejb-name="Book">
        <jndi-name>Book</jndi-name>
        <cmp>
            <jdbc>
                <table-name>BOOK_TABLE</table-name>
                <field-mapping>
                    <field-name>id</field-name>
                    <column-name>BOOK_ID</column-name>
                </field-mapping>
                <field-mapping>
                    <field-name>title</field-name>
                    <column-name>TITLE</column-name>
                </field-mapping>
                <field-mapping>
                    <field-name>price</field-name>
                    <column-name>PRICE</column-name>
                </field-mapping>
            </jdbc>
        </cmp>
    </entity>
...
<cmp-resource>
        <jndi-name>jdbc/CMPSample</jndi-name>
```

```
    </cmp-resource>
...
<ejb>
...
```

- 最后，开发者还需要在组件类中声明对容器管理持久性域的抽象访问方法，即Java Beans规范中的setter和getter方法，在组件被部署到应用服务器后，容器将在运行时根据前面指定的配置信息实现这些方法，因此，开发者在定义这些方法时，必须把它们声明为抽象（abstract）方法。以Book组件为例，组件类BookEJB有如下代码片断：

```
...
public abstract class BookEJB implements EntityBean {
  // access methods for persistent fields
  public abstract void setId(java.lang.String id);

  public abstract void setTitle(java.lang.String title);

  public abstract void setPrice(double price);

  public abstract double getPrice();

  public abstract java.lang.String getTitle();

  public abstract java.lang.String getId();
...
}
```

在Entity Bean的开发过程中，对组件状态的访问必须通过上面定义的抽象方法进行，例如，在BookEJB中的ejbCreate方法被调时，可直接调用以上访问方法进行状态的初始化，如下：

```
...
public String ejbCreate(String id, String title, double price) throws
CreateException {
  setId(id);
  setTitle(title);
  setPrice(price);
  return null;
}
...
```

- 如果，业务实体的状态对客户是可读或可写的，则可以通过组件接口决定是否暴露状态域的setter和getter方法，也可根据实际情形，决定暴露哪个方法给客户端。以Book组件为例，其两个容器管理持久性域对客户而言是可读和可写的，因此，本地接口Book中关于客户访问的代码片断如下：

```
...
public interface Book extends EJBLocalObject {
  public String getId();

  public void setTitle(String title);

  public String getTitle();

  public void setPrice(double price);

  public double getPrice();
...
}
```

当组件部署到Apusic应用服务器，在运行时容器为组件生成所有的数据访问代码，并对状态的同步、并发访问、事务控制等进行管理，组件中其它需要访问持久存储数据的代码，则通过上面组件类中定义的setter和getter方法进行访问，客户端则通过组件接口中暴露的方法，对组件状态进行访问。

二. 容器管理关系（cmr）与容器管理关系域（cmr-field）

前面描述过，EJB1.1中，容器管理持久性的Entity Bean在处理实体间关系上的不足之处，因此，在EJB2.1规范中，容器管理持久性的Entity Bean编程模型，定义了容器管理关系模型，以此解决EJB1.1中的不足。

容器管理持久性的Entity Bean的容器管理关系，有以下三种：

- one-to-one，每个Bean实例关联另一个Bean的单个实例；
- one-to-many，一个Bean关联另一个Bean的多个实例；
- many-to-many，Bean的多个实例关联另一个Bean的多个实例；

容器管理关系的方向可以是双向或单向的。在一个双向的关系中，涉及的Bean都有一个关系域与另外的Bean关联，通过关系域，可以从一个Bean的实例中访问关系另一方的Bean对象，反之亦可。在一个单向的关系中，只有一个Bean拥有关联其他Bean的关系域，只能从这个Bean的实例访问被关联的Bean对象，而不可从被关联的Bean对象访问到这个实例。

Entity Bean编程模型中定义的EJB QL查询语言用于对具有容器管理关系的Entity Bean实例之间进行定位。容器管理关系的方向决定了从一个Bean定位另一个Bean的能力。如Book作为关系中的一方，出版社PublishingHouse作为关系中的另一方，如果二者拥有双向关系，即从出版社可以知道出版了哪些书，而由书实例的状态数据亦可知道由某个出版社出版，那么，就可以通过这种双向关系，可以从出版社定位到书籍，也可以从书籍定位到出版社。

容器管理关系通过在部署描述中对需要由容器管理关系域进行声明，通过容器管理关系域来实现容器管理关系。如上面的例子，为完成PublishingHouseEJB与BookEJB间的关系，PublishingHouseEJB中需要定义关联到BookEJB的容器管理关系域为books，类型为java.util.Collection，表示关联BookEJB实例集合，通过books域可以定位到关联的BookEJB实例；在BookEJB中定义的关联到PublishingHouseEJB的容器管理关系域为publishingHouse域，类型为PublishingHouseEJB的本地接口，通过publishingHouse域，BookEJB实例可以定位到关联的PublishingHouseEJB实例。

因此，在容器管理关系中，构成关系的基础是容器管理关系域，但是，并不是所有用于在容器管理关系间进行定位的容器管理关系域，都会映射到持久存储中，以Book 与PublishingHouse间的关系为例，Book中的容器管理关系域publishingHouse，被对应到BOOK_TABLE中的PUB_FID列，而PublishingHouse中的books域，则不需要保存到持久数据存储，容器将根据开发者定义的关系对books域进行求值。同时对books的类型java.util.Collection赋予与标准的Java API不尽相同的语义。

根据容器管理关系的类型，决定了关系具有基数性（cardinality）,即一个cmr-field的域的get方法必须返回一个Entity Bean的本地接口或由Entity Bean的本地接口构成的集合（java.util.Collection或java.util.Set），同样，set方法的参数必须是一个Entity Bean的本地接口或由Entity Bean的本地接口构成的集合（java.util.Collection或java.util.Set）。

需要注意的是，容器管理关系定义的范围限于在同一个部署描述文件ejb-jar.xml中，即当两个Entity Bean在不同的部署描述文件ejb-jar.xml中进行声明，则不能对他们之间建立起容器管理关系，只能对在同一个部署描述文件中声明的Entity Bean之间定义容器管理关系。

在Apusic应用服务器2.0版本以上的容器管理持久性Entity Bean中，对容器管理关系的声明主要通过以下步骤进行：

- 在ejb-jar.xml文件中，使用relationships元素中的ejb-relation元素对每个容器管理关系进行定义。在ejb-relation元素中，需要使用ejb-relationship-name元素定义标识关系的关系名称，使用ejb-relationship-role对关系中涉及的角色进行定义，每个角色涉及的角色名、多重性（one or many)、对应的Entity Bean组件、容器管理关系域，依次使用ejb-relationship-role-name、multiplicity、relationship-role-source、cmr-field元素对其进行定义。以本章中Book实体与PublishingHouse实体间的关系为例，则有如下ejb-jar.xml文件有关容器管理关系的片断：

```
<ejb-jar>
...
<relationships>

...
<ejb-relation>
            <ejb-relation-name>BookAndPublishingHouse</ejb-relation-name>
            <!-- One role: Book -->
            <ejb-relationship-role>
                <ejb-relationship-role-name>

Book
</ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>Book</ejb-name>
```

```
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>publishingHouse</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>
            <!-- The other role: PublishingHouse -->
            <ejb-relationship-role>
                <ejb-relationship-role-name>

PublishingHouse
</ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>PublishingHouse</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>books</cmr-field-name>
                    <cmr-field-type>java.util.Collection</cmr-field-type>
                </cmr-field>
            </ejb-relationship-role>
        </ejb-relation>

...
</relationships>

...
</ejb-jar>
```

- 在apusic-application.xml中，开发者需要使用ejb元素中包含的relationship-mapping元素对构成容器管理关系的容器管理关系域与持久存储中的数据模型进行映射。其中，使用ejb-relation-name元素定义关系的名字（需要与ejb-jar.xml文件中对应的ejb-relation-name元素的值相同）、使用table-name定义关系域所存在的表名字、使用source-role和sink-role元素定义参与关系的角色双方，完成组件模型中的容器管理关系域与数据库表中列的映射。source-role和sink-role元素被用于区分两个不同的角色，需要与在ejb-jar.xml文件中使用ejb-relationship-role元素定义的角色对应；如角色存在容器管理关系域在持久存储中的映射，则需要在source-role或sink-role元素中，使用field-mapping元素声明关系域与数据库表列的映射。如角色不存在容器管理关系域在持久存储中的映射，则可以使用空标记声明角色。

field-mapping元素中，使用field-name元素声明本角色在ejb-jar.xml文件中定义的primkey-field域，使用column-name元素将其映射到引用此域以保存关系的列。

以Book与PublishingHouse间的关系为例，此关系中只有Book中的关系域publishingHouse存在持久数据存储中的对应列PUB_FID，因此，有关apusic-application.xml片断如下：

```
...
<apusic-application>

...
<module>

...
<ejb>

...
<relationship-mapping>
                <ejb-relation-name>

BookAndPublishingHouse
</ejb-relation-name>
                <table-name>BOOK_TABLE</table-name>
                <auto-create-table />
                <source-role />
                <sink-role>
                    <field-mapping>
                        <field-name>id</field-name>
                        <column-name>PUB_FID</column-name>
                    </field-mapping>
                </sink-role>
            </relationship-mapping>

...
</ejb>

...
</module>

...
</apusic-application>
```

- 在关系双方的组件类中，声明所使用的容器管理关系域的抽象访问方法，如关系是单向的，则只需在可定位到另一方的组件类中，对域进行声明。方法必须是公有（public）和抽象（abstract）方法。以Book与PublishingHouse间的关系为例，BookEJB和PublishingHouseEJB中的相关代码片断如下：

```
//BookEJB.java
public abstract class BookEJB implements EntityBean {
...
  public abstract void setPublishingHouse(PublishingHouse pub);

  public abstract PublishingHouse getPublishingHouse();
...
}// END BookEJB.java
 // PublishingHouseEJB.java
...


public abstract class PublishingHouseEJB implements EntityBean {
...
  public abstract void setBooks(Collection books);

  public abstract Collection getBooks();
...
}// END PublishingHouse.java
```

- 通过组件接口定义客户端对组件的访问接口，即使用组件接口决定是否允许客户端访问容器管理关系域。Book与PublishingHouse相关的组件接口代码如下：

```
//Book.java
public interface Book extends javax.ejb.EJBLocalObject {
...
  public void setPublishingHouse(PublishingHouse pub);

  public PublishingHouse getPublishingHouse();
...
}// END Book.java
 // PublishingHouse.java


public interface PublishingHouse extends EJBLocalObject {
...
  public void setBooks(Collection books);

  public Collection getBooks();
```

```
...
}//END PublishingHouse.java
```

容器管理持久性域和容器管理关系域是EJB2.1中容器管理持久性的Entity Bean的基础，在提供对它们的定义之后，即可在组件中使用EJB QL定义各种finder和select方法，实现对业务数据的查询；同时，在EJB QL中，通过容器管理关系对关系中的业务实体进行定位。

依靠容器管理关系模型，Apusic应用服务器将组件业务逻辑与数据持久性分离，开发者可以更高效、快速地进行业务实体组件的开发，同时，将与持久数据存储类型有关的部分从业务逻辑中永久分离出来，由Apusic应用服务器提供高效和安全的管理。

三. **辅助值对象**（dependent value object）

辅助值对象是企业应用中，较为常用一种具体（concrete）类，通常用于封装实体的全部或部分状态，用于组件内部或者通过组件接口使用辅助值对象与客户端交互。

在容器管理持久性的Entity Bean中，可使用辅助值对象表示封装实体的全部或部分状态，并作为Entity Bean中的容器管理持久性域，但是，辅助值对象不可作为容器管理关系域。

如某Customer实体包含了ContactInfo的辅助值对象，用于封装联系人姓名（contact_name）、电话（contact_phone）、传真（contact_fax）和电子邮件地址（contact_email）信息。

要将ContactInfo用于容器管理持久性的CustomerEJB，则需要将ContactInfo声明为一个容器管理持久性的域，持久性管理器将对此域及其映射在持久存储中的数据进行管理。当使用getter访问方法访问此域时，容器将返回一个辅助值对象的实例拷贝，当使用setter方法方法进行赋值时，容器将使用参数实例的拷贝进行赋值。

如果Bean使用自动建表功能，则不需要对ContactInfo与持久存储进行映射；当需要将辅助值对象映射到现有的表模型，则需要以下步骤：

- 在组件类中声明此辅助值对象的抽象访问方法，并在ejb-jar.xml文件中，将此域声明为一个容器管理持久性的域。

CustomerEJB.java中的相关代码片断如下：

```
...
public abstract class CustomerEJB implements EntityBean {
...
  public abstract ContactInfo getInfo();

  public abstract void setInfo(ContactInfo info);
...
}
```

- 在apusic-application.xml中，映射辅助值对象的域与表模型中的列，域声明使用<容器管理持久性域的名称>.<辅助值对象的域>，如本例中，ContactInfo在CustomerEJB中以info为域名，而ContactInfo包含name、phone、fax和email域，因此，apusic-application.xml有如下配置片断:

```
...
<entity ejb-name="CustomerEJB">
    <jndi-name>CustomerEJB</jndi-name>
    <cmp>
        <jdbc>

...
<field-mapping>
                <field-name>info.name</field-name>
                <column-name>customer_name</column-name>
            </field-mapping>
            <field-mapping>
                <field-name>info.phone</field-name>
                <column-name>phone</column-name>
            </field-mapping>
            <field-mapping>
                <field-name>info.fax</field-name>
                <column-name>fax</column-name>
            </field-mapping>
            <field-mapping>
                <field-name>info.email</field-name>
                <column-name>email</column-name>
            </field-mapping>

...
</jdbc>
    </cmp>
</entity>

...
```

辅助值对象的类必须实现java.io.Serializable接口。如果辅助值对象中的域是另一个辅助值对象类型，则前面的映射方式是可以递归的，如Customer实体中，contact_name被替换为contact_firstname和contact_lastname，在辅助值对象ContactName包含两个域firstName和lastName，以实现对contact_firstname和contact_lastname的映射，ContactName作为辅助值对象ContactInfo中的一个name域，则上面的apusic-application.xml文件中可进行如下对应：

```
...
<entity ejb-name="CustomerEJB">
    <jndi-name>CustomerEJB</jndi-name>
```

```
    <cmp>
        <jdbc>

...
<field-mapping>
                <field-name>info.name.firstName</field-name>
                <column-name>contact_firstname</column-name>
            </field-mapping>
            <field-mapping>
                <field-name>info.name.lastName</field-name>
                <column-name>contact_lastname</column-name>
            </field-mapping>
            <field-mapping>
                <field-name>info.phone</field-name>
                <column-name>phone</column-name>
            </field-mapping>
            <field-mapping>
                <field-name>info.fax</field-name>
                <column-name>fax</column-name>
            </field-mapping>
            <field-mapping>
                <field-name>info.email</field-name>
                <column-name>email</column-name>
            </field-mapping>

...
</jdbc>
    </cmp>
</entity>
```

因此，对于嵌套的辅助值对象的声明和映射，可使用“.”进行定位。在EJB QL中，也可在路径表达式中对辅助值对象进行定位。

四. **自动建表与自动生成主键**

以上部分描述了通过业务实体模型建立关系型数据库表模型，然后使用容器管理持久性的Entity Bean提供业务实体的面向对象表示。这方面的能力在使用传统数据库数据和传统应用数据方面具有重要的意义。

另一方面，通过Apusic应用服务器的持久性管理器，可以从业务实体模型建立起面向对象的实体组件，由容器在关系型数据库中自动创建对应的表模型和关系模型，对于企业应用的快速开发和应用开发过程中的开发、测试等过程，能起到提高开发效率和降低开发成本的作用，企业应用中需要维护的内容降到了更低的程度。

因为不同的关系型数据库提供的功能不同，因此，以往依赖于数据库的某个功能的企业应用，如使用数据库提供的自动生成主键值的功能，在一定程度上限制了企业应用的可移植特性。

Apusic应用服务器提供了一系列的功能，使应用逻辑不依赖于特定的数据库系统，并且，也不影响在某种数据库系统上的应用的性能优化。

下面对Apusic应用服务器提供的容器管理持久性Entity Bean的自动建表和自动生成主键的功能进行介绍：

(1). **自动建表**

以Book业务实体为例，前面提供了映射容器管理持久型域与数据库表列的范例部署描述，即如下的apusic-application.xml文件：

```
...
<entity ejb-name="BookEJB">
    <jndi-name>BookEJB</jndi-name>
    <cmp>
        <jdbc>
            <table-name>BOOK_TABLE</table-name>
            <field-mapping>
                <field-name>id</field-name>
                <column-name>BOOK_ID</column-name>
            </field-mapping>
            <field-mapping>
                <field-name>title</field-name>
                <column-name>TITLE</column-name>
            </field-mapping>
            <field-mapping>
                <field-name>price</field-name>
                <column-name>PRICE</column-name>
            </field-mapping>
        </jdbc>
    </cmp>
</entity>

...
```

实现自动建表的功能只需在上述部署描述文件中加入auto-create-table元素，同时，不再需要对容器管理持久性域与数据库表列的映射声明，如下所示：

```
...
<entity ejb-name="BookEJB">
    <jndi-name>BookEJB</jndi-name>
```

```
    <cmp>
        <jdbc>
            <datasource-name>jdbc/mssql</datasource-name>
            <table-name>BOOK_TABLE</table-name>
            <auto-create-table />
        </jdbc>
    </cmp>
</entity>

...
```

持久性管理器将在启动应用时，通过开发者指定的与数据库的连接和表名，在数据库中生成相应的表结构。

另外，对容器管理关系的自动创建，也可使用自动建表的功能。这一点，对于many-to-many的关系，可被广泛利用，以减轻对关联表的管理。如本章使用的例子中的Author与Book之间的many-to-many关系，在不使用自动建立关系表的功能时，需要作如下设置：

```
...
<apusic-application>

...
<module>

...
<ejb>

...
<relationship-mapping>
                <ejb-relation-name>BookAndAuthor</ejb-relation-name>
                <table-name>BOOK_AUTHOR_TABLE</table-name>
                <source-role>
                    <field-mapping>
                        <field-name>id</field-name>
                        <column-name>BOOK_FID</column-name>
                    </field-mapping>
                </source-role>
                <sink-role>
                    <field-mapping>
                        <field-name>id</field-name>
                        <column-name>AUTHOR_FID</column-name>
                    </field-mapping>
```

```
                </sink-role>
            </relationship-mapping>

...
</ejb>

...
</module>

...
</apusic-application>
```

在使用自动建表时，可用如下设置：

```
...
<apusic-application>

...
<module>

...
<ejb>

...
<relationship-mapping>
                <ejb-relation-name>BookAndAuthor</ejb-relation-name>
                <table-name>BOOK_AUTHOR_TABLE</table-name>
                <auto-create-table />
            </relationship-mapping>

...
</ejb>

...
</module>

...
</apusic-application>
```

(2). **自动生成主键**

企业应用中时常会使用不代表实体内部状态的数据来表示实体的主键。在基于J2EE平台的企业应用开发中，表现为应用的装配者或部署者在将应用部署到实际的企业应用平台时，使用一些特定的数据库提供的功能，自动为业务数据生成主键数据，这些数据可能是任何类型，如整型值、字符串、UUID等。

因此，开发者在开发容器管理持久性的Entity Bean时，如果Entity Bean的主键将在部署时进行指定，则开发者可以使用java.lang.Object来指定未知的主键数据类型，持久性管理器将自动为Entity Bean实例和持久存储数据生成主键。

在Apusic应用服务器中，可以按照如下方法使用自动生成主键的功能：

- 如果声明一个Entity的主关键字类型为java.lang.Object时，持久性管理器会自动为Entity生成一个主关键字，这时不需要在apusic-application.xml中声明auto-generate-key标记。在home接口中的findByPrimaryKey方法中，使用java.lang.Object作为参数类型；在组件类中的ejbCreate方法，必须声明返回java.lang.Object类型；
- 不可声明一个容器管理持久性的域来映射此主键，只需在ejb-jar.xml文件中使用prim-key-class元素声明Primary Key类型为java.lang.Object，且不要在ejb-jar.xml文件中使用primkey-field元素声明主键域；
- 可调用组件接口中的getPrimaryKey方法，得到主键的值。

另外，Apusic应用服务器还提供另一种更为方便的自动生成主键的功能，即使用整型类型的主键值。即声明一个Entity的主关键字类型为java.lang.Integer，并且在apusic-application.xml中声明了auto-generate-key标记，则持久性管理器也会在创建EJB实例时自动生成一个主关键字。使用这种方法，可以简化开发者与部署者之间的交流，而且在Entity Bean内部和Entity Bean之间，简化需要对主键进行的访问操作。

当使用Apusic应用服务器中的持久性管理器自动生成实体的主键数据的功能时，可以指定主键类型为java.lang.Integer类型，主键值缺省状态下以20为单位递增，即0、20、40…。

配置容器自动生成整型类型的主键值的功能比较简单，需要在apusic-application.xml中的jdbc元素中，声明auto-generate-key空标记，并注意如下规则：

- 在home接口中的findByPrimaryKey方法中，使用java.lang.Integer作为参数类型；在组件类中的ejbCreate方法，必须声明返回java.lang.Integer类型；
- 不可声明一个容器管理持久性的域来映射此主键，只需在ejb-jar.xml文件中使用prim-key-class元素声明Primary Key类型为java.lang.Integer，且不要在ejb-jar.xml文件中使用primkey-field元素声明主键域；
- 可调用组件接口中的getPrimaryKey方法，得到主键的值，主键值可使用java.lang.Integer进行造型。

五. 装载单元

对于企业应用中的实体，并非每次对实体的操作都会涉及实体的所有状态，一般来说，某些常用的操作决定了实体内部状态被使用的频度，例如，雇员纪录中包括住址、工作经历、技能、历史工作纪录、姓名、部门、职位等相关内容，当应用中对雇员纪录的查询大多数为对姓名、部门、职位内容的查询时，每次操作都读入所有的记录数据是不必要的。因此，对于实体状态的装载，需要更为细致的控制，即当处理实例状态的装载请求时，所有状态的集合可划分为多个子集合，只有当操作涉及某个子集合时，才对此子集合进行装载。

Apusic应用服务器中，对于容器管理持久性的Entity Bean，开发者可以通过将状态划分为不同的装载单元，以实现上面描述的这种装载机制。

装载单元涉及组件的容器管理持久性域和容器管理关系域，开发者可以在apusic-application.xml中使用load-unit元素，对装载单元进行指定。如下：

```
...
<entity ejb-name="CustomerEJB">

...
<cmp>
        <jdbc>

...
<load-unit>
                <unit-name>customer-usaual-info<unit-name>
                        <cmp-field>info.name</cmp-field>
                        <cmp-field>info.email</cmp-field>
                        <cmp-field>info.phone</cmp-field>
                        <load-unit>
                        </jdbc>
                    </cmp>
                </entity>

...
```

Apusic应用服务器使用了惰性装载（Lazy loading），充分提高了企业应用的效率，当对组件使用划分明确的装载单元后，可以更大幅度的提高企业应用的响应能力。

六. **关系的赋值语义**

关系多重性（multiplicity）的引用完整语义决定了容器管理关系的赋值操作具有特殊的语义。

例如，一个表示订单的Bean（OrderEJB）与表示订单中的采购项的Bean（LineItemEJB），从OrderEJB到LineItemEJB，这两个EJB之间被定义具有one-to-many的关系，即一个具体的订单，关联多个采购项，OrderEJB中包含一个setLineItems的方法，用于赋值给OrderEJB关联的所有LineItemEJB实例。

当o-1是一个订单实例，关联的采购项实例集合为ls1，o-2是另一个订单实例，o-2关联的采购项集合为空，当客户调用o-2的setLineItems方法，将ls1集合赋值给o-2 关联的LineItemEJB实例集合时，在同一个事务上下文中，o-2关联的LineItemEJB的实例集合将被从空更改为ls1，而o-1关联的LineItemEJB实例集合为空。看起来好像ls1从o-1被移动到了o-2。

在one-to-many和many-to-many的关系中，java.util.Collection API或Entity Bean实例的setter方法，可以被用来操作一个集合类型的cmr-field域的内容。

下面将分别描述使用java.util.Collection API和使用setter这两种方法。

(1). **使用java.util.Collection API更新关系**

在容器管理持久性的Entity Bean中，具备容器管理关系的组件实例，java.util.Collection接口被用于表示具有集合值的容器管理关系域（cmr-field），如上面的订单与采购项的例子，订单拥有一个具有集合值的容器管理关系域（cmr-field）lineItems，而这些采购项集合通过java.util.Collection接口表示，使用java.util.Collection接口方法拥有的标准语义可以操纵这个集合，如size 方法返回集合中元素的数目、iterator返回集合的迭代器等等，但由于one-to-many这种关系具有的引用完整性，使得这种情况下的java.util.Collection接口中的add和addAll方法具有与java.util.Collection接口方法拥有的标准语义不同的特殊语义。

- 如果add方法的参数是在容器管理关系中定义的具有集合值类型的容器管理关系集合中的元素，如上例中的ls1集合中的某个LineItemEJB实例，则在同一事务上下文中，此元素将被从原有的关系中移出，添加到现有的关系中，如上例，假设l-1是ls1中的一个元素，当对表示o-2关联的的LineItemEJB的实例集合ls2 调用add方法，其参数是l-1时，在同一事务上下文中，l-1将被从ls1中清除，并被添加到ls2中；
- 对于addAll方法，当在one-to-many的容器管理关系中，具有与add方法相同的语义，对addAll参数中的每个元素使用add方法的语义。

由于在one-to-many关系中，java.util.Collection API所具有的特别的语义，因此，对集合的迭代器（iterator）进行操作，有如下限制：当由集合产生的迭代器在运行中，对集合元素的进行的添加或删除的操作没有使用java.util.Iterator API的remove方法时，容器将在对迭代器执行下一操作时抛出java.lang.IllegalStateException异常。下面的范例说明处理这一情况的正确方法：

```
Collection ls1 = o1.getLineItems();
Collection ls2 = o2.getLineItems();
Iterator i = o1.iterator();
LineItem lineItem;
// a wrong way to transfer the line-item
while (i.hasNext()) {
  lineItem = (LineItem) i.next();
  ls2.add(salesrep); // removes line-items from ls1
}
// this is a correct and safe way to transfer the line-items
while (i.hasNext()) {
  lineItem = (lineItem) i.next();
  i.remove();
  ls2.add(salesrep);
}
```

(2). 使用setter方法更新关系

对于容器管理关系域的setter方法，同样具有由关系的多重性带来的引用完整语义。

在one-to-many的关系中，某一实例中容器管理关系域的集合对象被赋值到拥有同种类性的容器管理关系的另一个实例时，集合中的对象即被移动，后一实例的集合域中的对象被前一实例集合域中的对象取代，但集合不变，只是前一实例中的集合对象元素被清空。

16.2.5.2.4 PRIMARY KEY

Entity Bean实例通常作为关系型数据库表中的一行数据，具有一个标识唯一性的Primary Key，通过Primary Key，开发者可以在部署描述对实例与其他类型的Entity Bean实例间的容器管理关系进行定义。

对于容器管理持久性的Entity Bean实例，容器在Primary Key的基础上维护一个Entity Bean的运行时的对象标识。类似于关系型数据库中的复合键（compound key），Primary Key可以由实例的一个或多个容器管理持久性的域构成。开发者通过部署描述中的prim-key-class元素指定Primary Key数据的类型，通过primkey-field元素指定Primary Key被映射到某个容器管理持久性的域。

容器必须可以操作实例的Primary Key的类型，因此具有容器管理持久性的Entity Bean的Primary Key的类型必须遵循如下规则：

- 必须是合法的RMI类型；
- 必须提供hashCode方法和equals方法的正确实现，以简化容器对Primary Key的管理；

有两种方法指定一个Primay Key的类（class）：

- 映射到Entity Bean类的一个域的类；

开发者可以使用部署描述中的primkey-field元素指定Entity Bean类中包含Primary Key的容器管理域，此域的类型必须是Primary Key的类型。

- 映射到Entity Bean类多个域的类；

Primary Key类必须是公有的（public），并且必须拥有一个无参数的共有构造方法（constructor）。

Primary Key类中的所有域必须是公有的（public）。

Primary Key类中的所有域的名字必须是所有容器管理域的名字集合的子集

16.2.5.2.5 实例的清除

对于Entity对象，实例的清除也涉及引用完整性方面的因素，可以通过以下两种方式对实例进行清除。

一. remove方法

开发者可以通过调用Entity对象的remove方法对Entity对象进行清除。在remove方法被调用后，容器将调用组件的ejbRemove方法，参见。容器将把Entity对象从它参与的所有关系中移除，并且清除其对应的持久数据表示。

- 当Entity对象被从关系中移除，关系所提供的可访问到此对象的方法将反映出对象被移除的结果。原通过one-to-one关系或many-to-one 关系可访问到此对象的方法，现会返回null值；而通过many-to-may关系可访问到此对象的方法将返回一个不包含此对象的集合对象。
- 对于已被移除的Entity对象的访问，如果是通过远程客户进行调用，容器将抛出java.rmi.NoSuchObjectException异常；如是通过本地客户进行调用，容器将抛出javax.ejb.NoSuchObjectLocalException异常。当把一个已被移除的Entity对象赋值给一个容器管理关系的域（无论是通过域的set方法或通过java.util.Collection API），容器将抛出java.lang.IllegalArgument异常。

另外，对于被移除对象参与的容器管理关系中关联的所有其他对象，如果其cascade-delete元素被指定，则移除操作将被级联（cascade）到这些对象。

对象的移除可能牵涉到多于一个的容器管理关系。如下例中，当Order对象关联的ShippingAddress对象被移除，则访问BillingAddress对象的方法将返回null。

```
public void changeAddress()

Address a = createAddress();

setShippingAddress(a);

setBillingAddress(a);

//both relationships now reference the same entity object
getShippingAddress().remove();

if (getBillingAddress() == null) //it must be
...

else
...
```

通常，对象的移除，即通过调用对象的remove方法进行的操作，只会导致此对象被移除。不会使移除操作级联（cascade）到其他对象。如要级联地自动移除其它对象，则需要使用级联（cascade）移除机制。

二. 级联（cascade）移除

在容器管理关系的部署描述中，cascade-delete元素被用于指定某些Entity对象的生命周期依赖于其它Entity对象的生存周期。换句话说，在某个容器管理关系中，当通过设置其cascade-delete元素，对参与关系的某一方设置其依赖的另一方时，如被依赖一方的对象实例被移除，容器将会把通过关系关联到的另一方的对象实例也移除。

cascade-delete元素包含在ejb-relationship-role元素中。ejb-relationship-role则被包含在ejb-relation元素中。在同一个ejb-relation元素中，只有当一个ejb-relationship-role元素的multiplicity被指定成为one时，才能对另一个ejb-relationship-role元素指定cascade-delete元素。即当容器管理关系是one-to-one的情况下，可以对关系的双方使用级联移除，当容器管理关系是one-to-many的情况下，只可对重复性为many的一方使用级联移除。

当容器管理关系中，存在指定了cascade-delete 元素的一方，假定被称为B方，另一方被称为A方，A1为某个A方的对象实例，A1通过容器管理关系关联的B方的对象或对象集合为B1，则当对A方的对象A1调用remove方法时，如B1是单个的对象，则B1也被容器通过类似调用remove方法的方式移除，如B1是对象集合，则对集合中的每个对象应用上一方法移除。

使用cascade-delete元素，只会导致关系中被指定的对象的移除。

如Book与PublishingHouse之间的关系，在ejb-jar.xml中，则可如下配置：

```
<ejb-relation>
    <ejb-relation-name>

BookAndPublishingHouse
```

```
</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>

Book
</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <cascade-delete />
        <relationship-role-source>
            <ejb-name>Book</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>publishingHouse</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>

PublishingHouse
</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
            <ejb-name>PublishingHouse</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>books</cmr-field-name>
            <cmr-field-type>

java.util.Collection
</cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
```

16.2.5.2.6 FINDER方法

开发者可以在Entity Bean的Home接口中定义零到多个finder方法，每个方法提供一种在同一Home接口中查找Entity对象和Entity对象集合的方式。finder方法必须以"find"开头，如findLargeAccounts(...)。finder方法的参数用在查询的实现中，用于finder 方法定位需要查找的Entity对象。

除findByPrimaryKey方法之外，每个finder方法必须在部署描述中定义query元素，另外，如果Finder方法符合容器自动生成查询QL语法定义此方法需要执行的查询。有关具体的EJB QL语法，参见EJBQL。

开发者在Entity Bean的本地Home接口和远程Home接口中，使用同样的名字和参数类型定义finder方法的情况下，在query元素中指定的查询语句同时定义这两个方法的语义。

一. 单个对象的查找

某些finder方法，被设计为返回一个Entity对象。对于这一类finder方法，如果是在Entity Bean的远程Home接口中定义，则方法的返回值类型必须是Entity Bean的远程接口；如果是在Entity Bean的本地Home接口中定义，则方法的返回值类型必须是Entity Bean的本地接口。

以下代码是远程Home接口中定义一个返回单个Entity对象的finder方法范例：

```
// Entity's home interface
public interface BookHome extends javax.ejb.EJBHome {
...
  public Book findByPrimaryKey(String bookId) throws FinderException;
...
}
```

	**注意**
	本地Home接口中定义的finder方法不可抛出java.rmi.RemoteException异常。

当使用除findByPrimaryKey之外的返回单个Entity对象的finder方法时，必须保证finder方法确实返回的是单个Entity对象。因为如果在部署描述中为finder方法指定的EJB QL语句中包含了对Entity Bean的Primary Key中某个域的等式判断（equality test），这种不返回单个Entity对象的情况就有可能发生。

二. 多个对象的查找

某些finder方法可能会被指定查找多个Entity对象，对于这一类finder方法，如果是在Entity Bean的远程Home接口中定义，则方法的返回值类型必须是由Entity Bean的远程接口构成的集合；如果是在Entity Bean的本地Home接口中定义，则方法的返回值类型必须是Entity Bean的本地接口构成的集合。

对于容器管理持久性的Entity Bean，开发者使用java.util.Collection接口定义返回值的集合类型。

当在finder方法的查询语句中的SELECT子句中，未使用DISTINCT进行限制，则返回的集合类型中可能会包含重复的元素。

当使用在远程Home接口中定义的finder方法时，将集合中的元素定位到Entity Bean的远程接口时，必须使用PortableRemoteObject.narrow(...)方法。

以下代码是远程Home接口中定义一个返回多个Entity对象的finder方法范例：

```
// Entity's home interface
public interface BookHome extends javax.ejb.EJBHome {
...
```

```
  public Collection findByTitle(String title) throws FinderException;
...
}
```

	**注意**
	本地Home接口中定义的finder方法不可抛出java.rmi.RemoteException异常。

三. **容器自动生成查询**

Apusic应用服务器还提供了自动生成缺省查询的功能，以提高应用程序的开发效率。在以下几种情况下，容器将自动为finder方法生成缺省的查询语句：

- 如在Home接口中声明了findAll方法， 容器将自动生成查询所有行的语句；
- 如在Home接口中定义的finder方法名为findBy<域名>，其中，域名是一个容器管理持久性域的名字，且第一个字母大写并且参数类型与容器管理持久性域类型相同，容器自动生成按此容器管理持久性域进行查找的查询语句。

16.2.5.2.7 SELECT方法

select方法是Entity对象中用于查询的方法。不同于Finder方法，select方法不能在Entity Bean中的Home接口中定义，只是开发者定义在Entity Bean组件类中的抽象方法，并且不通过Home接口或组件接口暴露给客户端。

与finder方法相同的是，select方法也使用EJB QL查询语句定义语义，但是，select方法可以返回相当于容器管理持久性域（cmp-field）或容器管理关系域（cmr-field）的值。

每个select方法必须在部署描述中定义query元素。通过此元素，开发者在部署描述中使用EJB QL语法定义此方法需要执行的查询。有关具体的EJB QL语法，参见“EJB QL”。

一般，select方法返回的Entity对象都是EJBLocalObject对象。如果select方法需要返回EJBObject对象，必须在select方法的部署描述元素query中设置result-type-mapping的值为Remote。

对select方法的调用并不基于被调用实例本身的对象标识，但是，开发者可以使用Entity对象的Primary Key作为select方法的参数，以定义特定Entity Bean实例查询的逻辑范围。

下表是select方法与finder方法语义的对比：

	finder方法	select方法
方法名	以“find”开头	以“ejbSelect”开头
可见性	客户端可见	Entity Bean类内部
实例	实例池状态的任意实例	当前实例，可以是实例池状态，也可是就绪状态

返回值	Entity Bean中的本地接口类型或远程接口类型	Entity Bean中的本地接口类型、远程接口类型或容器管理持久性域（cmp-field）类型

一. 返回单个对象的select方法

某些select方法被指定只能返回一个值，当开发者定义这一类方法，必须保证方法只能返回一个值或者一个对象。如果select方法对应的查询语句对指定的类型返回了多个值，容器将抛出javax.ejb.FinderException异常。

select方法通常被定义为返回多个对象。

二. 返回多个对象的select方法

某些select方法被指定能返回多个值，对于这一类方法，返回值类型通常是对象集合。select方法返回的值对象集合需要使用java.util.Collection接口或java.util.Set接口作为返回值类型。集合中的元素类型取决于相应的EJB QL语句的SELECT子句。如果使用java.util.Collection作为返回值类型，并且SELECT子句不包含DISTINCT作为限制，则返回的java.util.Collection集合中可能包含重复的元素；如果使用java.util.Set作为返回值类型，并且SELECT子句不包含DISTINCT作为限制，则容器将把查询语句替换为SELECT DISTINCT。

下面是BookEJB中返回多个对象的select方法的范例：

```
// Book implementation class
public abstract class BookEJB implements javax.ejb.EntityBean
{

...

public abstract java.util.Collection
ejbSelectAllOrderedAuthors()

throws FinderException;

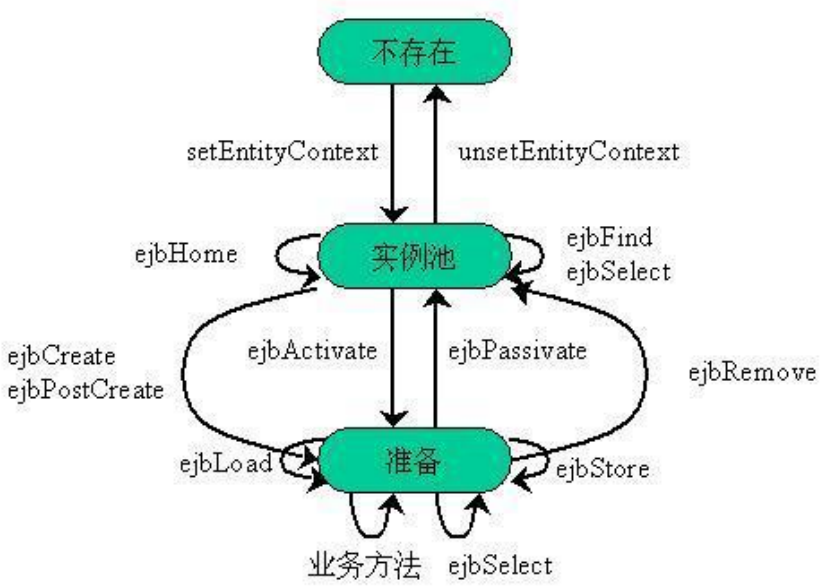
// internal finder method to find all authors ordered
...
```

16.2.5.2.8 实例的生存周期与开发中的约定

在"Entity Bean"中，讲述了Entity Bean共有的组件模型和组件接口，同时，从客户端的角度描述了Entity Bean的生存周期，但是，容器管理持久性的Entity Bean和Bean管理持久性的Entity Bean在组件类和组件接口的开发上，存在很多不同，因此，本节内容包含容器管理持久性Entity Bean的详细生存周期，以及开发容器管理持久性Entity Bean时，需要注意的一些约定和规则，关于Bean管理持久性的Entity Bean的开发，参见 "Bean管理持久性的Entity Bean"。

一. 实例的生存周期

实例的生存周期



容器管理持久性的Entity Bean实例有如下几个状态：

- 不存在状态；
- 实例池状态。处于实例池状态中的实例没有特别的对象标识；
- 就绪状态。处于就绪状态的实例已被分配对象标识。

容器管理持久性的Entity Bean实例的生存周期如下：

- 实例的生存周期开始于容器创建新的Entity实例，然后容器调用实例的setEntityContext方法，将容器保存的伴随实例的EntityContext对象传递给实例。EntityContext对象使实例拥有访问容器提供的服务和取得客户端调用信息的能力；
- 实例进入实例池中，成为可用实例。每个Entity Bean拥有其自身的实例池。实例池中的所有实例无对象标识，且都是相等的，任何实例都可通过分配对象标识进入就绪状态。对于实例池中的实例，容器可以调用实例的ejbFind方法和ejbHome方法，这两种方法的对应客户端视图在组件的Home接口中定义，执行这些方法不会导致实例改变其状态，同时可以在ejbHome方法中调用实例的ejbSelect方法。
- 当客户端对Entity对象进行调用，容器选择某实例以服务此调用时，实例从实例池状态转移到就绪状态。这种转换由两种可能的方式：一是通过ejbCreate方法和ejbPostCreate方法，即当客户端调用了Home接口中定义的create方法，容器在创建Entity对象的过程中，将实例分配给Entity对象；另一种方式是当对已存在的Entity对象的调用发生，当没有合适的处于就绪状态的实例可分配给客户端请求时，容器通过调用ejbActivate方法，使实例由实例池的状态转移到就绪状态；
- 处于就绪状态中的实例拥有指定的对象标识，当实例状态发生变化或与底层数据存储中的数据不一致，容器将会决定何时调用ejbLoad方法和ejbStore方法，以对实例状态和业务数据进行同步。当实例处于就绪状态，ejbSelect方法可以由业务方法或ejbStore或ejbLoad方法所调用；
- 容器可以在事务边界中选择钝化一个Entity Bean实例。容器首先调用ejbStore方法保证实例状态与业务数据的同步，之后调用ejbPassivate方法，使实例转移到实例池的状态；
- 最终，容器将实例转移到实例池状态。有三种方式完成转移，一种是通过ejbPassivate方法，通过ejbRemove方法，还有就是因ejbCreate、ejbPostCreate和ejbRemove等方法的事务回滚发生的转移（上图中未标注）。容器在需要解除实例的对象

标识但不移除对象的情况下，调用ejbPassivate。当组件接口中或Home接口中的remove方法被调用后，容器调用ejbRemove方法移除Entity对象。当ejbCreate方法、ejbPostCreate方法和ejbRemove方法被调用，但发生了事务回滚，容器将把实例转移到实例池状态。

- 当实例返回实例池中，不再拥有对象标识。容器可以把实例分配给任何同一Home接口的Entity对象；
- 容器通过调用实例的unsetEntityContext方法清除实例。

二. 开发中的约定

容器管理持久性Entity Bean的开发者除必须提供EJB组件具有的Home接口、组件接口之外，提供的组件类必须是实现（extends）javax.ejb.EntityBean接口的抽象（abstract）类，而且必须按照如下约定编写组件类：

- 提供一个无参数的公有构建器（public constructor）；
- 实现setEntityContext方法。

容器通过此方法将一个EntityContext接口的引用传递给实例。如果实例希望在其生存周期内使用此引用，一般需要定义一个变量以保存此引用。在此方法中，实例的对象标识不可用，在此方法内部不可访问实例的持久状态和关系。

通过此方法，开发者可以对实例生存周期内将要使用的资源进行分配，此类资源不可被指定到特定的Entity对象标识，因为在实例的生存周期内部，实例可能会服务于多个Entity对象标识。

- 实现unsetEntityContext方法。

在结束实例的生存周期之前，容器将会调用此方法。在此方法中，实例的对象标识不可用，在此方法内部不可访问实例的持久状态和关系。通过此方法，开发者可以对实例生存周期内占用的资源进行释放，此类资源通常在setEntityContext内分配。

- 实现ejbCreate方法。

Entity Bean的组件类中可以定义零个到多个ejbCreate方法。其方法签名必须与组件的Home接口中定义的create方法一一对应。在客户端调用create方法时，容器将调用对应的ejbCreate方法。

开发者通过getter和setter方法，使用ejbCreate方法的输入参数，对实例进行初始化，当ejbCreate方法返回时，在底层数据存储中创建对象的持久数据表示。在此方法中，对容器管理域的getter方法的调用将返回Java语言中的缺省值（如integer型的域返回0，指针类型返回null），除了集合类型的容器管理关系域，对它的getter方法的调用将返回一个空集合。在此方法中，开发者不能试图去修改容器管理关系域，而应该在ejbPostCreate方法中进行此操作。

ejbCreate方法创建的Entity对象必须拥有一个非重复的Primary Key，即不同于所有已经存在的，同一Home接口的Entity对象已经拥有的Primary Key。

	**注意**
	开发者应该在ejbCreate方法中编码返回null值，如下：...public String ejbCreate(String id,String title,double price) throws CreateException {setId(id);setTitle(title);setPrice(price);return null;}...

ejbCreate方法的事务上下文由对应的create方法指定的事务属性决定。容器在开发者实现的ejbCreate方法完成后，在同一事务上下文中完成数据的插入操作。

- 实现ejbPostCreate方法。

每个ejbCreate方法都有一个对应的ejbPostCreate方法，这两个对应方法拥有同样的参数数目和类型，但ejbPostCreate方法的返回值类型为void，容器在调用ejbCreate方法后，使用同样参数调用ejbPostCreate方法。容器可以调用随同实例的EntityContext对象引用的getPrimaryKey方法，得到实例的Primary Key。

在ejbPostCreate方法中，对象标识是可用的，例如，可以在方法中取得对象的组件接口并作为参数传递给另一个EJB。

ejbPostCreate方法与ejbCreate方法在同一事务上下文中执行。

- 实现ejbActivate方法。

当容器从实例池中取得实例并分配给实例一个指定对象标识时，容器将调用此方法，开发者可以在此方法中为实例将在就绪状态中使用的资源进行分配。在此方法中，不可访问实例的持久状态和关系。

实例可以通过伴随实例的EntityContext对象引用的getPrimaryKey、getEJBLocalObject或getEJBObject等方法取得Entity对象的标识。

- 实现ejbPassivate方法。

当容器决定清除实例的对象标识时，容器将调用此方法，并将实例返回到实例池中。开发者可以在此方法中为实例在就绪状态中占用的资源进行释放，通常这些资源在ejbActivate方法中分配。在此方法中，不可访问实例的持久状态和关系。

实例仍然可以通过伴随实例的EntityContext对象引用的getPrimaryKey、getEJBLocalObject或getEJBObject等方法取得Entity对象的标识。

- 实现ejbRemove方法。

当客户端调用组件的Home接口或组件接口中的remove方法，或者作为级联移除的结果，容器将调用实例的ejbRemove方法。当ejbRemove方法结束，实例转移到实例池的状态。

在实例对应的持久业务数据被清除之前，开发者可以在此方法中进行必要的操作。容器在调用ejbRemove方法之前，将会对实例状态与持久存储中的业务数据进行同步。

ejbRemove 方法的事务上下文决定于引发ejbRemove方法的对应remove方法的事务属性。在此方法中，实例仍然可以通过伴随实例的EntityContext对象引用的getPrimaryKey、getEJBLocalObject或getEJBObject等方法取得Entity对象的标识。

当ejbRemove方法返回，容器将在同一个事务上下文中，将实例从参与的所有容器管理关系从持久存储中移除。

在此方法执行完成之后的实例状态与实例钝化后的状态相同，准备进入实例池，因此，实例必须释放通常在ejbPassivate方法中释放的资源。

- 实现ejbLoad方法。

当容器决定同步实例状态与持久数据时，容器将调用此方法。在ejbLoad方法被调用之前，容器已经将持久数据装入，因此，开发者需要在此方法中对依赖于持久数据的实例变量进行重新计算和初始化。例如，开发者可以通过ejbLoad方法，对通过getter访问方法返回的值进行重新计算，如文本解压缩或进行解密操作。

执行ejbLoad方法的事务上下文取决于引发ejbLoad的业务方法的事务属性。

- 实现ejbStore方法。

当容器决定同步持久数据与实例状态时，容器将调用此方法。在同步实例状态之前，开发者应该使用域访问方法（getter、setter）更新实例，如在使用ejbStore存储数据到持久存储之前，将某文本域的值进行压缩等。

- 实现ejbFind方法。

对于容器管理持久性的Entity Bean，开发者不必编写ejbFind方法的内容。只需使用EJB QL查询语言在部署描述中指定查询语句。有关EJB QL查询语言，参见 "EJB QL"。有关finder方法的具体内容，参见"Finder方法"。

- 实现ejbHome方法。

当客户端通过Home接口引用调用Home方法时，容器将选择一个实例并调用对应的ejbHome方法。当容器调用实例的ejbHome方法时，实例处于实例池中，在方法调用返回之后，实例又回到实例池中。

执行ejbHome方法的事务上下文取决于对应的Home接口中Home方法的事务属性。

开发者必须在ejbHome方法中提供对Home接口中定义的Home方法的实现。在此方法中，不可访问实例的持久状态和关系。

- 实现ejbSelect方法。

开发者可以在组件类中定义零到多个ejbSelect方法。ejbSelect方法并不暴露在Home接口和组件接口中，对客户端是不可见的。通常，开发者通过业务方法调用ejbSelect方法。

ejbSelect方法必须被定义为抽象（abstract）的。

ejbSelect方法由开发者在部署描述中指定的信息决定，开发者不需要对ejbSelect方法进行编码。

定义ejbSelect方法的语法参见EJB QL。有关select方法的具体内容，参见"Select方法"。

执行ejbSelect方法的事务上下文取决于调用ejbSelect方法的业务方法的事务上下文。

16.2.5.3 必须遵守的规则

本节提供对开发容器管理持久性的Entity Bean过程中必须遵循的一些规则。

16.2.5.3.1 类与接口

开发者必须提供如下类文件：

- Entity Bean组件类；
- Primary Key类文件；
- 如Entity Bean提供客户端远程访问，必须提供远程接口和远程Home接口；
- 如Entity Bean提供客户端本地访问，必须提供本地接口和本地Home接口；

16.2.5.3.2 组件类

开发者在编写组件类时必须遵守如下规则：

- 组件类必须实现javax.ejb.EntityBean接口；
- 类必须声明为公有（public）类，并且必须是抽象（abstract）类；
- 类不能定义finalize方法；
- 类可以实现Enttiy Bean的组件接口。如果类实现了组件接口，则类必须提供组件接口中定义的方法的无操作（no-op）实现，因为在实例的生存周期内，容器将不会调用这些方法；

- 类必须提供业务方法、ejbCreate方法、ejbPostCreate方法。
- 在组件的Home接口中定义的Home业务方法，开发者必须在组件类中提供对应的ejbHome方法实现。当实例在实例池状态时，可以调用这些方法。
- 组件类中必须提供对Bean的抽象持久模式（abstract persistent schema）的get和set访问方法，这些方法必须被声明为抽象（abstract）的。
- 组件类可以继承某个类或接口，如果组件类继承某个类，则业务方法、ejbCreate方法、ejbPostCreate方法等从javax.ejb.EntityBean接口继承的方法实现，可以在组件类或组件类继承的类中提供。
- 组件类不实现finder方法。finder方法由容器提供实现。
- 必须实现抽象（abstract）的ejbSelect方法。

16.2.5.3.3 辅助类

对于Entity Bean的辅助类，必须遵循如下规则：

- 类必须被定义为公有（public）类 ，并且不能被定义为（abstract）类；
- 类必须可串行化。

16.2.5.3.4 EJBCREATE方法

Entity Bean的Home接口中定义的create方法，组件类中必须提供对应的ejbCreate方法。Entity Bean可有零到多个ejbCreate 方法，开发者必须遵循如下ejbCreate方法的签名规则：

- 方法名必须由“ejbCreate”开头；
- 方法必须是公有的（public）；
- 方法不能声明为final或static；
- 返回值类型必须是Entity Bean的Primary Key的类型；
- 如果ejbCreate方法对应于远程Home接口中定义的create方法，则方法参数和值类型必须是合法的RMI类型；
- 抛出子句必须包含对javax.ejb.CreateException异常的声明，可包含任意应用级异常。

16.2.5.3.5 EJBPOSTCREATE方法

对每个ejbCreate 方法，开发者必须提供对应的ejbPostCreate方法实现，并遵循如下规则：

- 方法名必须由“ejbPostCreate”开头；
- 方法必须是公有的（public）；
- 方法不能声明为final或static；
- 返回值类型必须是void类型；
- 必须与对应的ejbCreate方法具有同样的参数；

- 抛出子句可包含任意应用级异常声明，包括javax.ejb.CreateException异常的声明。

16.2.6 EJB QL

摘要

Enterprise JavaBeans Query Language，简称EJB QL，是用于对容器管理持久性的Entity Bean的查询进行定义的语言。EJB QL使Entity Bean的查询方法的语义定义更具可移植性。

16.2.6.1 概述

EJB QL是用于容器管理持久性Entity Bean中finder方法和select方法的标准查询语言。EJB QL可被编译成为目标语言，如数据库或其他持久存储所使用的SQL语言，从而将查询转移到由持久存储提供的本地（native）语言运行环境。因此，EJB QL在提供可移植能力的同时可提高执行效率。

EJB QL使用Entity Bean的抽象持久模式（abstract persistence schema）和关系作为数据模型。在这些数据模型的基础上定义了运算符和表达式。

基于在部署描述中定义的抽象持久模式和关系，开发者使用EJB QL编写查询。相互关联的Entity Bean中定义的cmp-field和cmr-field决定Entity Bean之间进行定位（navigation）与选取（selection）的能力，EJB QL则取决于这种能力。通过在EJB QL中使用cmr-field域的名字，开发者可以从一个Entity Bean定位到其他的Entity Bean。

如在同一个部署描述文件中定义了Entity Bean的抽象持久模式与EJB QL查询，则开发者可以在查询中使用Entity Bean的抽象模式类型。

EJB QL有以下两种使用方式：

- 通过定义在Entity Bean的Home接口中的finder方法，EJB QL可用于编写选取（select）Entity对象的查询。查询的结果可被Entity Bean的客户端使用。
- 通过定义在Entity Bean的组件类中的select方法，EJB QL可用于编写选取（select）Entity对象或从Entity Bean的抽象模式类型派生的其他值的查询。开发者可以使用select方法查找对象或与Entity Bean状态相关的值，且结果不会暴露给客户端。

一个EJB QL查询语句必须包含一个SELECT子句和一个FROM子句，可以包含WHERE子句和ORDER BY子句。

16.2.6.2 定义

EJB QL使用类似于SQL的语法，基于Entity Bean的抽象模式类型和关系，对值或对象进行选取。对于使用Entity Bean的cmr-field定义的关系，可使用EJB QL中的路径表达式（path expressions）在关系中进行定位。

本节提供EJB QL语言的完整定义。

EJB QL是包含以下四个部分的字符串：

- SELECT子句，用于定义被选取对象或值的类型；
- FROM子句，对于在SELECT子句中表达式需要指定的范围和应用于WHERE子句的查询范围，可使用FROM子句进行声明；
- 可选的WHERE子句，用于限制查询返回的结果。
- 可选的ORDER BY子句，用于对查询结果进行排序。

EJB QL可使用BNF语法定义如下：

EJB QL ::= select_clause from_clause [where_clause] [order by_clause]

EJB QL查询必须有一个SELECT子句和一个FROM子句，方括号表示WHERE和ORDER BY是可选的。

对应于使用EJB QL查询定义语义的finder方法和select方法，查询可以使用输入参数。

EJB QL查询使用ejb-ql部署描述元素进行定义。

16.2.6.2.1 抽象持久类型与查询范围

EJB QL是基于EJB2.1容器管理持久性类型模型而设计的类型表达式语言。每个EJB QL表达式都拥有类型。表达式的类型取决于表达式的结构、标记变量声明（indetification variable declaration）的抽象模式类型、对cmp-field和cmr-field的进行求值的类型和字面值的类型。EJB QL中容许类型是Entity Bean的抽象模式类型和cmp-field的定义类型。

由Entity Bean类和部署描述中提供的信息得到Entity Bean的抽象模式类型。抽象模式类型和Entity Bean之间存在一对一的关系。定义在abstract-schema-name部署描述元素中的抽象模式的名字，用于在EJB QL中标识抽象模式类型。

抽象模式类型拥有以下特征：

- 对于每个对应部署描述中的一个cmp-field元素的Entity Bean类的getter访问方法，存在一个域，其类型与cmp-field元素指定的类型相同。
- Entity Bean类中，对于每个对应部署描述中的cmr-field元素的getter访问方法，存在一个域，其类型是Entity Bean的抽象模式类型。此Entity Bean由包含在对应的ejb-relationship-role元素中的ejb-name子元素指定，如角色的多重性（multiplicity）为many，则类型是包含此Entity Bean抽象模式类型实例的集合。

EJB QL查询的范围，由同一个部署描述中的所有容器管理持久性的Entity Bean组成。

EJB QL查询的可定位范围，由Entity Bean抽象模式类型中的cmr-field决定。使用cmr-field，查询可以选取关联的Entity Bean和在查询中使用关联的Entity Bean的抽象模式类型。

16.2.6.2.2 命名

在EJB QL查询字符串中使用Entity Bean的抽象模式名来指定Entity Bean。

在部署过程中，开发者需要为每个容器管理持久性的Entity Bean分配一个唯一的抽象模式名字，以在查询中使用此名字来指定Entity Bean。名字的作用域范围在整个部署描述文件之内。

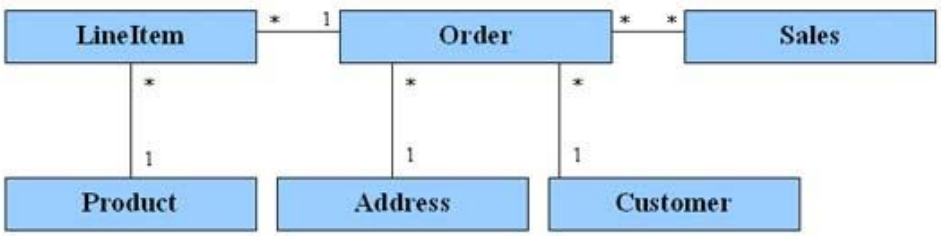
16.2.6.2.3 范例

本节中所使用的Entity Bean范例名字遵循如下约定：Entity Bean使用<name>EJB方式进行命名，Entity Bean类和抽象模式类型使用<name>方式进行命名。如表示订单的Entity Bean命名为OrderEJB，其Entity Bean类和抽象模式类型名称为Order。

例如某个部署描述文件中包含如下几个Entity Bean，OrderEJB、ProductEJB、LineItemEJB、ShippingAddressEJB和BillingAddressEJB，对应的抽象模式类型名字为Order、Product、LineItem、ShippingAddress和BillingAddress。只有OrderEJB和ProductEJB拥有远程和远程Home接口。

以上Entity Bean之间的关系如下图：

范例关系图



开发者可以通过Order和LineItem中定义的cmr-field和cmp-field，为OrderEJB定义finder查询，查找所有未完成的Order的finder查询可用如下方式编写：

```
SELECT DISTINCT OBJECT(o)

FROM Order AS o, IN(o.lineItems) AS l
WHERE l.shipped = FALSE
```

查询定位到Order抽象模式类型中的cmr-field域lineItems，并在其中中遍历对lineItems进行查找，并使用LineItem中的cmp-field域shipped选择至少有一个lineItem未发送的Order的集合。

尽管上面的例子使用了如SELECT、DISTINCT、OBJECT、FROM、AS、IN、WHERE和FALSE等大写的保留字，但是，EJB QL中的保留字是大小写无关的。

上例中的SELECT子句指定了查询的返回值类型是Order，如这个查询语句用于定义一个在Entity Bean的远程Home接口中的finder方法，则对应于由查询所选取的抽象模式类型实例，方法返回值类型为Entity Bean的远程接口类型。如这个查询语句用于定义一个在Entity Bean的本地Home接口中的finder方法，则方法返回值类型为Entity Bean的本地接口类型。

因为同一个部署描述中定义了相关的Entity Bean的抽象持久模式，所以同样可以在OrderEJB的查询中使用ProductEJB的抽象模式类型，也可使用Order和Product抽象模式类型中的cmp-field和cmr-field。例如，Product抽象模式类型中包含一个product_type的cmp-field，OrderEJB中的finder查询可以使用这个cmp-field。如"查找包含产品种类为'办公用品'的订单"，则可以用如下查询语句定义查询方法：

```
SELECT DISTINCT OBJECT(o)

FROM Order o, IN(o.lineItems) l
WHERE l.product.product_type = 'office_supplies'
```

通过Order与LineItem、LineItem和Product的关联，Order关联到Product，为表达此查询，必须通过lineItem和product 这两个cmr-field进行定位（navigation）。通过在整个语句范围内指定使用OrderEJB的抽象模式名称Order指定了查询的类型。

本部分文档中的余下部分将对本节中的例子进行扩展，以详细说明EJB QL的特性和用法。

16.2.6.2.4 返回值类型

EJB QL查询使用Entity Bean的抽象模式类型进行编写。使用SELECT子句指定的返回值类型，应该是Entity Bean的抽象模式类型或者是cmp-field类型。finder和select方法和在部署描述文件中定义的查询语句，决定了finder和select方法返回的结果被映射到何种Java

类型。

查询结果如何映射取决于查询语句定义的查询方法是finder方法还是select方法，也取决于finder方法是定义在远程Home接口中还是定义在本地Home接口中。

如前面的例子：“查找包含产品种类为‘办公用品’的订单”

- 如查询被用于远程Home接口中的finder方法，finder方法的结果是Entity Bean的远程接口（或是远程接口的集合）。如查询被用于本地Home接口中的finder方法，finder方法的结果是Entity Bean的本地接口（或是本地接口的集合）。
- 同样的查询如被用于select方法，返回值类型是Entity Bean的远程接口或本地接口实例，最终返回的何种接口则取决于包含在query元素中的部署描述元素result-type-mapping的值是Local还是Remote，缺省为Local。

16.2.6.2.5 FROM子句与定位声明

EJB QL的FROM子句通过声明标记变量（indetification variable）定义查询的范围。范围可以使用路径表达式进行限制。

标记变量（indetification variable）指定特定的Entity Bean的抽象模式类型实例。通过在标记变量（indetification variable）之间使用逗号分隔，FROM子句可以包含多个标记变量（indetification variable）。

```
from_clause ::=

FROM identification_variable_declaration
[, identification_variable_declaration]*

identification_variable_declaration ::=

collection_memeber_declaration |range_member_declaration
collection_member_declaration ::=

IN ( colleciton_value_path_expression) [AS] identifier
range_variable_declaration ::= abstract_schema_name [AS]
identifier
```

下面章节讨论FROM子句的构成。

一. 标识符（Identifier）

标识符是不限长的字符序列，首字符必须使用Java标识符的首字符（start character）开头，其他字母必须是Java标识符的组成字符（part character）。首字符可以是任何使用Character.isJavaIdentifierStart方法返回真值的字符，包括下划线（_）字符和美元符（$）。其他字符则可以是使用Character.isJavaIdentifierPart方法返回真值的字符，问号（?）是EJB QL的保留字符。

以下是EJB QL的保留字列表：SELECT、FROM、WHERE、DISTINCT、OBJECT、NULL、TRUE、FALSE、NOT、AND、OR、BETWEEN、LIKE、IN、AS、UNKNOWN、EMPTY、MEMBER、OF、IS

保留字是大小写无关的。

建议不要在EJB QL查询语句中使用SQL的保留字作为标识符，EJB QL的以后版本可能会使用此类保留字。

二. 标记变量(indentification variables)

标记变量（indetification variable）是EJB QL查询的FROM子句中声明的一个有效标识符。标记变量（indetification variable）可以使用特殊操作符IN和可选的操作符AS进行声明。

标记变量（indetification variable）必须声明在FROM子句中。

一个标记变量（indetification variable）是一个标识符。标记变量（indetification variable）不能是保留字或：

- 抽象模式名（abstract-schema-name）
- EJB名字（ejb-name）

标记变量（indetification variable）是大小写有关的。

标记变量（indetification variable）的求值使用声明变量的表达式类型。如上面OrderEJB中finder方法查询语句：

```
SELECT DISTINCT OBJECT(o)

FROM Order o, IN(o.lineItems) l
WHERE l.product.product_type = 'office_supplies'
```

FROM子句中声明了 IN(o.lineItems) l，则标记变量（indetification variable）l对Order直接关联的所有lineItem进行求值。cmr-field域lineItems是抽象模式类型LineItem的实例集合，l是集合中的一个元素，l的类型是LineItem的抽象模式类型。

标记变量（indetification variable）用于指定一个Entity Bean的抽象模式类型的实例或者Entity Bean的抽象模式类型实例集合中的一个元素。

因此，标记变量（indetification variable）代表对某个值的引用，并以以下两种方式进行声明：范围（range）变量或者集合成员变量：

- 范围变量使用Entity Bean的抽象模式名进行声明；
- 集合成员变量使用得到集合结果的路径表达式进行声明。

在 FROM子句中，标记变量（indetification variable）声明由左到右进行求值。查询语句中后续的标记变量（indetification variable）声明可以可以使用前面标记变量（indetification variable）声明的结果。

三. 范围变量声明

一个标记变量（indetification variable）包括一个Entity Bean的抽象模式类型。将标记变量（indetification variable）声明为一个范围变量的EJB QL语法与SQL相同：使用可选的AS操作符进行声明。

关联到Entity Bean的对象和变量一般使用路径表达式进行定位。但是，定位不可能针对所有的对象。对通过定位不能到达的对象，范围变量声明允许开发者为其指派一个"root"。

因此，开发者可在查询中使用多个范围变量声明对相同抽象模式类型的多个值进行比较。参见"SELECT子句"。

四. **集合成员变量声明**

一个使用collection_member_declaration声明的标记变量（indetification variable）包括通过路径表达式定位取得的集合中的所有值。路径表达式表示包含Entity Bean抽象模式类型的cmr-field的定位操作。因为一个路径表达式可以基于另一个路径表达式，则定位操作可以使用关联的Entity Bean的cmr-field。参见"路径表达式"。

集合成员变量声明使用特殊操作符IN进行声明。用IN操作符指定的功能表达式接收一个集合值类型的路径表达式参数。路径表达式使用集合类型进行求值，此集合类型是对Entity Bean抽象模式类型中一个集合类型的cmr-field的定位操作，所得结果的类型。

例如，OrderEJB中finder方法的查询语句中，FROM子句包含如下标记变量（indetification variable）声明子句：

```
IN(o.lineItems) l
```

lineItems 是一个cmr-field域的名字，其值是LineItemEJB的抽象模式类型LineItem的实例集合。标记变量（indetification variable）l是集合中的一个成员，一个单个的LineItem实例。本例中的o是抽象模式类型Order的一个标记变量（indetification variable）。

五. **范例**

下面的FROM子句包含两个标记变量（indetification variable）的声明子句，第一个子句中声明的标记变量（indetification variable）被第二个子句使用，此子句声明两个变量o和l。Order AS o是一个范围变量声明，将变量o指定为范围变量，类型为抽象模式类型Order。变量l的类型为抽象模式类型LineItem。因子句从左到右求值，变量l可以使用o的结果进行定位操作。

```
FROM Order AS o, IN(o.lineItems) l
```

六. **路径表达式**

路径表达式包含一个变量，变量后的定位操作符（.）和一个cmp-field或一个cmr-field三个部分。

根据其定位能力，指向一个cmr-field的路径表达式可以更进一步进行组合。如果基本的路径表达式是对某个cmr-field进行单一值类型（非集合）的求值，则路径表达式可以由其他路径表达式组成。路径表达式的类型是由定位操作的结果计算出来的。归结到一个cmp-field域的路径表达式是最终形式，不能进行更进一步的组合。

单一值路径表达式和集合值路径表达式的语法定义如下：

```
single_valued_path_expression ::=

{single_valued_navigation | identification_variable} . cmp_field |

single_valued_navigation
single_valued_navigation ::=

identification_variable . [single_valued_cmr_field .]*
single_valued_cmr_field
collection_valued_path_expression ::=
```

```
identification_variable . [single_valued_cmr_field .]*
collection_valued_cmr_field
```

一个single_valued_cmr_field的指定需要使用one-to-many或many-to-one关系中的一个cmr-field的名字。单一值cmr-field的路径表达式类型是关联的Entity Bean的抽象模式类型。

一个collection_valued_cmr_field的指定需要使用one-to-many或many-to-many关系中的一个cmr-field的名字。表达式的类型是关联的Entity Bean的抽象模式类型。collection_valued_cmr_field的类型是关联的Entity Bean的抽象模式类型的值的集合。

对关联Entity Bean进行定位操作得到的值，其类型是此关联Entity Bean的抽象模式类型。

例如，如l是表示LineItem类型实例的标记变量（indetification variable），路径表达式l.product的类型为抽象模式类型Product。

对路径表达式的求值以一个cmr-field终止，将得出一个此cmr-field的Java类型的值。如l.product.name得到的类型是java.lang.String。

一个路径表达式由一个类型为集合的路径表达式构成，在语法上是正确的。例如，如果o的类型是Order，路径表达式o.lineItems.product 是正确的，因为对lineItems的定位将得到一个集合。当验证查询语句的时候，这种情况将产生错误。为处理这种定位，在FROM子句中必须声明一个用于包含lineItems集合元素的标记变量（indetification variable），必须在查询语句的WHERE子句中，使用另一个路径表达式，才能对每个元素进行定位操作。如下例：

```
SELECT OBJECT(o)

FROM Order AS o, IN(o.lineItems) l
WHERE l.product.name = 'widget'
```

在Apusic应用服务器中，如果Entity Bean的抽象持久类型中包含辅助值对象（Dependent Value Object）时，可使用“.”运算符访问Entity Bean内部的辅助值对象的域。如上一章“容器管理持久性的Entity Bean”中使用的范例，Customer实体中包含了类型为ContactInfo的info域，则可以使用如下EJB QL查询语句查询ContactInfo中name为“Jim Clark”的实例：

```
SELECT OBJECT(o)

FROM Customer AS o
WHERE o.info.name = 'Jim Clark'
```

七. WHERE子句与条件表达式

WHERE子句由用于选取对象和值的条件表达式构成。WHERE子句对查询的结果进行限制。

WHERE子句的定义如下：

```
where_clause ::= WHERE conditional_expression
```

(1). **字面值（literals）**

字符串的字面值由单引号括起，如：'literal'。包含单引号的字符串字面值，将使用两个连续的单引号表示一个单引号，例如：'literal''s'。EJB QL中的字符串字面值如同Java中的字符串字面值一样，使用Unicode字符编码。

一个精确数字字面值是一个不带小数点的数字值，如57、-957、+62。精确数字字面值支持Java中long型的数值范围。精确数字字面值使用Java中的整数语法。

一个近似数字字面值是使用科学计数法表示的数字值，如7E3、-59.7E2，或是带小数点的数字值，如7.、-95.7、+6.2。精确数字字面值支持Java中double型的数值范围。精确数字字面值使用Java中的浮点数语法。

可以使用Java语言规范中规定的后缀指明字面的确切类型。

布尔值的字面值为大小写无关的TRUE和FALSE。

(2). **标记变量（indetification variable）**

在EJB QL查询语句中，所有的用在WHERE子句的标记变量（indetification variable）必须在FROM子句中声明，参见 "标记变量(indentification variables)"。

在 WHERE子句中，标记变量（indetification variable）被实际量化，即标记变量（indetification variable）代表Entity Bean的抽象模式类型的单个实例或实例集合中的成员。标记变量（indetification variable）不能被代表一个表示整个集合的值。

如标记变量（indetification variable）代表一个空集合中的成员，则标记变量（indetification variable）的值为unknown。

(3). **路径表达式**

在 WHERE子句中，除了empty_collection_comparison_expression和collection_member_expression，使用collection_valued_path_expression作为条件表达式的一部分是不合法的。

如果路径表达式由一个代表unknown值的标记变量（indetification variable）构成，则路径表达式的值为unknown。

(4). **输入参数**

以下是关于输入参数的规则。输入参数只能用于查询中的WHERE子句。

- 输入参数由问号（?） 前缀和一个后续整数表示，例如：?1。
- 输入参数由1开始编号。
- EJB QL查询中不同输入参数的编号不能超出finder或select方法的输入参数编号。EJB QL查询中不必使用所有的finder或select方法的参数。
- 输入参数只能用于comparison_expressions或collection_member_expressions。参见 "EJB QL BNF"。
- 输入参数使用查询关联的finder或select方法中的对应参数类型进行求值。
- 如finder或select方法的输入参数是一个EJBObject类型或EJBLocalObject类型，输入参数将会对应到正确的抽象模式类型。

(5). **条件表达式构成**

一个条件表达式由其他条件表达式、比较操作、逻辑操作、求布尔值的路径表达式和布尔值的字面值构成。

算术表达式可用于比较表达式中。一个算术表达式由其他算术表达式、算术操作、求数值的路径表达式和数值的字面值构成。

算术操作使用Java中的数值提升（numeric promotion）规则。

支持使用括号（）对表达式的求值顺序进行排序。

条件表达式定义如下：

```
conditional_expression ::=

conditional_term | conditional_expression OR conditional_term
conditional_term ::=

conditional_factor | conditional_term AND conditional_factor
conditional_factor ::= [ NOT ] conditional_test
conditional_test :: = conditional_primary
conditional_primary ::=

simple_cond_expression | (conditional_expression)

simple_cond_expression ::=

comparison_expression | between_expression | like_expression |

in_expression | null_comparison_expression |

empty_collection_comparison_expression |

collection_member_expression
```

(6). **运算符与优先级**

按照由高到低的优先级，运算符排列如下：

- 定位运算符（.）；
- 算术运算符：

+，-一元运算符

*，/乘除运算符

+，-加减运算符

- 比较运算符：=、>、>=、<、<=、<> （不等于）；

- 逻辑运算符：NOT、AND、OR

下面描述在特殊表达式中的运算符。

(7). BETWEEN表达式

在条件表达式中，使用比较运算符[NOT] BETWEEN的语法如下：

```
arithmetic_expression [NOT] BETWEEN arithmetic-expr AND arithmetic-expr
```

例如：

```
p.age BETWEEN 15 AND 19
```

等价于

```
p.age >= 15 AND p.age <= 19
```

如用在一个BETWEEN表达式中的算术表达式的值是NULL，则BETWEEN表达式的值为UNKNOWN。

(8). IN表达式

在条件表达式中使用比较运算符[NOT]IN的语法如下：

```
single_valued_path_expression [NOT] IN (string-literal [,string-literal]*)
```

其中，single_valued_path_expression必须是字符串值。

例如：

```
o.country IN ('UK', 'US', 'France')
```

等价于

```
(o.country = 'UK') OR (o.country = 'US') OR (o.country = 'France')
```

另一个例子：

```
o.country NOT IN ('UK', 'US', 'France')
```

等价于

```
NOT ((o.country = 'UK') OR (o.country = 'US') OR (o.country = 'France'))
```

在IN表达式中定义的字符串字面值列表中，使用逗号分隔，必须包含至少一个字符串字面值。

如用在一个IN表达式中的single_valued_path_expression表达式的值是NULL，则IN表达式的值为UNKNOWN。

(9). LIKE表达式

在条件表达式中，使用比较运算符[NOT] LIKE的语法如下：

```
single_valued_path_expression [NOT] LIKE pattern-value [ESCAPE escape-
character]
```

其中，single_valued_path_expression必须是字符串值。pattern-value是一个字符串字面值，pattern-value中的下划线（_）表示一个单个的字符、百分号（%）表示字符串序列（包括一个空序列），所有的其他字符表示自身。可选的escape-character是一个单字符的字符串字面值，用于将pattern-value中出现的下划线（_）和百分号（%）进行转义。

例如：

```
address.phone LIKE '12%3'
```

当address.phone代表的值为“123”，“12443”等值时，此表达式为真；当address.phone代表的值为“124”，“1234”等值时，此表达式为假。

```
asentence.word LIKE 'l_se'
```

当asentence.word代表的值为“lose”时，此表达式为真；当asendence.word代表的值为“loose”时，此表达式为假。

```
aword.underscored LIKE '\_%' ESCAPE '\'
```

当aword.underscored代表的值为“_foo”时，此表达式为真；当aword.underscored代表的值为“bar”时，此表达式为假；

```
address.phone NOT LIKE '12%3'
```

当address.phone代表的值为“12”，“1234”等值时，此表达式为真；当address.phone代表的值为“123”，“12443”等值时，此表达式为假。

如用single_valued_path_expression表达式的值是NULL，则表达式的值为UNKNOWN。

(10). NULL比较表达式

在条件表达式中，使用比较运算符IS NULL的语法如下：

```
single_valued_path_expression IS [NOT] NULL
```

NULL比较表达式检查single_valued_path_expression代表的值是否为NULL值。

包含NULL值的路径表达式在求值期间返回NULL值。

(11). **空集合比较表达式**

在empty_collection_comparison_expression表达式中，使用比较运算符IS [NOT] EMPTY的语法如下：

```
collection_valued_path_expression IS [NOT] EMPTY
```

此表达式检查集合值路径表达式代表的集合是否包含元素。

如果一个集合值路径表达式代表的集合被用在一个空集合比较表达式中，则不能在FROM子句中声明此集合为一个标记变量（indetification variable）。如果标记变量（indetification variable）被显式声明为一个集合中的元素，则表示存在一个非空的关系，即集合一定包含元素，检查这样的的一个集合是否为空是与前提矛盾的。因此，类似于下例的查询无效：

```
SELECT OBJECT(o)

FROM Order o, IN(o.lineItems) l
WHERE o.lineItems IS EMPTY
```

如用在一个空集合比较表达式中的集合值路径表达式的值不包含元素，则空集合比较表达式的值为UNKNOWN。

(12). **集合成员表达式**

在collection_member_expression（集合成员表达式）中，使用比较运算符IS MEMBER OF的语法如下：

```
{single_valued_navigation | identification_variable |input_parameter} [NOT]
MEMBER [OF] collection_valued_path_expression
```

表达式检查指定的值是否是一个集合的成员，此集合由集合值路径表达式指定。

如果在集合成员表达式中的集合值路径表达式代表的集合不包含元素，则此表达式的值为FALSE。

(13). **功能表达式**

EJB QL包含以下内置的功能表达式。

字符串函数：

- CONCAT(String, String)，返回一个字符串；
- SUBSTRING(String, start, length)，返回一个字符串；

- LOCATE(String, String [, start])，返回一个字符串；
- LENGTH(String)，返回一个整型值；

start和length是int型的参数，用于指定字符串中的位置。

算术函数：

- ABS(number)
- SQRT(double)

16.2.6.2.6 SELECT子句

SELECT 子句表示查询的结果。因为finder方法不能返回任意类型的值，定义fnder方法的查询语句中的SELECT子句必须对应finder方法返回的Entity Bean的抽象模式类型；对于select方法，可以返回Entity Bean的抽象模式类型和cmp-field域的值。

SELECT子句的语法如下：

```
select_clause ::=SELECT [DISTINCT ] {select_expression |OBJECT
(identification_variable)}

select_expression ::= single_valued_path_expression
|aggregate_select_expression

aggregate_select_expression ::=
{AVG |MAX |MIN |SUM |COUNT }( [DISTINCT ] cmp_path_expression) |
COUNT ( [DISTINCT ] identification_variable
|single_valued_cmr_path_expression)
```

SELECT子句中所有的标记变量（indetification variable）必须使用OBJECT运算符进行限制。SELECT子句中不能使用OBJECT运算符对路径表达式进行限制。

DISTINCT关键字从查询结果中删除重复的值。

如应用查询语句的方法返回值类型是java.util.Collection，如查询语句中未使用DISTINCT关键字，则返回的集合中可包含重复的元素；如方法的返回值类型是java.util.Set，查询语句中未使用DISTINCT关键字，则返回的集合中也不包含重复的元素。

例子:

```
SELECT l.product FROM Order AS o, IN(o.lineItems) l
```

注意SELECT子句必须指定返回的是一个单值表达式。下例是无效的查询语句：

```
SELECT o.lineItems FROM Order AS o
```

如希望通过比较多个Entity Bean抽象模式类型来选取值，则需要在FROM子句中声明多个抽象模式类型的标记变量（indetification variable）。

下面的finder方法的查询语句返回订单数量大于John Smith的订单数量的订单。本例中使用了两个不同的标记变量（indetification variable），都使用Order抽象模式类型。

```
SELECT DISTINCT OBJECT(o1)

FROM Order o1, Order o2
WHERE o1.quantity > o2.quantity AND
o2.customer.lastname = 'Smith' AND
o2.customer.firstname= 'John'
```

下例返回订单的所有订单项：

```
SELECT OBJECT(l)

FROM Order o, IN(o.lineItems) l
```

下例返回所有的订单项：

```
SELECT OBJECT(l)

FROM LineItems AS l
```

16.2.6.2.7 ORDER BY字句

ORDER BY字句的作用是对查询返回的对象或值进行排序。

ORDER BY字句的语法如下：

```
orderby_clause ::=ORDER BY orderby_item [, orderby_item]*

orderby_item ::= cmp_path_expression [ASC | DESC]
```

当使用了ORDER BY字句时，SELECT字句必须符合以下形式：

- 为标记变量（indetification variable），如：OBJECT(o);
- single_valued_cmr_path_expression;
- cmp_path_expression;

ORDER BY字句的排序项必须是SELECT 字句中的返回项或返回对象所包含的字段。

举例：以下前两个EJB QL表达式是正确的，后两个是错误的

```
SELECT OBJECT(o)

FROM Customer c, IN(c.orders) o
WHERE c.address.state = 'CA'

ORDER BY o.quantity, o.totalcost
SELECT o.quantity
FROM Customer c, IN(c.orders) o
WHERE c.address.state = 'CA'

ORDER BY o.quantity
SELECT l.product.product_name
FROM Order o, IN(o.lineItems) l
WHERE o.customer.lastname = 'Smith' AND o.customer.firstname = 'John'

ORDER BY l.product.price
SELECT l.product.product_name
FROM Order o, IN(o.lineItems) l
WHERE o.customer.lastname = 'Smith' AND o.customer.firstname = 'John'

ORDER BY o.quantity
```

如果在ORDER BY字句中有多个排序项（对应sql中有多个排序字段），排序规则是从左到右进行处理的。

ASC表示升序，DESC表示降序，默认是ASC。

当记录中排序项的值存在null值时，这些记录排在所有非空值记录的前面或者排在所有非空值记录后面。

16.2.6.2.8 NULL值

当引用的对象不存在持久存储中，其值被视为NULL。SQL92中的NULL的语义定义了对包含NULL值的条件表达式的求值方式。

下面是这些语义的简短说明：

- 使用NULL值的算术操作或比较操作总是产生一个UNKNOWN值；
- 两个NULL值并不相等，两个NULL值的比较产生一个UNKNOWN值；
- 使用UNKNOWN值的算术操作或比较操作总是产生一个UNKNOWN值；
- 在求值期间包含NULL值的路径表达式返回NULL值；

- IS NULL和IS NOT NULL运算符将一个包含NULL值的cmp-field或单值类型的cmr-field的值转换为TRUE或FALSE；
- 布尔运算符的三个计算逻辑如下：

AND运算符的定义

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

OR运算符的定义

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

NOT运算符的定义

NOT	
T	F
F	T
U	U

	注意
	EJB QL中，空字符串''，是零长度的字符串，不等于NULL。但当查询被映射到持久存储时，空字符串和NULL值并不总是可以明确分辨的。因此，开发者不应依赖EJB QL中涉及空字符串和NULL值的比较操作。

16.2.6.2.9 相等语义

EJB QL只允许能对其使用like运算符的类型的值进行比较。但这个规则有一个例外，近似数字和精确数字是可以比较的（Java的数值提升提供了必要的类型转换）。

包括上文中提到的NULL值的例外，值的比较必须考察Java语言中的相关语义。例如，使用Java基本（primitive type）类型定义的cmp-field，不能假定其为NULL值。如此cmp-field域需要使用NULL值，则必须使用对应的引用（reference type）类型，如使用Integer替代int。持久存储中的存储机制不影响对字符串的比较，如数据填充（padding）的因素。当且仅当字符串包含同样的字符序列时，才可以决定两个字符串相等，这点与标准的SQL不同。

当且仅当同样抽象模式类型的Entity对象的Primary Key相等，才可以决定它们是相等的。

16.2.6.2.10 查询语句的限制

日期和时间值应使用Java标准中的long类型的值，使用毫秒为单位。生成毫秒值的标准方法是使用java.util.Carlendar。

尽管SQL支持在算术表达式中，进行定点十进制数的比较。但是EJB QL并不支持。因为EJB QL限制精确数字类型的数字字面值中不可包含小数点（并且使用小数点作为近似类型数字的表示）。

字符串和布尔值的比较只能使用=，和<>。

在EJB QL查询语句中不能使用注释。

容器管理持久性的数据模型目前不支持继承。因此，不同类型的Entity对象或值不能进行比较。包含此类比较操作的查询语句无效。

16.2.6.3 范例

以下例子用于演示EJB QL的语法与语义。例子基于本节开始时介绍的范例。

16.2.6.3.1 简单查询

查找所有订单：

```
SELECT OBJECT(o)
FROM Order o
```

查找所有需要运到加里福利亚的订单：

```
SELECT OBJECT(o)
FROM Order o
WHERE o.shipping_address.state = 'CA'
```

查找所有订单涉及的州：

```
SELECT DISTINCT o.shipping_address.state
FROM Order o
```

16.2.6.3.2 使用关系的查询

查找有line item的订单：

```
SELECT DISTINCT OBJECT(o)
FROM Order o, IN(o.lineItems) l
```

查找没有line item的订单：

```
SELECT OBJECT(o)
FROM Order o
```

```
WHERE o.lineItems IS EMPTY
```

查找未完成订单：

```
SELECT DISTINCT OBJECT(o)
FROM Order o, IN(o.lineItems) l
WHERE l.shipped = FALSE
```

查找所有发货地址与收款地址不一样的订单，本例假定发货地址与收款地址都使用Entity Bean：

```
SELECT OBJECT(o)
FROM Order o
WHERE
NOT (o.shipping_address.state = o.billing_address.state AND
o.shipping_address.city = o.billing_address.city AND
o.shipping_address.street = o.billing_address.street)
```

如果在两个不同的关系中发货地址与收款地址使用相同的Entity Bean表示，基于"NULL比较表达式"中的定义，以上查询可以简化为：

```
SELECT OBJECT(o)
FROM Order o
WHERE o.shipping_address <> o.billing_address
```

查询所有包含书名为"EJB deveplopment"的书的订单：

```
SELECT DISTINCT OBJECT(o)
FROM Order o, IN(o.lineItems) l
WHERE l.product.type = 'book' AND
l.product.name = 'Applying Enterprise JavaBeans:
Component-Based Development for the J2EE Platform'
```

16.2.6.3.3 使用输入参数的查询

查询名字由输入参数指定的所有订单：

```
SELECT DISTINCT OBJECT(o)
FROM Order o, IN(o.lineItems) l
WHERE l.product.name = ?1
```

对于此查询，参数必须是产品名称的类型，java.lang.String。

16.2.6.3.4 定义SELECT方法的查询

下面查询演示值的选取而非Entity Bean的选取：

选取所有已被定购的产品的名称：

```
SELECT DISTINCT l.product.name
FROM Order o, IN(o.lineItems) l
```

以下查询在一个指定订单号码的订单中查找所有包含产品的产品名称。订单号码由一个输入参数指定，并且订单号码是订单的Primary Key。

```
SELECT l.product.name
FROM Order o, IN(o.lineItems) l
WHERE o.ordernumber = ?1
```

16.2.6.3.5 EJB QL与SQL

EJB QL与SQL相似，将FROM子句作为笛卡尔积，与SQL一样，即使未使用WHERE子句，已声明的标记变量（indetification variable）也会影响查询的结果。必须小心地定义标记变量，因为已声明类型的值是否存在，决定了查询的范围。

例如，下面的FROM子句定义了对订单的查询，订单必须包含line item和包含已有产品。如持久存储中不存在产品实例，查询的范围为空且不选取订单。

```
SELECT OBJECT(o)
FROM Order AS o, IN(o.lineItems) l, Product p
```

16.2.6.4 EJB QL BNF

EJB QL标记概述：

- {...} 组
- [...] 可选结构
- 黑体

关键字

EJB QL的完整BNF定义：

```
EJB QL ::= select_clause from_clause [where_clause]

from_clause ::=FROM identification_variable_declaration
[, identification_variable_declaration]*
```

```
identification_variable_declaration ::=

collection_member_declaration |

range_variable_declaration
collection_member_declaration ::=

IN (collection_valued_path_expression) [AS ] identifier
range_variable_declaration ::=

abstract_schema_name [AS ] identifier
single_valued_path_expression ::=

{single_valued_navigation | identification_variable}.cmp_field |

single_valued_navigation
single_valued_navigation ::=

identification_variable.[single_valued_cmr_field.]*

single_valued_cmr_field
collection_valued_path_expression ::=

identification_variable.[single_valued_cmr_field.]*

collection_valued_cmr_field
select_clause ::=

SELECT [DISTINCT ] {single_valued_path_expression |

OBJECT (identification_variable)}

where_clause ::=WHERE conditional_expression
conditional_expression ::=

conditional_term | conditional_expression OR conditional_term
conditional_term ::=

conditional_factor | conditional_term AND conditional_factor
conditional_factor ::= [NOT ] conditional_test
```

```
conditional_test :: = conditional_primary
conditional_primary ::=

simple_cond_expression | (conditional_expression)

simple_cond_expression ::=

comparison_expression | between_expression | like_expression |

in_expression | null_comparison_expression |

empty_collection_comparison_expression |

collection_member_expression
between_expression ::=

arithmetic_expression [NOT ]BETWEEN
arithmetic_expression AND arithmetic_expression
in_expression ::=

single_valued_path_expression [NOT ]IN (string_literal
[, string_literal]* )

like_expression ::=

single_valued_path_expression [NOT ]LIKE
pattern_value [ESCAPE escape-character]

null_comparison_expression ::=

single_valued_path_expression IS [NOT ] NULL
empty_collection_comparison_expression ::=

collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::=

{single_valued_navigation |

identification_variable |
```

```
input_parameter}

[NOT ]MEMBER [OF ] collection_valued_path_expression
comparison_expression ::=

string_value { =|<>} string_expression |

boolean_value { =|<>} boolean_expression} |

datetime_value { = | <> | > | < } datetime_expression |

entity_bean_value { = | <> } entity_bean_expression |

arithmetic_value comparison_operator single_value_designator
arithmetic_value ::=

single_valued_path_expression | functions_returning_numerics
single_value_designator ::= scalar_expression
comparison_operator ::=

= | > | >= | < | <= | <>

scalar_expression ::= arithmetic_expression
arithmetic_expression ::=

arithmetic_term | arithmetic_expression
{ + | - } arithmetic_term
arithmetic_term ::=

arithmetic_factor | arithmetic_term { * | / } arithmetic_factor
arithmetic_factor ::= { + |- } arithmetic_primary
arithmetic_primary ::=

single_valued_path_expression | literal |

(arithmetic_expression) |

input_parameter | functions_returning_numerics
string_value ::=
```

```
single_valued_path_expression | functions_returning_strings
string_expression ::= string_primary | input_expression
string_primary ::=

single_valued_path_expression | literal |

(string_expression) | functions_returning_strings
datetime_value ::= single_valued_path_expression
datetime_expression ::= datetime_value | input_parameter
boolean_value ::= single_valued_path_expression
boolean_expression ::=

single_valued_path_expression | literal | input_parameter
entity_bean_value ::=

single_valued_navigation | identification_variable
entity_bean_expression ::= entity_bean_value | input_parameter
functions_returning_strings ::=

CONCAT (string_expression, string_expression) |

SUBSTRING (string_expression, arithmetic_expression,
arithmetic_expression)

functions_returning_numerics::=

LENGTH (string_expression) |

LOCATE (string_expression, string_expression[,
arithmetic_expression]) |

ABS (arithmetic_expression) |

SQRT (arithmetic_expression)
```

16.2.7 Java Persistence API

16.2.7.1 概览

Java Persistence API为开发人员提供了一种对象/关系映射(O-R Mapping)工具，用来管理Java应用中的关系数据。Java Persistence包括以下三个方面：

- Java Persistence API (Java持久应用开发接口)
- 查询语言
- 对象/关系映射元数据

16.2.7.2 实体

16.2.7.2.1 什么是实体

实体是轻量级的持久的领域对象。通常，实体代表了关系数据库中的一个数据表，而每个实体的实例对应了数据表中的一条记录。虽然实体可能会使用一些帮助类，但实体类才是JPA中主要的编程对象。

实体的持久状态代表了实体的持久值域或持久属性。通过对象/关系映射注解，这些值域或属性把实体与实体间的关系映射到数据库的关系数据模型中。

16.2.7.2.2 实体类的一些需求

实体类必须符合以下要求：

- 必须以javax.persistence.Entity注解进行标记；
- 必须拥有一个公共或保护的无参数构造方法，此外也允许拥有其他构造方法；
- 实体类不能被声明为final。它的任何方法与任何持久的实例变量也不能被声明为final；
- 如果实体的实例将会被作为分布对象进行值传递，例如作为会话Bean的参数或返回值，则该实体类必须实现Serializable接口；
- 实体类的父类可以是实体类或非实体类，非实体类也可以继承自实体类；
- 持久的实例变量必须声明为私有,保护或包私有，且只能由实体类的方法直接访问。客户端必须通过访问器或业务方法来访问实体的状态。

16.2.7.2.3 持久类的持久值域与持久属性

可以通过实体的值域或JavaBean风格的属性访问实体的持久状态。这些值域或属性必须是以下类型之一：

- Java基本类型
- java.lang.String
- 其他可序列化的类型，包括：
    - Java基本类型的包装类
    - java.math.BigInteger
    - java.math.BigDecimal
    - java.util.Date
    - java.util.Calendar
    - java.sql.Date
    - java.sql.Time
    - java.sql.TimeStamp
    - 自定义的可序列化类型
    - byte[]
    - Byte[]

    - char[]
    - Character[]
- Enumerated类型
- 其他实体类或实体类的集合
- 可嵌入类

实体可以自由选择使用持久值域或持久属性。如果使用持久注解标记实体的实例变量，则使用持久值域；如果使用持久注解标记实体的getter方法，则使用持久属性。但不能在同一个实体中同时标记持久值域和持久属性。

16.2.7.2.4 持久值域

如果实体类使用了持久值域，持久机制将在运行期直接访问实体类的实例变量。所有没有用javax.persistence.Transient注解标记或没有标注为transient的值域都会被持久化到数据存储设备中。必须为这些实例变量指定对象/关系映射注解。

16.2.7.2.5 持久属性

如果实体类使用了持久属性，实体类必须遵照JavaBean的命名规则。JavaBean风格的属性使用getter与setter方法，这些方法通常基于实例变量的名称来命名。规则是，一个类型为Type，属性名为property的持久属性，它的getter名称应为getProperty，setter名称应为setProperty。如果这个属性是布尔类型，则也可以用isProperty来作为getter名称。

单值持久属性的getter与setter签名应如下：

```
Type getProperty()

void setProperty(Type value)
```

集合持久值域或持久属性必须使用JPA支持的集合接口：

- java.util.Collection
- java.util.Set
- java.util.List
- java.util.Map

如果实体类使用了集合类型持久属性，它的getter与setter的类型必须为以上的集合类型，或这些集合类型的泛型变体。例如，如果一个Customer实体拥有一个集合持久属性来记录多个电话号码，它应该具有以下方法：

```
Set<PhoneNumber> getPhoneNumbers() {}

void setPhoneNumbers(Set<PhoneNumber> phoneNumbers) {}
```

必须对持久属性的getter方法使用对象/关系映射注解，同时不能对被注解为@Transient或标记为transient的值域或属性使用映射注解。

16.2.7.2.6 实体的主键

每个实体对象都必须具有一个唯一标识，例如客户实体可以使用客户号码作为标识。这个唯一标识称为主键，客户端利用它来定位实体中某个特定的实例。实体可以使用简单主键或复合主键。

使用简单主键时，可用javax.persistence.Id注解来标记主键属性或值域。

复合主键必须在主键类中定义，使用javax.persistence.EmbeddedId与javax.persistence.IdClass来标记。

简单主键与复合主键中的属性或值域，必须是以下Java类型之一：

- Java基本类型
- Java基本类型的包装类
- java.lang.String
- java.util.Date
- java.sql.Date

不能使用浮点类型作为主键。若使用自动生成的主键，只有整数类型主键是平台无关的。

16.2.7.2.7 主键类

主键类必须符合以下需求：

- 必须声明为public；
- 如果使用基于属性的访问机制，主键类的属性必须声明为public或protected；
- 必须拥有一个公共的无参数构造方法；
- 必须实现hashCode()与equals(Object other)方法；
- 必须可序列化；
- 复合主键必须映射到或被表示为实体类中的多个值域或属性，也可以映射到或被表示为一个可嵌入类；
- 如果主键类被映射到实体类的多个值域或属性，则其中的主键值域或属性的名称和类型必须与实体类中的对应值域或属性匹配。

以下是一个代表复合主键的主键类范例，orderId与itemId值域共同唯一标识了一个实体实例：

```
public final class LineItemKey implements Serializable {
  public Integer orderId;
  public int itemId;

  public LineItemKey() {}

  public LineItemKey(Integer orderId, int itemId) {
    this.orderId = orderId;
```

```
    this.itemId = itemId;
  }

  public boolean equals(Object otherOb) {
    if (this == otherOb) {
      return true;
    }
    if (!(otherOb instanceof LineItemKey)) {
      return false;
    }
    LineItemKey other = (LineItemKey) otherOb;
    return ((orderId == null ? other.orderId == null :
orderId.equals(other.orderId)) && (itemId == other.itemId));
  }

  public int hashCode() {
    return ((orderId == null ? 0 : orderId.hashCode()) ^ ((int) itemId));
  }

  public String toString() {
    return "" + orderId + "-" + itemId;
  }
}
```

16.2.7.3 实体的关系

16.2.7.3.1 实体关系的多重性

实体关系中存在四种多重性：一对一，一对多，多对一，与多对多。

- 一对一(One-to-one)：一个实体实例关联到另一个实体的一个实例。通过在相关的持久值域或属性上使用javax.persistence.OneToOne注解来表示一对一关系。
- 一对多(One-to-many)：一个实体实例关联到另一个实体的多个实例。通过在相关的持久值域或属性上使用javax.persistence.OneToMany注解来表示一对多关系。
- 多对一(Many-to-one)：一个实体多个实例关联到另一个实体的一个实例，与一对多关系正好相反。通过在相关的持久值域或属性上使用javax.persistence.ManyToOne注解来表示多对一关系。
- 多对多(Many-to-many)：一个实体的多个实例关联到另一个实体的多个实例。通过在相关的持久值域或属性上使用javax.persistence.ManyToMany注解来表示多对一关系。

16.2.7.3.2 实体关系的方向

实体关系可以是双向或单向的。双向关系同时具有持有端与反向端。单向关系则只具有持有端。关系的持有端决定了持久机制运行环境如何对数据库中的关系进行更新。

一. 双向关系

在双向关系中，每个实体都拥有一个关系值域或属性引用另一个实体。一个实体类的代码可以通过这个关系值域或属性来访问关联的实体对象。如果实体具有一个关系值域，则称为此实体"知道"它所关联的对象。例如，如果一个Order（订单）实体知道它所持有的LineItem（订单明细项）实例，同时LineItem实体知道它所属的Order实例，则这是一个双向关系。

- 双向关系中的反向端必须使用@OneToOne、@OneToMany、@ManyToMany注解中的mappedBy元素来引用持有端的关联属性。mappedBy元素标注了关系持有者的关系值域或属性；
- 多对一双向关系的多方不能定义mappedBy元素。多方必须是关系中的持有者；
- 对于一对一双向关系，持有者是指包含关系外键的一方；
- 对于多对多双向关系，持有者可以是关系中的任何一方。

二. 单向关系

在单向关系中，只有一个实体拥有关系值域或属性引用关联的另一方。例如，LineItem（订单明细项）实体应有一个关系值域引用其关联的Product（产品）实体，但Product实体不应具有到LineItem的引用。也就是说，LineItem知道Product，但Product 并不知道对应的LineItem。

三. 查询与关系方向

Java Persistence查询语言经常需要通过关系来导航。关系的方向决定了一条查询是否可以从一端的实体导航到另一端。例如，一条查询可以从LineItem导航到Product但反之不然。对于Order与LineItem，由于它们是双向关系，查询可以沿双向导航。

四. 级联删除关系

由关系联结的实体往往依赖于关系另一端的实体而存在。例如订单的明细项是订单的一部分，一旦订单被删除了，相关的订单明细项也需要删除。这种关系称为级联删除关系(cascade delete relationship)。

级联删除关系通过@OneToOne与@OneToMany关系中的cascade=REMOVE元素来标记。例如：

```
@OneToMany(cascade = REMOVE, mappedBy = "customer")
public Set<Order> getOrders() {
  return orders;
}
```

16.2.7.3.3 实体的继承关系

实体支持类继承、多态关联与多态查询。实体类可以继承自非实体类，同时非实体类也可以继承自实体类。实体类可以是抽象或具体的。

一. 抽象实体

可以使用@Entity注解把抽象类标记为抽象实体。抽象实体与具体实体的唯一差别是它不能被实例化。

查询抽象实体类与查询具体实体类一样。查询抽象实体类时，该查询将对抽象实体类的所有具体子类进行操作。

```
@Entity
public abstract class Employee {
  @Id
  protected Integer employeeId;
...
}


@Entity
public class FullTimeEmployee extends Employee {
  protected Integer salary;
...
}


@Entity
public class PartTimeEmployee extends Employee {
  protected Float hourlyWage;
}
```

二. 映射超类(Mapped superclass)

实体可以继承自包含持久状态与映射信息的非实体的超类。也就是说，这些超类没有被@Entity注解标记，因而不会被Java持久机制提供者作为实体进行映射。这种超类通常用于多个实体子类具有某些公共持久状态与映射信息的情况。

映射超类通过使用javax.persistence.MappedSuperclass注解进行标记。

```
@MappedSuperclass
public class Employee {
  @Id
  protected Integer employeeId;
...
}


@Entity
public class FullTimeEmployee extends Employee {
  protected Integer salary;
...
}
```

```
@Entity
public class PartTimeEmployee extends Employee {
  protected Float hourlyWage;
...
}
```

映射超类可以是抽象类或具体类。不能对映射超类进行查询，且不能在EntityManager或Query操作中使用映射超类，在这些操作中，必须使用映射超类的实体子类。映射超类不能参与实体关系。

在数据库中没有对应映射超类的数据表，数据表映射是由实体子类定义的。例如，以上代码范例中的映射数据表是FULLTIMEEMPLOYEE与PARTTIMEEMPLOYEE，而不存在EMPLOYEE数据表。

三. **非实体超类**

实体类可以从不包含任何持久状态与映射信息的非实体超类中继承，这些超类可以是抽象类或具体类。从非实体类中继承的状态都是非持久的，在非实体类中使用任何映射或关系注解都会被忽略。非实体类不能在EntityManager或Query操作中使用。

16.2.7.3.4 继承结构映射策略

使用javax.persistence.Inheritance注解，可以配置Java持久机制如何将继承结构结构映射到关系数据库中。可以从以下三种映射策略中选择：

- 把整个继承结构映射到一个数据表中；
- 把每个具体实体类映射到一个数据表中；
- 使用"表连接"策略，超类中的公共属性以及各子类中的独特属性分别映射到不同的数据表中。

通过设置@Inheritance注解的strategy元素来指定使用何种策略，取值可以从javax.persistence.InheritanceType的枚举元素中选取：

```
public enum InheritanceType {
  SINGLE_TABLE, JOINED, TABLE_PER_CLASS
};
```

当没有在继承结构的根类中指定映射策略时，缺省的策略是InheritanceType.SINGLE_TABLE。

一. **单表映射继承结构策略**

这种策略对应InheritanceType.SINGLE_TABLE映射类型。在这种策略中，继承结构中的所有实体类都被映射到一个单一的数据表中。这个数据表包括了继承结构中的实体类所有可能使用的持久值域或属性的映射字段。对某个特定的实体实例，若它不包含某些映射字段对应的值域或属性，则该字段取空值null。单表映射策略使用一个额外的区分列字段来区分每条记录映射的实际类型，可以使用javax.persistence.DiscriminatorColumn注解对这个字段映射进行配置，缺省情况下，该字段的名称为DTYPE，使用实体子类的名称作为标识。

这一策略对覆盖整个继承结构的多态关系与查询提供了很好的支持。但它要求映射到子类持久属性的字段必须定义为可取空值，且当继承层次较深时，数据表将含有大量字段。

二. **单表映射具体实体类策略**

这种策略对应InheritanceType.TABLE_PER_CLASS映射类型。在这种策略中，每个具体类都被映射到一个独立的数据表中。具体子类中所有持久值域或属性，包括从超类中继承的持久值域和属性，都被映射到一个数据表对应的字段上。

这个策略对多态关系的支持并不好，而且在对整个继承结构进行查询时，往往需要使用SQL UNION子句或使用多条查询语句分别查询每个子类的数据表。EJB3.0规范规定，是否支持这种策略是可选的，因此并不保证所有持久机制提供者都支持此策略。

三. **子类连接策略**

这种策略对应InheritanceType.JOINED映射类型。在这种策略把继承结构的父类映射到独立的数据表中，每个子类都使用仅仅包含其特有属性的独立数据表。也就是说，子类数据表中不包含继承而来的值域或属性。这些子表都具有表示主键的一个或多个字段，这个主键同时作为连接到父类数据表的外键。

这种策略对多态关系提供了很好的支持，但在实例化子类或对整个继承结构进行查询时，需要执行一个或多个额外的SQL JOIN操作。当继承结构较复杂时会对性能造成影响。

使用此策略时，父类映射数据表中包含区分列字段，默认字段名为DTYPE，字段类型为DiscriminatorType.STRING类型。若开发者自行定义关系数据模型，需要保证父类映射数据表中包含此字段。开发者可以使用javax.persistence.DiscriminatorColumn注解对这个字段映射进行配置。

16.2.7.4 管理实体

实体由实体管理器javax.persistence.EntityManager的实例进行管理。每个EntityManager实例都关联到一个持久上下文中。持久上下文是在一个特定的数据存储中一系列受管实体实例的集合。EntityManager接口定义了与持久上下文交互的方法。

16.2.7.4.1 实体管理器ENTITY MANAGER

实体管理器EntityManager接口提供了创建与删除持久实体实例、通过主键查找实体、以及在实体中执行查询的方法。

一. **容器管理的实体管理器**

使用容器管理的实体管理器时，EntityManager实例关联的持久上下文自动由容器注入到所有应用组件中。这些组件将在单一的JTA（Java Transaction Architecture）事务中使用这个EntityManager实例。

JTA 事务通常涉及到应用组件之间的调用，这些组件往往需要访问单一的持久上下文来完成这个事务。当EntityManager通过javax.persistence.PersistenceContext注解注入到应用组件时，持久上下文自动被传递到当前的JTA事务中。同时，映射到同一持久单元的EntityManager引用将提供访问此事务中的持久上下文的接口。通过自动传递持久上下文，多个应用组件不需要相互传递各自的EntityManager实例就能对在同一个事务中运作。Java EE容器管理这些实体管理器的生存周期。

可以通过向应用组件注入实体管理器的方式获取一个EntityManager的实例：

```
@PersistenceContext
EntityManager em;
```

二. **应用管理的实体管理器**

使用应用管理的实体管理器时，持久上下文不会被自动传递到应用组件中，EntityManager实例的生存周期由应用进行管理。

当应用需要访问一个不会随JTA事务在多个EntityManager中传递的持久上下文时，可使用应用管理的实体管理器。此时，每个EntityManager都将创建各自的持久上下文。这些EntityManager与它们所关联的持久上下文需要由应用显式创建或清除。

应用可以先使用javax.persistence.PersistenceUnit注解向应用组件注入javax.persistence.EntityManagerFactory实例，然后使用该实例的createEntityManager()方法来创建EntityManager实例：

```
@PersistenceUnit
EntityManagerFactory emf;
...
EntityManager em = emf.createEntityManager();
```

三. 使用EntityManager查找实例

EntityManager的find方法可以用来从数据存储中通过主键查找实体。

```
@PersistenceContext
EntityManager em;

public void enterOrder(int custID, Order newOrder) {
  Customer cust = em.find(Customer.class, custID);
  cust.getOrders().add(newOrder);
  newOrder.setCustomer(cust);
}
```

16.2.7.4.2 管理实体实例的生存周期

可以通过调用EntityManager上的方法来管理实体实例。实体实例总是处于四种状态之一：新建，受管，分离，已清除。

新建的实体实例没有持久标识，没有与任何持久上下文关联。

受管的实体实例拥有持久标识，与一个持久上下文关联。

分离的实体实例拥有持久标识，但没有与持久上下文关联。

已清除的实体实例拥有持久标识，与一个持久上下文关联，并被标记为计划从数据存储中清除该实例。

一. 持久化实体实例

新建的实体实例可以通过两种方式成为受管的持久实例。一是对这个实例使用persist方法，二是该与关联的实体上传递过来的级联持久化操作。第二种方式要求实体所关联的另一实体在关系注解中使用cascade=PERSIST或cascade=ALL设置。受管表示与持久操作相关的事务成功完成时，实体的持久状态将存储到数据库中。如果实体已经处于受管状态，持久化动作将会被忽略，但如果该实体设置了cascade=PERSIST或cascade=ALL，这一持久化动作仍然会被级联传递。如果对一个处于已清除状态的实体实例使用persist方法，则该实例会变为受管状态。如果对一个处于分离状态的实体实例使用persist方法，将会引起一个IllegalArgumentException异常，或者该事务会提交失败。

```
@PersistenceContext
EntityManager em;

...
public LineItem createLineItem(Order order, Product product, int quantity) {
  LineItem li = new LineItem(order, product, quantity);
  order.getLineItems().add(li);
  em.persist(li);
  return li;
}
```

若进行持久化动作的实体关系注解的cascade元素设置为ALL或PERSIST，持久化动作将会级联传递到与之关联的所有实体实例。

```
@OneToMany(cascade = ALL, mappedBy = "order")
public Collection<LineItem> getLineItems() {
  return lineItems;
}
```

二. 清除实体实例

调用EntityManager的remove方法可以清除受管的实体实例，若实体的关系注解设置为cascade=REMOVE或cascade=ALL，则该清除动作将会级联传递到与之关联的受管实体实例。若对一个处于新建状态的实体实例使用remove方法，则该操作将会被忽略，但该实体若设置了相关级联元素，则这一清除动作仍会级联传递。若对一个处于分离状态的实体实例使用remove方法，则会引起一个IllegalArgumentException异常或事务失败。如果对一个处于已清除状态的实体实例使用remove方法，则该操作将会被忽略。当事务完成或显式调用了EntityManager的flush方法，处于已清除状态的实体数据将会从数据库中删除。

```
public void removeOrder(Integer orderId) {

try {

Order order = em.find(Order.class, orderId);

em.remove(order);

}...
```

这个例子中，若Order的getLineItems访问器的关系注解中设置了cascade=ALL，则所有与order实体实例关联的LineItem实体实例都会被级联清除。

三. 同步实体数据到数据库

当持久实体关联的事务成功完成时，实体的持久状态将会被自动同步到数据库中。如果一个受管实体与其他受管存在双向关系，数据将基于关系的持久端进行同步。

调用flush方法可以强制把受管实体同步到数据库。如果实体实例关联到其他实体，且关系注解中设置了cascade为ALL或PERSIST，则被关联的实体将级联同步到数据中。

如果实体实例处于已清除状态，调用flush方法将从数据库中删除该实体实例。

16.2.7.4.3 创建查询

使用EntityManager.createQuery方法或EntityManager.createnamedQuery方法可以创建使用Java Persistence查询语言的查询。

createQuery方法用于创建动态查询，这些查询直接在应用的业务逻辑中定义：

```
public List findWithName(String name) {
  return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE
:custName").setParameter("custName", name).setMaxResults(10).getResultList();
}
```

createNamedQuery方法用于创建静态查询，这些查询在javax.persistence.NamedQuery元数据注解中定义。@NameQuery注解的name元素指定了在createNamedQuery中使用的名称，query元素指定了查询的本体。

```
@NamedQuery(

name="findAllCustomersWithName",

query="SELECTc FROM Customer c WHERE c.name LIKE :custName"

)
```

可以这样创建以上定义的静态查询

```
@PersistenceContext
public EntityManager em;
...
customers =
em.createNamedQuery("findAllCustomersWithName").setParameter("custName",
"Smith").getResultList();
```

16.2.7.4.4 查询中的命名参数

在查询中可使用冒号(:)前缀定义命名参数。创建查询后可通过javax.persistence.Query.setParameter(String name, Object value)方法对命名参数赋值。在以下例子中，setParameter方法把实参name赋值到:custName命名参数中。

```
public List findWithName(String name) {
  return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE
:custName").setParameter("custName", name).getResultList();
}
```

命名参数对大小写敏感。在动态查询或静态查询中均可使用。

16.2.7.4.5 查询中的顺序参数

除了命名参数，在查询中还可以使用顺序参数。顺序参数以问号(?)为前缀，后跟一个数字表明该参数的在查询中的位置顺序，顺序从1开始。创建查询后可通过javax.persistence.Query.setParameter(int position, Object value)方法对参数赋值。

```
public List findWithName(String name) {
  return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE ?
1").setParameter(1, name).getResultList();
}
```

16.2.7.5 持久单元

持久单元定义了应用中由EntityManager实例管理的所有实体类的集合。这个集合代表在一个特定数据存储中存储的数据。

持久单元通过persistence.xml配置文件定义。若一个JAR文件或目录的META-INF子目录包含persistence.xml文件，则此文件或目录被称为持久单元的根。持久单元的根决定了持久单元的有效范围。在有效范围内的每个持久单元必须取唯一的名称。

持久单元可以打包为WAR或EJB JAR文件，或者打包为JAR文件然后包含于WAR或EAR文件中。

如果持久单元以类的形式打包到EJB JAR文件中，persistence.xml应位于EJB JAR的META-INF目录中。

如果持久单元以类的形式打包到WAR文件中，persistence.xml应位于WAR文件的WEB-INF/classes/META-INF目录中。

如果持久单元以JAR文件的形式打包到WAR或EAR文件中，此JAR文件应位于：

- WAR文件的WEB-INF/lib目录中
- EAR文件的顶层目录中
- EAR文件的库目录中

16.2.7.5.1 PERSISTENCE.XML文件

persistence.xml配置文件定义了一个或多个持久单元。以下是一个persistence.xml文件的例子：

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description>This unit manages orders and customers.

It does not rely on any vendor-specific features and can
```

```
therefore be deployed to any persistence provider.
</description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

这个文件定义了名为OrderManagerment的持久单元，使用支持JTA的数据源jdbc/MyOrderDB。jar-file与class元素指定了受管的持久类：实体类、可嵌入类、映射超类等。其中jar-file元素指定对持久单元可见的包含受管持久类的JAR包，class元素则显式地指定受管的持久类。

jta-data-source元素（用于支持JTA的数据源）与non-jta-data-source元素（用于不支持JTA的数据源）指定了容器需要使用的数据源的全局JNDI名称。

16.2.8 Java持久查询语言

16.2.8.1 摘要

Java 持久查询语言（Java Persistence query language）是EJB3.0规范对EJB-QL（Enterprise JavaBeans query language）的扩展。本节重点介绍在EJB3.0规范中新增或改进的特性。EJB2.1规范中定义的EJB-QL说明可参考 "EJB QL"

16.2.8.2 新特性

Java持久查询是EJB-QL的扩展，在原EJB-QL的基础上，增加或改进了一系列新特性，包括批量更新与删除、JOIN操作、GROUP BY从句、HAVING从句、投影、子查询、支持静态查询与动态查询、支持顺序参数与命名参数

16.2.8.2.1 批量更新与删除

Java持久查询语言支持一次性完成多条记录的更新与删除动作。批量更新与删除每次只能对一个实体类的多个实例进行操作（包括任何子类实例）。在FROM与UPDATE子句中只能指定一个抽象模式类型（关于抽象模式类型的说明可参考"抽象持久类型与查询范围"）。批量更新与删除操作的BNF语法结构如下：

```
update_statement ::= update_clause [where_clause]

update_clause ::= UPDATE abstract_schema_name [[AS]
identification_variable]

SET update_item {, update_item}*

update_item ::= [identification_variable.]{state_field |

single_valued_association_field} =

new_value
```

```
new_value ::=

simple_arithmetic_expression |

string_primary |

datetime_primary |

boolean_primary |

enum_primary
simple_entity_expression |

NULL
delete_statement ::= delete_clause [where_clause]

delete_clause ::= DELETE FROM abstract_schema_name [[AS]
identification_variable]
```

使用批量删除与更新时须注意以下几点：

- 批量删除操作只对指定的类及其子类执行，不会级联删除关联的实体；
- 批量更新中指定的新值的类型必须与数据库中的目标字段类型兼容；
- 批量更新直接对数据库进行操作，绕过了乐观锁检查。应用需要自行更新与校验version字段的值；
- 持久化上下文不会同操作结果进行同步。因此使用批量更新与删除可能会令持久化上下文中的实体与数据库记录不一致。通常来说，批量更新与删除应该在单独的事务中使用；或者在事务刚开始，可能会受到影响的实体还未被访问时使用。

以下是一些批量删除与更新的Java持久查询语言例子：

```
DELETE
FROM Customer c
WHERE c.status = 'inactive'

DELETE
FROM Customer c
WHERE c.status = 'inactive'

AND c.orders IS EMPTY
UPDATE customer c
```

```
SET c.status = 'outstanding'

WHERE c.balance < 10000
AND 1000 > (SELECT COUNT(o)

FROM customer cust JOIN cust.order o)
```

16.2.8.2.2 连接操作

连接（JOIN）是关系数据库中常见的操作。Java持久查询语言支持内连接与左连接。

可以通过在WHERE子句中指定连接条件的方式来隐式使用内连接：

```
select c from Customer c, Employee e where c.hatsize = e.shoesize
```

通常，当连接条件没有涉及已定义的实体关系时，使用这种方式进行内连接。当通过已定义的实体关系进行连接时，使用显式的连接语法。

显式连接的BNF语法结构如下：

```
join ::= join_spec join_association_path_expression [AS]
identification_variable
fetch_join ::= join_spec FETCH join_association_path_expression
join_association_path_expression ::=
join_collection_valued_path_expression |

join_single_valued_association_path_expression
join_spec::= [ LEFT [OUTER] | INNER ] JOIN
```

一. 内连接（关系连接）

内连接的BNF语法结构是：

```
[ INNER ] JOIN join_association_path_expression [AS]identification_variable
```

Java持久查询语言的默认连接方式是内连接。例如，以下查询语句通过实体关系连接了顾客与订单两个实体。一般来说，这个连接相当于在数据库中通过外键进行连接。

```
SELECT c FROM Customer c JOIN c.orders o WHERE c.status = 1
```

也可以显式使用INNER关键字：

```
SELECT c FROM Customer c INNER JOIN c.orders o WHERE c.status = 1
```

以上查询语句等价于先前的EJB-QL中的IN结构，查询出所有status为1，且至少关联了一个订单的顾客：

```
SELECT OBJECT(c) FROM Customer c, IN(c.orders) o WHERE c.status = 1
```

二. 左连接（LEFT JOIN）

左连接，或者叫左外连接(LEFT OUTER JOIN)，在进行连接时保证JOIN关键字左边的表（主表）的实例在结果表中至少出现一次，允许被匹配的值（右表的值）为空。

左连接的语法为：

```
LEFT [OUTER] JOIN join_association_path_expression [AS]
identification_variable
```

例如：

```
SELECT c FROM Customer c LEFT JOIN c.orders o WHERE c.status = 1
```

也可以显式使用OUTER关键字：

```
SELECT c FROM Customer c LEFT OUTER JOIN c.orders o WHERE c.status = 1
```

三. 获取连接（FETCH JOIN）

获取连接允许在执行查询时同时获取关联的实体。获取连接必须基于实体与其所关联的实体进行。

获取连接的语法为：

```
fetch_join ::= [ LEFT [OUTER] | INNER ] JOIN FETCH
join_association_path_expression
```

FETCH JOIN子句右边的实体引用必须是结果实体所持有的关系引用。在获取连接中，不能为FETCH JOIN子句右边的实体引用指定标识变量，因此这个隐式获取的实体引用不允许在查询语句的其它位置出现。

以下的查询返回一个部门（department）实体的集合，同时获取这些部门所关联的雇员（employee），但雇员实体并不显式地在结果表中出现。雇员实体中所有声明为"立即获取"的持久值域或属性都将被初始化。雇员的关系属性的初始化动作则由雇员实体类中的元数据决定。

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

获取连接在语义上同内连接与左连接一样，不同之处在于JOIN子句右边的被关联实体不会在结果中出现，也不能在查询语句的其它位置引用。

获取连接的优先级比关系中指定的FetchType优先级更高，因此一个常见使用场景是当实体的元数据指定为延迟获取时，使用左连接获取（LEFT JOIN FETCH）来对查询结果所关联的实体进行预先获取。

16.2.8.2.3 GROUP BY和HAVING子句

GROUP子句根据一个属性集合对结果进行分类聚合运算。HAVING子句可以指定基于分类的条件，进一步对结果进行过滤。

GROUP BY与HAVING子句的BNF语法描述如下：

```
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*

groupby_item ::= single_valued_path_expression |
identification_variable
having_clause ::= HAVING conditional_expression
```

使用GROUP BY和HAVING子句时，应注意以下几点：

- 如果查询中同时含有WHERE子句与GROUP BY子句，执行效果相当于先执行WHERE子句，然后根据GROUP BY子句对过滤结果进行分组，最后根据HAVING子句对分组进行进一步过滤；
- 除了聚合函数（COUNT,SUM,AVG等）外，在SELECT子句中出现的任何参数必须在GROUP BY子句中出现；
- 在进行分类时，两个空值被认为是相等的；
- 允许使用实体作为分类依据，但此实体不能包含已序列化的持久值域或实际取值为LOB类型的持久值域；
- HAVING子句指定的条件必须基于GROUP BY子句的参数和它们的聚合函数。

例如：

```
SELECT c.status, avg(c.filledOrderCount), count(c)

FROM Customer c
GROUP BY c.status
HAVING c.status IN (1, 2)

SELECT c.country, COUNT(c)
```

```
FROM Customer c
GROUP BY c.country
HAVING COUNT(c.country) > 3
```

16.2.8.2.4 投影

投影允许查询中的SELECT子句指定返回实体的特定属性。在不要求返回整个实体的场合中，使用投影可以提高执行效率。例如：

```
SELECT
b.name, p.name
FROM
Book b, Publisher p
WHERE
e.publisher = p
```

使用投影时，应注意以下两点：

- SELECT子句的每项参数必须返回单值。因此以下查询是不合法的：

```
SELECT o.lineItems FROM Order AS o
```

- 查询返回的结果集为Vector类型。若投影中只涉及一个属性，则结果集的元素类型为该属性类型或其包装类（当属性为Java基本类时）；若投影中涉及多个属性，则结果集的元素为对象数组Object[]，该对象数组的元素依次为SELECT列表中的属性的类型或其包装类。如果不想使用对象数组作为结果集元素，可使用构造表达式，以投影属性对创建的类实例进行初始化。

16.2.8.2.5 SELECT子句中的构造表达式

在SELECT列表中允许使用一个构造表达式来创建并返回一个类的实例。这个类不需要是实体类。引用构造器时必须使用全路径类名。构造表达式通常结合投影一起使用，返回以投影属性进行初始化的类实例。

```
SELECT NEW com.acme.example.CustomerDetails(c.id, c.status, o.count)

FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

若在构造表达式中指定了实体类，则返回的实体实例将处于新建状态（参考"管理实体实例的生存周期"）。

16.2.8.2.6 子查询

在WHERE子句与HAVING子句中可以使用子查询。EJB3.0规范暂不支持在FROM子句中使用子查询。

子查询的BNF语法结构如下：

```
subquery ::= simple_select_clause subquery_from_clause [where_clause]
```

```
[groupby_clause] [having_clause]

simple_select_clause ::= SELECT [DISTINCT]
simple_select_expression
subquery_from_clause ::=

FROM subselect_identification_variable_declaration
{, subselect_identification_variable_declaration}*

subselect_identification_variable_declaration ::=

identification_variable_declaration |

association_path_expression [AS] identification_variable |

collection_member_declaration
simple_select_expression::=

single_valued_path_expression |

aggregate_expression |

identification_variable
```

例如：

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (

SELECT spouseEmp
FROM Employee spouseEmp
WHERE spouseEmp = emp.spouse)

SELECT c
FROM Customer c
WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

注意在某些场合下，要求子查询的结果必须为单值。例如：

```
SELECT goodCustomer
FROM Customer goodCustomer
WHERE goodCustomer.balanceOwed < (

SELECT avg(c.balanceOwed) FROM Customer c)
```

16.2.8.2.7 查询中的命名参数

在查询中可使用冒号(:)前缀定义命名参数。关于命名参数详情可参考"查询中的命名参数"。

16.2.8.3 Java持久查询语言BNF

标记概述

{ ... } 分组

[...] 可选结构

粗体关键字

- 出现不定次数 （包括零次或一次）

| 选择项目

Java持久查询语言的完整BNF定义：

```
QL_statement ::= select_statement | update_statement |
delete_statement
select_statement ::= select_clause from_clause [where_clause]
[groupby_clause]

[having_clause] [orderby_clause]

update_statement ::= update_clause [where_clause]

delete_statement ::= delete_clause [where_clause]

from_clause ::=

FROM identification_variable_declaration
{, {identification_variable_declaration |
collection_member_declaration}}*

identification_variable_declaration ::= range_variable_declaration {
join | fetch_join }*
```

```
range_variable_declaration ::= abstract_schema_name [AS]
identification_variable
join ::= join_spec join_association_path_expression [AS]
identification_variable
fetch_join ::= join_spec FETCH join_association_path_expression
association_path_expression ::=

collection_valued_path_expression |
single_valued_association_path_expression
join_spec::= [ LEFT [OUTER] | INNER ] JOIN
join_association_path_expression ::=
join_collection_valued_path_expression |

join_single_valued_association_path_expression
join_collection_valued_path_expression::=

identification_variable.collection_valued_association_field
join_single_valued_association_path_expression::=

identification_variable.single_valued_association_field
collection_member_declaration ::=

IN (collection_valued_path_expression) [AS]
identification_variable
single_valued_path_expression ::=

state_field_path_expression |
single_valued_association_path_expression
state_field_path_expression ::=

{identification_variable |
single_valued_association_path_expression}.state_field
single_valued_association_path_expression ::=

identification_variable.{single_valued_association_field.}*
single_valued_association_field
collection_valued_path_expression ::=

identification_variable.
```

```
{single_valued_association_field.}*collection_valued_association_field
state_field ::= {embedded_class_state_field.}*simple_state_field
update_clause ::= UPDATE abstract_schema_name [[AS]
identification_variable]

SET update_item {, update_item}*

update_item ::= [identification_variable.]{state_field |
single_valued_association_field} =

new_value
new_value ::=

simple_arithmetic_expression |

string_primary |

datetime_primary |

boolean_primary |

enum_primary
simple_entity_expression |

NULL
delete_clause ::= DELETE FROM abstract_schema_name [[AS]
identification_variable]

select_clause ::= SELECT [DISTINCT] select_expression {,
select_expression}*

select_expression ::=

single_valued_path_expression |

aggregate_expression |

identification_variable |

OBJECT(identification_variable) |
```

```
constructor_expression
constructor_expression ::=

NEW constructor_name ( constructor_item {, constructor_item}* )

constructor_item ::= single_valued_path_expression |
aggregate_expression
aggregate_expression ::=

{ AVG | MAX | MIN | SUM } ([DISTINCT] state_field_path_expression)
|

COUNT ([DISTINCT] identification_variable |
state_field_path_expression |

single_valued_association_path_expression)

where_clause ::= WHERE conditional_expression
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*

groupby_item ::= single_valued_path_expression |
identification_variable
having_clause ::= HAVING conditional_expression
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*

orderby_item ::= state_field_path_expression [ ASC | DESC ]

subquery ::= simple_select_clause subquery_from_clause [where_clause]

[groupby_clause] [having_clause]

subquery_from_clause ::=

FROM subselect_identification_variable_declaration
{, subselect_identification_variable_declaration}*

subselect_identification_variable_declaration ::=

identification_variable_declaration |
```

```
association_path_expression [AS] identification_variable |

collection_member_declaration
simple_select_clause ::= SELECT [DISTINCT]
simple_select_expression
simple_select_expression::=

single_valued_path_expression |

aggregate_expression |

identification_variable
conditional_expression ::= conditional_term | conditional_expression OR
conditional_term
conditional_term ::= conditional_factor | conditional_term AND
conditional_factor
conditional_factor ::= [ NOT ] conditional_primary
conditional_primary ::= simple_cond_expression |
(conditional_expression)

simple_cond_expression ::=

comparison_expression |

between_expression |

like_expression |

in_expression |

null_comparison_expression |

empty_collection_comparison_expression |

collection_member_expression |

exists_expression
between_expression ::=
```

```
arithmetic_expression [NOT] BETWEEN
arithmetic_expression AND arithmetic_expression |

string_expression [NOT] BETWEEN string_expression AND string_expression
datetime_expression [NOT] BETWEEN
datetime_expression AND datetime_expression
in_expression ::=

state_field_path_expression [NOT] IN ( in_item {, in_item}* |
subquery)

in_item ::= literal | input_parameter
like_expression ::=

string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]

null_comparison_expression ::=

{single_valued_path_expression | input_parameter} IS [NOT] NULL
empty_collection_comparison_expression ::=

collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_expression
[NOT] MEMBER [OF] collection_valued_path_expression
exists_expression::= [NOT] EXISTS (subquery)

all_or_any_expression ::= { ALL | ANY | SOME} (subquery)

comparison_expression ::=

string_expression comparison_operator {string_expression |
all_or_any_expression} |

boolean_expression { =|<>} {boolean_expression |
all_or_any_expression} |

enum_expression { =|<>} {enum_expression |
all_or_any_expression} |

datetime_expression comparison_operator
```

```
{datetime_expression | all_or_any_expression} |

entity_expression { = | <> } {entity_expression |
all_or_any_expression} |

arithmetic_expression comparison_operator
{arithmetic_expression | all_or_any_expression}

comparison_operator ::= = | > | >= | < | <= | <>

arithmetic_expression ::= simple_arithmetic_expression | (subquery)

simple_arithmetic_expression ::=

arithmetic_term | simple_arithmetic_expression { + | - }
arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term { * | / }
arithmetic_factor
arithmetic_factor ::= [{ + | - }] arithmetic_primary
arithmetic_primary ::=

state_field_path_expression |

numeric_literal |

(simple_arithmetic_expression) |

input_parameter |

functions_returning_numerics |

aggregate_expression
string_expression ::= string_primary | (subquery)

string_primary ::=

state_field_path_expression |

string_literal |
```

```
input_parameter |

functions_returning_strings |

aggregate_expression
datetime_expression ::= datetime_primary | (subquery)

datetime_primary ::=

state_field_path_expression |

input_parameter |

functions_returning_datetime |

aggregate_expression
boolean_expression ::= boolean_primary | (subquery)

boolean_primary ::=

state_field_path_expression |

boolean_literal |

input_parameter |

enum_expression ::= enum_primary | (subquery)

enum_primary ::=

state_field_path_expression |

enum_literal |

input_parameter |

entity_expression ::=

single_valued_association_path_expression |
simple_entity_expression
```

```
simple_entity_expression ::=

identification_variable |

input_parameter
functions_returning_numerics::=

LENGTH(string_primary) |

LOCATE(string_primary, string_primary[,
simple_arithmetic_expression]) |

ABS(simple_arithmetic_expression) |

SQRT(simple_arithmetic_expression) |

MOD(simple_arithmetic_expression, simple_arithmetic_expression) |

SIZE(collection_valued_path_expression)

functions_returning_datetime ::=

CURRENT_DATE|

CURRENT_TIME |

CURRENT_TIMESTAMP
functions_returning_strings ::=

CONCAT(string_primary, string_primary) |

SUBSTRING(string_primary,

simple_arithmetic_expression, simple_arithmetic_expression)|

TRIM([[trim_specification] [trim_character] FROM]
string_primary) |

LOWER(string_primary) |
```

```
UPPER(string_primary)

trim_specification ::= LEADING | TRAILING | BOTH
```

16.2.9 EJB的环境

摘要

本节提供对Apusic应用服务器中的EJB环境的介绍，包括作为EJB环境的JNDI命名上下文（Context）的介绍、如何使用EJB的环境项（environment entry）、如何在EJB环境中设置对资源连接创建器的引用、如何在EJB环境中设置对应用服务器中被管理对象的引用，最后是存在于应用服务器命名环境中、可访问的javax.transaction.UserTransaction接口的介绍。

16.2.9.1 概述

EJB 组件编程模型提供了一个将企业应用业务逻辑的开发和部署分开的灵活模型。通常在企业应用开发、部署和运行过程中，随着业务需求的不断变化，应用的业务逻辑也在随之发生变化，另外，为提高EJB的可重用特性，需要一种标准的方式，将组件逻辑与实际生产环境中需要访问的其他资源的具体信息分离，因此，EJB编程模型中，引入了EJB环境。

EJB环境实际上是伴随EJB的一个运行时由容器负责管理的JNDI命名上下文，其内容来自于开发者在部署描述中设置和声明的信息。通常，对于业务逻辑中易发生变化的内容，以及组件中需要访问的实际生产环境中的资源（如数据库、邮件服务等）信息，开发者可以将其从组件代码中分离出来，放置到EJB环境有关的部署描述中，这样，在运行时，可只以通过修改部署描述来修改应用的业务逻辑并使组件适应新的部署环境。

EJB环境中提供对如下内容的设置和定制：

- 使用环境项（environment entry）进行一般的业务逻辑定制；
- 定制对其它EJB组件的引用；
- 通过资源管理器的连接创建器引用（resouce manager connection factory reference）指定资源管理器的连接创建器；
- 使用资源环境引用定义伴随某资源的被管理对象接口；

当组件被部署到应用服务器，容器将对部署描述中有关EJB环境设置的内容进行管理，并将其绑定到一个容器管理的命名上下文（naming context）上，组件代码中通过JNDI API访问EJB环境中定义的内容。

需要注意的是，对于不同的EJB组件，其部署描述中定义的环境项（environment entry）集合是不能共享的；只有同一个EJB组件的所有实例，才能共享相同的环境项集合。另外，在EJB规范中规定，在运行时，实例只能对环境项的进行只读操作，不能试图修改环境项。

如果同一个EJB组件被多次部署到容器中，则每一次部署都会被视为不同的组件而生成不同的Home实例，并对不同的Home实例伴随不同的环境项集合，只有属于同一个Home实例的EJB实例，才共享相同的环境项集合。

16.2.9.2 环境项（environment entry）

EJB组件模型中的环境项，提供了在装配和部署时，通过修改部署描述，改变组件业务逻辑的方法。

16.2.9.2.1 访问环境项

当组件被部署到应用服务器时，容器将把组件部署描述中设置和声明的环境项保存到EJB环境中的一个名为"java:comp/env的"JNDI命名上下文中，包括此命名上下文直接或非直接的子上下文。在运行时组件可通过JNDI API对这些环境项进行访问。环境项的类型则由

开发者在部署描述中声明。下面是在EJB组件业务方法中访问环境项的范例：

```
public class EmployeeServiceEJB implements SessionBean {

...

public void setTaxInfo(int numberOfExemptions, ...)

throws InvalidNumberOfExemptionsException {

...

// Obtain the enterprise bean's environment naming context.

Context initCtx = new InitialContext();

Context myEnv = (Context)initCtx.lookup("java:comp/env");

// Obtain the maximum number of tax exemptions
// configured by the Deployer.

Integer max = (Integer)myEnv.lookup("maxExemptions");

// Obtain the minimum number of tax exemptions
// configured by the Deployer.

Integer min = (Integer)myEnv.lookup("minExemptions");

// Use the environment entries to customize business logic.

if (numberOfExeptions > Integer.intValue(max) ||

numberOfExemptions < Integer.intValue(min))

throw new InvalidNumberOfExemptionsException();

// Get some more environment entries. These environment
// entries are stored in subcontexts.

String val1 = (String)myEnv.lookup("foo/name1");
```

```
Boolean val2 = (Boolean)myEnv.lookup(“foo/bar/name2”);

// The enterprise bean can also lookup using full pathnames.

Integer val3 = (Integer)

initCtx.lookup("java:comp/env/name3");

Integer val4 = (Integer)

initCtx.lookup("java:comp/env/foo/name4");

...

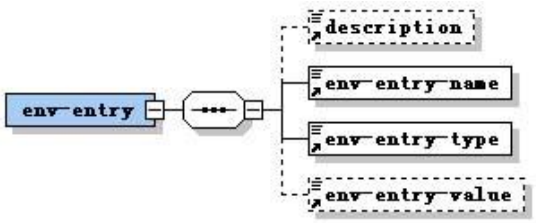
}

}
```

16.2.9.2.2 设置环境项

对EJB代码中访问的环境项，需要在部署描述文件中使用env-entry元素进行声明，每个env-entry元素代表EJB环境中的一个环境项。env-entry元素的结构如下图：

env-entry元素结构



一个env-entry元素中包含一个可选的描述环境项的子元素description；一个声明相对于java:comp/env命名上下文的环境项名字的元素env-entry-name；声明环境项值的Java类型（即通过JNDI的lookup方法返回的对象类型）的元素env-entry-type；一个表示环境项值的可选元素env-entry-value。

由于在运行时，每个EJB组件的环境项的作用域仅存在于本组件内部，因此，不同EJB组件是访问不到其它组件中定义的环境项的，因此不同的EJB组件采用相同的环境项名字，不会发生冲突。

环境项的值只能是以下几种类型：

- java.lang.String;;
- java.lang.Character;

- java.lang.Integer；
- java.lang.Boolean；
- java.lang.Double；
- java.lang.Byte；
- java.lang.Short；
- java.lang.Long；
- java.lang.Float。

除了java.lang.Character类型的环境项值必须是一个字符之外，每一种类型的环境项值，必须符合对应的Java类型中定义的只接受一个字符串参数的构建器方法，所能接受的字符串参数的格式。

对应上一节中的例子，访问多个环境项的EmployeeServiceEJB，有如下的部署描述文件设置：

```
<enterprise-beans>
<session>

...
<ejb-name>EmployeeService</ejb-name>
<ejb-class>EmployeeServiceEJB</ejb-class>

...
<env-entry>
<description>

The maximum number of tax exemptions
allowed to be set.
</description>
<env-entry-name>maxExemptions</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>15</env-entry-value>
</env-entry>
<env-entry>
<description>

The minimum number of tax exemptions
allowed to be set.
</description>
<env-entry-name>minExemptions</env-entry-name>
```

```
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>1</env-entry-value>
</env-entry>
<env-entry>
<env-entry-name>foo/name1</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>value1</env-entry-value>
</env-entry>
<env-entry>
<env-entry-name>foo/bar/name2</env-entry-name>
<env-entry-type>java.lang.Boolean</env-entry-type>
<env-entry-value>true</env-entry-value>
</env-entry>
<env-entry>
<description>Some description.</description>
<env-entry-name>name3</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
</env-entry>
<env-entry>
<env-entry-name>foo/name4</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>10</env-entry-value>
</env-entry>

...
</session>
</enterprise-beans>

...
```

16.2.9.2.3 EJB引用

企业应用中，组件之间的相互引用可能会限制单个组件的可移植和可重用特性，在EJB组件的编程模型中，引入了EJB引用。即当某个EJB组件中需要访问应用中的其他EJB时，可以在部署描述中声明对其他EJB的引用，当应用被部署到服务器，由容器解析此引用，并将引用保存在EJB环境中，不需要在组件代码中包含特定的EJB的JNDI名称，这样，当组件被部署到不同的生产环境或被引用的组件发生变化，不需要修改组件代码即可改变EJB的引用目标。

在使用EJB引用时，开发者需要在部署描述中使用ejb-ref元素指定一个到其他EJB的引用项。此引用项包含在EJB环境中，并包含在java:comp/env命名上下文中，根据EJB规范，此引用的命名应在EJB环境中的ejb子上下文中，如：java:comp/env/ejb/RefName，组件代码中即可在EJB环境中使用JNDI的API查找被引用EJB的Home接口。

一. 访问EJB引用

下面是访问EJB引用的代码范例：

```
public class EmployeeServiceEJB implements SessionBean {

public void changePhoneNumber(...) {

...

// Obtain the default initial JNDI context.

Context initCtx = new InitialContext();

// Look up the home interface of the EmployeeRecord
// enterprise bean in the environment.

Object result = initCtx.lookup(

"java:comp/env/ejb/EmplRecord");

// Convert the result to the proper type.

EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)

javax.rmi.PortableRemoteObject.narrow(result,

EmployeeRecordHome.class);

...

}

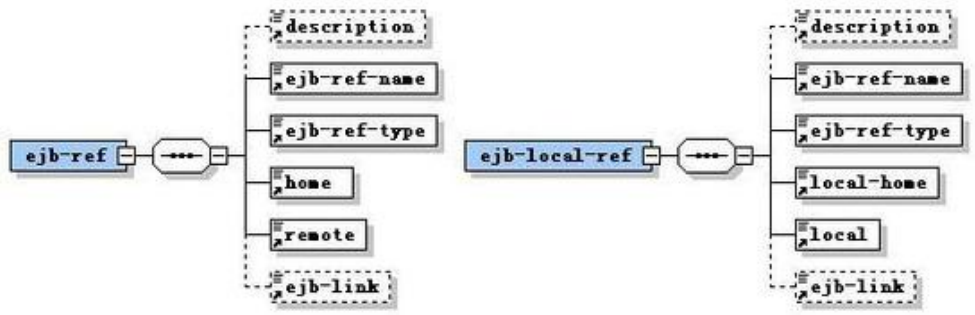
}
```

上面的例子中，在EmployeeServiceEJB的EJB环境中指定了一个名为ejb/EmplRecord的EJB引用，通过此引用可访问被引用的EJB的Home接口。

二. 声明EJB引用

尽管EJB引用属于EJB环境中的一个项目，但不可在部署描述中使用env-entry元素对引用进行声明，必须使用ejb-ref或ejb-local-ref元素进行声明，每个ejb-ref或ejb-local-ref代表一个EJB引用。ejb-ref和ejb-local-ref元素的结构如下图：

ejb-ref和ejb-local-ref元素结构



ejb-ref和ejb-local-ref元素都描述了对其他EJB的引用，所不同的是，ejb-ref声明的是通过远程Home接口和使用远程接口访问被引用的EJB，而ejb-local-ref声明的是通过本地Home接口和使用本地接口访问被引用的EJB。

ejb-ref和ejb-local-ref都包含一个可选的description元素，用于对此引用提供描述信息；ejb-ref-name元素指定此EJB 引用的引用名；ejb-ref-type用于指定引用的EJB组件的类型，可以是"Session"或"Entity"；home、local-home、remote和local元素分别用于指定被引用的EJB组件的远程Home接口、本地Home接口、远程接口和本地接口。

由于在运行时，每个EJB组件的EJB引用的作用域仅存在于本组件内部，因此，不同EJB组件是访问不到其它组件中定义的EJB引用的，因此不同的EJB组件采用相同的EJB引用名字，不会发生冲突。

下面是声明EJB引用的部署描述范例：

```
...
<enterprise-beans>
    <session>

...
<ejb-name>EmployeeService</ejb-name>
        <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>

...
<ejb-ref>
            <description>

This is a reference to the entity bean that
encapsulates access to employee records.
</description>
            <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>EmployeeRecordHome</home>
            <remote>EmployeeRecord</remote>
        </ejb-ref>
        <ejb-ref>
```

```
            <ejb-ref-name>ejb/Payroll</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>PayrollHome</home>
            <remote>com.aardvark.payroll.Payroll</remote>
        </ejb-ref>
        <ejb-ref>
            <ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <home>PensionPlanHome</home>
            <remote>PensionPlan</remote>
        </ejb-ref>

...
</session>

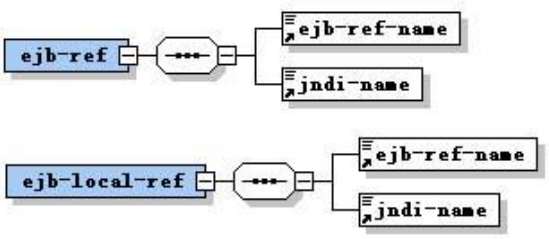
...
</enterprise-beans>

...
```

另外，开发者还需要在apusic-application.xml文件中，对应的EJB组件的设置中，使用ejb-ref或ejb-local-ref元素将引用名与被引用的EJB组件的JNDI名字进行映射。

apusic-application.xml中的ejb-ref和ejb-local-ref元素的结构如下图：

ejb-ref和ejb-local-ref元素结构



则上例中的EJB引用的配置在apusic-application中的配置如下：

```
<apusic-application>

...
<module>

...
```

```
<ejb>

...
<session ejb-name="EmployeeService">

...
<ejb-ref>
                    <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
                    <jndi-name>ejb/EmplRecord</jndi-name>
                </ejb-ref>
                <ejb-ref>
                    <ejb-ref-name>ejb/Payroll</ejb-ref-name>
                    <jndi-name>ejb/Payroll</jndi-name>
                </ejb-ref>
                <ejb-ref>
                    <ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
                    <jndi-name>ejb/PensionPlan</jndi-name>
                </ejb-ref>

...
</session>

...
</ejb>

...
</module>

...
</apusic-application>
```

(1). ejb-link

开发者也可在ejb-jar部署描述中的ejb-ref或ejb-local-ref元素里使用可选的ejb-link元素来将ejb引用连接到具体的EJB组件。ejb-link元素的值必须是被引用的EJB组件的名字，即被引用的EJB组件的部署描述元素ejb-name的值。被引用的EJB组件可以是同一个应用中，与声明引用的EJB组件相同或不同的EJB模块中的EJB组件。

为了避免整个应用中在发生引用时所产生的EJB名称冲突，ejb-link元素的值必须遵循如下规则，即当被引用的EJB组件位于应用中的其他EJB模块，则在被引用的EJB组件的名字前必须加上被引用组件位于的EJB模块的模块路径，在被引用模块路径和被引用组件名之间使用"#"字符进行分隔。被引用模块路径是相对于当前的引用模块的路径。

如下例：

```
<enterprise-beans>
    <session>

...
<ejb-name>EmployeeService</ejb-name>
        <ejb-class>com.wombat.empl.EmployeeServiceEJB</ejb-class>

...
<ejb-ref>
            <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.wombat.empl.EmployeeRecordHome</home>
            <remote>com.wombat.empl.EmployeeRecord</remote>
            <ejb-link>EmployeeRecord</ejb-link>
        </ejb-ref>

...
</session>

...
<entity>
        <ejb-name>EmployeeRecord</ejb-name>
        <home>com.wombat.empl.EmployeeRecordHome</home>
        <remote>com.wombat.empl.EmployeeRecord</remote>

...
</entity>

...
</enterprise-beans>

...
```

下例是引用同一个应用中不同模块的EJB组件时，ejb-link的用法：

```
<entity>
...
<ejb-name>OrderEJB</ejb-name>
    <ejb-class>com.wombat.orders.OrderEJB</ejb-class>
```

```
...
<ejb-ref>
        <ejb-ref-name>ejb/Product</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>com.acme.orders.ProductHome</home>
        <remote>com.acme.orders.Product</remote>
        <ejb-link>../products/product.jar#ProductEJB</ejb-link>
    </ejb-ref>

...
</entity>
```

	**注意**
	当在ejb-jar模块部署描述中的ejb-ref或ejb-local-ref元素里使用了ejb-link元素将EJB引用连接到了具体的EJB组件， 则在apusic-application部署描述可以不使用ejb-ref或ejb-local-ref元素将引用名与被引用EJB组件的JNDI名进行映射。

16.2.9.3 资源管理器的连接创建器引用（resource manager connection factory reference）

Java EE平台中定义了三种可管理资源，JMS消息系统、可通过JDBC连接的数据库系统以及符合Java Connector Architecture规范的后端系统，应用可通过以上资源的连接创建器（Connection Factory）创建对以上资源的连接。应用服务器则对以上资源提供资源管理器，对应用中创建的资源连接进行管理，如对资源连接提供连接池、在事务中自动对连接进行征用（enlist）等。

Apusic应用服务器中提供对以上内容的实现，例如当应用通过javax.sql.DataSource接口从连接池中获取java.sql.Connection连接时，应用服务器中的资源管理器即对连接进行管理，不过，这些工作对于应用而言则是透明的，应用只根据接口提供的标准API对连接进行操作即可。

EJB组件中，不可避免将会使用某些资源，但是，实际生产环境中的资源情况是不同的，尤其是当EJB组件被装配到不同的生产环境的时候，因此，EJB组件模型在EJB环境中提供了资源管理器的连接创建器引用的机制，即应用中的代码通过资源连接创建器的逻辑名称，取得资源连接创建器对象，然后通过此对象创建与资源的连接。这样，当应用被部署到不同的生产环境，通过将逻辑名称映射到不同的资源，即可提高应用或组件的可重用特性，不用修改代码即可适应新的生产环境。

16.2.9.3.1 使用资源管理器的连接创建器引用

连接创建器的引用必须在EJB组件的部署描述中进行声明，引用是作为EJB环境中的一个特殊项而被声明的，根据EJB规范，推荐开发者所声明的引用位于不同的EJB环境中的子上下文（subcontext）中，如有关数据库连接的创建器引用名应位于java:comp/env/jdbc的子上下文中、有关JMS消息系统的连接创建器引用名应位于java:comp/env/jms的子上下文中、有关JavaMail的连接创建器引用名应位于java:comp/env/mail的子上下文中、有关URL连接创建器引用名应位于java:comp/env/url的子上下文中。

在部署描述中声明了连接创建器引用后，必须在部署描述中将引用映射到实际的资源，这个工作需要在apusic-application.xml中使用相应的部署描述元素将引用名与资源的JNDI名进行映射。

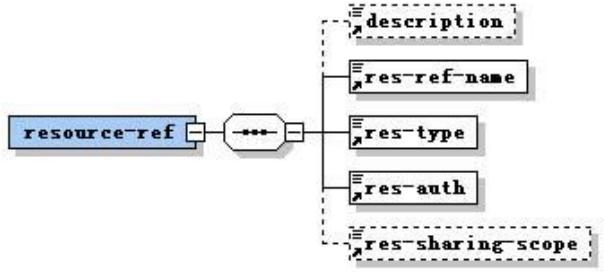
在完成了引用名与实际资源的映射后，组件在运行时即可通过JNDI的API对引用进行查找，然后通过连接创建器创建与特定资源的连接。

16.2.9.3.2 声明连接创建器引用

虽然连接创建器引用作为EJB环境的一部分，但是连接创建器的引用在ejb-jar中使用resource-ref元素声明每一个引用，而不是使用env-entry元素。

resource-ref元素的结构如下图：

resource-ref元素结构



EJB 组件部署描述中的每个resource-ref元素声明一个连接创建器的引用，包含一个可选的用于对此引用提供文档信息的description元素；包含一个必需的声明引用名的res-ref-name元素、一个声明引用类型的res-ref元素、一个声明引用身份鉴定方式的res-auth元素和一个可选的引用共享范围的res-sharing-scope元素。

res-ref-name元素的值是EJB代码中使用的引用名称，相对于java:comp/env上下文，如值为jdbc/EmployeeAppDB的引用，实际上被视为java:comp/env/jdbc/EmployeeAppDB上下文。

res-type元素的值是引用的资源连接创建器的Java类型，如javax.sql.DataSource。

res-auth-type元素的值用于指定是否在EJB代码中包含登录资源的身份鉴定信息或是按照部署描述中的安全信息通过容器登录资源，如通过代码则值必须是Application，如通过容器则值必须是Container。

res-sharing-scope元素的值用于指定通过连接创建器引用创建的连接是否可以共享。其值可以是Shareable或是Unshareable，如未指定res-sharing-scope元素，则容器按照值为Shareble的情况进行处理。

由于在运行时，每个EJB组件的连接创建器引用的作用域仅存在于本组件内部，因此，不同EJB组件是访问不到其它组件中定义的连接创建器引用的，因此不同的EJB组件采用相同的连接创建器引用名字，不会发生冲突。

下面是EJB组件中声明连接创建器的部署描述文件的例子：

```
...
<enterprise-beans>
    <session>

...
<ejb-name>EmployeeService</ejb-name>
        <ejb-class>EmployeeServiceEJB</ejb-class>

...
```

```
<resource-ref>
            <description>

A data source for the database in which
the EmployeeService enterprise bean will
record a log of all transactions.
</description>
            <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
            <res-sharing-scope>Shareable</res-sharing-scope>
        </resource-ref>

...
</session>
</enterprise-beans>

...
```

下面是声明一个JMS消息系统连接创建器引用的例子:

```
...
<enterprise-beans>
    <session>

...
<resource-ref>
            <description>

A queue connection factory used by the
MySession enterprise bean to send
notifications.
</description>
            <res-ref-name>jms/qConnFactory</res-ref-name>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Container</res-auth>
            <res-sharing-scope>Unshareable</res-sharing-scope>
        </resource-ref>

...
```

```
</session>
</enterprise-beans>

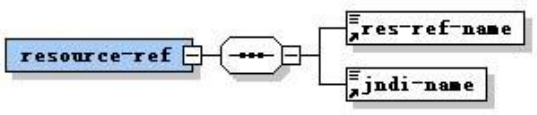
...
```

16.2.9.3.3 引用的映射

对连接创建器引用的映射需要在apusic-application部署描述中使用resource-ref元素进行定义。

apusic-application部署描述中resource-ref元素的结构如下：

resource-ref元素结构



开发者需要将EJB组件中使用res-ref-name元素定义的引用名与实际的资源连接创建器的JNDI名进行映射。如上节中的例子，对应的apusic-application.xml部署描述文件有如下内容：

```
<apusic-application>

...
<module>

...
<ejb>

...
<session ejb-name="EmployeeService">

...
<resource-ref>
                    <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
                    <jndi-name>jdbc/ConcreteDataSource</jndi-name>
                </resource-ref>

...
</session>

...
</ejb>
```

```
...
</module>

...
</apusic-application>
```

16.2.9.3.4 访问引用

下面的例子演示如何在EJB组件代码中使用上面两节中定义的数据库连接的连接创建器创建JDBC连接：

```
public class EmployeeServiceBean implements SessionBean {

EJBContext ejbContext;

public void changePhoneNumber(...) {

...

// obtain the initial JNDI context
Context initCtx = new InitialContext();

// perform JNDI lookup to obtain resource manager
// connection factory
javax.sql.DataSource ds = (javax.sql.DataSource)

initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");

// Invoke factory to obtain a connection. The security
// principal is not given, and therefore
// it will be defined in related deployment descriptor.

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

...

}

}
```

16.2.9.3.5 标准的连接创建器类型

可在连接创建器引用声明中出现的连接创建器引用包括如下类型：

- JDBC javax.sql.DataSource
- JMS消息系统 javax.jms.QueueConnectionFactory javax.jms.TopicConnectionFactory
- Java Mail javax.mail.Session
- URL java.net.URL

16.2.9.4 资源环境引用（resource environment references）

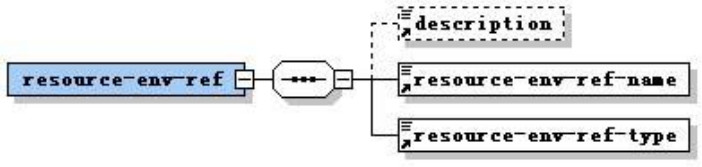
在EJB编程模型中，组件可通过某资源的被管理对象（如JMS消息服务中的队列或主题）的逻辑名字，进而取得对此被管理对象的引用。这一类逻辑名字，称为资源环境引用，属于EJB环境的一个组成部分。资源环境引用在EJB组件的部署描述中声明，通过这类引用，EJB组件即可适应不同的实际运行环境。

16.2.9.4.1 声明资源环境引用

资源环境引用属于EJB环境项中的一个特殊组成部分，不通过env-entry元素进行声明，必须在ejb-jar中组件相关的部署描述的resource-env-ref元素进行声明；在部署到实际生产环境之前，还必须在apusic-application中组件相关的部署描述中使用resource-env-ref元素将引用与实际的被管理对象进行映射。

下图是ejb-jar中resource-env-ref元素的结构：

resource-env-ref元素结构



在ejb-jar中的resource-env-ref元素，包含一个可选的为此引用提供文档信息的description元素，一个声明引用名称的resource-env-ref-name元素和声明引用对象Java类型的resource-env-ref-type元素。

resource-env-ref-name是相对于java:comp/env上下文的一个JNDI名字。因此，如果在代码中使用了一个名为java:comp/env/jms/StockQueue的JMS队列的引用，则在resource-env-ref-name中的值应为jms/StockQueue，而不是java:comp/env/jms/StockQueue。

由于在运行时，每个EJB组件的资源环境引用的作用域仅存在于本组件内部，因此，不同EJB组件是访问不到其它组件中定义的资源环境引用的，因此不同的EJB组件采用相同的资源环境引用名字，不会发生冲突。

下面是在ejb-jar中声明资源环境引用的例子：

```
...
<resource-env-ref>
    <description>

This is a reference to a JMS queue used in the
```

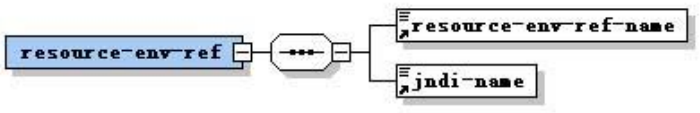
```
processing of Stock info
</description>
    <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>

...
```

在完成对资源环境引用的声明之后，在部署到实际运行环境之前，开发者还需要将已声明的引用与实际的被管理对象进行映射。映射在整个应用的部署描述文件apusic-application.xml中进行，使用resource-env-ref元素。

apusic-applicatoin中的resource-env-ref元素结构如下图：

*resource-env-ref元素结构 *



resource-env-ref元素中，必须包含一个resource-env-ref-name，其值为ejb-jar中对应的resource-env-ref元素中resource-env-ref-name元素的值；还必须包含一个jndi-name元素，此元素的值为被管理对象的JNDI名字。

上面例子对应的apusic-application中的resource-env-ref元素声明如下：

```
...
<resource-env-ref>
    <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
    <jndi-name>jms/StockInfoQueue</jndi-name>
</resource-env-ref>

...
```

本部署描述片断，将引用jms/StockInfo映射到一个名为jms/StockInfoQueue的JMS消息队列。

16.2.9.4.2 访问资源环境引用

在EJB组件代码中，则可以使用JNDI的API，根据资源环境引用名查找并获取对实际运行环境中被管理对象的引用，如下例：

```
public class StockServiceBean implements SessionBean {

public void processStockInfo(...) {
```

```
...

// Obtain the default initial JNDI context.

Context initCtx = new InitialContext();

// Look up the JMS StockQueue in the environment.

Object result = initCtx.lookup(

"java:comp/env/jms/StockQueue");

// Convert the result to the proper type.

javax.jms.Queue queue = (javax.jms.Queue)result;

}

}
```

16.2.9.5 javax.transaction.UserTransaction接口

在 EJB规范中，Bean管理事务的Session Bean和Message-driven Bean可以通过JNDI API查找"java:comp/UserTransaction"名字，取得对javax.transaction.UserTransaction对象的引用。

如下例：

```
public MySessionEJB implements SessionBean {

...

public someMethod(){

Context initCtx = new InitialContext();

UserTransaction utx = (UserTransaction)initCtx.lookup(

"java:comp/UserTransaction");

utx.begin();
```

```
...
utx.commit();
}
...
}
```

在功能上，上例等价于：

```
public MySessionEJB implements SessionBean {
SessionContext ctx;
...
public someMethod(){
UserTransaction utx = ctx.getUserTransaction();
utx.begin();
...
utx.commit();
}
...
}
```

16.2.10 EJB的事务

摘要

本节包含如何在EJB开发中使用事务，包括容器管理事务和Bean管理事务的描述，EJB的事务处理模型的介绍，以及在Apusic应用服务器中的EJB开发中，如何正确地使用事务模型。

本节假设读者已经熟悉JDBC中的事务模型和JTA API，有关以上两个部分，可以参考Apusic应用服务器中的Sun的JTA文档和Sun的JDBC教程。

16.2.10.1 概述

金蝶Apusic应用服务器的事务服务，提供了JTA API的完整实现，支持EJB1.1和EJB2.1的两阶段提交（two-phase commit）协议。两阶段提交协议是一种在两个或以上的资源管理器之间协调一个事务的方法。两阶段提交协议通过对所有参与事务的资源（如数据库、消息系统等）进行事务性的数据更新来保证数据的完整性。

金蝶Apusic应用服务器中，EJB组件可以在代码中进行程序型的事务边界划分，也可通过部署描述对事务进行声明型的划分。使用这两种方式，开发者可以简单地划分事务边界和确定事务逻辑，运行时的具体底层细节将由Apusic事务服务中的事务管理器进行管理。

在J2EE™平台中，可管理的事务资源包括三种，数据库连接池、消息系统连接和符合J2EE™连接器架构（J2EE Connector Architecture，JCA）的资源。Apusic应用服务器中的事务管理器提供对以上三种资源的事务管理。

在了解EJB的事务划分之前，需要了解EJB客户端的事务与EJB之间的关系，同时也需要了解EJB方法的事务上下文（transaction context）。

16.2.10.2 客户端事务与未指明的事务上下文

在一节中，概述了EJB的客户端类型，为简化问题的描述，本节中不对每种客户端类型进行单独讨论。统称EJB客户端。

EJB规范关于事务处理模型的内容，定义了现阶段的EJB事务处理模型并不包含嵌套事务（nested transaction），只提供平直事务（flat transaction）的支持。

所谓嵌套事务，指一个事务边界内部可包含多个子事务边界，子事务的提交或回滚并不影响父事务的提交或回滚，而父事务的提交或回滚可决定所有包含在此事务边界中的子事务的提交或回滚。

所谓平直事务，指事务不能相互嵌套，事务内部不能包含子事务。事务与事务之间是独立的，某个事务的提交或回滚不会直接影响其他事务的提交或回滚。

因此，按照以上的平直事务模型的定义，根据EJB客户端是否包含在事务边界中，应用服务器中的事务管理器在运行时按照开发者指定的事务管理类型（容器管理或Bean管理），对EJB实例方法的事务边界与客户端事务边界进行划分。通常有如下策略：

- 客户端无事务

容器根据开发者指定的事务管理类型，在新的事务上下文中完成EJB实例的方法调用，或在未指明的事务上下文（unspecified transaction context）中完成方法调用；如开发者指定调用此EJB方法时，客户端必须伴随事务上下文，则容器将抛出异常。

- 客户端有事务

根据开发者指定的事务管理类型，容器有以下策略：暂停客户端事务，在新的事务上下文中运行EJB实例的方法，调用完成后，恢复客户端事务；暂停客户端事务，在未指明的事务上下文（unspecified transaction context）中完成EJB实例的方法调用，调用完成后，恢复客户端事务；在客户端的事务上下文中，完成EJB实例的方法调用；另外，如开发者指定调用此EJB方法时，的客户端不可伴随事务上下文，则容器将抛出异常。

按照以上策略，可以了解到金蝶Apusic应用服务器中的事务管理器对EJB实例的方法调用进行事务控制的大致情形。必须注意，这些策略都保证了整个调用过程中，事务之间不会相互嵌套。

一般来讲，对于JSP、Servlet、Java Application和Applet等客户端，不提倡在其中进行事务划分，因为这将使易出错的代码增加，并且失去了不修改代码即可改变系统行为的能力，而且，这种方式将系统的关键功能（事务的管理）移到了整个系统的前端或GUI部

分，违反了企业应用的分层原理，因此，在涉及时，应尽量将划分事务边界的操作放到EJB包含的业务逻辑中，并尽量使用容器管理事务的方式。

在前面，提到的未指明的事务上下文（unspecified transaction context），是指在EJB体系结构中，对执行EJB方法的事务语义，没有进行完整定义的情况。在这种情况下的EJB方法，其中对事务性资源的操作（如数据库连接、消息服务连接）不具有可恢复的能力，例如，某个未指明的事务上下文（unspecified transaction context）的EJB方法中，对数据库进行插入、或更新数据的操作，当客户端调用此EJB方法时，伴随了事务，容器将在方法执行前暂停客户端事务，在方法执行完成之后恢复客户端事务，而之后的客户端事务的结果无论是提交或回滚，都不能影响到前面的EJB方法对数据库的操作，这些操作实际上不包含在事务边界中，属于非事务性的不可恢复的操作。

因此，在未指明的事务上下文（unspecified transaction context）的EJB方法中，开发者必须慎重考虑对事务性资源的操作。一般，不提倡开发者在这些方法中对事务性资源进行操作。EJB体系结构中，具体的未指明的事务上下文（unspecified transaction context）的EJB方法，可参考本节中的"容器管理事务（声明型的事务划分）"。

从以上对客户端事务的描述，可以看到，EJB的事务处理与客户端请求的事务上下文存在很紧密的关系，与此同时，EJB的事务处理与容器管理EJB实例的方式也有存在着紧密的关系。我们将在具体的事务划分方式中对以上因素加以分析。

16.2.10.3 EJB的事务划分

提高企业应用中有关事务的开发和维护效率，降低EJB的开发难度，使开发者不必过多考虑事务处理的底层细节，Apusic应用服务器根据EJB规范和其他J2EE平台™规范，提供了简单的易于开发者使用的事务划分方式。

EJB 的规范中，定义了开发者可使用的两种事务划分方式：一种是Bean管理事务，即程序型的事务划分；另一种是容器管理事务，即声明型的事务划分。当使用程序型的事务划分的方式时，开发者必须使用通过JNDI访问容器提供的javax.transaction.UserTransaction接口的引用，使用UserTransaction接口对事务边界进行划分，每个在UserTransaction.beging()和UserTransaction.commit()之间的对资源（如数据库）的操作，作为事务的一个组成部分。当使用声明型的事务划分时，开发者通过配置EJB的部署描述，开发者在部署描述中使用事务相关的元素对方法的事务属性进行设置，完成方法级别的事务划分。

两种方式都最大限度地简化了在事务方面的开发难度，而且具有不同的特点。

对于EJB组件模型中的Session Bean，Entity Bean和Message-driven Bean，Session Bean和Entity Bean可指定使用程序型的事务划分或者声明型的事务划分，但不能在一个Bean内部同时使用两种事务划分方式。Entity Bean必须采用声明型的事务划分方式。

16.2.10.3.1 BEAN管理事务（程序型的事务划分）

当使用Bean管理事务时，开发者必须通过JNDI API查找并获取由服务器提供的javax.transaction.UserTransaction接口，或者通过伴随EJB实例的javax.ejb.EJBContext接口的getUserTransaction方法，获取javax.transaction.UserTransaction接口，并使用此接口对事务进行划分。另外，还必须在EJB的部署描述文件中，使用transaction-type元素指定其事务管理类型为Bean管理事务类型。

使用JNDI查找UserTransaction接口的范例代码如下：

```
...

try
{

Context initial = new InitialContext();
```

```
Object objref = initial.lookup("java:comp/UserTransaction");

javax.transaction.UserTransaction tx =

(javax.transaction.UserTransaction)objref;

tx.begin();

...

}

catch(...)

...
```

Session Bean中使用javax.ejb.EJBContext取得UserTransaction接口的范例代码如下：

```
try
{

...

javax.ejb.EJBContext ctx = getSessionContext();

javax.transaction.UserTransaction tx =

ctx.getUserTransaction();

...

tx.begin();

...

}

catch(...)
```

```
...
```

tansaction-type元素的设置范例如下：

```
...
<ejb-jar>
    <display-name>CartEJB</display-name>
    <enterprise-beans>
        <session>
            <display-name>CartEJB</display-name>

...
<transaction-type>Bean</transaction-type>

...
</session>

...
</enterprise-beans>

...
</ejb-jar>
```

有关javax.transaction.UserTransaction的详细使用说明，请参考Sun的JTA文档。

一. 使用Bean管理事务的限制

从上面对客户端事务的描述，可以看到，EJB的事务处理模型与客户端请求存在很紧密的关系，从"简介"一节中，我们了解到EJB容器对客户端请求的处理方式，即有状态的Session Bean实例与客户端是一一对应的，无状态的Session Bean实例可由容器分配给任意客户端使用，而Entity Bean实例则有多个客户端共享，Message-driven Bean则与客户端无关。按照EJB组件实例与客户端的关系，对各种组件模型使用Bean管理事务有如下限制：

- 对于有状态的Session Bean实例，客户端是唯一的，因此，在方法调用之间的事务状态可以得到保持，对于实例的客户端而言，可以使用javax.transaction.UserTransaction接口在方法中清晰地划分事务边界：如在某方法内部使用UserTransaction.begin()方法开始事务边界和使用UserTransaction.commit()方法提交事务；也可在某方法中使用UserTransaction.begin()方法开始事务边界，在进行一系列的业务方法调用之后，使用UserTransaction.commit()方法提交事务；
- 对于无状态的Session Bean，实例可被容器分配给不同的客户端使用，而且，实例不会保持会话状态，因此，当使用Bean管理事务时，如在方法中使用UserTransaction接口划分事务，必须在同一个方法内部开始事务边界并在方法返回之前提交或回滚事务，否则容器将回滚事务并抛出异常；

- 对于Message-Driven Bean而言，如果开发者在onMessage方法中使用了UserTransaction方法划分事务边界，则必须在方法返回前提交或回滚事务，否则容器将回滚事务；
- 对于Entity Bean而言，由于实例在不同的客户端之间共享，容器使用事务对不同的客户端调用进行同步，因此，Entity Bean的事务类型不能是Bean管理事务，只能是容器管理事务，当在Entity Bean中调用javax.ejb.EntityContext接口的getUserTransaction方法时，容器将抛出java.lang.IllegalStateException异常。

二. 事务上下文

当客户端对Bean管理事务的Session Bean进行调用时，根据客户端调用是否伴随事务上下文，容器将使用如下策略对实例的事务进行管理：

- 客户端请求不伴随事务

当客户端请求不伴随事务，并且实例无关联事务时，容器将在未指明的事务上下文（unspecified transaction context）中调用实例方法，这些方法中的对资源的操作不具有事务性，参见本节中的"客户端事务与未指明的事务上下文"。

当客户端请求不伴随事务，并且实例被关联到事务T2时（当客户端对有状态Session Bean实例进行的前一个方法调用开始了一个事务边界，而且在调用完成后未提交或回滚事务时，可能发生这种情况），容器将在此关联事务T2中进行方法调用，对于无状态的Session Bean，这种情况永远不会发生。

- 客户端请求伴随事务

当客户端请求伴随事务T1，并且实例无关联事务时，容器将暂停客户端事务，在未指明的事务上下文（unspecifiedtransaction context）中进行方法调用，在方法调用完成后，容器恢复原客户端伴随的事务T1。

当客户端请求伴随事务T1，并且实例被关联到事务时T2时（当客户端对有状态Session Bean实例进行的前一个方法调用开始了一个事务边界，而且在调用完成后未提交或回滚事务时，可能发生这种情况），容器将暂停T1，并在T2中进行方法调用，在方法完成后恢复T1。对于无状态的Session Bean，这种情况永远不会发生。

由于Message-driven Bean与客户端事务上下文无关，并且，使用容器管理事务的Message-driven Bean必须在onMessage方法中开始事务和在方法返回之前提交或回滚事务，因此，对于Bean管理事务的Message-driven Bean的方法调用运行在未指明的事务上下文中。

16.2.10.3.2 容器管理事务（声明型的事务划分）

当使用容器管理事务时，开发者需要在部署描述文件中使用transaction-type元素指定事务管理类型为容器管理事务，并在Entity Bean的assembly-descriptor元素中，使用container-transaction元素对EJB方法的事务属性进行声明。

使用容器管理事务的EJB的业务方法和Message-driven Bean中的onMessage方法中，不可对相关的资源调用任何特定的事务管理方法，以免干扰容器对事务边界的划分。这些方法包括如java.sql.Connection接口的commit、setAutoCommit和rollback方法，javax.jms.Session接口的commit和rollback等方法。

使用容器管理事务的EJB的业务方法和Message-driven Bean中的onMessage方法中，不能试图访问和使用javax.transaction.UserTransaction接口。

下面是一个容器管理事务的EJB的业务方法的例子，此方法使用JDBC连接更新两个数据库中的相关数据，容器按照部署描述中的事务属性进行事务划分：

```
...

public class MySessionEJB implements javax.ejb.SessionBean {

EJBContext ejbContext;

public void someMethod(...) {

java.sql.Connection con1;

java.sql.Connection con2;

java.sql.Statement stmt1;

java.sql.Statement stmt2;

// 获取con1和con2连接对象

con1 = ...;

con2 = ...;

stmt1 = con1.createStatement();

stmt2 = con2.createStatement();

//

// 在con1和con2上进行更新操作。容器使用自动征用con1和

// con2对象，提供对con1和con2的操作的事务管理。

//

stmt1.executeQuery(...);

stmt1.executeUpdate(...);

stmt2.executeQuery(...);
```

```
stmt2.executeUpdate(...);

stmt1.executeUpdate(...);

stmt2.executeUpdate(...);

// 释放连接

con1.close();

con2.close();

}

...

}
```

此范例Session Bean的事务属性相关的部署描述文件如下：

```
<ejb-jar>
    <display-name>MySessionEJB</display-name>
    <enterprise-beans>
        <session>
            <display-name>MySessionEJB</display-name>

...
</session>

...
</enterprise-beans>
    <assembly-descriptor>

...
<container-transaction>

...
<method>
```

```
...
<ejb-name>MySessionEJB</ejb-name>
                <method-name>someMethod</method-name>

...
</method>
            <trans-attribute>Required</trans-attribute>

...
</container-transaction>

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

一. setRollbackOnly与getRollbackOnly方法

使用容器管理持事务划分的EJB实例可调用伴随此实例javax.ejb.EJBContext接口中的setRollbackOnly和getRollbackOnly方法，对实例方法的事务的回滚标记进行设置和对当前事务是否已设置回滚标记进行检查。

通常情况下，开发者需要在抛出应用级异常之前，设置调用实例伴随的EJBContext接口的setRollbackOnly方法，把当前事务标记为回滚以保持数据完整性，因为应用级异常不会使容器自动回滚事务。例如，在一个进行汇款操作的业务方法中，可能涉及两个账户的操作，当减少一个账户的余额成功而增加另一账户余额的操作失败时，则需要调用setRollbackOnly方法，将当前事务标记为回滚。

使用容器管理事务的EJB可调用javax.ejb.EJBContext的getRollbackOnly方法检查当前事务是否被标记为回滚。此标记可由实例本身进行设置，也可由其他EJB或其他组件进行设置。

二. 使用JMS API的考虑

如采用了容器管理事务类型的EJB，开发者不能在单个事务中使用JMS的请求和响应模式（如发送一条JMS消息之后，同步地接收消息的响应）。这是因为如果直到事务提交时，消息仍未到达终点，则此事务边界内将永远不可能接收到消息的响应。

由于容器将代表EJB，对JMS Session进行事务性的征用（enlistment），createQueueSession(boolean transacted, int acknowledgeMode)和createTopicSession(boolean transacted,int acknowledgeMode)方法中的参数将被忽略。并且根据EJB规范，推荐开发者指定此transacted参数为真，并指定acknowledgeMode为零。

另外，开发者不能在一个事务内或一个未指明事务上下文的事务内使用JMS API中的acknowledge方法。未指明事务上下文的事务内的消息通知由容器进行。

三. 事务属性

对容器管理事务的Session Bean和Entity Bean的Home接口和组件接口中定义的方法，以及Message-driven Bean中的onMessage方法，开发者可以设置其事务属性。当客户端通过Home接口或组件接口调用这些方法时，或者当消息到达时onMessage方法被调用时，容器将按照开发者指定的事务属性对以上方法的事务进行管理。

当使用容器管理事务的EJB时，开发者必须指定以下方法的事务属性：

- Session Bean中，除了在javax.ejb.EJBObject接口和javax.ejb.EJBLocalObject接口中定义的方法，其他定义在组件接口中的方法，以及定义在组件接口直接或非直接地继承的接口中的方法，开发者必须指定方法的事务属性。Session Bean的Home接口中的方法不可指定其事务属性；
- Entity Bean中，除了getEJBHome、getEJBLocalHome、getHandler、getPrimaryKey和isIdentical方法之外，其他定义在组件接口中的方法，以及定义在组件接口直接或非直接地继承的接口中的方法，开发者必须指定方法的事务属性。除了远程Home接口中的getEJBMetaData和getHomeHandler方法，其他定义在Home接口中的方法，以及定义在Home接口直接或非直接地继承的接口中的方法，开发者必须指定方法的事务属性；
- Message-driven Bean中，开发者必须指定onMessage方法的事务属性。

开发者可指定的事务属性包括以下几种：NotSupported、Required、Supports、RequiresNew、Mandatory和Never。

下面对这六种事务属性具有的事务语义分别进行描述。

(1). NotSupported

当某方法的事务属性被指定为NotSupported时，容器将在未指明的事务上下文中调用此方法。具体情形如下：

当客户端请求伴随事务上下文，容器将在调用EJB的业务方法之间，暂停当前线程伴随的事务上下文。当方法调用完成之后，容器恢复前面暂停的事务上下文。伴随客户端调用事务上下文不会传递到资源管理器或业务方法中调用的其他组件。 如此业务方法调用了其他组件，则此调用不伴随任何事务。

当客户端请求不伴随事务上下文，容器也将不在事务上下文中进行此方法调用。

另外，如果方法中调用了其他的组件，则调用将不会伴随事务上下文。

(2). Required

当某方法的事务属性被指定为Required时，容器将在有效的事务上下文中调用此方法。具体情形如下：

当客户端请求伴随事务上下文，则容器在此事务上下文中调用此业务方法。

如客户端请求不伴随事务上下文，容器在调用此业务方法之前，将自动开始一个新的事务，并在此事务中自动征用（enlist）此业务方法访问的所有事务性资源，如方法中调用了其他的EJB组件，则调用将伴随此容器开始的事务。当容器对业务方法的调用完成，容器将试图提交事务。在方法调用结果返回客户端之前，容器将执行事务的提交协议（如两阶段提交协议）。

(3). Supports

当某方法的事务属性被指定为Supports时，容器将按如下规则调用此方法：

- 如客户端请求伴随事务上下文，容器按照事务属性为Required的情况，使用相同步骤进行处理；
- 如客户端请求不伴随事务上下文，容器按照事务属性为NotSupported的情况，使用相同步骤进行处理。

	注意
	使用Supports事务属性时必须注意，因为根据客户端请求是否伴随事务上下文，决定了不同的执行模式，进而决定了不同的方法事务语义，只有在这两种不同的执行模式中都能正确运行的方法，才能考虑使用Supports事务属性。

(4). RequiresNew

当某方法的事务属性被指定为RequiresNew时，容器将在一个新的事务上下文中调用此方法。具体情形如下：

当客户端请求不伴随事务上下文，容器在调用此业务方法之前，将自动开始一个新的事务，并在此事务中自动征用（enlist）此业务方法访问的所有事务性资源，如方法中调用了其他的EJB组件，则调用将伴随此容器开始的事务。当容器对业务方法的调用完成，容器将试图提交事务。在方法调用结果返回客户端之前，容器将执行事务的提交协议（如两阶段提交协议）。

当客户端请求伴随事务上下文，容器在开始一个新的事务和调用业务方法之前，将自动暂停当前客户端线程伴随的事务上下文。在方法调用和容器开始的事务完成之后，容器将恢复前面暂停的客户端事务。

(5). Mandatory

当某方法的事务属性被指定为Mandatory时，容器将在客户端的事务上下文中调用此方法。客户端调用此方法时，必须伴随事务：

- 如客户端请求伴随事务上下文，容器按照事务属性为Required的情况，使用相同步骤进行处理；
- 如客户端请求不伴随事务上下文，如果客户端为远程客户端，则容器将抛出javax.transaction.TransactionRequiredException异常；如果客户端为本地客户端，则容器将抛出javax.transaction.TransactionRequiredLocalException异常；

(6). Never

当某方法的事务属性被指定为Never时，容器将不会在任何EJB规范中规定的事务上下文中调用此方法。客户端调用此方法时，不能伴随事务：

- 如客户端请求伴随事务上下文，如果客户端为远程客户端，则容器将抛出java.rmi.RemoteException异常；如果客户端为本地客户端，则容器将抛出javax.ejb.EJBException异常；
- 如客户端请求不伴随事务上下文，容器按照事务属性为NotSupported的情况，使用相同步骤进行处理；

四. 必须遵守的规则

- 对于Message-driven Bean中的方法，只能使用Required和NotSupported事务属性；
- 对于EJB2.1规范中的容器管理持久性的Entity Bean中的方法，只能使用Required、RequiresNew或Mandatory等三个事务属性；
- 如EJB实现了javax.ejb.SessionSynchronization接口，则方法只能使用Required、RequiresNew或Mandatory等三个事务属性；

16.2.10.4 使用Bean管理事务与容器管理事务

从以上部分中，可以了解到Apusic应用服务器中，不同的事务划分方式的特点与用法，以及其对应的事务语义。除了本章中列出的有关EJB事务方面的开发规则之外，开发者在编写EJB还应该注意以下准则：

- 应尽可能地使用声明型的、容器管理的事务类型，避免代码中充斥着对事务管理API的调用，这不仅能减少开发者需要做的工作，同时也能减少应用最终发布时容器产生错误的代码，而且，可以改变应用的行为而不需要修改代码；
- 处于分布式环境中的事务要在尽量小的时间内提交或回滚，事务应该在接收到客户端请求之后开始并在对客户端进行响应时结束。当用户对响应返回的的数据进行操作时，事务不应该还处于活动的状态。这样，因为数据库的锁被保持在更短的时间

之内，可以减少应用对资源的争用；

- Session Bean通常作为一组相互关联的Entity Bean的前端，以将Entity Bean中相互关联的方法组合到一个事务中，这样，Session Bean的方法即可作为一个单个的工作单元。只有在方法内部需要控制更高级、更复杂的业务逻辑时，才采用Bean管理事务的方式。但是，使用容器管理事务的方式必须作为优先的考虑；

- 应避免在JSP、Servlet、Java应用客户端等客户端中划分事务的方式，这种方式具有使用Bean管理事务带来的缺点，同时还违反了多层企业应用中划分层次的原则，错误地把系统中的关键部分（事务管理）放到了整个系统的前端或图形界面的逻辑中。

综上所述，EJB中的事务模型划分需要开发者进行全面的考虑，同时也需要注意采用不同的事务划分方式对应用开发带来的影响。

16.2.11 EJB的安全管理

摘要

本节描述Apusic应用服务器中EJB体系结构中，有关安全管理方面的内容，包括安全角色、安全主体（Security Principal）、安全角色引用等概念的介绍；包括EJB体系结构中对组件方法级别的访问控制进行声明性的设置；包括组件模型中提供的用于安全管理方面的API的介绍；包括如何映射系统的安全主体与定义组件方法访问控制的安全角色，以及如何映射代码中的角色引用和安全角色的方法；包括如何配置组件方法在执行过程中，使用不同于调用者身份的其他身份对其他组件进行调用。

16.2.11.1 安全模型

为降低开发者在保护应用安全方面的负担，并且使企业应用可以在在部署时灵活地制定安全策略，EJB体系结构中，采用的是基于角色授权的安全模型。

基于角色授权的安全模型使开发者不再在应用的业务逻辑中对安全策略进行硬编码，而且，大多数EJB组件的业务方法中也不应包含与安全有关的逻辑，在应用被部署到不同的生产环境时，根据不同的需要，可以改动应用的安全策略而不需要更改应用中的代码。

16.2.11.1.1 安全角色（SECURITY ROLE）与方法权限

EJB 组件模型中的安全，是基于安全角色进行授权的。所谓安全角色，实际上它代表逻辑上的权限组合，例如，一个名为"teller"的安全角色，可能代表了能执行某个账户组件中的"checkTransactionHistory"和"checkBalance"方法的能力。

EJB体系结构中安全角色是指语义上的权限组合。权限指为成功运行应用，应用中给定类型的用户所必须具有的权限。

开发者在EJB的部署描述中声明安全角色，并使用已声明的安全角色对定义在组件接口和Home接口中的方法进行授权。通常，每个安全角色具有调用一组定义在组件接口和Home接口中的方法的权限。

16.2.11.1.2 安全主体（SECURITY PRINCIPAL）

在Apusic应用服务器中，系统提供定义系统用户与角色进行映射的工具。通常意义上，安全主体指系统中代表某个具体组织或个人的用户标识。在用户登陆系统后，系统将为用户的调用赋予一个安全主体对象，在整个操作过程中，系统将使用此对象来区分不同的用户调用和按照安全策略对调用进行授权。

16.2.11.1.3 安全角色与安全主体的映射

在应用部署到实际生产环境之前，按照EJB部署描述和应用部署描述中的关于安全角色的声明信息，部署者或系统管理员对安全角色与安全主体进行映射。因此，部署应用时，部署者看到的是代表权限的安全角色，和代表实际用户的安全主体，不需要对单个的方法进行权限设置，降低了部署者将应用部署到特定生产环境中去的负担。

使用这种将应用安全逻辑与实际运行环境分开的安全模型，提高了应用的可移植特性和组件的可重用特性，应用的安全策略通过修改部署描述文件即可改变。

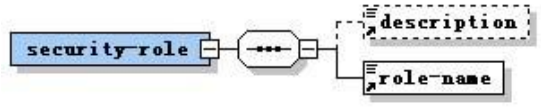
16.2.11.2 使用安全模型

EJB的安全体系结构提供了一个灵活而易于配置的模型，本节将详细描述在Apusic应用服务器中，EJB安全管理的配置。同时，因为所有的安全策略并非都能进行声明性的表示，因此，EJB的编程模型还提供了一个简单的接口，使开发者可以在业务方法中访问安全上下文。

16.2.11.2.1 声明安全角色

开发者需要在组件的部署描述文件ejb-jar.xml中，使用assembly-descriptor的子元素security-role对组件中的安全角色进行声明，每个安全角色对应一个security-role元素。security-role元素的结构如下图：

security-role元素



开发者使用description元素提供此安全角色的描述信息给应用的装配者，并使用role-name元素声明此安全角色。

如下：

```
...
<assembly-descriptor>
    <security-role>
        <description>

This role includes the employees of the
enterprise who are allowed to access the
employee self-service application. This role
is allowed only to access his/her own
information.
</description>
        <role-name>employee</role-name>
    </security-role>
    <security-role>
        <description>

This role includes the employees of the human
resources department. The role is allowed to
view and update all employee records.
</description>
```

```
        <role-name>hr-department</role-name>
    </security-role>
    <security-role>
        <description>

This role includes the employees of the payroll
department. The role is allowed to view and
update the payroll entry for any employee.
</description>
        <role-name>payroll-department</role-name>
    </security-role>
    <security-role>
        <description>

This role should be assigned to the personnel
authorized to perform administrative functions
for the employee self-service application.

This role does not have direct access to
sensitive employee and payroll information.
</description>
        <role-name>admin</role-name>
    </security-role>

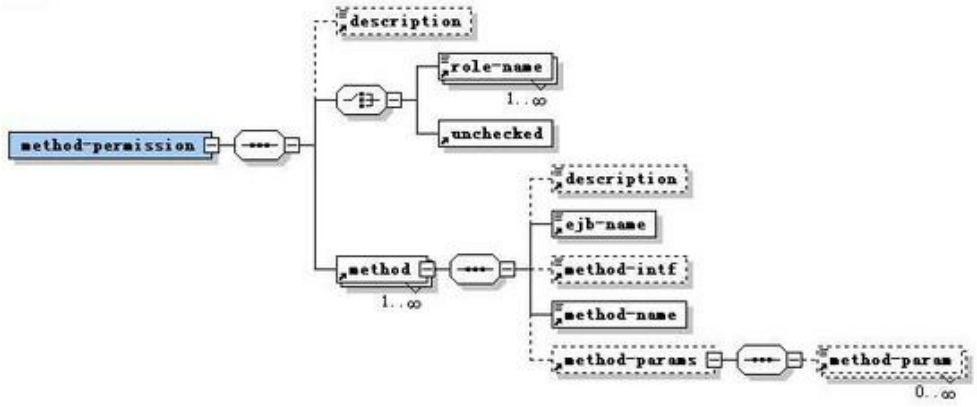
...
</assembly-descriptor>
```

16.2.11.2.2 定义方法许可

开发者通过在ejb-jar.xml文件中，使用assembly-descriptor的子元素method-permission，对security-role元素中声明的安全角色集合，以及在Session Bean和Entity Bean组件接口和Home接口中的方法（包括组件接口和Home接口继承的接口中定义的方法）集合之间的二元关系集合进行定义。即当且仅当安全角色R可以调用方法M时，则此二元关系集合中即存在一个元素E（R，M）。

method-permission元素的结构如下图：

method-permission元素



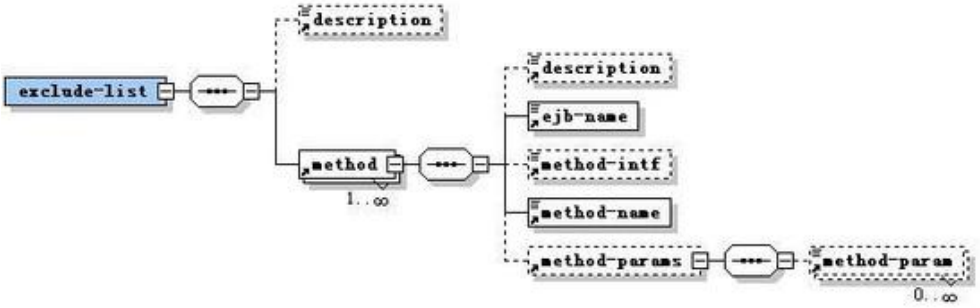
method-permission元素中，开发者可以使用role-name元素定义一个或多个安全角色，使用method元素定义一个或多个方法元素，其中，每个安全角色允许调用使用method元素定义的所有方法。

开发者可以使用method-permission元素中可选的description元素对此方法许可进行描述。

开发者可使用一个unchecked元素取代method-permission中的role-name元素列表，此uncheck元素用于指定本method-permission元素中定义的方法集合中的每个方法，在对方法进行调用之前，不需要进行授权。

另外，开发者也可使用exclude-list元素指定不应被调用的方法集合。exclude-list元素的结构如下图：

exclude-list元素



对于每个在exclude-list元素中指定的方法，无论调用者的使用何种身份还是方法已在method-permission元素中定义，对此方法的调用都将被拒绝。

method-permission元素和exclude-list中的method元素，使用ejb-name、method-name和method-params 元素来指定定义在EJB组件接口或Home接口中的一个或多个方法，下面是几种定义method元素的正确方法：

```
<method>
    <ejb-name>Foo</ejb-name>
    <method-name>*</method-name>
</method>
```

上例指所有定义在名为Foo的EJB组件的接口和Home接口中的方法。

```
<method>
    <ejb-name>Foo</ejb-name>
    <method-name>Bar</method-name>
</method>
```

上例指定义在名为Foo的EJB组件的接口和Home接口中的所有名为Bar的方法。如组件接口和Home接口中定义了多个Bar方法，则本例包含所有这些方法。

```
<method>
    <ejb-name>Foo</ejb-name>
    <method-name>Bar</method-name>
    <method-params>
        <method-param>PARAMETER_1</method-param>

...
<method-param>PARAMETER_N</method-param>
    </method-params>
</method>
```

上例指定义在名为Foo的EJB组件的接口和Home接口中的名为Bar并具有相同参数的方法。

另外，开发者可使用可选的method-intf元素指定具体接口中的方法，如下：

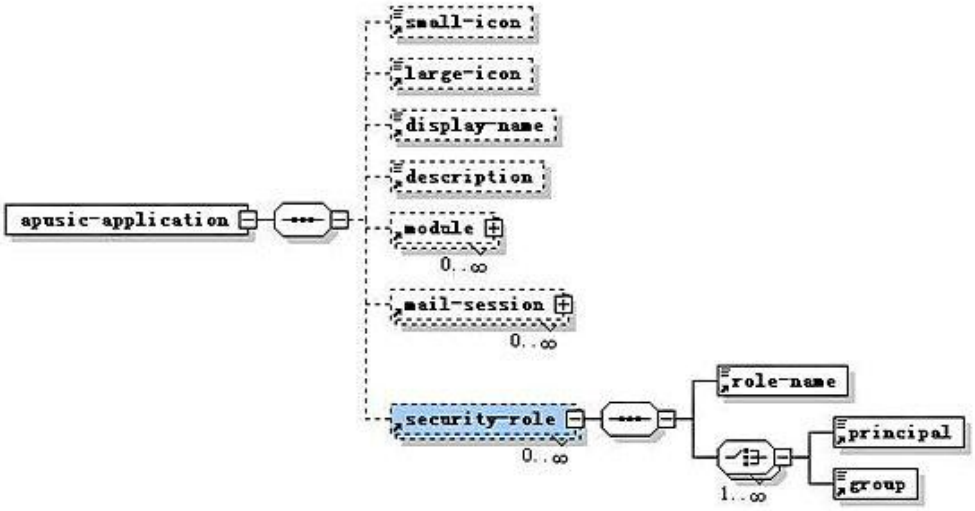
```
<method>
    <ejb-name>Foo</ejb-name>
    <method-intf>Home</method-inf>
    <method-name>*</method-name>
</method>
```

本例指名为Foo的EJB的Home接口中的所有方法。method-intf元素的值可以是Home、LocalHome、Remote和Local分别对应EJB组件模型中的远程Home接口、本地Home接口、远程接口和本地接口。

16.2.11.2.3 定义角色映射

在应用被部署到生产环境之前，部署人员需要将组件中的安全角色映射到实际环境中的安全主体，映射在整个应用的部署描述文件apusic-application.xml中进行，使用apusic-application中的security-role元素进行映射。apusic-applicaton以及其中的security-role元素的结构如下图：

apusic-applicaton元素



由于此映射属于应用范畴，因此，本节不对此元素进行进一步描述。

16.2.11.2.4 调用中的身份传播

从前面的EJB的安全模型的描述中，可以了解到当客户端调用组件接口或Home接口中的方法时，容器将检查方法的访问权限，如果方法的访问许可被设置，则容器将根据伴随调用的身份信息进行授权，判断调用者是否具有调用此方法的权限，如成功，则容器将调用组件实例的方法。

需要考虑到，被调用的方法也可能调用其他组件或对某些受保护资源进行操作，因此，需要确定这种情况下，伴随调用的身份信息如何传播。

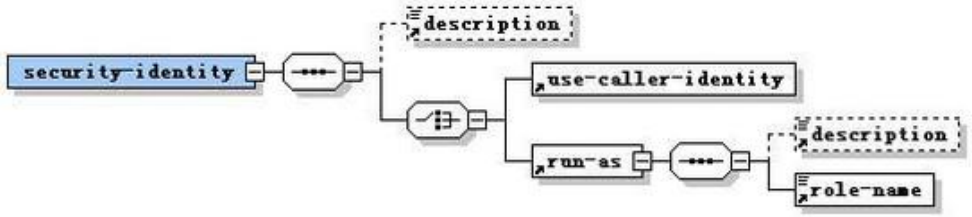
一种情况是，组件在执行时不具有特定调用者的身份信息（如Message-driven Bean的onMessage方法），在这种情况下，组件只能调用其他组件的未设置访问权限的方法，和不受保护的资源。

另一种情况是，组件在执行时具有特定调用者的身份信息，并且在组件中需要调用其他组件的方法，或者访问受保护的资源，在这种情况下，组件只能对具有同样访问权限或没有定义访问权限的组件方法或资源进行访问。

因此，EJB组件模型提供了一种声明性的身份传播方式，使一个组件在调用其它组件或访问受保护资源时，采用原调用者的身份信息还是使用开发者定义的其他安全角色进行。同时，必须注意这种传播机制不会影响方法的原调用者的信息。

开发者可以使用security-identity元素来定义这种身份传播行为。security-identity元素的结构如下图：

security-identity元素



开发者可以使用use-caller-identity元素定义组件在访问其他组件时，使用组件调用者的身份。

	注意
	Message-driven Bean不能使用use-caller-identity元素

开发者可以使用run-as元素定义组件在访问其他组件时，使用其它的调用者身份，这里的调用者身份实际上是一个逻辑角色，必须对应一个ejb-jar.xml部署描述文件中使用security-role元素定义的安全角色。

下面是一个定义run-as元素的例子：

```
...
<enterprise-beans>

...
<session>
        <ejb-name>EmployeeService</ejb-name>

...
<security-identity>
            <run-as>
                <role-name>admin</role-name>
            </run-as>
        </security-identity>

...
</session>

...
</enterprise-beans>

...
```

在运行时，当此EmployeeService组件的实例中访问了其他的组件或资源，则伴随此访问传播的将是admin的身份信息。

16.2.11.2.5 关于安全角色引用

安全角色引用指开发者使用security-role-ref元素对EJB组件代码中使用的所有安全角色名称进行声明，将其映射到实际生产环境中定义的安全角色。以提高应用中安全控制策略的灵活性，同时，使安全策略仍保持在组件代码之外。

通常，涉及调用者对组件方法的调用的授权和访问控制策略对调用者是透明的。但声明型的安全策略有时不能完全满足应用的需求，因此，EJB规范中提供了可在组件代码中访问调用者安全上下文的API。在javax.ejb.EJBContext接口中提供了两个方法，开发者可以通过这两个方法访问调用者的安全相关的信息。

```
public interface javax.ejb.EJBContext
{

...

//

// The following two methods allow the EJB class
// to access security information.

//

java.security.Principal getCallerPrincipal();

    boolean isCallerInRole(String roleName);

    //

    // The following two EJB 1.0 methods are deprecated.

    //

    java.security.Identity getCallerIdentity();

    boolean isCallerInRole(java.security.Identity role);

...

}
```

以上这两个方法，开发者可以在实例具有了安全上下文之后进行调用。如以下列表中的方法：

- 有状态Session Bean

ejbCreate方法；

ejbRemove方法；

ejbActivate方法；

ejbPassivate方法；

定义在组件接口中的业务方法；

对于容器管理事务的有状态Session Bean，可以在从SessionSynchronization接口中的afterBegin、beforeCompletion、afterCompletion等方法中调用上面提到的两个方法。

- 无状态Session Bean

ejbCreate方法；

ejbRemove方法；

定义在组件接口中的业务方法。

- Entity Bean

ejbCreate方法；

ejbPostCreate方法；

ejbRemove方法；

ejbHome方法；

ejbLoad和ejbStore方法；

16.2.11.2.6 GETCALLERPRINCIPAL

开发者可以调用getCallerPrincipal方法取得一个使用java.security.Principal接口表示的调用者的信息。

必须注意，getCallerPrincipal方法返回的是EJB调用者的身份信息，不代表任何run-as元素指定的身份信息。

以下是调用getCallerPrincipal方法的例子：

```
public class EmployeeServiceBean implements SessionBean {

EJBContext ejbContext;

public void changePhoneNumber(...) {

...

// Obtain the default initial JNDI context.

Context initCtx = new InitialContext();

// Look up the remote home interface of the EmployeeRecord
// enterprise bean in the environment.

Object result = initCtx.lookup(
```

```
"java:comp/env/ejb/EmplRecord");

// Convert the result to the proper type.

EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)

javax.rmi.PortableRemoteObject.narrow(

result,

EmployeeRecordHome.class);

// obtain the caller principal.

callerPrincipal = ejbContext.getCallerPrincipal();

// obtain the caller principal's name.

callerKey = callerPrincipal.getName();

// use callerKey as primary key to EmployeeRecord finder
EmployeeRecord myEmployeeRecord =

emplRecordHome.findByPrimaryKey(callerKey);

// update phone number
myEmployeeRecord.changePhoneNumber(...);

...

}

}
```

16.2.11.2.7 ISCALLERINROLE

对于较难使用部署描述中的声明型的方法许可进行定义的安全检查，开发者可以通过调用此方法来完成。

开发者使用isCallerInRole(String roleName)方法检查调用者是否被指定到通过roleName参数代表的角色名。

下面是调用isCallerInRole方法的例子：

```
public class PayrollBean...
{

EntityContext ejbContext;

public void updateEmployeeInfo(EmplInfo info) {

oldInfo = ... read from database;

// The salary field can be changed only by callers
// who have the security role "payroll"

if (info.salary != oldInfo.salary &&

!ejbContext.isCallerInRole("payroll")) {

throw new SecurityException(...);

}

...

}

...

}
```

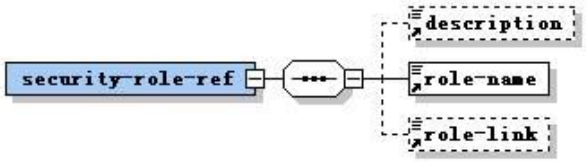
当开发者在代码中调用了isCallerInRole，则必须在EJB的部署描述中使用security-role-ref元素，将代码中的安全角色引用名称，映射到实际的安全角色名。

16.2.11.2.8 声明代码中使用的安全角色引用

开发者需要在部署描述中，使用security-role-ref元素声明代码中使用的安全角色名。这样，在应用被部署到实际生产环境中时，可以通过部署描述将代码中使用的安全角色名和使用security-role元素定义的安全角色进行映射。

当开发者使用了isCallerInRole方法，则必须对每个代码中引用的安全角色进行声明，security-role-ref元素结构如下图。

security-role-ref元素



开发者需要使用role-name元素声明代码中使用的角色名，使用role-link元素将其映射到某个使用security-role元素声明的安全角色名上。如下例：

```
...
<enterprise-beans>

...
<entity>
<ejb-name>AardvarkPayroll</ejb-name>
<ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>

...
<security-role-ref>
<description>

This role should be assigned to the
employees of the payroll department.

Members of this role have access to
anyone's payroll record.

The role has been linked to the
payroll-department role.
</description>
<role-name>payroll</role-name>
<role-link>payroll-department</role-link>
</security-role-ref>

...
</entity>

...
<assembly-descriptor>

...
```

```
<security-role>
<description>

This role includes the employees of the payroll
department. The role is allowed to view and
update the payroll entry for any employee.
</description>
<role-name>payroll-department</role-name>
</security-role>

...
</assembly-descriptor>

...
</enterprise-beans>

...
```

上例中，使用了security-role-ref元素，将代码中使用的安全角色名payroll映射到了部署描述中使用security-role元素声明的payroll-department安全角色。

16.2.12 EJB Timer Service

摘要

EJB Timer Service是一个由EJB容器管理的定时器服务程序。

16.2.12.1 概述

EJB Timer Service是让用户可以基于时间调度表调用指定的程序，这种服务是由容器管理的。它可以指定事件在某一个时间点触发，也可以指定事件过多长时间然后触发，还可以指定事件以某个固定的周期来触发，这解决了在EJB容器内部几种Bean不能用多线程来达到类似功能的困难。

EJB Timer Service是一个粗粒度的时间点触发服务，它只能提供应用程序级别的时间调度和时间触发服务，不能用于精确度很高的实时系统的时间精度要求。

同时EJB Timer Service是一个可靠的、事务性的服务。

16.2.12.2 组件模型单元

16.2.12.2.1 接口描述

- TimerService接口
- TimedObject接口
- Timer接口

- TimerHandle接口

16.2.12.2.2 TIMERSERVICE接口

TimerService的实现对象从EJBContext接口的实现获得，方法为getTimerService。TimerService接口定义了如下方法：

```
public interface javax.ejb.TimerService {

public Timer createTimer(long duration,

java.io.Serializable info);

public Timer createTimer(long initialDuration,

long intervalDuration, java.io.Serializable info);

public Timer createTimer(java.util.Date expiration,

java.io.Serializable info);

public Timer createTimer(java.util.Date initialExpiration,

long intervalDuration, java.io.Serializable info);

public Collection getTimers();

}
```

info参数用来标识时间到期的意义和描述

getTimers返回和当前Enterprise Bean相关的Timers

16.2.12.2.3 TIMEDOBJECT接口

商务逻辑实现bean对象需要实现的接口，该接口的 ejbTimeout 方法用来供Timer到期时进行回调。

```
public interface javax.ejb.TimedObject {

public void ejbTimeout(Timer timer);

}
```

16.2.12.2.4 TIMERHANDLE接口

该接口定义用来获得Timer对象。

```
public interface javax.ejb.TimerHandle extends java.io.Serializable{

public javax.ejb.Timer getTimer();

}
```

16.2.12.2.5 TIMER接口

该接口可以进行取消Timer的动作，也可以获得一些Timer的其它信息，例如：下次到期是什么时候，离下次到期还有多长时间等。

```
public interface javax.ejb.Timer {

public void cancel();

public long getTimeRemaining();

public java.util.Date getNextTimeout();

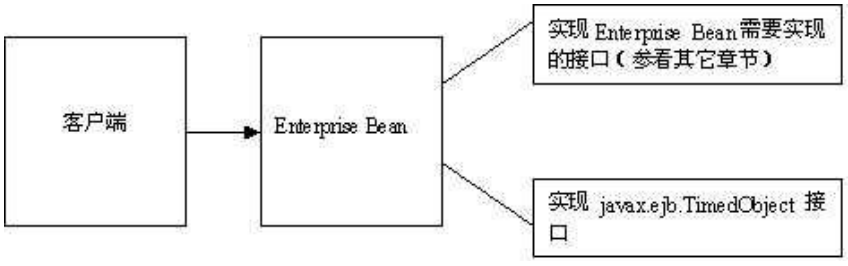
public javax.ejb.TimerHandle getHandle();

public java.io.Serializable getInfo();

}
```

16.2.12.2.6 功能实现框架

实现javax.ejb.TimedObject接口，javax.ejb.TimedObject接口只有一个ejbTimeout()方法，该方法就是事件到期时触发的方法，需要把Timer需要执行的商业逻辑放在该方法面里。



16.2.12.3 使用EJB Timer Service

当一个bean的相关时间点到达的时候，容器将会调用该对象的ejbTimeout()方法，方法中，开发人员可以将需要执行的内容写在这个方法内。

16.2.12.3.1 使用范围

stateless session bean、 message-driven bean、entity bean可以在定时器中登记，然后定时器在一定的时间点触发，记住：statefull session bean不能实现TimedObject接口。

16.2.12.3.2 创建TIMER

从context对象获取TimerService对象

```
TimerService timerService = context.getTimerService();
Timer timer = timerService.createTimer(intervalDuration, "created timer");
```

Timer是持久性的，假如应用服务器关机，当应用服务器重新启动时，Timer将被重新激活。

createTimer方法Date和long类型的参数表示到毫秒级的精度，但是Timer不是为实时系统提供的，所以它触发的精度可能不能达到精确的毫秒级，对于普通的应用程序，它的精度已经足够。

16.2.12.3.3 定义TIMER需要执行的内容

使用Timer的Bean必须实现javax.ejb.TimedObject接口，javax.ejb.TimedObject接口只有一个ejbTimeout()方法，该方法就是事件到期时触发的方法，需要把Timer需要执行的商业逻辑放在该方法面里。例如：

```
public class TimerSessionBean implements SessionBean, TimedObject {
  private SessionContext context;

  public TimerSessionBean() {}

  public void myCreateTimer(long intervalDuration) {
    System.out.println("TimerSessionBean: start createTimer ");
    TimerService timerService = context.getTimerService();
    Timer timer = timerService.createTimer(intervalDuration, "created
timer");
  }

  public void ejbTimeout(Timer timer) {
    System.out.println("TimerSessionBean: ejbTimeout ");
  }

  public void setSessionContext(SessionContext sc) {
    System.out.println("TimerSessionBean: setSessionContext");
    context = sc;
  }

  public void ejbCreate() {
    System.out.println("TimerSessionBean: ejbCreate");
```

```
    }

    public void ejbRemove() {}

    public void ejbActivate() {}

    public void ejbPassivate() {}
}
```

16.2.12.3.4 取消和保存TIMER

Timer可以由以下几种方式取消：

- 当一个单事件的Timer到期时，EJB容器调用完ejbTimeout方法后取消Timer
- 当entity bean的实例被移除时，EJB容器取消和被移除实例相关的Timer
- 当调用Timer的cancel方法时，EJB容器取消Timer

16.2.12.3.5 获得TIMER的信息

除了cancel 和 getHandle 方法之外，Timer接口还定义了一些其它方法来获得Timer的部分信息

```
public long getTimeRemaining();

public java.util.Date getNextTimeout();

public java.io.Serializable getInfo();
```

getInfo方法返回调用createTimer方法时最后一个参数的值。

16.2.12.3.6 TIMER和交易

当一个 enterprise bean在交易的内部创建Timer，当交易回滚时，创建Timer的动作也同时回滚。

当一个 enterprise bean在交易的内部取消Timer，当交易回滚时，Timer取消的动作也同时被废除。

16.3 Web Service开发手册

16.3.1 概述

16.3.1.1 Web Services与SOA

Web Services是一套用于支持联通的主机间跨网络交互与通信的技术标准。Web Services以多种平台中立的XML技术为基础，为异构平台提供了标准的互操作方式。

最近我们常听到的一个概念是SOA（Service Oriented Architecture）——面向服务的架构。将企业级应用转换为面向服务的架构能达到标准化、服务重用、服务抽象、低耦合、服务组合等多重优良特性。

SOA是一种架构模型，不同的公司可能采取不同的实现方案。但在当前市场环境下，Web Services可以说是更符合SOA原则且使用最为广泛的技术平台。

	**注意**
	注：下文WS或Web服务义同Web Services

16.3.1.2 Web服务类型

在概念层面上，服务是指通过网络端点对外提供的软件组件。服务消费者和提供者专注于使用标准格式的文档数据交换请求和响应信息，对消息在目的端的处理则不作任何假设。

在技术层面上，服务端对外提供的标准或非正式契约将服务实现与使用者隔离开来，Web服务可以采用任何方式实现，只须服务行为与契约描述相符。

通常使用的Web服务有三类：

- RPC方式的Web服务
- 基于SOAP的Web服务
- REST风格Web服务

16.3.1.2.1 RPC方式的WEB服务

表面看来，RPC方式和基于SOAP的Web服务都使用了WSDL和SOAP，其基本特点是通过WSDL（Web Service Description Language）提供服务契约描述并通过SOAP（Simple Object Access Protocol）消息协议进行服务调用和响应。

RPC是一种传统的Web服务实现方式，实质上它只是通过一个简单的映射，将SOAP消息中携带的调用请求分发到由某一特定语言编写的函数或方法上，然后将调用结果封装返回。从另一角度看，RPC方式的Web服务不过是新瓶装旧酒，基于SOAP的另一种远程过程调用实现方式而已。这种以通用标准迎合老旧技术方案的做法造成服务器端、客户端及其所使用WS实现之间的紧密耦合性，难以达成Web服务设计主要目的之一的互操作性，因而不被WS-I基本协议集（WS-I Basic Profile）支持。

Java平台上通过JAX-RPC提供了RPC方式的Web服务支持。

16.3.1.2.2 基于SOAP的WEB服务

基于SOAP 的Web服务提供了一套完备的分布式系统技术方案，这中完备性主要体现在如下几个方面：

1. 规范化的两大基础构件：WSDL和SOAP，基于XML，标准的描述与数据结构易于实现自动化处理，使得WS能够被方便地引入到具体语言平台中。同时，SOAP的协议栈机制为WS调用提供了灵活的处理模型。
2. 基于SOAP的 Web服务是WS-*兼容的，其中最基本的是WS-I Basic Profile，该规范致力于Web服务互操作性而对Web服务协议集进行限定，这意味着采用SOAP Web服务体系的应用能够方便地集成异构子系统、与外部老旧系统进行整合。另外，其它WS-扩展标准可用于满足企业级分布式系统中某些复杂的非功能性需求，这些标准包括WS安全（WS-Security）、WS可靠性（WS-Reliability）、WS可靠消息（WS-ReliableMessaging）、WS事务（WS-Transaction）和WS寻址（WS-Addressing）。因此，基于SOAP的Web服务也常被称作WS-* Web服务。

3. 在具体的语言平台上，如Java中，由JAX-WS为客户端异步服务调用提供支持。

在Java EE 6环境中，JAX-WS（JSR-224）和JWS（JSR-109）为基于SOAP的Web服务提供功能性支持，定义了客户端和服务器端编程模型与API。这也是本手册介绍的主要内容。

16.3.1.2.3 REST风格WEB服务

REST（Representatinal State Transfer）的概念源自Roy Fielding博士的论文——Architectural Styles and the Design of Network-based Software Architectures，REST是一种设计风格而不是标准，通常基于HTTP、URI、XML和HTML等广泛流行的协议与标准，该风格将网络应用看作是一种通过网页表征其内部状态的虚拟状态机，并通过用户对链接的选择操作由对应URI代表的服务器资源，进而驱动其状态的转变。

REST 风格（RESTFul）Web服务是一种使用HTTP并遵循REST原则的Web服务。RESTFul Web服务从资源URI、服务接受的MIME类型以及约定固化的方法语义三方面定义了一个轻量级基础架构，REST风格Web服务不需要复杂的SOAP消息和WSDL描述，减小了服务开发的复杂度，降低了技术实用的门槛。适用于临时或基本的Web服务整合场景。Java EE 6中通过JAX-RS（JSR-311）提供了RESTFul Web服务的支持。

适用RESTful Web服务的场景通常需要满足如下条件：

1. 服务完全无状态
2. 提供者和消费者对于服务契约达成一致理解
3. 允许采用caching机制提高性能
4. 资源受限的场合，如PDA等
5. 在现有网站中整合与分发Web服务

16.3.1.2.4 选择适用的WEB服务类型

一般来说，由于相对轻量级，RESTful Web服务常用于简单服务发布和跨Web服务集成等场景；另一方面，对于需要灵活定制、异构系统整合、系统间互操作性及服务质量(QoS)有较高要求的企业级应用集成场合，基于SOAP的Web服务则较为适用；同时，配合UDDI，采用WSDL和SOAP的Web服务将更容易发布和发现。

由于业务领域主要面向企业级应用，金蝶Apusic应用服务器对于JAX-RPC（RPC方式Web服务）/JAX-WS（基于SOAP的Web服务）和JWS（WS in JavaEE）规范的支持已经达到了极为完善的程度。在后续版本中我们将加入对JAX-RS的支持。

16.3.2 在Apusic AS中开发WS

在Java EE环境中，按照JWS规范的规定，端口组件（Port Component）是指一个Web服务的服务器端视图，端口组件也称端口（Port），每一个端口组件对应着WSDL中定义的一个端口地址，并在该地址上服务于WSDL里端口类型（PortType）中声明的操作请求。

端口组件Java EE环境中WS所定义的编程模型包含如下构件：

1. WSDL文档：并非严格必须，WSDL文档是可以自由发布给第三方的标准Web服务描述。WSDL文档和服务端点接口之间必须符合JAX-RPC/JAX-WS规范中定义的WSDL与Java间的映射规则。
2. 服务端点接口：Service Endpoint Interface（SEI），是WSDL中的端口类型在Java语言平台上的映射。该接口定义了应该由服务实现Bean所实现的方法。

3. 服务实现Bean：Service Implemention Bean（SIB），一个用于提供Web服务业务逻辑的Java类。表明了端口组件与JavaEE容器间的契约，使得Web服务业务逻辑能够与容器所提供的服务交互。服务实现Bean应该同SEI中定义的方法和签名保持一致，但不要求SIB在Java语法上实现SEI。在Java EE容器环境中有两种类型的服务实现Bean：托管于Web容器中的JAX-RPC/JAX-WS POJO和运行在EJB容器中的无状态会话Bean（Stateless EJB）
4. 安全角色引用：Security Role References，端口声明在部署描述符中的角色名将在部署时被映射为物理角色，从而允许服务提供实例级的安全检查。

开发人员编写服务实现Bean，细心处理SEI与WSDL之间的映射，然后由部署人员在部署描述符中声明端口组件，及其所关联的包含端口类型和服务绑定描述的WSDL文档，这是JAX-RPC时代典型的WS开发方式，耗时费力。如果使用JAX-WS，部署描述符将是可选的，大部分配置、映射与定制将由对业务代码和使用情况更为熟悉的开发人员通过加诸于SEI或SIB上的注解完成，如若必要才通过部署描述符进行调整，这种开发方式显然更为便捷。

作为企业级的中间件产品，金蝶Apusic应用服务器出于兼顾功能与实用性的考虑，加强了对JAX-WS的支持，同时全面兼容JWS规范（JSR-109），在Apusic应用服务器中开发、部署、发布和使用Web服务将是快捷而高效的。

通常来说，Web服务的开发有两种方式：

1. 自顶向下，以独立的的WSDL服务描述为蓝本，基于具体平台开发兼容的服务业务逻辑，这种方式主要适用于现有系统整合与业务集成的场景。
2. 自底向上，从代码开始有目的地构建并对外发布Web服务，这种方式主要适用于以Web服务的方式对外提供增值服务。

两种方式的选择取决于是否预先就存在独立的WSDL服务描述，其根本要求是保证业务代码与服务描述的语义兼容，本质上并无区别。

由于Apusic应用服务器完全兼容规范、且在Apusic上开发部署Web服务相当简单，因而我们并没有对此提供独立开发工具的支持，当今所有支持企业级开发的工具完全能胜任此项工作，这也包括金蝶中间件公司早先提供的基于Eclipse的开发工具——OperaMasks Studio。

下面，我们将以一个Hello Web服务为例，采用自底向上的开发方式演示在Apusic应用服务器中开发、部署和发布符合规范Web服务的基本过程：

16.3.2.1 通用构件

为了简化开发步骤、降低使用复杂度，部署的Apusic应用服务器中的Web服务端口组件可选地可以提供下列通用构件：

1. WSDL，服务描述
2. SEI，服务端点接口
3. webservices.xml，部署描述符

16.3.2.1.1 WSDL

可以在应用中包含服务的WSDL描述，然后通过JAX-WS中定义的类型级别@WebService注解的wsdlLocation属性引用此描述，或者通过部署描述符webservices.xml配置此描述，SEI和SIB必须同WSDL中的内容保持语义兼容，否则应用部署将失败。

如果没有在应用中提供WSDL，Web服务成功发布后，Apusic应用服务器将为每一个Web服务自动生成对应的WSDL，并在服务的绑定地址上按照一定的URL模式对外提供web访问。得到此WSDL的客户端将能成功访问相应的Web服务。

由于是自底向上开发方式，WSDL内容可参照服务发布完成后产生的WSDL文档。

WSDL规范链接:http://www.w3.org/TR/wsdl

16.3.2.1.2 SEI

SEI是WSDL服务描述中PortType部分对应于Java平台的接口定义，SEI中的方法对应于PortType中定义的操作，SEI需要与提供的WSDL保持语义相同。

端口组件SEI在Apusic应用服务器中也是可选的。如果没有为SIB显式指定SEI，服务器将自动为SIB生成默认SEI，该默认SEI方法组成如下：

1. 所有存在注解@WebMethod且exclude属性为false（默认值）的public方法。
2. 否则，所有非继承自Object类的public方法，排除包含@WebMethod注解且exclude属性为true的那些。

本手册中所使用的SEI代码如下：

```
package com.apusic.ws.endpoint.sei;

import javax.jws.WebService;

@WebService
public interface HelloSEI {
  public String hello(String input);
}
```

说明：

1. 必须通过注解@WebService标记一个接口是某个Web服务在JavaEE环境下的SEI。
2. 接口中的所有方法代表了Web服务所提供的操作集合，可通过方法上的@WebMethod注解对这些操作进行定制

Java与WSDL 1.1之间的映射规则可参考JAX-WS规范。

16.3.2.1.3 WEBSERVICES.XML

在JAX-WS规范中，部署描述符webservices.xml是可选的，其原有功能已由JAX-WS或JWS规范中的相关注解取代。

但另一方面，可以通过该部署描述符在部署时（而不是开发期）对端口组件进行一定的装配，如服务SEI或WSDL引用等，因为相对于注解，该部署描述符中的内容具有更高优先级。本示例未提供webservices.xml。相关结构与内容可参考JSR-109规范。

下面将要介绍本示例中所使用到的SIB，不同的SIB运行于不同的容器中，使用不同的容器服务，提供Web服务的业务逻辑。SIB是区分不同端口组件类型的关键。

16.3.2.2 JAX-WS POJO

本示例中所使用的JAX-WS POJO服务实现Bean代码如下：

```
package com.apusic.ws.endpoint.jaxws;

import javax.jws.WebService;

@WebService(endpointInterface = "com.apusic.ws.endpoint.sei.HelloSEI")
public class HelloPOJO {
  public String hello(String input) {
    return "hello " + input + " from jaxws pojo endpoint";
  }
}
```

说明：

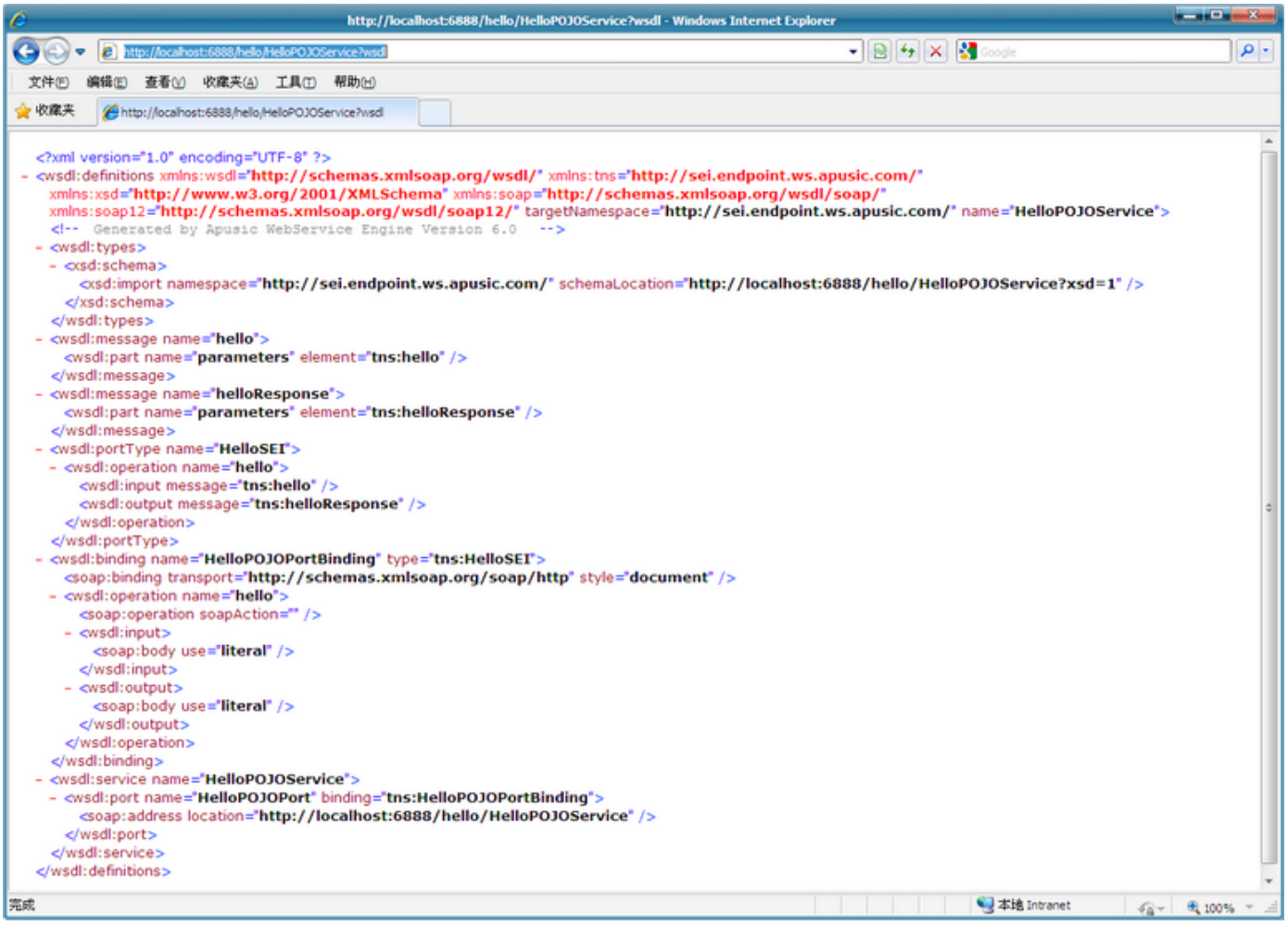
1. 通过@WebService注解声明类HelloPOJO是一个SIB
2. 通过endpointInterface属性引用SIB所实现的SEI，从代码可以看出，HelloPOJO并不需要在Java语法上实现HelloSEI，只须在方法签名上同HelloSEI保持一致。（其实只需要在WSDL语义上同HelloSEI保持一致即可）
3. 引用SEI后，在生成的WSDL行，SEI将对应于服务抽象定义的部分，如类型、消息和接口类型；SIB则主要被映射到WSDL定义中的绑定、服务和端口部分。上述各部分内容都可以通过JSR-181或JSR-224中定义的注解进行定制。详请参照相关规范。

JAX-WS POJO运行于web容器中，和其它运行在web容器中的应用组件一样，也能够得到由容器所提供的线程池、请求分发、资源注入、生命周期回调等服务。详情请参考JSR-109中的相关规定。

至此，一个JAX-WS POJO Web服务端口组件开发完成。

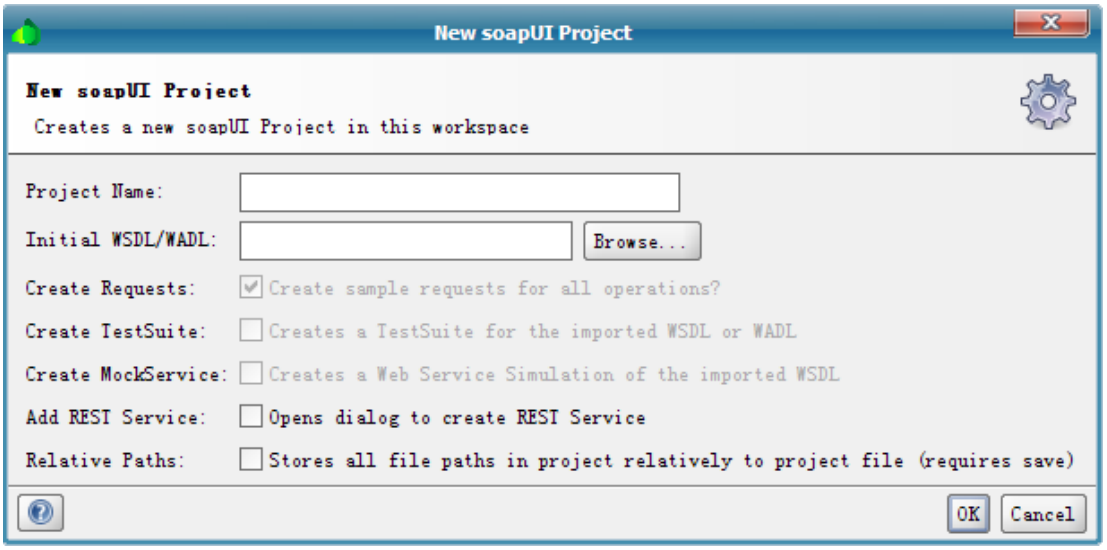
接下来，进行部署、发布和测试：

1. 将上述代码编译后打包到任一Web应用war包中（或者可以直接使用目录部署，使用ear包同样可行），本示例将其加入一个context-root为hello的web应用中。
2. 启动应用服务器，将上述war包复制到所启动域的applications目录下，应用将被自动部署。
3. 打开IE浏览器，输入地址http://localhost:6888/hello/HelloPOJOService?wsdl（假设应用部署在本地），可以看到如下由应用服务器自动生成的服务WSDL描述：

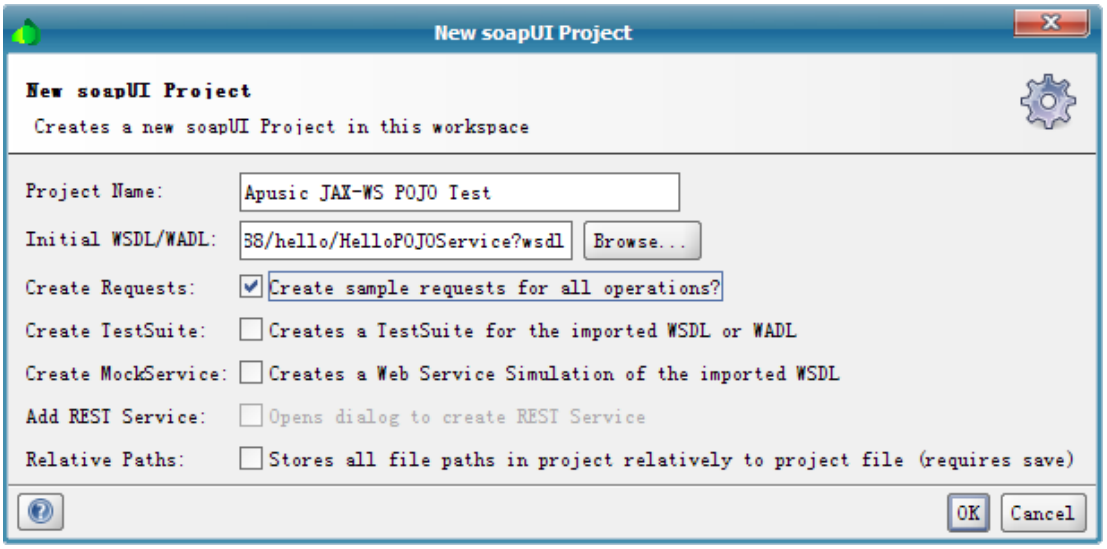


4. 使用Soap-UI测试Hello Web服务：

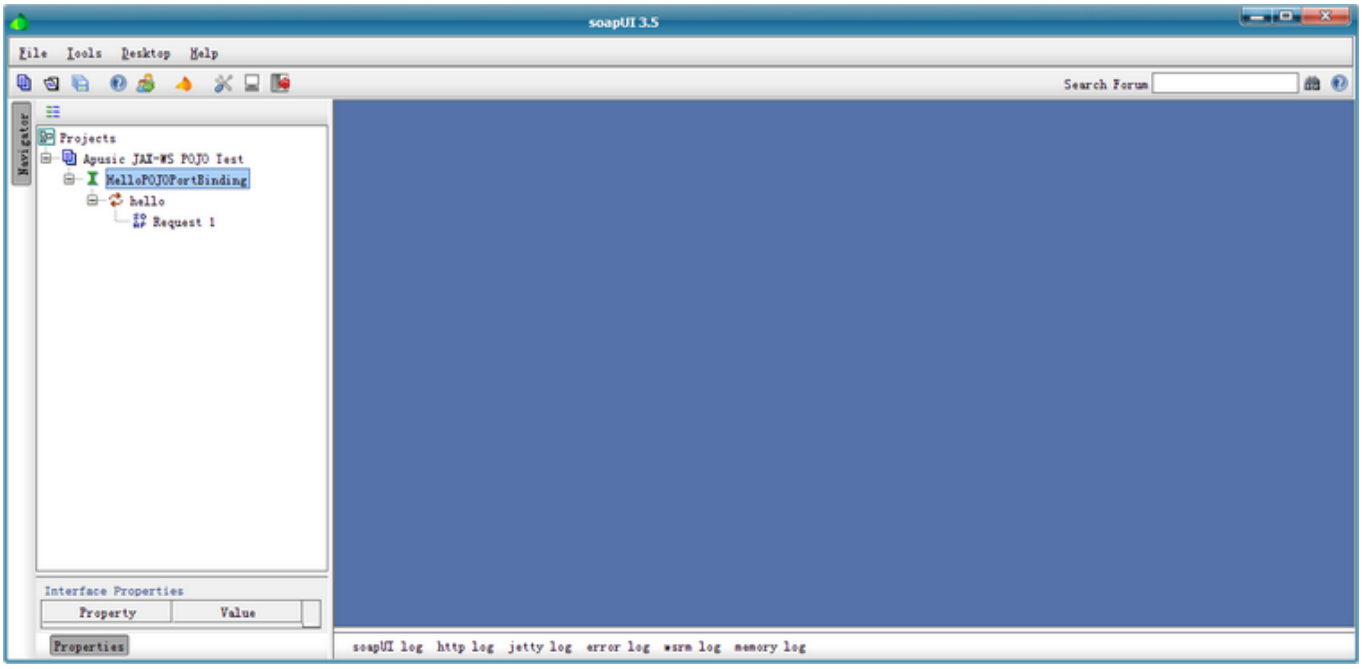
1. 打开Soap-UI，选择File-->New soapUI Project，界面如下图：



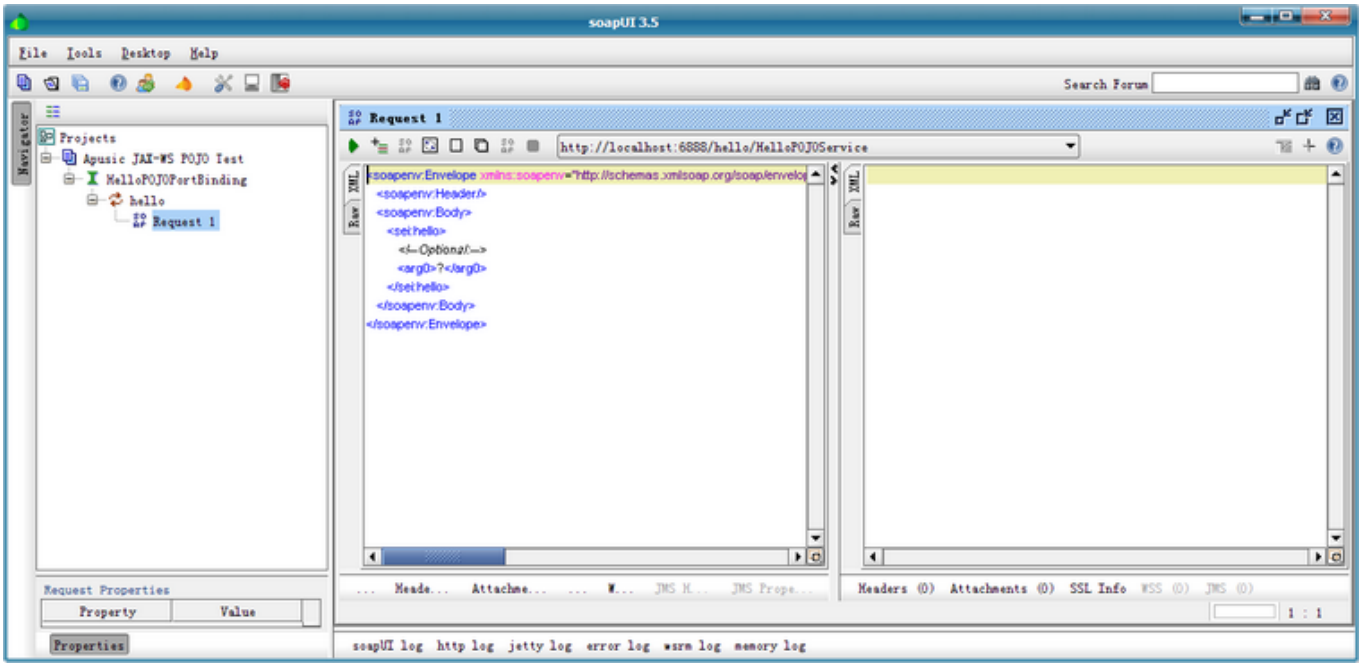
2. 在Project Name一栏任意输入，如Apusic JAX-WS POJO Test，在Initial WSDL/WADL一栏输入上述生成的WSDL web访问地址，选择Create Requests，如下图：



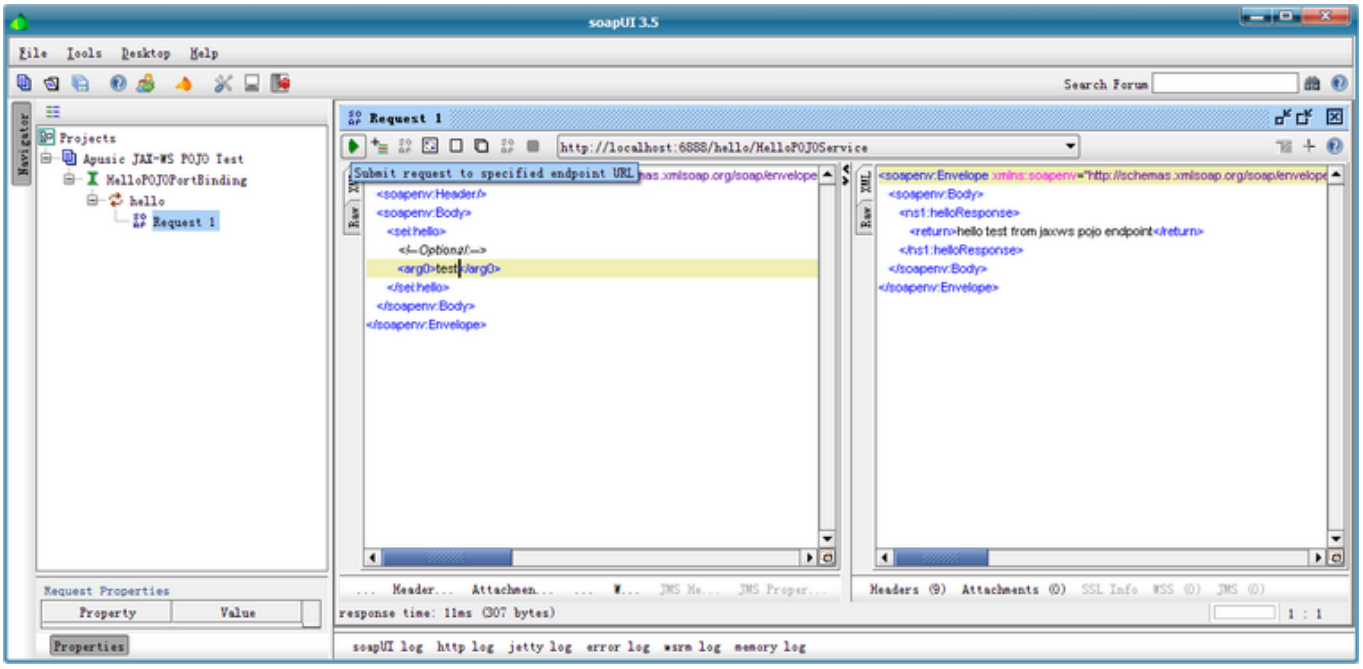
3．单击OK，界面左侧工程列表栏出现新建的soapUI工程，如下图：



4．双击左侧Request 1节点，右侧工作区间出现测试请求响应显示界面，如下图：



5．在工作区间左侧的请求区域的arg0元素中输入待测试消息如test，并单击上方的提交按钮，右侧响应区域将显示成功调用Hello Web服务后的响应消息，如下图：



5. 测试通过。

下面，将演示开发Hello Web服务的Stateless EJB SIB版本：

16.3.2.3 Stateless EJB

开发一个普通的EJB 3 stateless EJB，并为其添加@WebService注解，引用SEI，代码如下：

```
package com.apusic.ws.endpoint.ejb;

import javax.ejb.Stateless;
import javax.jws.WebService;

@Stateless
@WebService(endpointInterface = "com.apusic.ws.endpoint.sei.HelloSEI")
public class HelloEJB {
  public String hello(String input) {
    return "hello " + input + " from stateless ejb endpoint";
  }
}
```

至此，一个Stateless EJB Web服务端口组件开发完成。上述SIB运行在EJB容器中，能够使用EJB容器所提供的各种服务，同时对外响应SOAP请求提供Web服务。

将上述SEI和SIB代码编译并打包到一个jar（或ear）包中，加入应用服务器的applications目录自动部署，其部署、发布与测试的过程与JAX-WS POJO端口组件类似。唯一不同在于其WSDL访问地址为：http://localhost:6888//HelloEJBService/HelloEJB?wsdl

可以看到，由于JavaEE相关规范中对WS编程模型的简化，以及注解、元数据映射等相关技术的引入，更得益于Apusic应用服务器对规范的完美支持，在Apusic应用服务器中开发和使用Web Service确实是相当快捷高效的。

另一方面，在一个简单的入门级示例中，Apusic应用服务器所支持的其他定制功能和高级特性必然难以尽数体现。但正如前文所说，Apusic应用服务器完美兼容规范，其它地方随处可见的JWS典型用法一般都适用于Apusic应用服务器，这也是无需更多琐碎示例的缘由。

16.3.3 开发WS客户端

Web Services作为一个分布式的技术平台，JAX-WS和JWS为其所定义的编程模型涵盖了服务器端和客户端两部分。Java环境中的WS客户端分为如下两类：

1. 独立的Java应用，桌面应用和applet可归为此类
2. JavaEE容器托管的的应用模块，包括容器托管的客户端、Web、EJB和JCA应用

根据JWS规范规定，Java客户端访问Web服务的方式有如下几类：

1. 通过工具类库生成的静态客户端代理
2. 在XML、SOAP或更低层面直接操作请求和响应消息的编程API——SAAJ、JAX-WS
3. 客户端异步Web服务调用机制——轮询或回调
4. JavaEE服务器托管的Web服务客户端代理——@WebServiceRef

这种分类依据是调用方式和抽象层次的不同，并不是一个严格泾渭分明、界限清晰的分类，某些类别在底层可能就使用了另一类别中的技术。

另外，客户端类别和Web服务使用方式之间并没有确定的对应关系，除了容器托管的服务客户端代理一般只能用于容器托管的应用模块，其他的方式两类客户端都可以使用。

下面将针对前面所的Hello示例Web服务（JAX-WS POJO版本，Stateless EJB版本使用方式类似）分别使用上述技术开发一个示例客户端：

16.3.3.1 静态代理

开发步骤：

1. 在已经设置好JDK环境变量的前提下，打开命令行窗口，使用JDK自带的工具wsimport，指定Web服务URL地址，生成客户端代理。如下图：



2. 进入目录查看，发现在包com.apusic.ws.client.proxy中产生了下图所示的类结构：



说明（各Java类的角色与作用）：

1. package-info：用于定制包级别的Java-WSDL映射
2. HelloPOJOService：服务的客户端视图，客户端调用的入口
3. HelloSEI：端口的客户端编程接口
4. Hello、HelloResponse：操作调用的请求和响应消息对象，遵循JAXB映射
5. ObjectFactory：上述JAXB对象的工厂类

3. 将上述代理加入类路径，即可开始客户端编写，最终使用静态客户端代理的WS客户端代码如下：

```
package com.apusic.ws.client;

import com.apusic.ws.client.proxy.HelloPOJOService;
import com.apusic.ws.client.proxy.HelloSEI;

public class StaticProxyClient {
  public static void main(String[] args) {
    HelloPOJOService service = new HelloPOJOService();
    HelloSEI port = service.getHelloPOJOPort();
    String response1 = port.hello("static-proxy-client");
    // response would be "hello static-proxy-client from jaxws pojo endpoint"
  }
}
```

代码演示了通过静态代理方式进行同步WS调用的步骤，可以看出，通过JAX-WS API屏蔽了底层消息封装、协议绑定和网络通信的细节，静态代理的方式无疑是最简单易用的。在示例中，手动生成代理类的方式加重了些许工作负担，但这项工作完全可以通过开发工具自动完成；另外，有些框架和工具中还提供了动态客户端的支持（类似于Java中的反射调用），完全不需要显式生成任何类。

16.3.3.2 直接操作XML消息动态调用

通过JAX-WS所提供的编程接口，可直接在XML消息层面上操作请求和响应消息进行WS调用，此时无需代理类的参与，下面的示例代码演示其中的一种方式：

```
package com.apusic.ws.client;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.concurrent.ExecutionException;
import javax.xml.namespace.QName;
import javax.xml.soap.MessageFactory;
```

```
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Response;
import javax.xml.ws.Service;
import org.w3c.dom.DOMException;

public class XMLMessageClient {
  public static void main(String[] args) throws MalformedURLException,
SOAPException, DOMException, InterruptedException, ExecutionException {
    String namespaceUri = "http://sei.endpoint.ws.apusic.com/";
    Service service = Service.create(new
URL("http://localhost:6888/hello/HelloPOJOService?wsdl"), new
QName(namespaceUri, "HelloPOJOService"));
    Dispatch<SOAPMessage> dispatch = service.createDispatch(new
QName(namespaceUri, "HelloPOJOPort"), SOAPMessage.class,
Service.Mode.MESSAGE);
    MessageFactory msgFactory = MessageFactory.newInstance();
    SOAPMessage request = msgFactory.createMessage();
    request.getSOAPBody().addBodyElement(new QName(namespaceUri, "hello",
"tns")).addChildElement(new QName("arg0")).addTextNode("dispatch-soap-
message");
    request.saveChanges();
    SOAPMessage response1 = dispatch.invoke(request);
    String responseMsg1 = ((SOAPElement) ((SOAPElement)
response1.getSOAPBody().getChildElements(new QName(namespaceUri,
"helloResponse")).next()).getChildElements(new QName("return")).next())
        .getFirstChild().getTextContent();
    Response<SOAPMessage> response2 = dispatch.invokeAsync(request);
    String responseMsg2 = null;
    while (response2.isDone())
      responseMsg2 = ((SOAPElement) ((SOAPElement)
response2.get().getSOAPBody().getChildElements(new QName(namespaceUri,
"helloResponse")).next()).getChildElements(new QName("return")).next())
          .getFirstChild().getTextContent();
    // responseMsg1 and responseMsg2 should both be "hello dispatch-soap-
message from jaxws pojo
    // endpoint"
```

```
    }
}
```

上面的代码同时演示了同步与异步（轮询）动态调用的情况，我们并没有对客户端代码进行任何修改，由此可以确认JAX-WS通过客户端编程模型支持异步WS调用。

16.3.3.3 JavaEE托管环境中的WS客户端注入

该类别的WS调用方式只适用与JavaEE服务器（如Apusic应用服务器）托管下的应用组件，这些组件既包括传统的Servlet、EJB等，甚至可以另一个需要使用其他Web服务的端口组件，这样可以通过一个集中的入口整合各种资源为客户提供一致的服务。

开发一个简单的Servlet，其中通过@WebServiceRef注解注入Hello Web服务客户端。值得注意的是，此种方式需要使用到前面通过工具生成的服务客户端视图和SEI接口，由此也导致了@WebServiceRef注解的两种使用方式，本示例只演示其中的一种，代码如下：

```
package com.apusic.ws.client.servlet;

import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;
import com.apusic.ws.client.proxy.HelloPOJOService;
import com.apusic.ws.client.proxy.HelloSEI;

public class ClientServlet extends HttpServlet {
  @WebServiceRef(name = "helloService", value = HelloPOJOService.class)
  private HelloSEI service;

  public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException {
    process(request, response);
  }

  public void doPost(HttpServletRequest request, HttpServletResponse
response) throws IOException {
    process(request, response);
  }

  private void process(HttpServletRequest request, HttpServletResponse
response) throws IOException {
    String wsMessage = service.hello("web-service-ref");
    response.getWriter().print(wsMessage);
```

```
    }
}
```

应用的web.xml配置文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">
    <servlet>
        <servlet-name>ClientServlet</servlet-name>
        <servlet-class>com.apusic.ws.client.servlet.ClientServlet</servlet-
class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ClientServlet</servlet-name>
        <url-pattern>/client</url-pattern>
    </servlet-mapping>
    <service-ref>
        <service-ref-name>helloService</service-ref-name>
        <service-
interface>com.apusic.ws.client.proxy.HelloPOJOService</service-interface>
        <service-ref-type>com.apusic.ws.client.proxy.HelloSEI</service-ref-
type>
        <wsdl-file>http://localhost:6888/hello/HelloPOJOService?wsdl</wsdl-
file>
    </service-ref>
</web-app>
```

当通过浏览器请求Servlet时会看到响应hello web-service-ref from jaxws pojo endpoint，表明Hello Web服务调用成功。

可以看到，采用容器注入方式的优势在于能够将使用代码和配置文件分离，这样Web服务被部署到其他位置，将只需要修改配置文件，而无需改动应用代码。

在众多的Web服务客户端调用方式中，本文只选取了其中典型的几个进行讲解，但这只是初涉皮毛，其他注意事项和调用方式请参考JWS和JAX-WS规范以及通用于JavaEE环境的各种教程。

16.3.4 Apusic AS的SPI支持

Apusic应用服务器支持标准的Handler和Provider处理机制，这些都只是更低级的XML消息处理SPI，详细使用请参见相关教程与资料。

16.4 客户端开发手册

16.4.1 客户端简介

金蝶Apusic应用服务器的客户端可以是Web浏览器或运行在客户端容器中的应用程序。金蝶Apusic应用服务器提供了不同类型的客户端来连接服务器，执行应用逻辑，并将结果返回。客户端应用往特定的URL发送请求，服务器接收到请求并处理它，然后返回。这样，客户端就执行了运行在服务器上的远程程序和功能。

16.4.1.1 客户端类型

这一节将介绍金蝶Apusic应用服务器支持的客户端类型。

16.4.1.1.1 WEB客户端

Web客户端包含两部分：

- 包含不同类型的标识语言（如HTML、XML）的动态Web页面，它们是由运行在Web服务器上的Web组件产生的。
- 浏览器，显示从服务器上接收的页面。

Web客户端有时被称作"瘦客户端"。瘦客户端的特点是不用去查询数据库或执行复杂的业务逻辑，而把这些操作交给应用服务器完成，由应用服务器去考虑安全、访问速度、提供服务和可靠性等问题。

16.4.1.1.2 应用客户端

应用客户端是运行于客户端Java 虚拟机中的客户端程序。应用客户端基于Java应用模型，即通过main方法调用并运行到虚拟机中止。但是，如同其他的应用组件，应用客户端依靠应用客户端容器提供系统服务。与其它J2EE容器相比，应用客户端容器可以说是相对简单的容器。

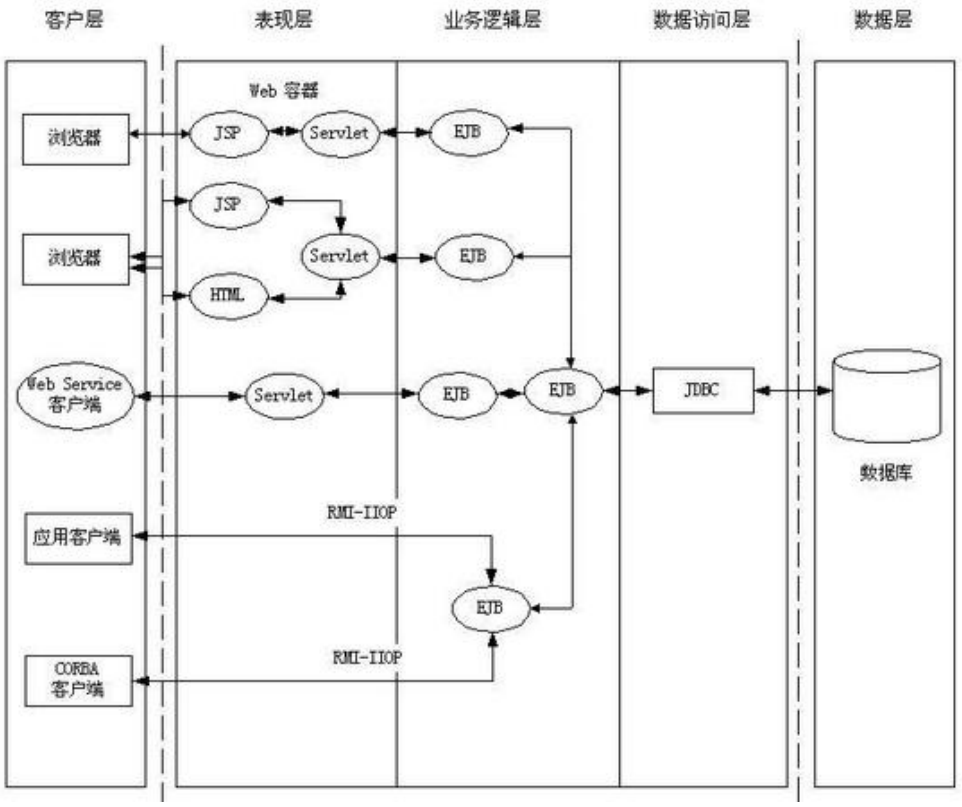
16.4.1.1.3 CORBA客户端

CORBA 客户端可以用通用对象请求代理（CORBA）支持的任何语言进行开发，包括Java、C++和C等。当独立运行的程序或其他应用服务器作为客户端访问部署在金蝶Apusic应用服务器上的EJB组件时，可以使用CORBA客户端。金蝶Apusic应用服务器符合EJB3.0/2.1规范，支持使用IIOP（Internet Inter-ORB Protocol）访问EJB。Java RMI-IIOP客户端使用JNDI（Java Naming and Directory Interface）定位EJB，使用RMI-IIOP（Java Remote Method Invocation-Internet Inter-ORB Protocol）访问远程EJB组件的业务方法。而标准的CORBA客户端使用CORBA Naming Service定位EJB，像访问普通的CORBA对象一样访问EJB。

16.4.1.1.4 WEB SERVICES客户端

下图描述了客户端和金蝶Apusic应用服务器的体系结构：

客户端和金蝶Apusic应用服务器体系结构



16.4.2 使用应用客户端容器

这一节介绍如何使用RMI/IIOP协议访问应用服务器，和如何使用应用客户端容器开发和打包应用客户端。

16.4.2.1 应用客户端容器简介

应用客户端容器是由一组Java类和必须的文件组成，它同客户端应用一起运行在客户端的Java虚拟机中，管理应用客户端组件的执行。象其他Java EE应用组件一样，应用客户端的执行依赖于客户端容器提供的系统服务。客户端容器和金蝶Apusic应用服务器通讯使用RMI/IIOP。和其他服务器端的Java EE容器相比，应用客户端容器可以说是相对简单的容器。

应用客户端容器提供的系统服务有：

- 创建客户端运行环境，负责和应用服务器进行通讯。
- 提供JNDI包装，使客户端能够使用"java:"名字空间。如其他J2EE 组件，应用客户端使用JNDI查找EJB，访问对资源管理器的访问、服务器被管理对象等等，应用客户端使用java:JNDI名字空间来访问以上这些内容。
- 认证客户端。应用客户端容器自动完成JAAS用户认证。

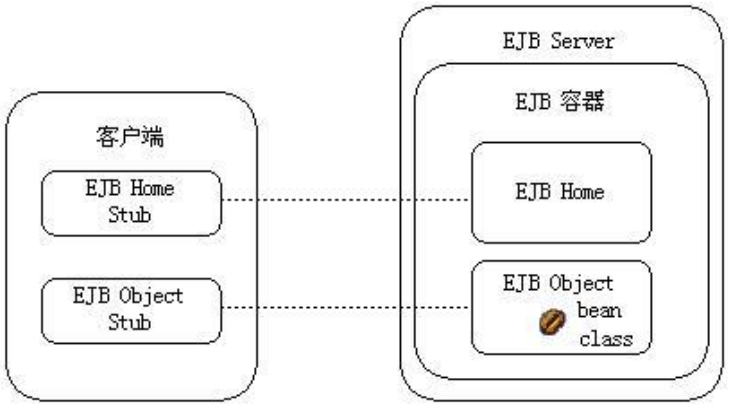
另外，应用客户端不需要直接访问Java EE平台的事务功能。J2EE产品不需要提供JTA的UserTransaction对象给应用客户端使用。当然，应用客户端可以调用开始事务边界的EJB，并可使用JDBC API 提供的事务功能。但是当应用客户端调用EJB时开始了一个JDBC API 事务，事务环境不会传播到EJB 服务器。

应用客户端拥有所有的Java 2 Standard Edition 平台功能，每个应用客户端在客户端的Java虚拟机中运行。通过设置应用客户端JAR文件中的manifest 文件的Main-Class属性，指定JAR文件中的Java类文件，运行时虚拟机查找Main-Class属性指定的类文件，执行其main方法（为准备容器环境，应用客户端容器在执行客户端程序之前运行，安装SecurityManager，初始化名字服务客户端类库，等等）。

16.4.2.2 开发应用客户端

EJB容器可以有多种不同的实现方式，但每一个容器都支持标准的EJB客户端视图。当开发客户端时，开发者通过home接口获得对EJB的访问，然后通过调用定义在远程接口中的方法，使用业务逻辑。这个简单的编程模型被应用到所有的EJB客户端。

使用EJB Home和远程接口



范例程序说明了客户端程序必须包含的内容：

- 获得JNDI initial context。
- 使用initial context获得EJB Home引用。
- 通过调用EJB Home的create或find方法获得EJB实例引用。
- 访问EJB的业务逻辑方法。

16.4.2.2.1 定位HOME接口

使用JNDI查找和定位EJB组件的Home接口。下面的代码描述了定位EJB组件Home接口的步骤：

```
//创建initial contenxt
Context initial = new InitialContext();
Object objref = initial.lookup("java:comp/env/ejb/Hello");
HelloHome home = (HelloHome) PortableRemoteObject.narrow(objref,
HelloHome.class);
```

16.4.2.2.2 创建EJB实例

客户端调用HelloHome对象的create方法创建EJB实例。create方法返回Hello类型的对象。远程接口Hello定义了提供给客户端调用的EJB业务方法。

```
Hello hello = home.create();
```

16.4.2.2.3 调用业务方法

开发者调用远程接口Hello上的业务方法，EJB容器将调用运行在服务器端的HelloBean实例上相应的方法。客户端调用sayHello业务方法的代码如下：

```
String str = hello.sayHello("apuaic");
System.out.println(str);
```

16.4.2.3 打包应用客户端

应用客户端被打包成JAR文件并且包含部署描述文件和MANIFEST.MF文件。部署描述文件是一个名为application-client.xml的xml文件，用于描述应用客户端所引用的EJB和外部资源。

application-client.xml文件必须符合application-client_1_4.xsd文档类型声明，并在文件中指定正确的schema：

```
version="1.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/application-client_1_4.xsd"
```

范例程序的客户端部署描述文件为，其中定义了EJB引用：

```
<?xml version="1.0" encoding="UTF-8"?>
<application-client version="1.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/application-client_1_4.xsd">
    <display-name>AppClient</display-name>
    <ejb-ref>
        <ejb-ref-name>ejb/Hello</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>samples.ejb.HelloHome</home>
        <remote>samples.ejb.HelloHome</remote>
    </ejb-ref>
</application-client>
```

应用客户端开始执行MANIFEST.MF文件中Main-Class属性指定的类的main方法，例如：

```
Manifest-Version: 1.0
Created-By: Apache Ant 1.5.1
Main-Class: samples.client.Client
```

应用客户端JAR文件结构如下图：

应用客户端JAR文件结构



16.4.2.4 运行应用客户端

首先将应用客户端JAR文件和EJB JAR文件打包成J2EE应用，部署到Apusic应用服务器上。详细步骤请参考"打包和部署Java EE应用""。

然后使用Apusic提供的ejbGen工具生成可独立运行的应用客户端JAR文件：

```
java -classpath %apusic_home%/lib/apusic.jar com.apusic.tools.ejbgen.Main
sample.ear appclient.jar
```

最后，在客户端容器中运行应用客户端：

```
java -classpath %apusic_home%/lib/apusic.jar -
Djava.security.auth.login.config==login.conf com.apusic.client.Main -jar
appclient.jar
```

	注意
	应用客户端运行在客户端容器中，会自动使用JAAS用户认证，因此需要提供LoginModules配置文件。配置文件中LoginModule的名字必须为"client"。客户端容器会自动使用名字为"client"的LoginModule进行JAAS用户认证。如果使用作为登录配置文件。也可以指定-noauth参数，以匿名身份访问服务器。

16.4.2.5 不使用应用客户端容器

客户端可以不使用应用客户端容器，在获得初始化context时需要指定java.naming.factory.initial和java.naming.provider.url：

//设置初始化属性

```
Properties env = new Properties();
env.put("java.naming.factory.initial",
```

```
"com.apusic.naming.jndi.CNContextFactory");
env.put("java.naming.provider.url", "iiop://localhost:6888");
//获得 initial context
Context initial = new InitialContext(env);
//定位Home接口
Object objref = initial.lookup("ejb/appclientsample");
```

定位Home接口使用的JNDI名称为部署EJB时指定的JNDI名称。

	注意
	初始化context factory的类名为：com.apusic.naming.jndi.CNContextFactoryProvider URL的格式为：iiop://<host>:<port>

16.4.2.6 关于范例

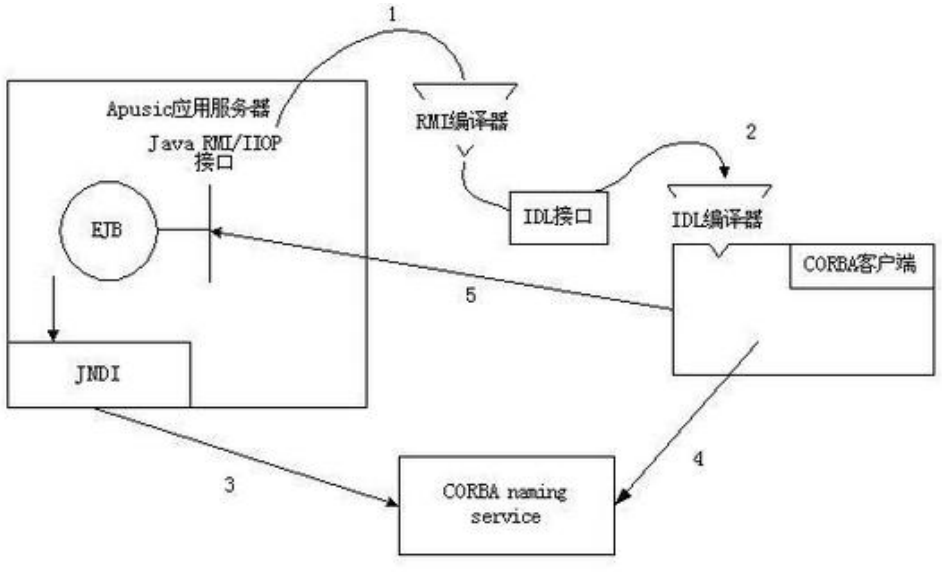
本节中的范例位于金蝶Apusic应用服务器安装目录中的docs/samples/client/rmi-iiop目录。

有关范例的内容、编译、部署与运行，可参考docs/samples/client/rmi-iiop目录下的readme.txt文件。

16.4.3 CORBA客户端开发

CORBA 客户端可以通过IIOP访问EJB组件。金蝶Apusic应用服务器完全支持标准的EJB-CORBA映射，也就是说，EJB的home和remote也是CORBA对象。开发者可以使用JDK提供的RMI编译器rmic-idl为EJB产生CORBA视图的IDL接口。用来定位EJB home接口的JNDI（Java naming service）被映射成CORBA naming service，这样，CORBA客户端就能查找和定位home接口并象CORBA对象一样使用它们。下图描述了Java到IDL映射的步骤：

Java到IDL映射



这些步骤包括：

1. 使用RMI编译器rmic -idl为EJB远程接口产生IDL；
2. 使用CORBA IDL编译器根据客户端的程序语言为客户端产生存根（stub）；
3. 配置EJB服务器使用CORBA名称服务。金蝶Apusic应用服务器缺省就使用CORBA名称服务作为JNDI 服务的提供者；
4. CORBA客户端在CORBA名称服务中查找和定位EJB；
5. CORBA客户端象访问普通CORBA对象一样访问EJB；

16.4.3.1 开发CORBA客户端

本小节通过一个范例展示了开发CORBA客户端的一般步骤。

本节中的范例位于金蝶Apusic应用服务器安装目录中的docs/samples/client/corba目录。有关范例的内容、编译、部署与运行，可参考docs/samples/client/corba目录下的readme.txt文件。

首先开发一个简单的EJB模块，编译并部署到Apusic应用服务器上。

然后使用RMI编译器rmic -idl为EJB远程接口产生IDL：

```
rmic -idl -noValueMethods -classpath
%apusic_home%/lib/apusic.jar;../ejb/build -d idl samples.ejb.Hello
samples.ejb.HelloHome
```

理论上可以使用任何CORBA规范支持的程序语言开发CORBA客户端。本范例展示的是Java CORBA客户端的开发，使用了IONA公司的Orbacus产品作为ORB和IDL编译器。可以到www.iona.com下载Orbacus，参考它的说明文档注册并安装，用户可以免费30天试用。注意，使用Orbacus最好在JDK1.3.1下进行开发。编译上一步得到的IDL：

```
jidl --all --output-dir src --package idl -Iidl -I"D:/JOOC/idl/OB" -
I"D:/JOOC/idl" ./idl/samples/ejb/Hello.idl
jidl --all --output-dir src --package idl -Iidl -I"D:/JOOC/idl/OB" -
I"D:/JOOC/idl" ./idl/samples/ejb/HelloHome.idl
```

其中，“d:\jooc”为Orbacus的安装目录。

开发者根据IDL编译器编译生成的Java文件，提供客户端实现，主程序的代码片断如下：

```
//设置系统属性，确定使用Orbacus作为ORB
java.util.Properties props = System.getProperties();
props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
props.put("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton");
props.put("ooc.orb.service.NameService",
"corbaloc::localhost:6888/NameService");
ORB orb = ORB.init(args, props);
```

```
//查找HelloHome，创建Hello，调用方法sayHello()
//注意，使用的CORBA Naming Service定位EJB home接口
System.out.println("Looking up Hello as CORBA object in CORBA naming
service");
NamingContextExt nc =
NamingContextExtHelper.narrow(orb.resolve_initial_references("NameService"));
idl.samples.ejb.HelloHome home =
idl.samples.ejb.HelloHomeHelper.narrow(nc.resolve_str("ejb/CORBAClientSample"))
idl.samples.ejb.Hello hello = home.create();
String str = (String) hello.sayHello("apusic,corba client");
System.out.println(str);
System.exit(0);
```

使用范例程序提供的ant脚本编译并运行客户端。

这个例子只是简单的展示了CORBA客户端能够访问EJB组件的方法，实际上CORBA和EJB的互操作还包含事务和安全的映射，情况比较复杂，因此要求开发者对CORBA规范和开发比较熟悉。而且Java到IDL的映射可能遇到很多问题，例如：ValueType映射，集合的使用等，这就使CORBA客户端访问EJB在很多情况下不能成功。有关EJB和CORBA客户端的开发还可以参考《Enterprise JavaBeansTM Components and CORBA Clients: A Developer Guide 》。

16.5 金蝶Apusic应用服务器系统服务

16.5.1 数据库连接池开发

16.5.1.1 概述

JDBC 是执行SQL 语句的JAVA API，由一系列使用Java 编写的类和接口组成。JDBC为工具和数据库开发人员提供标准API并使使用纯JAVA API 编写数据库应用成为可能。JDBC 是个低级的接口，意味着它只是用来直接调用SQL 命令。另外，JDBC 也作为诸如JMS和EJB等高级界面和接口的基础。

金蝶Apusic应用服务器支持JDBC4.0版本，JDBC4.0是本文档书写时JDBC API最新发布版。 有关JDBC3.0的特性，参见JDBC4.0特性。

JAVA 数据库连接(JDBC)提供对后台数据库的访问，JAVA 应用通过使用JDBC驱动程序（数据库供应商指定的数据库接口）取得对JDBC 的使用。尽管任何JAVA 应用都可以装载JDBC 驱动程序、连接数据库、执行数据库操作，Apusic应用服务器还提供一个高效的JDBC 数据库连接池。JDBC 数据库连接池是一个通过金蝶Apusic应用服务器管理的JDBC连接的命名组。金蝶Apusic应用服务器启动时打开JDBC 数据库连接并把他们加入到JDBC数据库连接池中。当应用请求一个连接时，应用从池中取得连接，使用后将连接返回到池中以备被其他应用使用。通常，建立数据库连接是个耗费时间和资源的操作。因此，通过限制连接操作次数的连接池，能提高服务器的效率。要在JNDI命名树中登记一个连接池，只需为它定义一个DataSource对象。Java 客户端应用通过对DataSource名字执行JNDI查找，即可从连接池中获取连接。服务器端Java类使用标准JDBC 接口，它是一个一般化的JDBC驱动程序接口。即使在数据层改变了数据库的类型，也不用更改应用的代码，使应用具有更强的可移植特性。

金蝶Apusic应用服务器提供已准备好的对DBMS 的连接池。连接池启动时连接即建立，消除了应用运行时建立连接的拥挤状态。可以从如HTTPServlet、EJB 等服务器端应用使用连接池和通过Apusic 的RMI 服务使用独立Java 客户端应用程序访问连接池。连接池需要一个两层的JDBC驱动程序生成Apusic 到DBMS 的连接。

16.5.1.2 使用JDBC连接池

16.5.1.2.1 服务器端JDBC应用

对于从服务器端应用如HTTP Servlet 等使用数据库连接池可使用JNDI 和DataSource对象，下例将在一个Servlet中使用数据源来演示从服务器端应用数据库连接池。

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.sql.*;
import java.sql.*;
import javax.naming.*;

public class ConnectionPoolServlet extends javax.servlet.http.HttpServlet {

public void service(HttpServletRequest req,

HttpServletResponse res)throws IOException{

try{

//取得InitialContext 对象

Context ctx=new InitialContext();

//通过JNDI 取得数据源

DataSource datasource=(DataSource)

ctx.lookup("jdbc/oracle");

//通过数据源取得数据库连接

Connection conn=datasource.getConnection();

...

}

}

...
```

16.5.1.2.2 客户端JDBC应用

金蝶Apusic应用服务器增加了JDBC 连接的远程访问接口，用户可以在Java客户端程序中通过RMI 调用JDBC连接池中的连接。

第一，如要实现在Java 客户端程序中通过RMI 调用JDBC连接池中的连接，必须在datasources.xml中，为相应的数据库连接设定其通过远程连接需要的户认证标记。方法如下,首先设定将使用此连接的Apusic组或用户，如：your_group<组名>或your_user<用户名>；然后在datasources.xml中添加<remote-acl>标记及其元素。范例如下：

```
<remote-acl>
    <user>username</user>
</remote-acl>
```

或

```
<remote-acl>
    <group>groupname</group>
</remote-acl>
```

第二，在客户端Java 程序中，将相关的验证信息加入到环境参数中，即Apusic用户名及密码。范例如下：

```
import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import javax.rmi.*;

public class RMIJdbc
public static void main(String[] args){

Hashtable env=new Hashtable();

Context initCtx=null;

DataSource datasource=null;

try{

env.put(Context.INITIAL_CONTEXT_FACTORY,"com.apusic.naming.jndi.CNContextFactor

env.put(Context.PROVIDER_URL,"iiop://APUSIC_ADDRESS:6888");

//插入相关验证信息
```

```
env.put(Context.SECURITY_CREDENTIALS,"your_username" ) ;

env.put(Context.SECURITY_PRINCIPAL,"your_password");

initCtx=new InitialContext(env);

//通过RMI 取得数据库连接

datasource=(DataSource)initCtx.lookup("jdbc/sample")

Connection conn=datasource.getConnection();

}catch(Exception e){

e.printStackTrace();

}

...

}
```

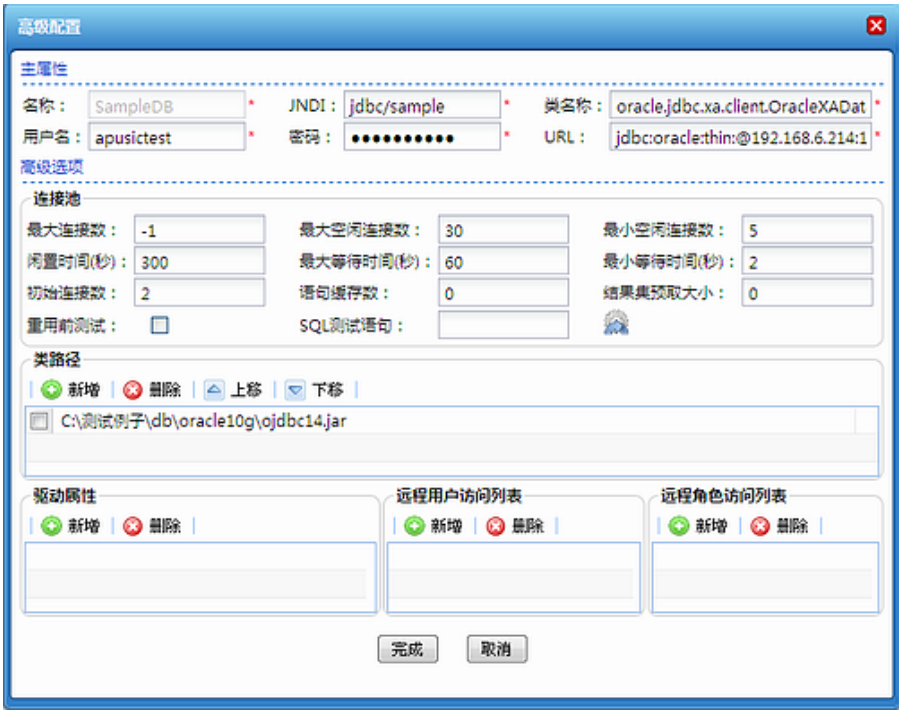
最后，编译客户端代码并执行。如不能连接，请检查是否可以通过servlet 或EJB 进行连接， 可参考上节"2.1服务器端应用"中相关内容。

16.5.1.2.3 范例

下面的例子是JAVA应用程序直接访问数据源的例子。数据库是Oracle，表是scott下的customer。

第一步：建立数据源。

在Apusic管理工具中，打开"驾驶舱"-->"数据源"-->"新增数据源",在"添加数据源"界面下连接池中填写如下信息：



如果需要提供可供远程访问的数据源，请参考"编辑datasources.xml文件"配置

第二步：JAVA应用程序代码：

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

public class JDBClient {
  private Connection con = null;
  private PreparedStatement prepStmt = null;
  /** dbName 对应第一步在创建数据源时的"基本"表格中"连接池JNDI名" */
  private String dbName = "jdbc/sample";

  public JDBClient() {}

  public void run() {
    try {
      Hashtable env = new Hashtable();
      // url 中的localhost可以替换为任意可连通的已安装Apusic应用服务器的主机名或地址
      String url = "iiop://localhost:6888";
      env.put(Context.INITIAL_CONTEXT_FACTORY,
```

```
"com.apusic.naming.jndi.CNContextFactory");
      env.put(Context.PROVIDER_URL, url);
      // 下面的用户必须是在Apusic中存在的，并且是在"远程访问控制用户列表"中定义的用户
      env.put(Context.SECURITY_PRINCIPAL, "scott");
      env.put(Context.SECURITY_CREDENTIALS, "tiger");
      InitialContext ic = new InitialContext(env);
      DataSource ds = (DataSource) ic.lookup(dbName);
      con = ds.getConnection();
      // customer 是Oracle自带的scott下的表
      String selectStatement = "select * from customer";
      prepStmt = con.prepareStatement(selectStatement);
      ResultSet rs = prepStmt.executeQuery();
      System.out.println("FIRSTNAME " + "LASTNAME");
      while (rs.next()) {
        System.out.println(rs.getString("FIRSTNAME") + " " +
rs.getString("LASTNAME"));
      }
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      cleanup(con, prepStmt);
    }
  }

  private void cleanup(Connection con, PreparedStatement ps) {
    try {
      if (ps != null) ps.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
    try {
      if (con != null) con.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    JDBClient _jdbctest = new JDBClient();
    _jdbctest.run();
```

```
    }
}
```

正确的执行结果为：

```
FIRSTNAME LASTNAME
sunny sunny Yuan
Judy Judy Li
John John Wan
Peter Peter Miao
```

16.5.2 消息服务开发

16.5.2.1 理解JMS

JAVA 消息服务(JMS)定义了Java 中访问消息中间件的接口。JMS只是接口，并没有给予实现，实现JMS接口的消息中间件称为JMS Provider，Apusic应用服务器实现了JMS接口，用户可以通过使用JMS接口，在Apusic中进行JMS编程。Apusic支持JMS1.1版本，JMS1.1是本文档书写时JMS API最新发布版。有关JMS1.1的特性，参见JMS1.1特性。

16.5.2.1.1 消息中间件

消息中间件提供企业数据的异步传输，通过消息中间件，一些原本互相孤立的业务组件可以组合成一个可靠的、灵活的系统。

消息中间件大致分为两类：Point-to-Point(PTP)和Publish-Subscribe(Pub/Sub),Apusic支持这两种模型。

PTP是点对点传输消息，建立在消息队列的基础上，每个客户端对应一个消息队列，客户端发送消息到对方的消息队列中，从自己的消息队列读取消息。

Pub/Sub是将消息定位到某个层次结构栏目的节点上，Pub/Sub通常是匿名的并能够动态发布消息，Pub/Sub必须保证某个节点的所有发布者(Publisher)发布的信息准确无误地发送到这个节点的所有消息订阅者(Subscriber)。

16.5.2.1.2 JMS接口描述

JMS 支持两种消息类型PTP 和Pub/Sub，分别称作：PTP Domain 和Pub/Sub Domain，这两种接口都继承统一的JMS Parent 接口，JMS 主要接口如下所示：

JMS Parent	PTPDomain	Pub/Sub Domain
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver,QueueBrowser	TopicSubscriber

以下是对这些接口的简单描述：

ConnectionFactory ： 连接工厂，JMS 用它创建连接

Connection ： JMS 客户端到JMS Provider 的连接

Destination ： 消息的目的地

Session： 一个发送或接收消息的线程

MessageProducer： 由Session 对象创建的用来发送消息的对象

MessageConsumer： 由Session 对象创建的用来接收消息的对象

16.5.2.1.3 JMS和其它企业级JAVA API的关系

JDBC： JMS 客户端可以使用JDBC 接口，可以将JDBC 和JMS包含在一个事务里。这种包含可以在EJB里，也可以直接调用JTA(Java Transaction API)接口实现。

JavaBeans： JavaBeans可以用JMS Session 发送接收消息。

EJB： EJB2.1 规范中定义了新的Message-Driven Beans 组件模型。

JTA(Java Transaction API)： JMS 客户端可以用JTA 启动事务。JMS Provider可以选择是否支持分布式事务。

JTS(Java Transaction Service)： JMS 可以和JTS一起组成一个分布式事务，如将发送接收消息和更新数据库包含在一个事务里。

JNDI：JMS客户端通过JNDI 调用JMS 中的对象。

16.5.2.1.4 JMS消息模型

JMS 消息由以下几部分组成：消息头，属性，消息体。

- 消息头(Header) - 消息头包含消息的识别信息和路由信息，消息头包含一些标准的属性如：JMSDestination,JMSMessageID 等。

消息头	由谁设置
JMSDestination	send 或 publish 方法
JMSDeliveryMode	send 或 publish 方法
JMSExpiration	send 或 publish 方法
JMSPriority	send 或 publish 方法
JMSMessageID	send 或 publish 方法
JMSTimestamp	send 或 publish 方法
JMSCorrelationID	客户
JMSReplyTo	客户
JMSType	客户

JMSRedelivered	JMS Provider

- 属性(Properties) - 除了消息头中定义好的标准属性外，JMS提供一种机制增加新属性到消息头中，这种新属性包含以下几种：
    - 应用需要用到的属性;
    - 消息头中原有的一些可选属性;
    - JMS Provider 需要用到的属性。

标准的JMS 消息头包含以下属性：

JMSDestination --消息发送的目的地。

JMSDeliveryMode --传递模式，有两种模式：PERSISTENT和NON_PERSISTENT，PERSISTENT表示该消息一定要被送到目的地，否则会导致应用错误。NON_PERSISTENT表示偶然丢失该消息是被允许的，这两种模式使开发者可以在消息传递的可靠性和吞吐量之间找到平衡点。

JMSMessageID 唯一识别每个消息的标识，由JMS Provider 产生。

JMSTimestamp 一个消息被提交给JMS Provider 到消息被发出的时间。

JMSCorrelationID 用来连接到另外一个消息，典型的应用是在回复消息中连接到原消息。

JMSReplyTo 提供本消息回复消息的目的地址。

JMSRedelivered如果一个客户端收到一个设置了JMSRedelivered属性的消息，则表示可能该客户端曾经在早些时候收到过该消息，但并没有签收(acknowledged)。

JMSType 消息类型的识别符。

JMSExpiration 消息过期时间，等于QueueSender 的send方法中的timeToLive值或TopicPublisher 的publish 方法中的timeToLive值加上发送时刻的GMT时间值。如果timeToLive值等于零，则JMSExpiration被设为零，表示该消息永不过期。如果发送后，在消息过期时间之后消息还没有被发送到目的地，则该消息被清除。

JMSPriority 消息优先级，从0-9 十个级别，0-4 是普通消息，5-9 是加急消息。JMS 不要求JMS Provider严格按照这十个优先级发送消息，但必须保证加急消息要先于普通消息到达。

- 消息体(Body) - JMS API定义了5种消息体格式，也叫消息类型，你可以使用不同形式发送接收数据并可以兼容现有的消息格式，下面描述这5种类型：

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
Message	没有消息体，只有消息头和属性

下例演示创建并发送一个TextMessage到一个队列：

```
TextMessage message = queueSession.createTextMessage();
message.setText(msg_text); // msg_text is a String
queueSender.send(message);
```

下例演示接收消息并转换为合适的消息类型：

```
Message m = queueReceiver.receive();
if (m instanceof TextMessage) {
  TextMessage message = (TextMessage) m;
  System.out.println("Reading message: " + message.getText());
} else {
  // Handle error
}
```

16.5.2.1.5 消息的同步异步接收

消息的同步接收

同步接收是指客户端主动去接收消息，JMS 客户端可以采用MessageConsumer的receive方法去接收下一个消息。

消息的异步接收

异步接收是指当消息到达时，主动通知客户端。JMS客户端可以通过注册一个实现MessageListener接口的对象到MessageConsumer，这样，每当消息到达时，JMS Provider 会调用MessageListener中的onMessage 方法。

16.5.2.1.6 PTP模型

PTP(Point-to-Point)模型是基于队列的，发送方发消息到队列，接收方从队列接收消息，队列的存在使得消息的异步传输成为可能。和邮件系统中的邮箱一样，队列可以包含各种消息，JMS Provider 提供工具管理队列的创建、删除。JMS PTP模型定义了客户端如何向队列发送消息，从队列接收消息，浏览队列中的消息。

下面描述JMS PTP 模型中的主要概念和对象：

名称	描述
Queue	由JMS Provider 管理，队列由队列名识别，客户端可以通过JNDI接口用队列名得到一个队列对象。
TemporaryQueue	由QueueConnection 创建，而且只能由创建它的QueueConnection 使用。
QueueConnectionFactory	客户端用QueueConnectionFactory 创建QueueConnection 对象。

QueueConnection	一个到JMS PTP provider 的连接，客户端可以用QueueConnection创建QueueSession来发送和接收消息。
QueueSession	提供一些方法创建QueueReceiver、QueueSender、QueueBrowser和TemporaryQueue。如果在 QueueSession关闭时，有一些消息已经被收到，但还没有被签收(acknowledged)，那么，当接收者下次连接到相同的队列时，这些消息还会被再次接收。
QueueReceiver	客户端用QueueReceiver接收队列中的消息，如果用户在QueueReceiver中设定了消息选择条件，那么不符合条件的消息会留在队列中，不会被接收到。
QueueSender	客户端用QueueSender 发送消息到队列。
QueueBrowser	客户端可以QueueBrowser 浏览队列中的消息，但不会收走消息。
QueueRequestor	JMS 提供QueueRequestor类简化消息的收发过程。QueueRequestor的构造函数有两个参数：QueueSession和queue，QueueRequestor 通过创建一个临时队列来完成最终的收发消息请求。
可靠性(Reliability)	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。

16.5.2.1.7 PUB/SUB模型

JMS Pub/Sub 模型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)。

主题可以被认为是消息的传输中介，发布者(publisher)发布消息到主题，订阅者(subscribe)从主题订阅消息。主题使得消息订阅者和消息发布者保持互相独立，不需要接触即可保证消息的传送。

下面描述JMS Pub/Sub 模型中的主要概念和对象：

名称	描述
订阅(subscription)	消息订阅分为非持久订阅(non-durable subscription)和持久订阅(durablesubscrip-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。持久订阅时，客户端向JMS注册一个识别自己身份的ID，当这个客户端处于离线时，JMS Provider 会为这个ID 保存所有发送到主题的消息，当客户再次连接到JMS Provider时，会根据自己的ID得到所有当自己处于离线时发送到主题的消息。
Topic	主题由JMS Provider 管理，主题由主题名识别，客户端可以通过JNDI接口用主题名得到一个主题对象。JMS没有给出主题的组织和层次结构的定义，由JMS Provider 自己定义。
TemporaryTopic	临时主题由TopicConnection创建，而且只能由创建它的TopicConnection使用。临时主题不能提供持久订阅功能。
TopicConnectionFactory	客户端用TopicConnectionFactory 创建TopicConnection 对象。
TopicConnection	TopicConnection 是一个到JMS Pub/Sub provider的连接，客户端可以用TopicConnection创建TopicSession来发布和订阅消息。
TopicSession	TopicSession提供一些方法创建TopicPublisher、TopicSubscriber、TemporaryTopic。它还提供unsubscribe方法取消消息的持久订阅。
TopicPublisher	客户端用TopicPublisher 发布消息到主题。

TopicSubscriber	客户端用TopicSubscriber接收发布到主题上的消息。可以在TopicSubscriber中设置消息过滤功能，这样，不符合要求的消息不会被接收。
Durable TopicSubscriber	如果一个客户端需要持久订阅消息，可以使用Durable TopicSubscriber，TopSession提供一个方法createDurableSubscriber创建Durable TopicSubscriber 对象。
恢复和重新派送(Recovery and Redelivery)	非持久订阅状态下，不能恢复或重新派送一个未签收的消息。只有持久订阅才能恢复或重新派送一个未签收的消息。
TopicRequestor	JMS 提供TopicRequestor类简化消息的收发过程。TopicRequestor的构造函数有两个参数：TopicSession和topic。TopicRequestor 通过创建一个临时主题来完成最终的发布和接收消息请求。
可靠性(Reliability)	当所有的消息必须被接收，则用持久订阅模式。当丢失消息能够被容忍，则用非持久订阅模

16.5.2.2 使用JMS

Apusic JMS 配置文件包括两部分：$APUSIC_HOME/config/apusic.conf和$APUSIC_HOME/config/jms.xml。

16.5.2.2.1 开发JMS CLIENT的步骤

广义上说，一个JMS 应用是几个JMS 客户端交换消息，开发JMS 客户端应用由以下几步构成：

1. 用JNDI 得到ConnectionFactory 对象；
2. 用JNDI 得到目标队列或主题对象，即Destination 对象；
3. 用ConnectionFactory 创建Connection 对象；
4. 用Connection 对象创建一个或多个JMS Session；
5. 用Session 和Destination 创建MessageProducer和MessageConsumer；
6. 通知Connection 开始传递消息。

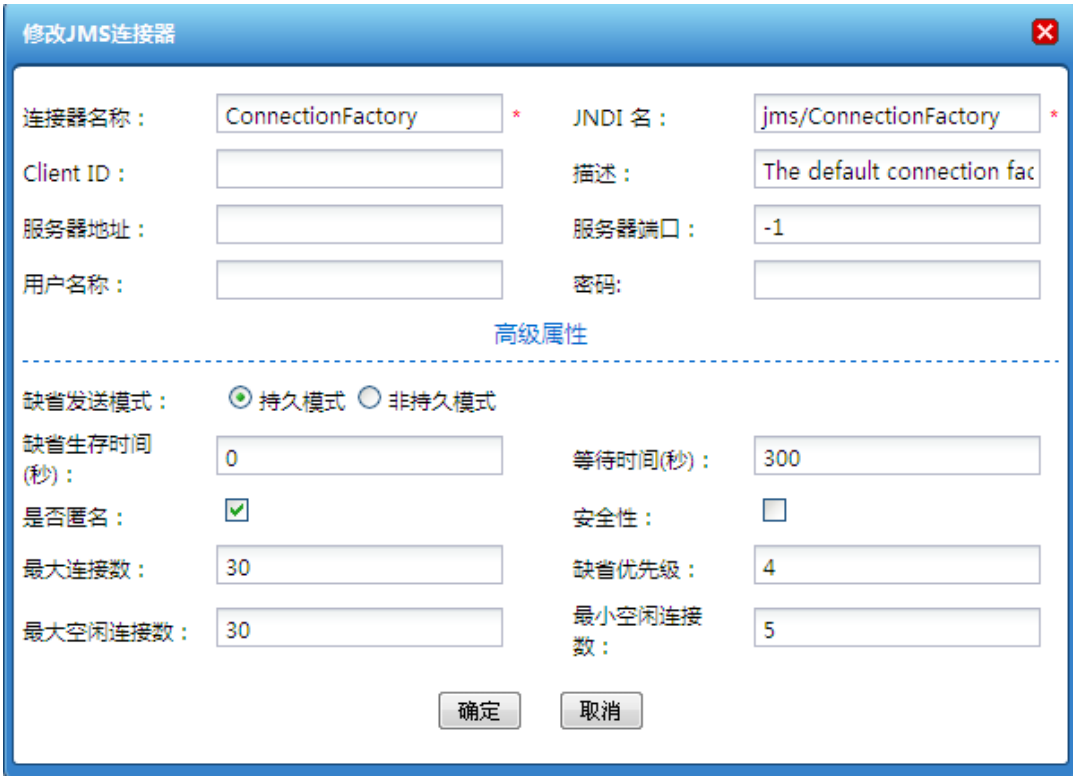
16.5.2.2.2 PTP模型应用

PTP 模型主要包含消息的发送和接收，下面我们分别举例说明：

发送消息

1. 启动Apusic应用服务器。
2. 打开管理工具：在浏览器中输入http://localhost:6888，进到Web管理，缺省管理员为admin，密码也是admin。
3. 配置队列连接创建器：在管理工具中点击“管理控制台”-->“消息服务”，点击“连接创建器”，如果存在JNDI名为“jms/QueueConnectionFactory”的队列连接创建器，则无须创建，否则点击“创建”，添加必要信息，“连接创建器名称”为“QueueConnectionFactory”，“队列连接创建器JNDI名”为“jms/QueueConnectionFactory”，“连接是否可以匿名访问”为“是”，其它项为缺省设置，点击“保存”。如下：

连接创建器



4. 配置队列：在管理工具中点击“管理控制台”-->“消息服务”，点击“队列（queue）”，如果存在JNDI名为“jms/QueueConnectionFactory”的队列，则无须创建，否则点击“创建”，添加必要信息，“队列名称”为“testQueue”，“队列JNDI名”为“testQueue”，其它项为缺省设置，点击“保存”。如下：

配置队列



5. 编写发送消息客户端代码：Send.java

```
import javax.jms.*;
import javax.naming.*;
import java.io.*;
import java.util.*;
```

```
import java.rmi.RemoteException;

/**
 * Title: JMS Description: JMS Test Copyright: Copyright (c) 2003 Company:
Apusic
 *
 * @author Michael
 *
 * @version 1.0
 */
public class Send {
  String queueName = "myQueue";
  QueueConnectionFactory queueConnectionFactory = null;
  Queue queue = null;
  QueueConnection queueConnection = null;
  QueueSession queueSession = null;
  QueueSender queueSender = null;
  TextMessage message = null;

  public static void main(String[] args) throws Exception {
    InitialContext ic = getInitialContext();
    Send sender = new Send();
    sender.init(ic);
    sender.sendMessage();
    sender.close();
  }

  public void init(InitialContext ctx) throws Exception {
    queueConnectionFactory = (QueueConnectionFactory)
ctx.lookup("jms/QueueConnectionFactory");
    queueConnection = queueConnectionFactory.createQueueConnection();
    queue = (Queue) ctx.lookup(queueName);
  }

  public void sendMessage() throws JMSException, RemoteException {
    queueSession = queueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
    queueSender = queueSession.createSender(queue);
    queueSender.setDeliveryMode(DeliveryMode.PERSISTENT);
    message = queueSession.createTextMessage();
```

```
    message.setText("The Message from myQueue");
    queueSender.send(message);
  }

  public void close() throws JMSException {
    if (queueConnection != null) queueConnection.close();
  }

  private static InitialContext getInitialContext() throws NamingException {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.apusic.naming.jndi.CNContextFactory");
    env.put(Context.PROVIDER_URL, "iiop://localhost:6888");
    return (new InitialContext(env));
  }
}
```

接收消息

如果管理工具中没有JNDI名为"jms/QueueConnectionFactory"的队列连接创建器和"myQueue"的队列，则依照上面一到四步进行设置，否则可直接编写接收消息客户端代码：Receive.java

```
import javax.jms.*;
import javax.naming.*;
import java.io.*;
import java.util.*;
import java.rmi.RemoteException;

/**
 *
 * Title: JMS Description: JMS Test Copyright: Copyright (c) 2003 Company:
Apusic
 *
 * @author Michael
 *
 * @version 1.0
 */

public class Receive {
```

```
    String queueName = "myQueue";

    QueueConnectionFactory queueConnectionFactory = null;

    Queue queue = null;

    QueueConnection queueConnection = null;

    QueueSession queueSession = null;

    QueueReceiver queueReceiver = null;

    TextMessage message = null;

    public static void main(String[] args) throws Exception {

        InitialContext ic = getInitialContext();

        Receive receiver = new Receive();

        receiver.init(ic);

        receiver.TBreceiveMessage();// 你可以在此处调用YBreceiveMessage
        receiver.close();

    }

    public void init(InitialContext ctx) throws Exception {

        queueConnectionFactory = (QueueConnectionFactory)
ctx.lookup("jms/QueueConnectionFactory");

        queueConnection = queueConnectionFactory.createQueueConnection();

        queue = (Queue) ctx.lookup(queueName);

    }

    public void TBreceiveMessage() throws NamingException, JMSException,
RemoteException {
```

```
        queueSession = queueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);

        queueReceiver = queueSession.createReceiver(queue);

        queueConnection.start();

        for (;;) {

            message = (TextMessage) queueReceiver.receive();

            System.out.println("Reading message: " + message.getText());

            if (message.getText().equals("quit"))

                break;

        }

    }

public void YBreceiveMessage() throws NamingException,
JMSException,RemoteException,IOException{

queueSession =
queueConnection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

queueReceiver = queueSession.createReceiver(queue);

//register my textListener which comes from MessageListener
TextMessageListener textListener = new TextMessageListener();

queueReceiver.setMessageListener(textListener);

queueConnection.start();

System.out.println("To end program, enter Q or q, then ");

InputStreamReader reader = new InputStreamReader(System.in);
```

```
char answer = '\\0';

while (!((answer == 'q') || (answer == 'Q')))

answer = (char)reader.read();

}

    public void close() throws JMSException {

        if (queueReceiver != null)

            queueReceiver.close();

        if (queueSession != null)

            queueSession.close();

        if (queueConnection != null)

            queueConnection.close();

    }

    private static InitialContext getInitialContext() throws NamingException
{

        Hashtable env = new Hashtable();

        env.put(Context.INITIAL_CONTEXT_FACTORY,

                "com.apusic.naming.jndi.CNContextFactory");

        env.put(Context.PROVIDER_URL, "iiop://localhost:6888");

        return (new InitialContext(env));

    }
```

```
}
```

其中异步接收时使用的TextMessageListener代码如下：

```
import javax.jms.MessageListener;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.JMSException;

/**
 * Title: JMS Description: JMS Test Copyright: Copyright (c) 2003 Company:
Apusic
 *
 * @author Michael
 *
 * @version 1.0
 */
public class TextMessageListener implements MessageListener {
  public TextMessageListener() {}

  public void onMessage(Message m) {
    TextMessage msg = (TextMessage) m;
    try {
      System.out.println("Async reading message: " + msg.getText() + "
(priority=" + msg.getJMSPriority() + ")");
    } catch (JMSException e) {
      System.out.println("Exception in onMessage(): " + e.toString());
    }
  }
}
```

最后，运行程序：在Windows下打开两个DOS窗口，首先运行Receive 然后运行Send。

如果是异步接收消息，在运行Receive 的窗口中可以看到输出：

```
To end program, enter Q or q, then
Async reading message: The Second Message from testQueue (priority=4)
```

如果是同步接收消息，在运行Receive 的窗口中可以看到输出：

```
Reading message: The Message from testQueue
```

16.5.2.2.3 PUB/SUB模型应用

Pub/Sub 模型主要包含消息的发布和订阅，下面我们分别举例说明：

发布消息

如果管理工具中没有JNDI名为"jms/TopicConnectionFactory"的连接创建器和"myTopic"的队列，则依照上面一到四步进行设置，否则可直接编写发布消息客户端代码：Published.java

```
import javax.jms.*;
import javax.naming.*;
import java.io.*;
import java.util.*;
import java.rmi.RemoteException;

/**
 * Title: JMS Description: JMS Test Copyright: Copyright (c) 2003 Company:
Apusic
 *
 * @author Michael
 *
 * @version 1.0
 */
public class Published {
  String topicName = "myTopic";
  TopicConnectionFactory topicConnectionFactory = null;
  Topic topic = null;
  TopicConnection topicConnection = null;
  TopicSession topicSession = null;
  TopicPublisher topicPublisher = null;
  String msgText = null;
  TextMessage message = null;

  public static void main(String[] args) throws Exception {
    InitialContext ic = getInitialContext();
    Published publisher = new Published();
    publisher.init(ic);
    publisher.publish();
    publisher.close();
```

```
  }

  public void init(InitialContext ctx) throws Exception {
    topicConnectionFactory = (TopicConnectionFactory)
ctx.lookup("jms/TopicConnectionFactory");
    topicConnection = topicConnectionFactory.createTopicConnection();
    topic = (Topic) ctx.lookup(topicName);
  }

  public void publish() throws NamingException, JMSException, RemoteException
{
    topicSession = topicConnection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);
    topicPublisher = topicSession.createPublisher(topic);
    message = topicSession.createTextMessage();
    msgText = "This is the published message";
    message.setText(msgText);
    topicPublisher.publish(message);
  }

  public void close() throws JMSException {
    if (topicConnection != null) topicConnection.close();
  }

  private static InitialContext getInitialContext() throws NamingException {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.apusic.naming.jndi.CNContextFactory");
    env.put(Context.PROVIDER_URL, "iiop://localhost:6888");
    return (new InitialContext(env));
  }
}
```

订阅消息

```
import javax.jms.*;
import javax.naming.*;
import java.io.*;
import java.util.*;
import java.rmi.RemoteException;
```

```
/**
 * Title: JMS Description: JMS Test Copyright: Copyright (c) 2003 Company:
Apusic
 *
 * @author Michael
 *
 * @version 1.0
 */
public class Subscriber {
  String topicName = "myTopic";
  TopicConnectionFactory topicConnectionFactory = null;
  TopicConnection topicConnection = null;
  Topic topic = null;
  TopicSession topicSession = null;
  TopicSubscriber topicSubscriber = null;
  TextMessage message = null;
  String id = "durable";

  public static void main(String[] args) throws Exception {
    InitialContext ic = getInitialContext();
    Subscriber subscriber = new Subscriber();
    subscriber.init(ic);
    subscriber.subscribe();
    subscriber.close();
  }

  public void init(InitialContext ctx) throws Exception {
    topicConnectionFactory = (TopicConnectionFactory)
ctx.lookup("jms/TopicConnectionFactory");
    topicConnection = topicConnectionFactory.createTopicConnection();
    topicConnection.setClientID(id);
    topic = (Topic) ctx.lookup(topicName);
  }

  public void subscribe() throws NamingException, JMSException,
RemoteException {
    topicSession = topicConnection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);
    topicSubscriber = topicSession.createDurableSubscriber(topic, id);
```

```
    topicConnection.start();
    message = (TextMessage) topicSubscriber.receive();
    System.out.println("SUBSCRIBER THREAD: Reading message: " +
message.getText());
  }

  public void close() throws JMSException {
    if (topicConnection != null) topicConnection.close();
  }

  private static InitialContext getInitialContext() throws NamingException {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.apusic.naming.jndi.CNContextFactory");
    env.put(Context.PROVIDER_URL, "iiop://localhost:6888");
    return (new InitialContext(env));
  }
}
```

最后，运行程序：在Windows下打开两个DOS窗口，首先运行Subscriber，然后运行Published，在运行Subscriber的窗口中可以看到输出：

```
SUBSCRIBER THREAD: Reading message: This is the published message
```

16.5.2.2.4 权限设置

利用Apusic JMS配置文件（jms.xml），你可以对队列和主题设定访问权限，可以详细规定Apusic中的用户和组，可以对队列和主题进行何种操作。

对于队列，可以设定发送(send)、接收(receive)和浏览(browse)权限；对于主题，可以设定发布(publish)、订阅(subscribe)、持久订阅(subscribe-durable)和取消持久订阅(unsubscribe)权限。

客户端用QueueConnectionFactory 的createQueueConnection(java.lang.String userName,java.lang.String password)方法创建队列连接时输入用户身份,或者用createTopicConnection(java.lang.String userName, java.lang.String password)方法创建主题连接时输入用户身份。

下面的例子授权用户larry 可以对队列testQueue 进行接收和浏览：

首先创建testQueue队列，如果Apusic不存在testQueue队列，则通过管理工具创建，方法同上。

建立安全角色(security-role)rbt，将该角色映射到用户larry，使larry拥有该角色包含的所有权限。如果Apuisc应用服务器不存在larry用户，在管理工具中"用户管理"-->"用户/组"中创建larry用户。然后可参考apusic.jar中jms.dtd的定义在jms.xml中加入如下片段：

```
<security-role>
    <role-name>rbt</role-name>
    <principal>larry</principal>
</security-role>
```

设置安全角色rbt的权限：

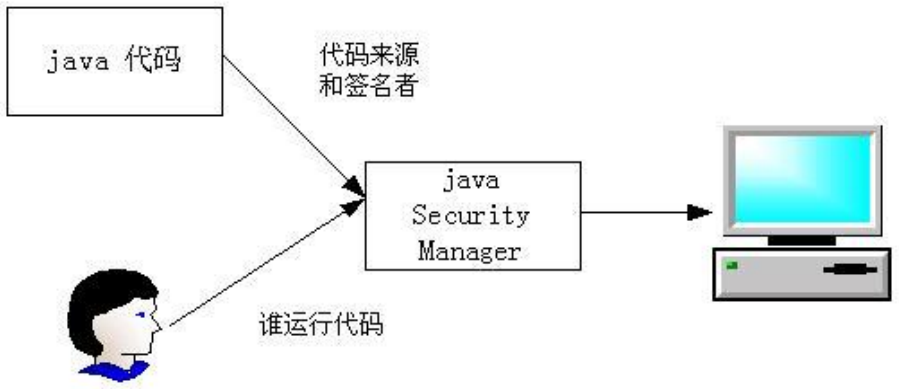
```
<destination-permission>
    <role-name>rbt</role-name>
    <destination-method>
        <queue-name>testQueue</queue-name>
        <method-name>receive</method-name>
        <method-name>browse</method-name>
    </destination-method>
</destination-permission>
```

16.5.3 使用JAAS用户认证

16.5.3.1 JAAS简介

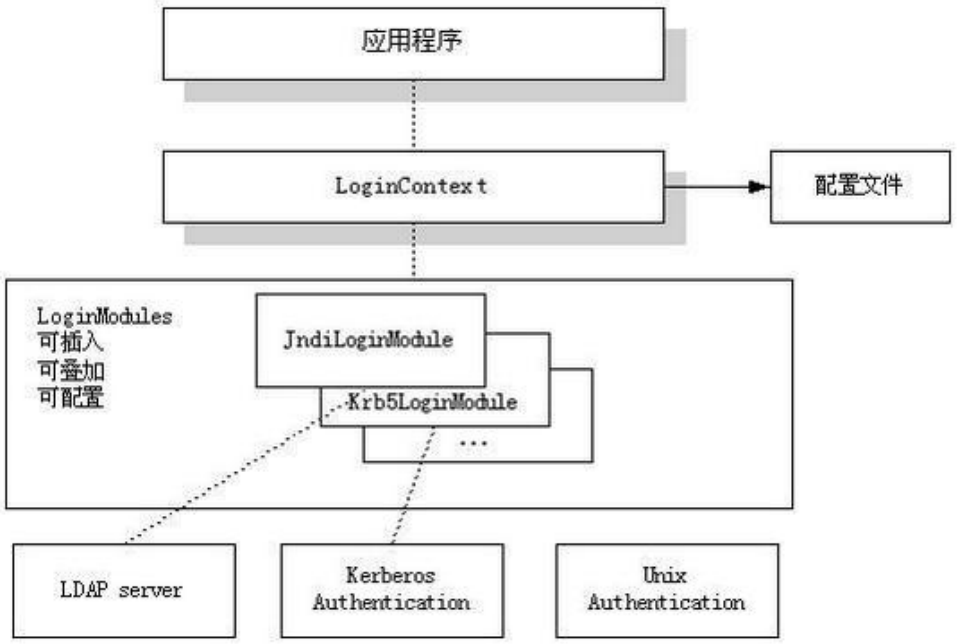
Java安全框架最初集中在保护用户运行潜在的不可信任代码，是基于代码的来源（URL）和谁创建的代码（certificate）来给移动代码进行授权。Java 2 SDK 1.3引入了JAAS（Java Authentication and Authorization Service）,增加了基于用户的访问控制能力，即根据谁在运行代码来进行授权。JAAS已经整合进了Java 2 SDK 1.4，作为标准的用户认证与授权模型。

JAAS用户认证框架



JAAS认证被实现为可插入的方式，允许应用程序同底层的具体认证技术保持独立，新增或者更新认证方法并不需要更改应用程序本身。应用程序通过实例化LoginContext对象开始认证过程，引用配置文件中的具体认证方法，即LoginModule对象，来执行认证。

JAAS：可插入式认证



一旦执行代码的用户通过了认证，JAAS授权组件将和核心Java访问控制模型一起工作，来保护对敏感资源的访问。从J2SDK 1.4开始，访问控制不仅基于代码的来源和签名者（CodeSource），而且还要检查谁在运行代码。执行代码的用户被表现为Subject对象，如果LoginModule认证成功，Subject对象被更新为相应的Principals和credentials。

16.5.3.1.1 一个简单的例子

本节通过一个简单的例子介绍JAAS开发的基本步骤。本节中的范例位于Apusic应用服务器安装目录中的docs/samples/jaas /simple目录。有关范例的内容、编译、部署与运行，可参考docs/samples/jaas/simple目录下的readme.txt文件。

范例程序的代码分为两部分，一部分为主程序，执行用户认证过程，源程序如下：

```
package samples;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import com.sun.security.auth.callback.TextCallbackHandler;

public class CountFiles {

    static LoginContext lc = null;

public static void main(String[] args) {

//使用配置文件中名字为"CountFiles"的条目

try {
```

```
lc = new LoginContext("CountFiles",

new TextCallbackHandler());

} catch (LoginException le) {

le.printStackTrace();

System.exit(-1);

}

try {

lc.login();

//如果没有异常抛出，则表示认证成功

} catch (Exception e) {

System.out.println("Login failed: " + e);

System.exit(-1);

}

//以认证用户的身份执行代码

Object o = Subject.doAs(lc.getSubject(), new CountFilesAction());

System.out.println("User " + lc.getSubject( ) + " found " + o + "
files.");

System.exit(0);

}

}
```

可以看出，主程序包含了三个重要的步骤：首先构造一个LoginContext对象，然后使用这个对象进行登录，最后，把用户作为doAs方法一个参数。

另一部分表示用户想要执行的具体操作，源程序如下：

```
package samples;

import java.io.File;
import java.security.PrivilegedAction;

class CountFilesAction implements PrivilegedAction {
  public Object run() {
    File f = new File(".");
    File[] files = f.listFiles();
    return new Integer(files.length);
  }
}
```

16.5.3.1.2 JAAS核心类和接口

JAAS相关的核心类和接口分为三类，公共、认证和授权。

- 公共类：Subject,，Principal，Credential
- 认证类和接口：LoginContext，LoginModule，CallbackHandler，Callback
- 授权类 ： Policy，AuthPermission，PrivateCredentialPermission 详细的描述请参考《》。

16.5.3.1.3 配置LOGINMODULES

JAAS认证被实现为一种可插入的方式，系统管理员可以通过配置文件为每一个应用程序配置LoginModuls来决定应用程序使用的认证技术。配置信息可以保存在文件或数据库中，通过javax.security.auth.login.Configuration对象进行读取。javax.security.auth.login.Configuration为抽象类，JDK提供了可实例化的子类com.sun.security.auth.login.ConfigFile，从文件中读取配置信息。配置文件中包含一个或多个条目，每一个条目指明了特定应用程序使用的认证方法。条目的结构如下：

```
<name used by application to refer to this entry> {
<LoginModule> <flag> <LoginModule options>;
<optional additional LoginModules, flags and options>;

};
```

可以看出，每一个条目由名字和一个或多个LoginModule组成。范例程序使用的配置文件login.conf内容如下：

```
CountFiles {

com.apusic.security.auth.login.ClientPasswordLoginModule required;

};
```

详细的描述信息可以参考Configuration。

16.5.3.1.4 编写POLICY文件

JAAS授权扩展了现有的Java安全体系结构，在给代码授权时可以包括一个多个Principal域，指出Principal代表的用户执行特定的代码时，具有分配的权限。因此，授权声明的基本形式为：

```
grant <signer(s) field>, <codeBase URL>
<Principal field(s)> {

permission perm_class_name "target_name", "action";

...

permission perm_class_name "target_name", "action";

};
```

缺省的策略文件实现和策略文件语法请参考《Default Policy Implementation and Policy File Syntax》。范例程序使用的策略文件policy.jaas内容如下：

```
grant codeBase "file:./build" {

permission java.security.AllPermission;

};

grant codeBase "file:/${apusic.home}/lib/apusic.jar" {

permission java.security.AllPermission;

};

grant codeBase "file:./build/actions" Principal
com.apusic.security.PrincipalImpl "admin" {
```

```
permission java.io.FilePermission "<<ALL FILES>>", "read";

};
```

可以看出，给主程序和apusic.jar授予了所有权限；当执行具体操作的用户为“admin”时，授予了读取所有文件的权限。

16.5.3.1.5 运行范例程序

范例程序提供了ant的build.xml脚本，请用户自己下载并安装ant。运行范例程序的步骤为：

1. 首先启动Apusic应用服务器， 范例程序将登录服务器。
2. 编译、运行程序。在simple目录下执行ant命令，会编译源程序CountFiles.java到build目录下，编译源程序CountFilesAction.java到build/actions目录下。然后会自动运行程序，相当于在命令行敲入下面的java命令：

```
java -classpath %APUSIC_HOME%/lib/apusic.jar;./build;./build/actions \
-Djava.security.manager \
-Djava.security.policy==policy.jaas \
-Djava.security.auth.login.config==login.conf \
-Dapusic.home=%APUSIC_HOME% samples.CountFiles
```

根据提示输入服务器，用户名和口令。如果用“admin”登录，程序将正常运行结束，若使用其他用户名登录，将抛出访问控制异常。

16.5.3.2 JAAS和Apusic

Apusic应用服务器提供了使用JAAS接口进行用户认证的方法，而访问控制授权则由Apusic自己的安全管理器进行管理。这样Apusic提供了两种用户认证方式，即传统的JNDI认证和新型的JAAS认证，对于Web应用则提供标准的HTTP认证。

Apusic提供的LoginModule有：

- com.apusic.security.auth.login.ClientPasswordLoginModule：标准的用户口令认证。
- com.apusic.security.auth.login.ClientKrb5LoginModule：Kerberos支持的用户认证。
- com.apusic.security.auth.login.ServerLoginModule：服务器端LoginModule，把当前线程关联的安全上下文中的principal和credentials直接存放在Subject中。

一般的，开发者使用ClientPasswordLoginModule和ClientKrb5LoginModule从客户端登录服务器。使用JAAS可以同时使用多个身份登录服务器，也可以同时登录到多个服务器，然后通过Subject.doAs方法以不同的身份调用需要授权的服务器操作。当使用JNDI认证时也可以同时等录到多个服务器，但每个服务器上只能有一个活动用户身份。

	注意
	JAAS认证与JNDI认证不能同时使用，当使用JAAS认证之后，在Subject.doAs块中创建InitialContext时不能指定Context.PROVIDER_URL, Context.SECURITY_PRINCIPAL, Context.SECURITY_CREDENTIALS属性，即使使用这些属性创建

> 了InitialContext也仍然使用JAAS认证的用户身份进行调用。

Apusic还提供了CallbackHandler，开发者可以根据需要进行选择：

- com.apusic.security.auth.callback.TextCallbackHandler：命令行方式的登录提示
- com.apusic.security.auth.callback.DialogCallbackHandler：Swing对话框窗口方式的登录提示
- com.apusic.security.auth.callback.DefaultCallbackHandler：在程序中通过构造函数传入登录信息

当应用运行在客户端容器中时，JAAS认证由客户端容器自动完成，开发者可以在application-client.xml中指定callback-handler。如果未指定callback-handler，容器会根据客户端的环境选择TextCallbackHandler或DialogCallbackHandler。

16.5.3.3 Apusic JAAS开发

一般，开发者使用JAAS认证从客户端登录到服务器。在应用客户端中使用JAAS认证，需要把执行的操作(例如调用服务器上部署的EJB）包装在PrivilegedAction中。由于Apusic只把JAAS作为认证用户使用，所以只需要配置LoginModules。策略文件（Policy）作为访问控制授权语义，在这里不需要使用。本节通过一个范例程序的开发，介绍使用Apusic JAAS的基本步骤。

本节中的范例位于Apusic应用服务器安装目录中的docs/samples/jaas/client目录。有关范例的内容、编译、部署与运行，可参考docs/samples/jaas/client目录下的readme.txt文件。

1. 开发客户端主程序，包含JAAS认证过程：

```
package samples;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import com.apusic.security.auth.callback.DialogCallbackHandler;

public class Client {
  static LoginContext lc = null;

  public static void main(String[] args) {
    // 使用配置文件中名字为"client"的条目
    try {
      lc = new LoginContext("client", new DialogCallbackHandler());
    } catch (LoginException le) {
      le.printStackTrace();
      System.exit(-1);
    }
    try {
      lc.login();
      // 如果没有异常抛出，则表示认证成功
```

```
    } catch (Exception e) {
      System.out.println("Login failed: " + e);
      System.exit(-1);
    }
    String name;
    if (args != null && args.length > 0) {
      name = args[0];
    } else {
      name = "apusic";
    }
    // 以认证用户的身份执行代码
    Object o = Subject.doAs(lc.getSubject(), new SampleAction(name));
    System.out.println(o);
  }
}
```

2. 客户执行的操作包装在PrivilegedAction子类的run方法中。在这个例子中客户端将调用EJB。

```
package samples;

import java.security.PrivilegedAction;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import samples.ejb.Hello;
import samples.ejb.HelloHome;

public class SampleAction implements PrivilegedAction {
  private String name;

  public SampleAction(String name) {
    this.name = name;
  }

  public Object run() {
    Object obj = null;
    try {
      Context ic = new InitialContext();
      Object objref = ic.lookup("ejb/HelloJAAS");
      HelloHome home = (HelloHome) PortableRemoteObject.narrow(objref,
```

```
HelloHome.class);
      Hello hello = home.create();
      obj = hello.sayHello(name);
    } catch (Exception e) {
      e.printStackTrace();
    }
    return obj;
  }
}
```

3. 开发EJB模块。本例中的EJB只实现了一个商业方法sayHello，

```
public java.lang.String sayHello(java.lang.String name) {
  return "hello, " + name;
}
```

在部署文件ejb-jar.xml中定义了两个角色：employee和manager，并且定义了方法访问许可，即只允许角色manager访问商业方法sayHello：

```
...
<security-role>
    <role-name>employee</role-name>
</security-role>
<security-role>
    <role-name>manager</role-name>
</security-role>
<method-permission>
    <role-name>manager</role-name>
    <method>
        <ejb-name>Hello</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>sayHello</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </method>
</method-permission>

...
```

4. 编译、打包和部署EJB模块。具体过程可以参考 "打包和部署EJB模块"。在apusic的部署描述文件apusic-application.xml中将ejb-jar.xml中定义的角色映射为服务器用户：

```
<security-role>
    <role-name>manager</role-name>
    <principal>admin</principal>
</security-role>
```

```
<security-role>
    <role-name>employee</role-name>
    <principal>j2ee</principal>
</security-role>
```

最后使用ejbgen工具生成客户端运行需要的类库appclient.jar。

5. 配置LoginModules。客户端使用JAAS认证需要读取配置文件来配置LoginModules，本范例的配置文件login.conf内容为：

```
client {

com.apusic.security.auth.login.ClientPasswordLoginModule required;

};
```

在运行客户端时通过 `-Djava.security.auth.login.config==login.conf` 将配置文件传入客户端主程序。如果Apusic配置了Kerberos支持，可以使用ClientKrb5LoginModule

```
client {

com.apusic.security.auth.login.ClientKrb5LoginModule required
url="http://localhost:6888";

};
```

	注意
	如果程序运行在客户端容器中，配置文件中LoginModule的名字必须为"client"。客户端容器会自动使用名字为"client"的LoginModule进行JAAS认证。如果使用作为登录配置文件。

6. 运行客户端。范例程序提供了ant脚本build.xml，可以自动编译、打包、部署EJB模块和运行客户端程序。运行客户端等价的命令行命令为：

```
java -classpath %apusic_home%/lib/apusic.jar;./build/client;appclient.jar \
-Djava.security.auth.login.config==login.conf \
samples.Client apusic
```

当客户端运行时，会弹出windows对话框，提示输入服务器、用户名和口令。如果使用"admin"登录，客户端将正常调用EJB的sayHello方法。如果使用其他用户登录，客户端将抛出访问控制异常：java.rmi.AccessException: 用户 'xxx' 没有权限访问EJB的业务方法。

16.5.4 使用Apusic事务服务

16.5.4.1 事务服务简介

事务的目的就是为了确保数据的完整性，一个事务通常被定义为一个包含多步操作的不可分割的单元。事务具有下列四个特征：

- 原子性（Atomicity）：事务能够以两种方式结束：提交或回滚。当事务被提交后，在这个事务边界内对数据的更改会被保存。当事务回滚，所有对数据的更改都要被恢复。
- 一致性（Consistency）：如果事务失败，数据的完整性将被保护。
- 隔离性（Isolation）：并发执行的事务都应该表现为独立执行。
- 持久性（Durability）：事务提交后的改变应该是持久的。

在J2EE平台中，可管理的事务资源包括三种，数据库连接池、消息系统连接和符合J2EE™连接器架构（J2EE Connector Architecture，JCA）的资源。Apusic应用服务器中的事务管理器提供对以上三种资源的事务管理。

J2EE通过两个规范支持分布式事务：

- Java Transaction API (JTA)
- Java Transaction Service (JTS)

JTA是上层的、中立于协议的应用程序接口，允许应用和应用服务器访问事务。JTS规定事务管理的实现必须支持JTA，并且实现OMG对象事务服务（Object Transaction Service，OTS）规范的Java映射。JTS使用IIOP协议传播事务。Apusic应用服务器的事务管理器完全支持JTS和JTA，EJB容器使用JTA和JTS进行交互。

16.5.4.2 配置Apusic事务服务

请参考。

16.5.4.3 分布式事务恢复

16.5.4.3.1 需配置的参数

- EnableLog:

打开事务日志。只有打开事务日志，才会启用分布式事务恢复功能。默认为False（关闭事务日志）。

- RetryTimeout

发生通讯故障时重试的时间，默认600秒，超出该时间将不再重试，而是通知用户因通讯故障而导致出错。0 表示会永远重试。

- RetryInterval

发生通讯故障时重试的时间间隔。默认60秒。

- HeuristicDecision

作为事务从属Coordinator的应用服务器如果在RetryTimeout指定的时间内无法从其上级Coordinator处获得事务完成方向的指示，会自行决定应该提交还是回滚。取值必须为"commit"或"rollback"两者之一，默认为"rollback"。若不想做出启发式决定，而是一直等待直到条件成熟时再从上级Coordinator得到指示，请记得把RetryTimeout设为0，此时会一直尝试，并且应用服务器启动过程暂停，不对外提供服务，直到所有事务恢复完成为止。

- TxServerID

事务管理器的ID，用于标识事务管理器的一个字符串，默认无需设置。每个应用服务器的实例包含一个事务管理器，在事务管理器把XAs资源纳入到事务中进行管理时，会给资源赋予一个事务分支ID，TxServerID的信息会被加入到这个事务分支ID中。在一个事务包含超过一个事务管理器时（即跨应用服务器的事务），如果不同应用服务器配置了相同参数的数据源，在事务恢复时，必须区分哪个XA资源是归属哪个事务管理器所管理，此时必须为不同的事务管理器配置不同的TxServerID，否则，会出现事务管理器与XA资源的失配，导致事务恢复过程出错。

16.5.4.3.2 使用方法

Apuisc应用服务器的分布式事务恢复功能默认关闭，若要开启该功能，需要配置apusic.conf中的TransactionService服务EnableLog参数为True。 其他参数酌情配置，请参考配置事务服务。

在应用服务器重启时，如果存在需要恢复的事务，会自动进行事务的恢复，有关事务恢复的信息会在Apusic的日志中体现，会告知共有多少个事务需要恢复，以及经过恢复后，有多少个事务已提交，有多少个事务已回滚，有多少个事务尚在存疑状态，有多少个事务恢复失败（等待条件成熟时下次重启应用服务器再次执行恢复）。

应用服务器关闭后，不论是正常关闭还是异常终止，如果有需要恢复的事务，logs/tx目录会留下事务日志文件，事务日志文件格式为二进制，只用于重启时的事务恢复，如果该目录为空，表示没有需要恢复的事务。

在重启恢复事务的过程中如果有事务恢复失败，可等条件成熟时再次重启进行恢复。若重启恢复事务过程中应用服务器再次崩溃，可再次重启应用服务器继续完成恢复过程。若应用服务器所在的机器无法工作，可以把logs/tx目录中的事务日志文件拷贝到其他机器上的相同配置的Apusic应用服务器的同名目录中，启动该应用服务器完成事务的恢复。

16.5.4.4 使用事务服务

Apusic应用服务器的事务服务支持在EJB和客户端中使用事务。

16.5.4.4.1 在EJB中使用事务服务

Apusic应用服务器的EJB容器支持两种类型的事务管理：

- Bean管理事务（程序型的事务划分）
- 容器管理事务（声明型的事务划分）

请参考。

16.5.4.4.2 在客户端中使用事务服务

客户端通过JNDI查找（lookup）java:comp/UserTransaction获得服务器的事务服务，返回的对象实现了javax.transaction.UserTransaction接口。客户端通过调用UserTransaction的begin、commit和rollback方法管理事务。使用JNDI查找UserTransaction接口的范例代码如下：

```
InitialContext ic = new InitialContext();
String txName = "java:comp/UserTransaction";
UserTransaction tx = (javax.transaction.UserTransaction) ic.lookup(txName);
tx.begin();
// transacted commands, such as JDBC calls
tx.commit();
```

17 部分VII 应用程序部署指南

17.1 打包和部署Java EE应用

17.1.1 摘要

根据Java EE5 规范，Java EE应用包含Web模块与EJB模块等模块，应用在部署到实际生产环境之前，必须对各个模块进行打包并装配。

本节介绍了Web模块、EJB模块和Java EE应用的结构和如何打包部署到金蝶Apusic应用服务器上。

17.1.2 打包和部署简介

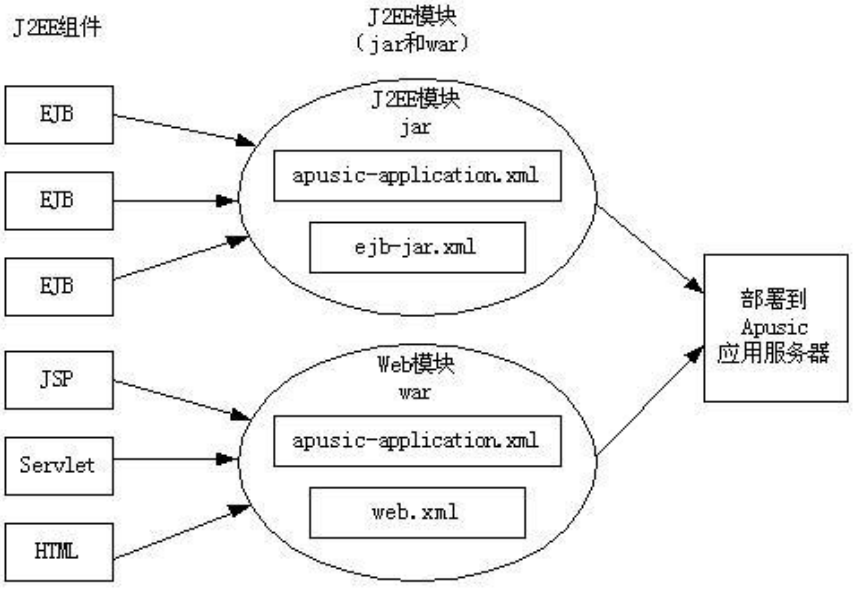
应用打包是这样一个过程：将应用中的各个分离组件打包到一个单元中，使它可以部署到符合Java EE规范的应用服务器上。包（package）可以是单独的模块，如Web模块war或EJB模块jar，也可以是完整的Java EE应用。

Java EE模块是一个或多个属于同一种容器类型（如Web容器，EJB容器）的Java EE组件的集合，并带有这种容器的部署描述文件。其中一种为Java EE标准的部署描述文件，另一种是金蝶Apusic应用服务器专有的部署描述文件。Java EE模块类型有：

- Web应用档案文件（WAR）：WAR文件由下列项目组成：servlets、JSP、JSP标记库、辅助类、静态页面、Java bean和部署描述文件（web.xml和可选的apusic-application.xml）。
- EJB JAR文件：EJB JAR文件是标准的EJB打包格式。JAR文件中包含home接口、远程接口、本地接口、用户的实现类、辅助类和部署描述文件（ejb-jar.xml，单独部署EJB JAR时还要包含apusic-application.xml）。
- 应用客户端JAR文件：应用客户端支持标准的Java EE客户端规范，它的部署描述文件是application-client.xml。

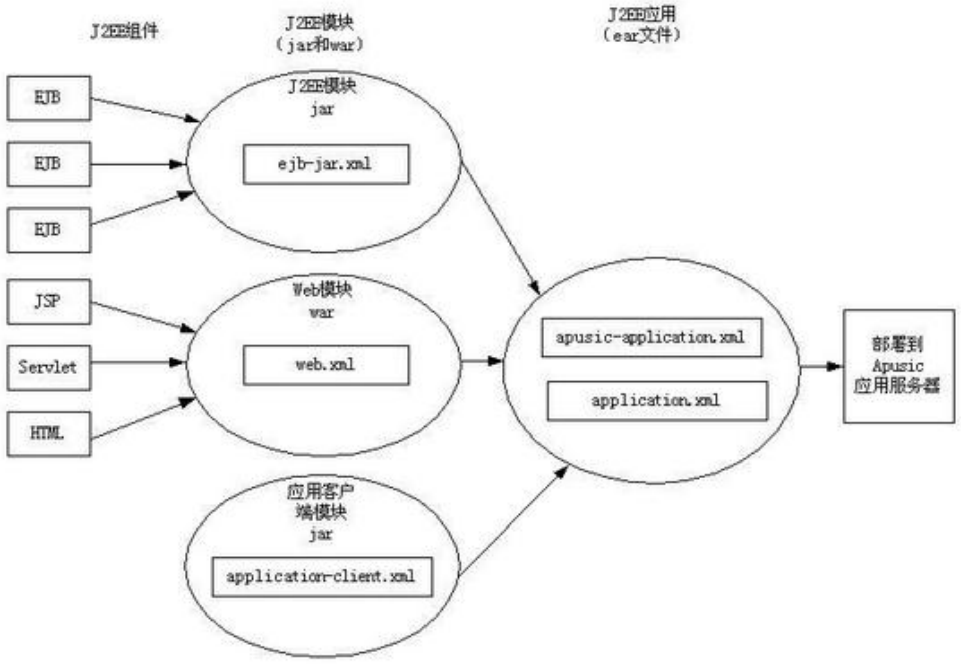
部署描述文件中的信息为声明式的，因此不需要更改源文件就能够更改这些信息。EJB JAR和Web模块可以单独部署，这时，它们应该包含apusic-application.xml。

EJB模块和Web模块单独部署



Java EE应用是一个或多个Java EE模块的逻辑集合，并包含部署描述文件。下图描述了组件是如何打包在模块中，然后再打包成Java EE应用EAR文件。

Java EE应用打包和部署



Java EE应用由一个或多个模块组成，使用Java应用档案文件格式把所有模块打包成扩展名为ear的文件，然后部署到金蝶Apusic应用服务器上。

	注意
	Web应用WAR作为模块单独部署时，可能需要包含金蝶Apusic应用服务器的配置文件：apusic-application.xml，放在目录META-INF下。EJB JAR作为模块单独部署时，必须包含金蝶Apusic应用服务器的配置文件：apusic-application.xml，放在目录META-INF下。EJB JAR和Web应用WAR作为模块打包在Java EE应用EAR中时，金蝶Apusic应用服务器的配置文件apusic-application.xml应该放在EAR文件的META-INF目录中。

金蝶Apusic应用服务器提供了打包和部署的功能，这些功能使用WAR、JAR和EAR作为组件和应用的标准打包格式，使用基于XML的部署描述文件配置参数。Java EE标准的部署描述文件在Java EE5规范中进行了详细的介绍。下面的表格列出了Java EE标准部署描述文件在规范中的具体位置。附录中列出了这些部署描述文件的文档类型定义（Document Type Definition，DTD）。

部署描述文件	规范中的具体位置
	Java EE 5规范，第8章：“Application Assembly and Deployment”。
	Java Servlet规范2.4，第13章：“Deployment Descriptor”和JavaServer Pages规范2.1，第7章，“Tag Extensions”。
	EJB3.0 核心规范，第19章：“Deployment Descriptor”。
	Java EE 5规范，第9章：“Application Clients”。

金蝶pusic应用服务器还使用附加的部署描述文件apusic-application.xml配置金蝶Apusic应用服务器相关的属性。关于金蝶Apusic应用服务器部署描述文件apusic-application.xml的详细介绍请见"Apusic部署描述文件"。

17.1.3 打包和部署Web模块

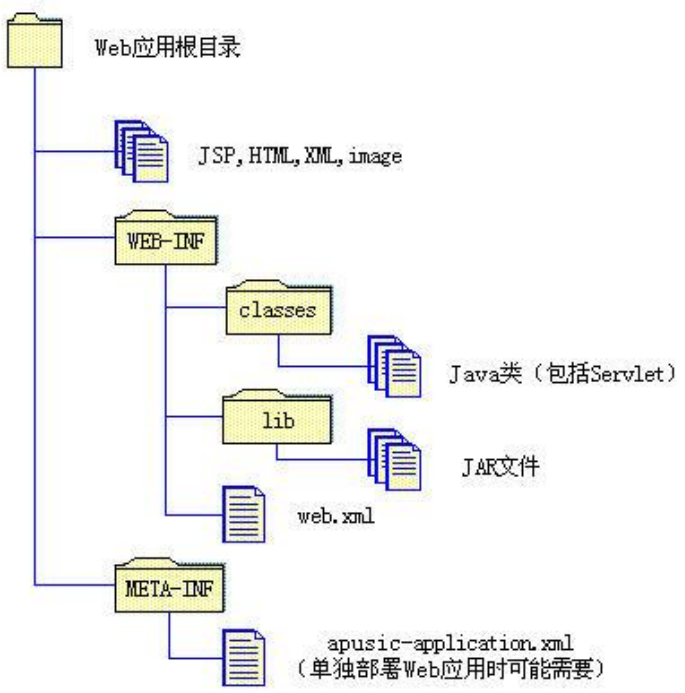
Web应用是最小的、可部署的一组可重用的Web资源。Web应用被打包和部署成Web ARchive(WAR) 文件，是一个带有 .war 后缀的JAR文件。一个Web应用可能包含：

- Servlets的类文件和相关的类
- JSP文件及其辅助类
- 静态文档(HTML,images等)
- Applet及其类文件
- Web模块部署描述文件web.xml
- 单独部署Web模块时，可能需要apusic-application.xml

17.1.3.1 Web应用的目录结构

Web应用使用层次结构存放Web资源，在开发阶段表现为文件系统的目录结构。如下图所示：

Web应用的目录结构



17.1.3.2 Web模块的部署描述文件

部署Web应用首先要配置web.xml文件。开发人员需要在Web.xml部署描述文件中配置：

- ServletContext初始化参数
- Session配置

- Servlet/JSP定义
- Servlet/JSP映射
- MIME类型映射
- 欢迎文件列表
- 错误页面
- 安全

Web 模块单独部署时可能还需要部署描述文件apusic-application.xml配置金蝶Apusic应用服务器相关的信息，如context-root、EJB引用、资源引用等。由于每个web模块都必须指定一个唯一的context-root，因此有三种方法指定一个独立web模块的context-root：

- 在部署时指定base-context，请参考appctl
- 在WAR中包含一个apusic-application.xml文件指定context-root；
- 如果以上两种方法都没有采用，则使用appctl命令中指定的应用名作为context-root。如果使用了自动部署，WAR文件名去掉.war后缀后作为应用名。

17.1.3.3 Web模块的两种形式

在金蝶Apusic应用服务器上，支持两种形式的Web-WAR，一种是标准的JAR文件形式，另一种则可以是目录形式的Web-JAR。通常，在Web应用的开发阶段，采用目录的Web-WAR，可以减少使用JAR工具打包的工作，并可方便的对模块进行修改。

而在实际的装配和部署阶段，往往采用易于管理的标准JAR文件格式。

当使用标准的JAR文件形式打包Web-WAR模块时，只需按照Web应用的结构，使用jar工具打包即可。

17.1.3.4 部署Web模块

使用金蝶Apusic应用服务器提供的appctl工具部署Web模块。金蝶Apusic应用服务器支持目录形式和打包形式的Web模块，例如：

```
appctl install web_app_name MyWeb.war
```

如果Web模块是目录的形式则Web应用无须安装，仅在server.xml中登记一项。如果Web模块是打包的形式则将应用自动解压缩到DOMAIN_NAME/deploy目录下的应用子目录中(其中DOMAIN_NAME为当前正在运行的金蝶Apusic应用服务器实例域名)，如果该档案文件发生变化，在重启应用时将对Web模块重新自动解包。在应用子目录中每个模块都建有一个临时文件夹，用于存放运行时生成的临时文件，此临时文件夹代替了原来的scratch目录。

17.1.3.5 自动部署

将打包的Web-WAR模块拷贝到DOMAIN_NAME/applications目录，Web应用会被自动部署。其中DOMAIN_NAME为正在运行的金蝶Apusic应用服务器域名。

17.1.4 打包和部署EJB模块

EJB-JAR文件是一个不包含装配信息的、由开发者提供的、包含一个或多个EJB组件的应用可装配单元文件，是一个一般的JAR文件。一般一个EJB-JAR文件代表一个EJB模块。当一个或多个EJB模块被装配到一个Java EE应用，并由装配者通过部署描述提供了装配信息

和具体运行环境中资源信息之后，即可部署到应用服务器。金蝶Apusic应用服务器支持EJB-JAR的单独部署，这时，EJB-JAR中就要包含装配信息。

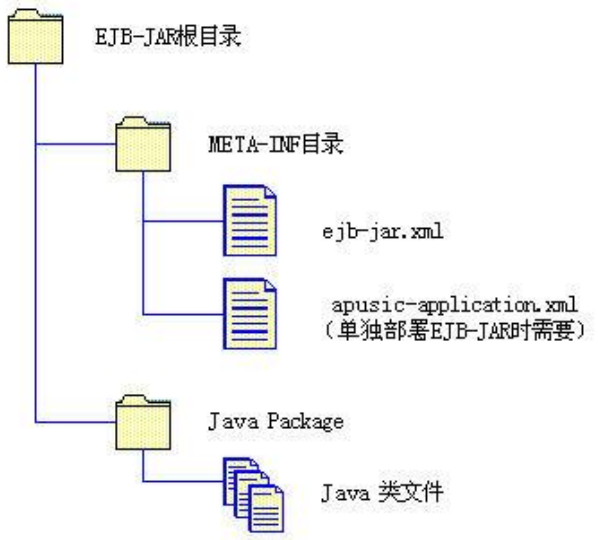
17.1.4.1 EJB-JAR文件结构

EJB-JAR文件中必须包含如下内容：

- 组件类文件；
- 组件接口类文件和组件Home接口的类文件；
- 如组件是Entity Bean，则还必须包含主键类文件；
- 组件依存的其他Java类文件；
- 一个模块的部署描述文件，ejb-jar.xml文件；
- 金蝶Apusic应用服务器相关装配信息的部署描述文件apusic-application.xml，EJB-JAR单独部署时需要提供；

EJB-JAR文件的结构如下图：

EJB-JAR文件结构



17.1.4.2 EJB模块的部署描述文件

部署描述文件ejb-jar.xml为EJB模块提供如下两方面的信息：

- 结构信息

ejb-jar.xml文件为模块中的EJB组件提供的结构信息并声明组件的外部依存关系，对于组件的开发者而言，结构信息是必须提供的，而且一般结构信息在结构信息在装配和部署时不能更改，以免破坏组件的功能。

- 装配信息

装配信息提供将组件装配到一起，形成一个更大的可装配单元的信息，装配信息对于组件提供者而言是可选的，并且在装配和部署时可以通过修改装配信息改变装配后的应用的行为。

ejb-jar.xml文件必须符合ejb-jar_2_1.xsd文档类型声明，并在文件中指定正确的xsd文件：

```
xmlns="http://java.sun.com/xml/ns/Java EE"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/Java EE
http://java.sun.com/xml/ns/Java EE/ejb-jar_3_0.xsd"

version="3.0"
```

EJB模块单独部署时还需要部署描述文件apusic-application.xml配置Apusic相关的信息，如EJB引用、资源引用、CMP2.0的域-数据库列映射、CMP2.0关系映射等。

17.1.4.3 EJB模块的两种形式

在金蝶Apusic应用服务器上，支持两种形式的EJB-JAR，一种是标准的JAR文件形式，另一种则可以是目录形式的EJB-JAR。通常，在组件开发阶段，采用目录形式的EJB-JAR，可以减少使用JAR工具打包的工作，并可方便的对模块进行修改。

而在实际的装配和部署阶段，往往采用易于管理的标准JAR文件格式。

当使用标准的JAR文件形式打包EJB-JAR模块时，只需按照前面描述的EJB-JAR的结构，使用jar工具打包即可。

17.1.4.4 部署EJB模块

使用金蝶Apusic应用服务器提供的appctl工具部署EJB模块。金蝶Apusic应用服务器支持目录形式和打包形式的EJB模块，例如：

```
appctl install ejb_app_name MyEJB.jar
```

如果EJB模块是目录的形式则无须安装，仅在server.xml中登记一项。如果EJB模块是打包的形式则将应用自动解压缩到$APUSIC_HOME/domains/<DOMAIN_NAME>/deploy目录下的应用子目录件（其中<DOMAIN_NAME>为正在运行的Apusic领域名），如果该档案文件发生变化，在重启应用时将对EJB模块重新自动解包。在应用子目录中每个模块都建有一个临时文件夹，用于存放运行时生成的临时文件，此临时文件夹代替了原来的scratch目录。

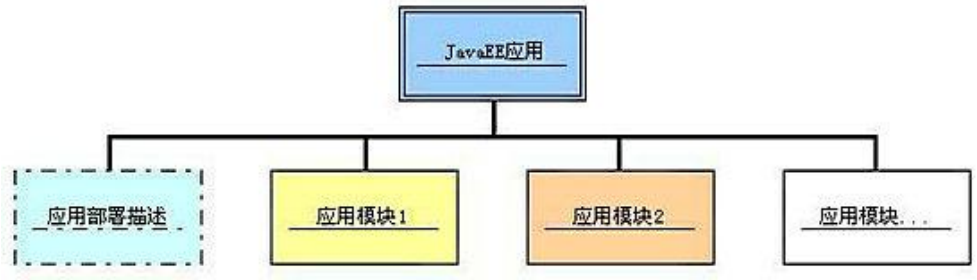
17.1.4.5 自动部署

将打包的EJB-WAR模块拷贝到$APUSIC_HOME/domains/<DOMAIN_NAME>/applications目录（其中<DOMAIN_NAME>为正在运行的Apusic领域名），EJB模块会被自动部署。

17.1.5 打包和部署Java EE应用

Java EE应用由一个或多个Java EE组件和部署描述文件组成。通常，Apusic应用服务器中的Java EE应用逻辑结构如下图：

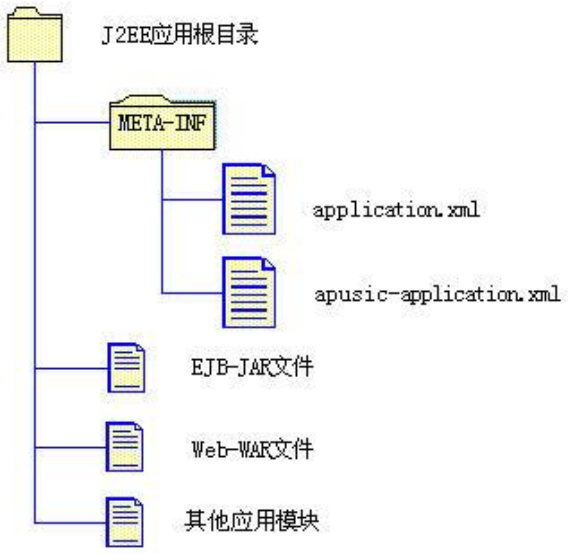
Java EE应用逻辑结构



17.1.5.1 Java EE应用的目录结构

在Apusic应用服务器上，实际的Java EE应用被打包成为一个后缀名为".ear"的EAR文件。实际的EAR文件中的目录结构如下图：

Java EE应用的目录结构



17.1.5.2 Java EE应用的部署描述文件

Java EE标准的部署描述文件application.xml列出了应用包含的所有Java EE模块。

Apusic的部署描述文件apusic-application.xml配置应用包含模块的Apusic相关的信息，如context-root、EJB引用、资源引用、CMP2.0的域-数据库列映射、CMP2.0关系映射等。

	注意
	当Web模块、EJB模块作为完整的Java EE应用其中一部分时，应该将它们的Apusic相关配置信息放在整个应用的META-INF目录下的apusic-application.xml中。

17.1.5.3 Java EE应用的两种形式

在Apusic应用服务器上，支持两种形式的Java EE应用，一种是标准的JAR文件形式，另一种则可以是目录形式。通常，在Java EE应用开发阶段，采用目录形式，可以减少使用JAR工具打包的工作，并可方便的对模块进行修改。

而在实际的装配和部署阶段，往往采用易于管理的标准JAR文件格式。

当使用标准的JAR文件形式打包Java EE应用时，只需按照前面描述的Java EE应用目录结构，使用jar工具打包即可。

17.1.5.4 部署Java EE应用

使用Apusic提供的appctl工具部署Java EE应用。Apusic支持目录形式和打包形式的Java EE应用，例如：

```
appctl install jee_app_name MyApp.ear0
```

如果Java EE应用是目录的形式则应用无须安装，仅在server.xml中登记一项。如果Java EE应用块是打包的形式则将应用自动解压缩到$APUSIC_HOME/domains/<DOMAIN_NAME>/deploy目录下的应用子目录中（其中<DOMAIN_NAME为正在运行的Apusic领域名>），如果该档案文件发生变化，在重启应用时将对Java EE应用重新自动解包。

17.1.5.5 自动部署

将打包的Java EE应用EAR文件拷贝到$APUSIC_HOME/domains/<DOMAIN_NAME>/applications目录（其中<DOMAIN_NAME为正在运行的Apusic领域名>），Java EE应用会被自动部署。

17.1.6 Apusic部署描述文件

独立的Web应用、EJB模块和完整的JavaEE应用都需要部署描述文件apusic-application.xml来配置Apusic相关的属性。apusic-application_5_0.dtd文件定义了apusic-application.xml的结构，确定了它能够包含的元素、子元素和元素的属性。定义在DTD中的元素能够包含：

- 子元素
- 数据
- 属性

元素能够包含子元素，例如下面的代码片断定义了apusic-application元素：

```
<!ELEMENT apusic-application (module*, mail-session*,
security-role*)>
```

可以看出，元素apusic-application能够包含子元素moduls、mail-session和security-role。通过可选的后缀修饰子元素来决定能够包含子元素的数量。具体含义见下表：

子元素后缀	含义
element*	能够包含零或多个这种子元素
element?	能够包含零或一个这种子元素
element+	必须包含一个或多个这种子元素
element	只能包含一个这种子元素

一些元素没有子元素而是包含字符数据。例如：

```
<!ELEMENT role-name (#PCDATA)>
```

在apusic-application.xml中就应该表现下面的形式。注意，字符数据前后都不要有空格：

```
<role-name>admin</role-name>
```

元素还能够带有属性，使用ATTLIST标记声明。声明的语法为：

```
<!ATTLIST element-name attribute-name
attribute-type default-value>
```

例如，在apusic-application_4_0.dtd中为module元素定义了名称为uri的属性：

```
<!ATTLIST module uri CDATA #IMPLIED>
```

属性的类型（attribute-type）可以为下表中所列的值：

值	说明
CDATA	字符数据
(en1	en2
ID	唯一的ID
IDREF	另外一个元素的ID
IDREFS	其他元素ID列表
NMTOKEN	一个有效的XML名称
NMTOKENS	有效的XML名称列表
ENTITY	实体
ENTITIES	实体列表
NOTATION	符合的名称
xml:	预先定义的XML值

属性的缺省值（default-value）可以为：

值	说明
value	value为属性的缺省值

#REQUIRED	元素必须包含这个属性值
#IMPLIED	元素可以不包含这个属性值
#FIXED value	属性值是固定值

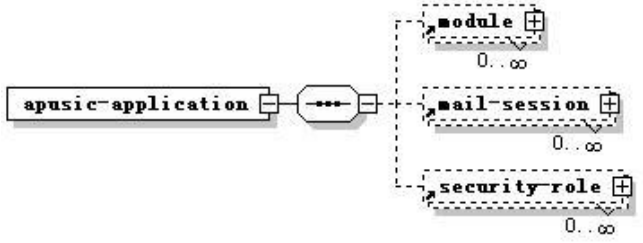
17.1.6.1 apusic-application.xml中的元素

这一节将介绍apusic-application.xml中包含的元素。

17.1.6.1.1 APUSIC-APPLICATION元素

apusic-application为根元素，包含下列子元素：

apusic-application元素

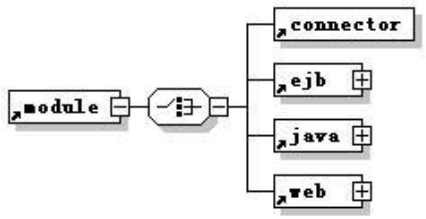


元素	包含数量	描述
module	零个或多个	代表单个Java EE模块，如Web、EJB模块
mail-session	零个或多个	
security-role	零个或多个	安全角色映射

17.1.6.1.2 MODULE元素

module元素代表单个Java EE模块，包含下列子元素之一：

module元素



module的子元素为枚举类型，因此它只包含四个子元素中的一个元素。

元素	包含数量	描述
connector	其中一个	暂不支持
ejb	其中一个	代表EJB模块

java	其中一个	代表应用客户端模块
web	其中一个	代表web应用模块

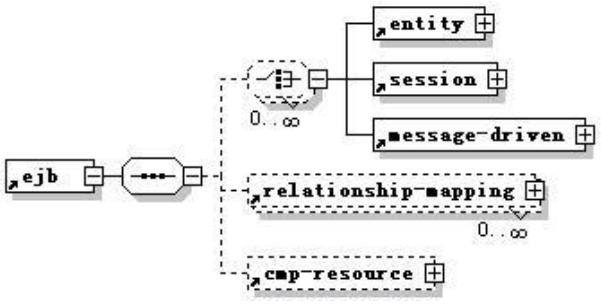
module元素还包含一个属性uri：

名称	类型	缺省值	描述
uri	字符	#IMPLIED	模块的URI，相对于应用的最顶层目录

17.1.6.1.3 EJB元素

ejb元素代表EJB模块，包含下列子元素：

ejb元素

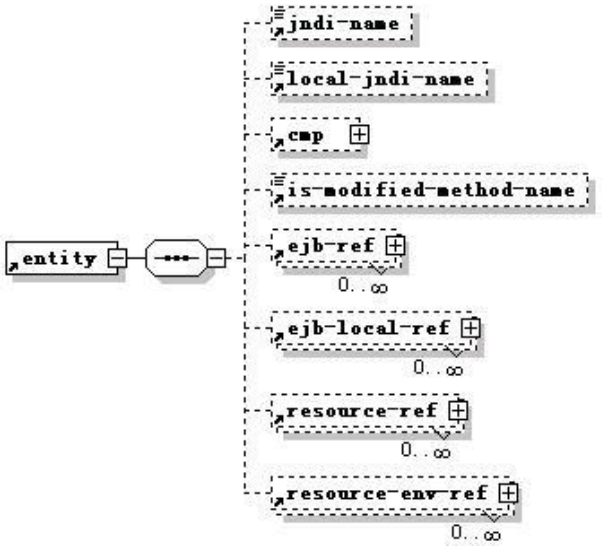


元素	包含数量	描述
entity	session	message-driven
relationship-mapping	一个或多个	CMP2.0关系映射
cmp-resource	零个或一个	CMP数据源JNDI名称

17.1.6.1.4 ENTITY元素

entity元素代表entity bean，包含下列子元素：

entity元素



元素	包含数量	描述
jndi-name	零个或一个	指定jndi名称
local-jndi-name	零个或一个	指定本地jndi名称
cmp	零个或一个	配置Apusic相关的CMP属性
is-modified-method-name	零个或一个	指定is-modified方法，容器根据此方法返回的布尔值来决定是否调用ejbStore方法将信息存储到数据库。is-modified方法在BMP 和CMP1.x的实体Bean中使用。CMP2.x的实体Bean采用另外的方式实现，无需设置is-modified方法。
ejb-ref	零个或多个	映射ejb-jar.xml中相应的EJB引用到JNDI名称
ejb-local-ref	零个或多个	映射ejb-jar.xml中相应的本地EJB引用到JNDI名称
resource-ref	零个或多个	映射ejb-jar.xml中相应的资源引用到JNDI名称
resource-env-ref	零个或多个	映射ejb-jar.xml中相应的资源环境引用到JNDI名称

entity元素还包含下列属性：

名称	类型	缺省值	描述
ejb-name	字符	#REQUIRED	匹配ejb-jar.xml中相应的ejb-name
pool-size	字符	#IMPLIED	实例池容量，组件实例池的大小，单位为实例的个数
cache-size	字符	#IMPLIED	缓冲池容量，可保存在EJB缓存中的EJB实例数
transaction-timeout	字符	#IMPLIED	事务的超时时间值，单位为秒
concurrency-strategy	字符	#IMPLIED	并发策略，可以为“Exclusive”，“Parallel”，“ReadOnly”。Exclusive - 容器将维护一个实体Bean的内部缓存，当客户访问其中一个EJB实例时在该实例上设置互斥锁，当事务完成时释放互斥锁，保证所有的事务都是可串行化的。Parallel - 容器将为每个并发访问的客户都分配一个EJB实例，每个事务独立执行，EJB实例状态和持久存储数据之间的同步由数据库系统完成。这种并发控制在EJB容器和数据库之间做的很薄，所以一般不推荐使用，只有在Exclusive方式下查询频繁出现死锁时才考虑使用这种模式。ReadOnly - 实例在运行过程中永远不会执行任何INSERT/UPDATE/DELETE操作，并根据一个可配置的时间周期性读取最新的数据。使实例可以获得最大程度的并发执行。
force-refresh	True	False	False
delay-updates	True	False	True
load-for-update	True	False	False
cmp11-promotion	字符	#IMPLIED	CMP1.1自动提升，对EJB1.1版本的容器管理持久性Entity Bean有效，在运行时是否需要将其自动升级为EJB2.1版本的容器管理持久性Entity Bean运行。默认值为“True”。
expiration-time	字符	#IMPLIED	缓冲过期时间，实例保持在就绪状态的最大空闲时间。单位是秒。

17.1.6.1.5 JNDI-NAME元素

jndi-name元素指定JNDI名称，不包含子元素。例如：

```
<jndi-name>jdbc/Hello</jndi-name>
```

17.1.6.1.6 LOCAL-JNDI-NAME元素

local-jndi-name元素指定本地JNDI名称，不包含子元素。

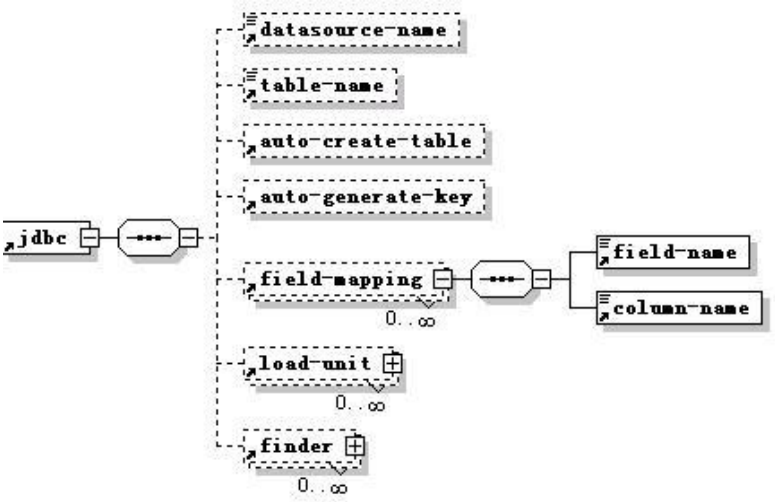
17.1.6.1.7 CMP元素

cmp元素代表Apusic相关的CMP配置信息，包含一个子元素jdbc。

17.1.6.1.8 JDBC元素

jdbc元素包含下列子元素：

jdbc元素



名称	包含数量	描述
datasource-name	零或一个	数据源JNDI名称，已不使用
table-name	零或一个	CMP映射的数据库表名
auto-create-table	零或一个	自动产生表，在数据库表不存在的情况下使用
auto-generate-key	零或一个	自动产生主键
field-mapping	零或多个	CMP域-数据库表列映射
load-unit	零或多个	装载单元，Apusic中对于容器管理持久性的Entity Bean，开发者可以通过将状态划分为不同的装载单元，实现Lazy Loading的特性
finder	零或多个	为CMP1.x find方法设置SQL子句

17.1.6.1.9 DATASOURCE-NAME元素

datasource-name不包含子元素，代表数据源JNDI名称，已不使用。

17.1.6.1.10 TABLE-NAME元素

table-name元素代表CMP映射的数据库表名，不包含子元素。

17.1.6.1.11 AUTO-CREATE-TABLE元素

auto-create-table元素代表自动产生表，在数据库表不存在的情况下使用。

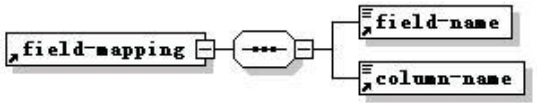
17.1.6.1.12 AUTO-GENERATE-KEY元素

auto-generate-key元素代表自动产生主键。

17.1.6.1.13 FIELD-MAPPING元素

field-mapping元素代表CMP域-数据库表列映射，包含下列子元素：

field-mapping元素



名称	包含数量	描述
field-name	一个	CMP域名称，必须匹配ejb-jar.xml中相应的cmp-field子元素field-name
column-name	一个	数据库表列名称

17.1.6.1.14 FIELD-NAME元素

field-name元素代表CMP域名称，不包含子元素。

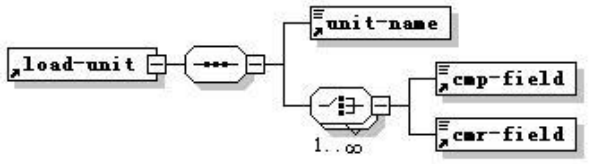
17.1.6.1.15 COLUMN-NAME元素

column-name元素代表数据库表列名称，不包含子元素。

17.1.6.1.16 LOAD-UNIT元素

load-unit元素代表装载单元，包含下列子元素：

load-unit元素



名称	包含数量	描述
unit-name	一个	装载单元名称
cmp-field	cmr-field	一个或多个

17.1.6.1.17 UNIT-NAME元素

unit-name元素代表装载单元名称，不包含子元素。

17.1.6.1.18 CMP-FIELD元素

cmp-field元素代表CMP域，必须匹配ejb-jar.xml中相应的cmp-field子元素field-name，不包含子元素。

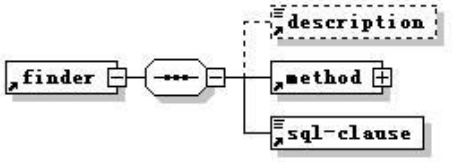
17.1.6.1.19 CMR-FIELD元素

cmr-field元素代表CMR域，必须匹配ejb-jar.xml中相应的cmr-field子元素cmr-field-name，不包含子元素。

17.1.6.1.20 FINDER元素

finder元素代表CMP1.x find方法，包含下列子元素：

finder元素



名称	包含数量	描述
description	零或一个	为父元素提供文本描述信息
method	一个	代表EJB home接口或组件接口中定义的方法
sql-clause	一个	CMP1.1查询语句

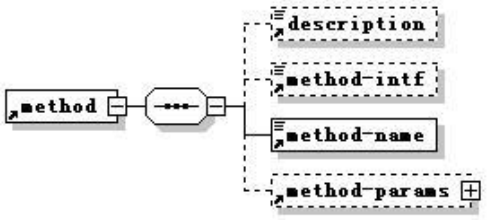
17.1.6.1.21 DESCRIPTION元素

description元素为父元素提供文本描述信息，不包含子元素。

17.1.6.1.22 METHOD元素

method元素代表EJB home接口或组件接口中定义的方法，包含下列子元素：

method元素



名称	包含数量	描述
description	零或一个	为父元素提供文本描述信息
method-intf	零或一个	用于区分home接口和组件接口中具有相同方法名和方法签名的方法。
method-name	一个	查询方法的名称
method-params	零或一个	方法参数列表

17.1.6.1.23 METHOD-INTF元素

method-intf元素用于区分home接口和组件接口中具有相同方法名和方法签名的方法，不包含子元素。method-intf元素必须为下面列表之一：

```
<method-intf>Home</method-intf>
<method-intf>Remote</method-intf>
<method-intf>LocalHome</method-intf>
<method-intf>Local</method-intf>
```

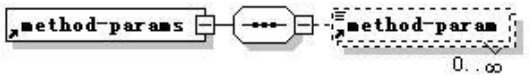
17.1.6.1.24 METHOD-NAME元素

method-name元素代表查询方法的名称，不包含子元素。

17.1.6.1.25 METHOD-PARAMS元素

method-params元素代表方法的参数列表，包含下列子元素：

method-params元素



名称	包含数量	描述
method-param	零或多个	方法参数的Java类型全名

17.1.6.1.26 METHOD-PARAM元素

method-param元素代表方法参数的Java类型全名，不包含子元素。

17.1.6.1.27 SQL-CLAUSE元素

sql-clause元素代表CMP1.1的查询语句，不包含子元素。

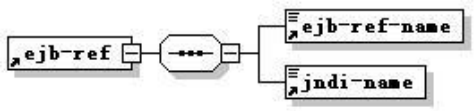
17.1.6.1.28 IS-MODIFIED-METHOD-NAME元素

is- modified-method-name元素用来指定is-modified方法，容器根据此方法返回的布尔值来决定是否调用ejbStore方法将信息存储到数据库。is-modified方法在BMP和CMP1.x的实体Bean中使用。CMP2.x的实体Bean采用另外的方式实现，无需设置is-modified方法。

17.1.6.1.29 EJB-REF元素

ejb-ref元素用来映射ejb-jar.xml中相应的EJB引用到JNDI名称，包含下列子元素：

ejb-ref元素



名称	包含数量	描述
ejb-ref-name	一个	必须匹配ejb-jar.xml中相应的ejb-ref-name元素
jndi-name	一个	指定JNDI名称

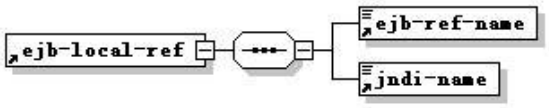
17.1.6.1.30 EJB-REF-NAME元素

ejb-ref-name元素必须匹配ejb-jar.xml中相应的ejb-ref-name元素。该元素不包含子元素。

17.1.6.1.31 EJB-LOCAL-REF元素

ejb-local-ref元素用来映射ejb-jar.xml中相应的本地EJB引用到JNDI名称，包含下列子元素：

ejb-local-ref元素

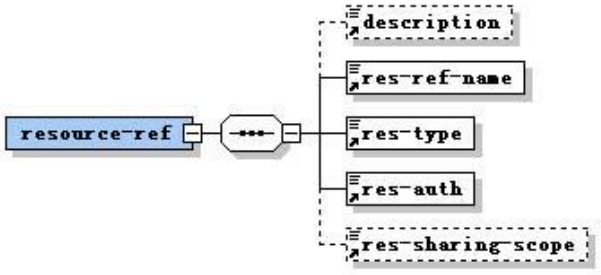


名称	包含数量	描述
ejb-ref-name	一个	必须匹配ejb-jar.xml中相应的ejb-ref-name元素
jndi-name	一个	指定JNDI名称

17.1.6.1.32 RESOURCE-REF元素

resource-ref元素用来映射ejb-jar.xml中相应的资源引用到JNDI名称，包含下列子元素：

resource-ref元素



名称	包含数量	描述
res-ref-name	一个	必须匹配ejb-jar.xml中相应的resource-ref的子元素res-ref-name。
jndi-name	一个	指定JNDI名称

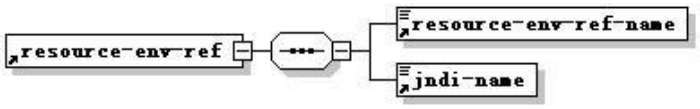
17.1.6.1.33 RES-REF-NAME元素

res-ref-name元素必须匹配ejb-jar.xml中相应的resource-ref的子元素res-ref-name。该元素不包含子元素。

17.1.6.1.34 RESOURCE-ENV-REF元素

resource-env-ref元素用来映射ejb-jar.xml中相应的资源环境引用到JNDI名称，包含下列子元素：

resource-env-ref元素



名称	包含数量	描述
resource-env-ref-name	一个	必须匹配ejb-jar.xml中相应的resource-env-ref的子元素resource-env-ref-name。
jndi-name	一个	指定JNDI名称

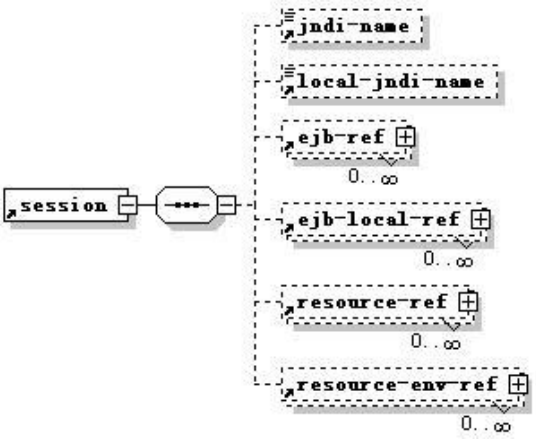
17.1.6.1.35 RESOURCE-ENV-REF-NAME元素

resource-env-ref-name元素必须匹配ejb-jar.xml中相应的resource-env-ref的子元素resource-env-ref-name。该元素不包含子元素。

17.1.6.1.36 SESSION元素

session元素代表session bean，包含下列子元素：

session元素



元素	包含数量	描述
jndi-name	零个或一个	指定jndi名称
local-jndi-name	零个或一个	指定本地jndi名称
ejb-ref	零个或多个	映射ejb-jar.xml中相应的EJB引用到JNDI名称
ejb-local-ref	零个或多个	映射ejb-jar.xml中相应的本地EJB引用到JNDI名称
resource-ref	零个或多个	映射ejb-jar.xml中相应的资源引用到JNDI名称
resource-env-ref	零个或多个	映射ejb-jar.xml中相应的资源环境引用到JNDI名称

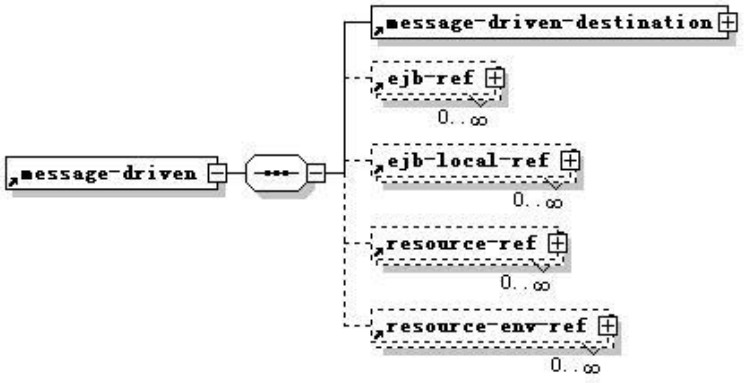
session元素还包含下列属性：

名称	类型	缺省值	描述
ejb-name	字符	#REQUIRED	匹配ejb-jar.xml中相应的ejb-name
pool-size	字符	#IMPLIED	实例池容量，组件实例池的大小，单位为实例的个数
cache-size	字符	#IMPLIED	缓冲池容量，可保存在EJB缓存中的EJB实例数
session-timeout	字符	#IMPLIED	Session Bean组件实例的超时时间
transaction-timeout	字符	#IMPLIED	事务的超时时间值，单位为秒

17.1.6.1.37 MESSAGE-DRIVEN元素

message-driven元素代表message-driven bean，包含下列子元素：

message-driven元素



元素	包含数量	描述
message-driven-destination	一个	配置Apusic相关的message-driven bean目的地
ejb-ref	零个或多个	映射ejb-jar.xml中相应的EJB引用到JNDI名称
ejb-local-ref	零个或多个	映射ejb-jar.xml中相应的本地EJB引用到JNDI名称
resource-ref	零个或多个	映射ejb-jar.xml中相应的资源引用到JNDI名称
resource-env-ref	零个或多个	映射ejb-jar.xml中相应的资源环境引用到JNDI名称

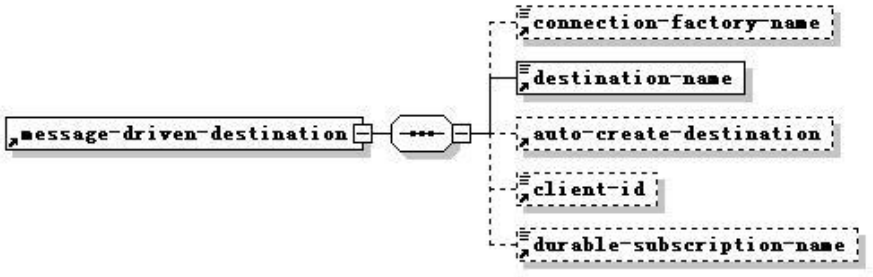
message-driven元素还包含下列属性：

名称	类型	缺省值	描述
ejb-name	字符	#REQUIRED	匹配ejb-jar.xml中相应的ejb-name
pool-size	字符	#IMPLIED	实例池容量，组件实例池的大小，单位为实例的个数

17.1.6.1.38 MESSAGE-DRIVEN-DESTINATION元素

message-driven-destination元素用来配置Apusic相关的message-driven bean目的地，包含下列子元素：

message-driven-destination元素



名称	包含数量	描述
connection-factory-name	零或一个	连接工厂的JNDI名字
destination-name	一个	JMS消息目的地JNDI名字
auto-create-destination	零或一个	自动创建目的地
client-id	零或一个	客户端标识
durable-subscription-name	零或一个	持久订阅名称

17.1.6.1.39 CONNECTION-FACTORY-NAME元素

connection-factory-name元素代表连接工厂的JNDI名字，不包含子元素。

17.1.6.1.40 DESTINATION-NAME元素

destination-name元素代表JMS消息目的地JNDI名字，不包含子元素。

17.1.6.1.41 AUTO-CREATE-DESTINATION元素

auto-create-destination元素代表自动创建目的地，不包含子元素。

17.1.6.1.42 CLIENT-ID元素

client-id元素客户端标识，不包含子元素。

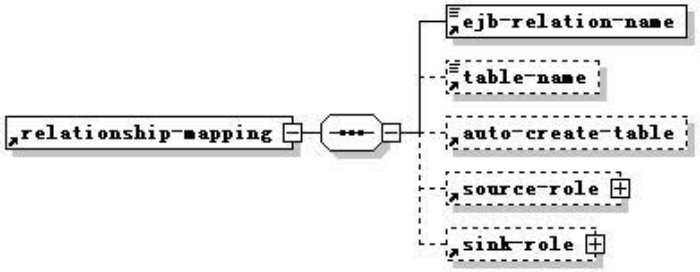
17.1.6.1.43 DURABLE-SUBSCRIPTION-NAME元素

durable-subscription-name元素代表持久订阅名称，不包含子元素。

17.1.6.1.44 RELATIONSHIP-MAPPING元素

relationship-mapping元素代表Apusic相关的CMP2.0关系映射配置，包含下列子元素：

relationship-mapping元素



名称	包含数量	描述
ejb-relation-name	一个	必须匹配ejb-jar.xml中ejb-relation的子元素ejb-relation-name
table-name	零或一个	辅助表名。单向一对多、单向多对多和双向多对多需要辅助表
auto-create-table	零或一个	自动创建辅助表
source-role	零或一个	CMP2.0关系中的一个角色
sink-role	零或一个	CMP2.0关系中的另一个角色

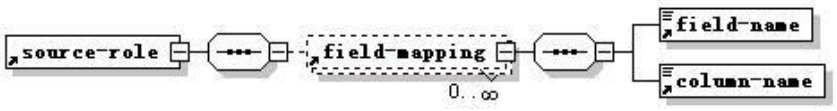
17.1.6.1.45 EJB-RELATION-NAME元素

ejb-relation-name元素必须匹配ejb-jar.xml中ejb-relation的子元素ejb-relation-name。该元素不包含子元素。

17.1.6.1.46 SOURCE-ROLE元素

source-role元素代表CMP2.0关系中的一个角色，包含下列子元素：

source-role元素

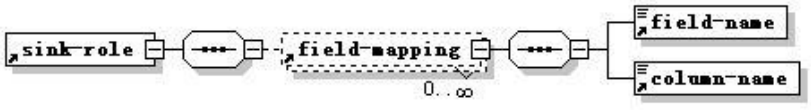


名称	包含数量	描述
field-mapping	零或多个	CMP域-外键映射

17.1.6.1.47 SINK-ROLE元素

sink-role元素代表CMP2.0关系中的一个角色，包含下列子元素：

sink-role元素



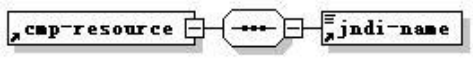
名称	包含数量	描述

field-mapping	零或多个	CMP域-外键映射

17.1.6.1.48 CMP-RESOURCE元素

cmp-resource元素代表CMP数据源JNDI名称，包含下列子元素：

cmp-resource元素

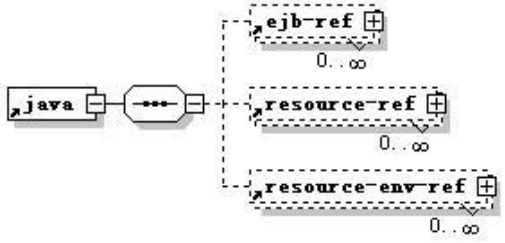


名称	包含数量	描述
jndi-name	零或一个	CMP数据源JNDI名称，是EJB模块中所有CMP实体EJB所使用的数据源

17.1.6.1.49 JAVA元素

java元素代表应用客户端模块，包含下列子元素：

java元素

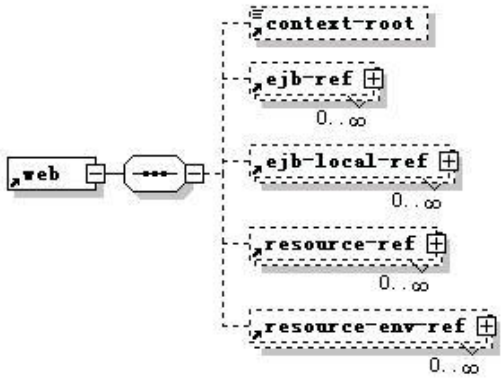


元素	包含数量	描述
ejb-ref	零个或多个	映射ejb-jar.xml中相应的EJB引用到JNDI名称
resource-ref	零个或多个	映射ejb-jar.xml中相应的资源引用到JNDI名称
resource-env-ref	零个或多个	映射ejb-jar.xml中相应的资源环境引用到JNDI名称

17.1.6.1.50 WEB元素

web元素代表web应用模块，包含下列子元素：

web元素



元素	包含数量	描述
context-root	零个或一个	web应用的上下文根路径
ejb-ref	零个或多个	映射ejb-jar.xml中相应的EJB引用到JNDI名称
ejb-local-ref	零个或多个	映射ejb-jar.xml中相应的本地EJB引用到JNDI名称
resource-ref	零个或多个	映射ejb-jar.xml中相应的资源引用到JNDI名称
resource-env-ref	零个或多个	映射ejb-jar.xml中相应的资源环境引用到JNDI名称

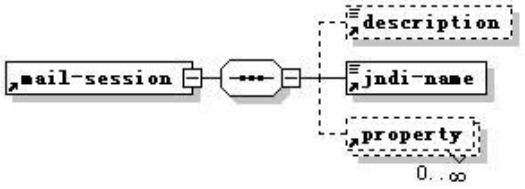
17.1.6.1.51 CONTEXT-ROOT元素

context-root元素代表web应用的上下文根路径，不包含子元素。

17.1.6.1.52 MAIL-SESSION元素

mail-session元素包含下列子元素：

mail-session元素



元素	包含数量	描述
description	零个或一个	为父元素提供文本描述信息
jndi-name	一个	指定JNDI名称
property	零个或多个	配置mail-session属性信息，一个设置两个属性：“mail.host”和“mail.from”

17.1.6.1.53 PROPERTY元素

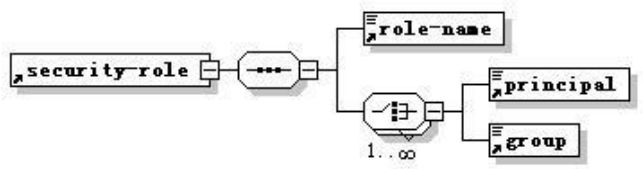
property元素代表配置相关的属性信息，不包含子元素，但有两个属性：

名称	类型	缺省值	描述
name	字符	#REQUIRED	属性名称
value	字符	#REQUIRED	属性值

17.1.6.1.54 SECURITY-ROLE元素

security-role元素代表Java EE应用安全角色到Apusic用户的映射，包含下列子元素：

security-role元素



元素	包含数量	描述
role-name	一个	Java EE应用中定义的角色名
principal	group	一个或多个

17.1.6.1.55 ROLE-NAME元素

role-name元素代表Java EE应用中定义的角色名，不包含子元素。

17.1.6.1.56 PRINCIPAL元素

principal元素代表Apusic应用服务器中的用户，不包含子元素。

17.1.6.1.57 GROUP元素

group元素代表Apusic应用服务器中的用户，不包含子元素。

17.1.7 application_5_0.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://java.sun.com/xml/ns/j2ee"
    xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified"
version="5.0">
    <xsd:annotation>
        <xsd:documentation>

@(#)application_5_0.xsds 1.15 01/04/05
```

```
</xsd:documentation>
    </xsd:annotation>
    <xsd:annotation>
        <xsd:documentation>

Copyright 2003-2005 Sun Microsystems, Inc.

4150 Network Circle
Santa Clara, California 95054
U.S.A
All rights reserved.

Sun Microsystems, Inc. has intellectual property rights
relating to technology described in this document. In
particular, and without limitation, these intellectual
property rights may include one or more of the U.S. patents
listed at http://www.sun.com/patents and one or more
additional patents or pending patent applications in the
U.S. and other countries.

This document and the technology which it describes are
distributed under licenses restricting their use, copying,

distribution, and decompilation. No part of this document
may be reproduced in any form by any means without prior
written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is
copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Solaris, Java, J2EE,

JavaServer Pages, Enterprise JavaBeans and the Java Coffee
Cup logo are trademarks or registered trademarks of Sun
Microsystems, Inc. in the U.S. and other countries.

Federal Acquisitions: Commercial Software - Government Users
Subject to Standard License Terms and Conditions.
</xsd:documentation>
    </xsd:annotation>
```

```
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[

This is the XML Schema for the application 5.0 deployment
descriptor. The deployment descriptor must be named
"META-INF/application.xml" in the application's ear file.

All application deployment descriptors must indicate
the application schema by using the J2EE namespace:

http://java.sun.com/xml/ns/j2ee
and indicate the version of the schema by
using the version element as shown below:
<application xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/application_5_0.xsd"

version="5.0">

...
</application>

The instance documents may indicate the published version of
the schema using the xsi:schemaLocation attribute for J2EE
namespace with the following location:

http://java.sun.com/xml/ns/j2ee/application_5_0.xsd
]]>
</xsd:documentation>
</xsd:annotation>
<xsd:annotation>
    <xsd:documentation>

The following conventions apply to all J2EE
deployment descriptor elements unless indicated otherwise.
```

```
- In elements that specify a pathname to a file within the
same JAR file, relative filenames (i.e., those not
starting with "/") are considered relative to the root of
the JAR file's namespace. Absolute filenames (i.e., those
starting with "/") also specify names in the root of the
JAR file's namespace. In general, relative names are
preferred. The exception is .war files where absolute
names are preferred for consistency with the Servlet API.
</xsd:documentation>
</xsd:annotation>
<xsd:include schemaLocation="j2ee_5_0.xsd" />
<!--
**************************************************
>
<xsd:element name="application" type="j2ee:applicationType">
<xsd:annotation>
<xsd:documentation>

The application element is the root element of a J2EE
application deployment descriptor.
</xsd:documentation>
</xsd:annotation>
<xsd:unique name="context-root-uniqueness">
<xsd:annotation>
<xsd:documentation>

The context-root element content must be unique
in the ear.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:module/j2ee:web"/>
<xsd:field xpath="j2ee:context-root"/>
</xsd:unique>
<xsd:unique name="security-role-uniqueness">
<xsd:annotation>
<xsd:documentation>

The security-role-name element content
must be unique in the ear.
</xsd:documentation>
```

```
</xsd:annotation>
<xsd:selector xpath="j2ee:security-role"/>
<xsd:field xpath="j2ee:role-name"/>
</xsd:unique>
</xsd:element>
<!--
**************************************************
>
<xsd:complexType name="applicationType">
<xsd:annotation>
<xsd:documentation>

The applicationType defines the structure of the
application.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="j2ee:descriptionGroup"/>
<xsd:element name="module"

type="j2ee:moduleType"

maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>

The application deployment descriptor must have one
module element for each J2EE module in the
application package. A module element is defined
by moduleType definition.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="security-role"

type="j2ee:security-roleType"

minOccurs="0"

maxOccurs="unbounded"/>
```

```
<xsd:element name="library-directory"

type="j2ee:pathType"

minOccurs="0"

maxOccurs="1">
<xsd:annotation>
<xsd:documentation>

The library-directory element specifies the pathname
of a directory within the application package, relative
to the top level of the application package. All files
named "*.jar" in this directory must be made available
in the class path of all components included in this
application package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="version"

type="j2ee:dewey-versionType"

fixed="5.0"

use="required">
<xsd:annotation>
<xsd:documentation>

The required value for the version is 5.0.
</xsd:documentation>
</xsd:annotation>
</xsd:attribute>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="moduleType">
```

```
<xsd:annotation>
<xsd:documentation>

The moduleType defines a single J2EE module and contains a
connector, ejb, java, or web element, which indicates the
module type and contains a path to the module file, and an
optional alt-dd element, which specifies an optional URI to
the post-assembly version of the deployment descriptor.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:choice>
<xsd:element name="connector"

type="j2ee:pathType">
<xsd:annotation>
<xsd:documentation>

The connector element specifies the URI of a
resource adapter archive file, relative to the
top level of the application package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="ejb"

type="j2ee:pathType">
<xsd:annotation>
<xsd:documentation>

The ejb element specifies the URI of an ejb-jar,

relative to the top level of the application
package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="java"

type="j2ee:pathType">
```

```
<xsd:annotation>
<xsd:documentation>

The java element specifies the URI of a java
application client module, relative to the top
level of the application package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="web"

type="j2ee:webType"/>
</xsd:choice>
<xsd:element name="alt-dd"

type="j2ee:pathType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The alt-dd element specifies an optional URI to the
post-assembly version of the deployment descriptor
file for a particular J2EE module. The URI must
specify the full pathname of the deployment
descriptor file relative to the application's root
directory. If alt-dd is not specified, the deployer
must read the deployment descriptor from the default
location and file name required by the respective
component specification.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="webType">
```

```
<xsd:annotation>
<xsd:documentation>

The webType defines the web-uri and context-root of
a web application module.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="web-uri"

type="j2ee:pathType">
<xsd:annotation>
<xsd:documentation>

The web-uri element specifies the URI of a web
application file, relative to the top level of the
application package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="context-root"

type="j2ee:string">
<xsd:annotation>
<xsd:documentation>

The context-root element specifies the context root
of a web application.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
</xsd:schema>
```

17.1.8 web-app_2_4.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://java.sun.com/xml/ns/j2ee"
    xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified"
version="2.4">
    <xsd:annotation>
        <xsd:documentation>

@(#)web-app_2_4.xsds 1.60 03/08/26
</xsd:documentation>
    </xsd:annotation>
    <xsd:annotation>
        <xsd:documentation>

Copyright 2003 Sun Microsystems, Inc., 901 San Antonio
Road, Palo Alto, California 94303, U.S.A. All rights
reserved.

Sun Microsystems, Inc. has intellectual property rights
relating to technology described in this document. In
particular, and without limitation, these intellectual
property rights may include one or more of the U.S. patents
listed at http://www.sun.com/patents and one or more
additional patents or pending patent applications in the
U.S. and other countries.

This document and the technology which it describes are
distributed under licenses restricting their use, copying,

distribution, and decompilation. No part of this document
may be reproduced in any form by any means without prior
written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is
copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Solaris, Java, J2EE,
```

```
JavaServer Pages, Enterprise JavaBeans and the Java Coffee
Cup logo are trademarks or registered trademarks of Sun
Microsystems, Inc. in the U.S. and other countries.

Federal Acquisitions: Commercial Software - Government Users
Subject to Standard License Terms and Conditions.
</xsd:documentation>
    </xsd:annotation>
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[

This is the XML Schema for the Servlet 2.4 deployment descriptor.

The deployment descriptor must be named "WEB-INF/web.xml" in the
web application's war file. All Servlet deployment descriptors
must indicate the web application schema by using the J2EE
namespace:

http://java.sun.com/xml/ns/j2ee
and by indicating the version of the schema by
using the version element as shown below:
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="..."

version="2.4">

...
</web-app>

The instance documents may indicate the published version of
the schema using the xsi:schemaLocation attribute for J2EE
namespace with the following location:

http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
]]>
```

```
</xsd:documentation>
</xsd:annotation>
<xsd:annotation>
    <xsd:documentation>

The following conventions apply to all J2EE
deployment descriptor elements unless indicated otherwise.

- In elements that specify a pathname to a file within the
same JAR file, relative filenames (i.e., those not
starting with "/") are considered relative to the root of
the JAR file's namespace. Absolute filenames (i.e., those
starting with "/") also specify names in the root of the
JAR file's namespace. In general, relative names are
preferred. The exception is .war files where absolute
names are preferred for consistency with the Servlet API.
</xsd:documentation>
</xsd:annotation>
<xsd:include schemaLocation="j2ee_1_4.xsd" />
<xsd:include schemaLocation="jsp_2_0.xsd" />
<!--
**************************************************
>
<xsd:element name="web-app" type="j2ee:web-appType">
<xsd:annotation>
<xsd:documentation>

The web-app element is the root of the deployment
descriptor for a web application. Note that the sub-elements
of this element can be in the arbitrary order. Because of
that, the multiplicity of the elements of distributable,

session-config, welcome-file-list, jsp-config, login-config,

and locale-encoding-mapping-list was changed from "?" to "*"

in this schema. However, the deployment descriptor instance
file must not contain multiple elements of session-config,

jsp-config, and login-config. When there are multiple elements of
```

```
welcome-file-list or locale-encoding-mapping-list, the container
must concatinate the element contents. The multiple occurance
of the element distributable is redundant and the container
treats that case exactly in the same way when there is only
one distributable.
</xsd:documentation>
</xsd:annotation>
<xsd:unique name="web-app-servlet-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The servlet element contains the name of a servlet.

The name must be unique within the web application.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:servlet"/>
<xsd:field xpath="j2ee:servlet-name"/>
</xsd:unique>
<xsd:unique name="web-app-filter-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The filter element contains the name of a filter.

The name must be unique within the web application.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:filter"/>
<xsd:field xpath="j2ee:filter-name"/>
</xsd:unique>
<xsd:unique name="web-app-ejb-local-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-local-ref-name element contains the name of an EJB
reference. The EJB reference is an entry in the web
application's environment and is relative to the
java:comp/env context. The name must be unique within
the web application.
```

```
It is recommended that name is prefixed with "ejb/".
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:ejb-local-ref"/>
<xsd:field xpath="j2ee:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="web-app-ejb-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-ref-name element contains the name of an EJB
reference. The EJB reference is an entry in the web
application's environment and is relative to the
java:comp/env context. The name must be unique within
the web application.

It is recommended that name is prefixed with "ejb/".
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:ejb-ref"/>
<xsd:field xpath="j2ee:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="web-app-resource-env-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>

The resource-env-ref-name element specifies the name of
a resource environment reference; its value is the
environment entry name used in the web application code.

The name is a JNDI name relative to the java:comp/env
context and must be unique within a web application.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:resource-env-ref"/>
<xsd:field xpath="j2ee:resource-env-ref-name"/>
</xsd:unique>
<xsd:unique name="web-app-message-destination-ref-uniqueness">
<xsd:annotation>
```

```
<xsd:documentation>

The message-destination-ref-name element specifies the name of
a message destination reference; its value is the
environment entry name used in the web application code.

The name is a JNDI name relative to the java:comp/env
context and must be unique within a web application.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:message-destination-ref"/>
<xsd:field xpath="j2ee:message-destination-ref-name"/>
</xsd:unique>
<xsd:unique name="web-app-res-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The res-ref-name element specifies the name of a
resource manager connection factory reference. The name
is a JNDI name relative to the java:comp/env context.

The name must be unique within a web application.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:resource-ref"/>
<xsd:field xpath="j2ee:res-ref-name"/>
</xsd:unique>
<xsd:unique name="web-app-env-entry-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The env-entry-name element contains the name of a web
application's environment entry. The name is a JNDI
name relative to the java:comp/env context. The name
must be unique within a web application.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:env-entry"/>
<xsd:field xpath="j2ee:env-entry-name"/>
</xsd:unique>
```

```
<xsd:key name="web-app-role-name-key">
<xsd:annotation>
<xsd:documentation>

A role-name-key is specified to allow the references
from the security-role-refs.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:security-role"/>
<xsd:field xpath="j2ee:role-name"/>
</xsd:key>
<xsd:keyref name="web-app-role-name-references"

refer="j2ee:web-app-role-name-key">
<xsd:annotation>
<xsd:documentation>

The keyref indicates the references from
security-role-ref to a specified role-name.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:servlet/j2ee:security-role-ref"/>
<xsd:field xpath="j2ee:role-link"/>
</xsd:keyref>
</xsd:element>
<!--
**************************************************
>
<xsd:complexType name="auth-constraintType">
<xsd:annotation>
<xsd:documentation>

The auth-constraintType indicates the user roles that
should be permitted access to this resource
collection. The role-name used here must either correspond
to the role-name of one of the security-role elements
defined for this web application, or be the specially
reserved role-name "*" that is a compact syntax for
indicating all roles in the web application. If both "*"
```

```
and rolenames appear, the container interprets this as all
roles. If no roles are defined, no user is allowed access
to the portion of the web application described by the
containing security-constraint. The container matches
role names case sensitively when determining access.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="j2ee:descriptionType"

minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="role-name"

type="j2ee:role-nameType"

minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="auth-methodType">
<xsd:annotation>
<xsd:documentation>

The auth-methodType is used to configure the authentication
mechanism for the web application. As a prerequisite to
gaining access to any web resources which are protected by
an authorization constraint, a user must have authenticated
using the configured mechanism. Legal values are "BASIC",

"DIGEST", "FORM", "CLIENT-CERT", or a vendor-specific
authentication scheme.

Used in: login-config
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:simpleContent>
<xsd:restriction base="j2ee:string"/>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="dispatcherType">
<xsd:annotation>
<xsd:documentation>

The dispatcher has four legal values: FORWARD, REQUEST, INCLUDE,

and ERROR. A value of FORWARD means the Filter will be applied
under RequestDispatcher.forward() calls. A value of REQUEST
means the Filter will be applied under ordinary client calls to
the path or servlet. A value of INCLUDE means the Filter will be
applied under RequestDispatcher.include() calls. A value of
ERROR means the Filter will be applied under the error page
mechanism. The absence of any dispatcher elements in a
filter-mapping indicates a default of applying filters only under
ordinary client calls to the path or servlet.
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="j2ee:string">
<xsd:enumeration value="FORWARD"/>
<xsd:enumeration value="INCLUDE"/>
<xsd:enumeration value="REQUEST"/>
<xsd:enumeration value="ERROR"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:simpleType name="encodingType">
<xsd:annotation>
<xsd:documentation>
```

```
The encodingType defines IANA character sets.
</xsd:documentation>
</xsd:annotation>
<xsd:restriction base="xsd:string">
<xsd:pattern value="[^\\s]+"/>
</xsd:restriction>
</xsd:simpleType>
<!--
**************************************************
>
<xsd:complexType name="error-codeType">
<xsd:annotation>
<xsd:documentation>

The error-code contains an HTTP error code, ex: 404
Used in: error-page
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="j2ee:xsdPositiveIntegerType">
<xsd:pattern value="\\d{3}"/>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="error-pageType">
<xsd:annotation>
<xsd:documentation>

The error-pageType contains a mapping between an error code
or exception type to the path of a resource in the web
application.

Used in: web-app
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
```

```
<xsd:choice>
<xsd:element name="error-code"

type="j2ee:error-codeType"/>
<xsd:element name="exception-type"

type="j2ee:fully-qualified-classType">
<xsd:annotation>
<xsd:documentation>

The exception-type contains a fully qualified class
name of a Java exception type.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:choice>
<xsd:element name="location"

type="j2ee:war-pathType">
<xsd:annotation>
<xsd:documentation>

The location element contains the location of the
resource in the web application relative to the root of
the web application. The value of the location must have
a leading `/'.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="filter-mappingType">
<xsd:annotation>
<xsd:documentation>

Declaration of the filter mappings in this web
```

```
application is done by using filter-mappingType.

The container uses the filter-mapping
declarations to decide which filters to apply to a request,

and in what order. The container matches the request URI to
a Servlet in the normal way. To determine which filters to
apply it matches filter-mapping declarations either on
servlet-name, or on url-pattern for each filter-mapping
element, depending on which style is used. The order in
which filters are invoked is the order in which
filter-mapping declarations that match a request URI for a
servlet appear in the list of filter-mapping elements.The
filter-name value must be the value of the filter-name
sub-elements of one of the filter declarations in the
deployment descriptor.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="filter-name"

type="j2ee:filter-nameType"/>
<xsd:choice>
<xsd:element name="url-pattern"

type="j2ee:url-patternType"/>
<xsd:element name="servlet-name"

type="j2ee:servlet-nameType"/>
</xsd:choice>
<xsd:element name="dispatcher"

type="j2ee:dispatcherType"

minOccurs="0" maxOccurs="4"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
```

```
>
<xsd:complexType name="filter-nameType">
<xsd:annotation>
<xsd:documentation>

The logical name of the filter is declare
by using filter-nameType. This name is used to map the
filter. Each filter name is unique within the web
application.

Used in: filter, filter-mapping
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:extension base="j2ee:nonEmptyStringType"/>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="filterType">
<xsd:annotation>
<xsd:documentation>

The filterType is used to declare a filter in the web
application. The filter is mapped to either a servlet or a
URL pattern in the filter-mapping element, using the
filter-name value to reference. Filters can access the
initialization parameters declared in the deployment
descriptor at runtime via the FilterConfig interface.

Used in: web-app
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="j2ee:descriptionGroup"/>
<xsd:element name="filter-name"

type="j2ee:filter-nameType"/>
<xsd:element name="filter-class"
```

```
type="j2ee:fully-qualified-classType">
<xsd:annotation>
<xsd:documentation>

The fully qualified classname of the filter.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="init-param"

type="j2ee:param-valueType"

minOccurs="0" maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>

The init-param element contains a name/value pair as
an initialization param of a servlet filter
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="form-login-configType">
<xsd:annotation>
<xsd:documentation>

The form-login-configType specifies the login and error
pages that should be used in form based login. If form based
authentication is not used, these elements are ignored.

Used in: login-config
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
```

```
<xsd:element name="form-login-page"

type="j2ee:war-pathType">
<xsd:annotation>
<xsd:documentation>

The form-login-page element defines the location in the web
app where the page that can be used for login can be
found. The path begins with a leading / and is interpreted
relative to the root of the WAR.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="form-error-page"

type="j2ee:war-pathType">
<xsd:annotation>
<xsd:documentation>

The form-error-page element defines the location in
the web app where the error page that is displayed
when login is not successful can be found.

The path begins with a leading / and is interpreted
relative to the root of the WAR.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="http-methodType">
<xsd:annotation>
<xsd:documentation>

The http-method contains an HTTP method recognized by the
web-app, for example GET, POST, ...
```

```
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="j2ee:string">
<xsd:enumeration value="GET"/>
<xsd:enumeration value="POST"/>
<xsd:enumeration value="PUT"/>
<xsd:enumeration value="DELETE"/>
<xsd:enumeration value="HEAD"/>
<xsd:enumeration value="OPTIONS"/>
<xsd:enumeration value="TRACE"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="locale-encoding-mapping-listType">
<xsd:annotation>
<xsd:documentation>

The locale-encoding-mapping-list contains one or more
locale-encoding-mapping(s).
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="locale-encoding-mapping"

type="j2ee:locale-encoding-mappingType"

maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="locale-encoding-mappingType">
<xsd:annotation>
<xsd:documentation>
```

```
The locale-encoding-mapping contains locale name and
encoding name. The locale name must be either "Language-code",

such as "ja", defined by ISO-639 or "Language-code_Country-code",

such as "ja_JP". "Country code" is defined by ISO-3166.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="locale"

type="j2ee:localeType"/>
<xsd:element name="encoding"

type="j2ee:encodingType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:simpleType name="localeType">
<xsd:annotation>
<xsd:documentation>

The localeType defines valid locale defined by ISO-639-1
and ISO-3166.
</xsd:documentation>
</xsd:annotation>
<xsd:restriction base="xsd:string">
<xsd:pattern value="[a-z]{2}(_|-)?([\\p{L}\\-\\p{Nd}]{2})?"/>
</xsd:restriction>
</xsd:simpleType>
<!--
**************************************************
>
<xsd:complexType name="login-configType">
<xsd:annotation>
<xsd:documentation>
```

```
The login-configType is used to configure the authentication
method that should be used, the realm name that should be
used for this application, and the attributes that are
needed by the form login mechanism.

Used in: web-app
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="auth-method"

type="j2ee:auth-methodType"

minOccurs="0"/>
<xsd:element name="realm-name"

type="j2ee:string" minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The realm name element specifies the realm name to
use in HTTP Basic authorization.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="form-login-config"

type="j2ee:form-login-configType"

minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="mime-mappingType">
<xsd:annotation>
<xsd:documentation>
```

```
The mime-mappingType defines a mapping between an extension
and a mime type.

Used in: web-app
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:annotation>
<xsd:documentation>

The extension element contains a string describing an
extension. example: "txt"
</xsd:documentation>
</xsd:annotation>
<xsd:element name="extension"

type="j2ee:string"/>
<xsd:element name="mime-type"

type="j2ee:mime-typeType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="mime-typeType">
<xsd:annotation>
<xsd:documentation>

The mime-typeType is used to indicate a defined mime type.

Example:

"text/plain"

Used in: mime-mapping
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:simpleContent>
<xsd:restriction base="j2ee:string">
<xsd:pattern value="[\\p{L}\\-\\p{Nd}]+/[\\p{L}\\-\\p{Nd}\\.]+"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="nonEmptyStringType">
<xsd:annotation>
<xsd:documentation>

This type defines a string which contains at least one
character.
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="j2ee:string">
<xsd:minLength value="1"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="security-constraintType">
<xsd:annotation>
<xsd:documentation>

The security-constraintType is used to associate
security constraints with one or more web resource
collections
Used in: web-app
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="display-name"

type="j2ee:display-nameType"
```

```
minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="web-resource-collection"

type="j2ee:web-resource-collectionType"

maxOccurs="unbounded"/>
<xsd:element name="auth-constraint"

type="j2ee:auth-constraintType"

minOccurs="0"/>
<xsd:element name="user-data-constraint"

type="j2ee:user-data-constraintType"

minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="servlet-mappingType">
<xsd:annotation>
<xsd:documentation>

The servlet-mappingType defines a mapping between a
servlet and a url pattern.

Used in: web-app
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="servlet-name"

type="j2ee:servlet-nameType"/>
<xsd:element name="url-pattern"
```

```
type="j2ee:url-patternType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="servlet-nameType">
<xsd:annotation>
<xsd:documentation>

The servlet-name element contains the canonical name of the
servlet. Each servlet name is unique within the web
application.
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:extension base="j2ee:nonEmptyStringType"/>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="servletType">
<xsd:annotation>
<xsd:documentation>

The servletType is used to declare a servlet.

It contains the declarative data of a
servlet. If a jsp-file is specified and the load-on-startup
element is present, then the JSP should be precompiled and
loaded.

Used in: web-app
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="j2ee:descriptionGroup"/>
```

```
<xsd:element name="servlet-name"

type="j2ee:servlet-nameType"/>
<xsd:choice>
<xsd:element name="servlet-class"

type="j2ee:fully-qualified-classType">
<xsd:annotation>
<xsd:documentation>

The servlet-class element contains the fully
qualified class name of the servlet.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="jsp-file"

type="j2ee:jsp-fileType"/>
</xsd:choice>
<xsd:element name="init-param"

type="j2ee:param-valueType"

minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="load-on-startup"

type="j2ee:xsdIntegerType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The load-on-startup element indicates that this
servlet should be loaded (instantiated and have
its init() called) on the startup of the web
application. The optional contents of these
element must be an integer indicating the order in
which the servlet should be loaded. If the value
is a negative integer, or the element is not
present, the container is free to load the servlet
```

```
whenever it chooses. If the value is a positive
integer or 0, the container must load and
initialize the servlet as the application is
deployed. The container must guarantee that
servlets marked with lower integers are loaded
before servlets marked with higher integers. The
container may choose the order of loading of
servlets with the same load-on-start-up value.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="run-as"

type="j2ee:run-asType"

minOccurs="0"/>
<xsd:element name="security-role-ref"

type="j2ee:security-role-refType"

minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="session-configType">
<xsd:annotation>
<xsd:documentation>

The session-configType defines the session parameters
for this web application.

Used in: web-app
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="session-timeout"
```

```
type="j2ee:xsdIntegerType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The session-timeout element defines the default
session timeout interval for all sessions created
in this web application. The specified timeout
must be expressed in a whole number of minutes.

If the timeout is 0 or less, the container ensures
the default behaviour of sessions is never to time
out. If this element is not specified, the container
must set its default timeout period.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
*****************************************************
>
<xsd:complexType name="transport-guaranteeType">
<xsd:annotation>
<xsd:documentation>

The transport-guaranteeType specifies that the communication
between client and server should be NONE, INTEGRAL, or
CONFIDENTIAL. NONE means that the application does not
require any transport guarantees. A value of INTEGRAL means
that the application requires that the data sent between the
client and server be sent in such a way that it can't be
changed in transit. CONFIDENTIAL means that the application
requires that the data be transmitted in a fashion that
prevents other entities from observing the contents of the
transmission. In most cases, the presence of the INTEGRAL or
CONFIDENTIAL flag will indicate that the use of SSL is
required.
```

```
Used in: user-data-constraint
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="j2ee:string">
<xsd:enumeration value="NONE"/>
<xsd:enumeration value="INTEGRAL"/>
<xsd:enumeration value="CONFIDENTIAL"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="user-data-constraintType">
<xsd:annotation>
<xsd:documentation>

The user-data-constraintType is used to indicate how
data communicated between the client and container should be
protected.

Used in: security-constraint
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="j2ee:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="transport-guarantee"

type="j2ee:transport-guaranteeType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

```
<!--
**************************************************
>
<xsd:complexType name="war-pathType">
<xsd:annotation>
<xsd:documentation>

The elements that use this type designate a path starting
with a "/" and interpreted relative to the root of a WAR
file.
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="j2ee:string">
<xsd:pattern value="/.*"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:simpleType name="web-app-versionType">
<xsd:annotation>
<xsd:documentation>

This type contains the recognized versions of
web-application supported. It is used to designate the
version of the web application.
</xsd:documentation>
</xsd:annotation>
<xsd:restriction base="xsd:token">
<xsd:enumeration value="2.4"/>
</xsd:restriction>
</xsd:simpleType>
<!--
**************************************************
>
<xsd:complexType name="web-appType">
<xsd:choice minOccurs="0" maxOccurs="unbounded">
<xsd:group ref="j2ee:descriptionGroup"/>
```

```
<xsd:element name="distributable"

type="j2ee:emptyType"/>
<xsd:element name="context-param"

type="j2ee:param-valueType">
<xsd:annotation>
<xsd:documentation>

The context-param element contains the declaration
of a web application's servlet context
initialization parameters.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="filter"

type="j2ee:filterType"/>
<xsd:element name="filter-mapping"

type="j2ee:filter-mappingType"/>
<xsd:element name="listener"

type="j2ee:listenerType"/>
<xsd:element name="servlet"

type="j2ee:servletType"/>
<xsd:element name="servlet-mapping"

type="j2ee:servlet-mappingType"/>
<xsd:element name="session-config"

type="j2ee:session-configType"/>
<xsd:element name="mime-mapping"

type="j2ee:mime-mappingType"/>
<xsd:element name="welcome-file-list"

type="j2ee:welcome-file-listType"/>
<xsd:element name="error-page"
```

```
type="j2ee:error-pageType"/>
<xsd:element name="jsp-config"

type="j2ee:jsp-configType"/>
<xsd:element name="security-constraint"

type="j2ee:security-constraintType"/>
<xsd:element name="login-config"

type="j2ee:login-configType"/>
<xsd:element name="security-role"

type="j2ee:security-roleType"/>
<xsd:group ref="j2ee:jndiEnvironmentRefsGroup"/>
<xsd:element name="message-destination"

type="j2ee:message-destinationType"/>
<xsd:element name="locale-encoding-mapping-list"

type="j2ee:locale-encoding-mapping-listType"/>
</xsd:choice>
<xsd:attribute name="version"

type="j2ee:web-app-versionType"

use="required"/>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="web-resource-collectionType">
<xsd:annotation>
<xsd:documentation>

The web-resource-collectionType is used to identify a subset
of the resources and HTTP methods on those resources within
a web application to which a security constraint applies. If
no HTTP methods are specified, then the security constraint
```

```
applies to all HTTP methods.

Used in: security-constraint
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="web-resource-name"

type="j2ee:string">
<xsd:annotation>
<xsd:documentation>

The web-resource-name contains the name of this web
resource collection.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="description"

type="j2ee:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="url-pattern"

type="j2ee:url-patternType"

maxOccurs="unbounded"/>
<xsd:element name="http-method"

type="j2ee:http-methodType"

minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
```

```
<xsd:complexType name="welcome-file-listType">
<xsd:annotation>
<xsd:documentation>

The welcome-file-list contains an ordered list of welcome
files elements.

Used in: web-app
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="welcome-file"

type="xsd:string"

maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>

The welcome-file element contains file name to use
as a default welcome file, such as index.html
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
</xsd:schema>
```

17.1.9 ejb-jar_3_0.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://java.sun.com/xml/ns/Java EE"
    xmlns:JavaEE="http://java.sun.com/xml/ns/Java EE"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified"
version="3.0">
    <xsd:annotation>
```

```
        <xsd:documentation>

@(#)ejb-jar_3_0.xsds 1.50 02/07/06
</xsd:documentation>
    </xsd:annotation>
    <xsd:annotation>
        <xsd:documentation>

Copyright 2003-2006 Sun Microsystems, Inc.

4150 Network Circle
Santa Clara, California 95054
U.S.A
All rights reserved.

Sun Microsystems, Inc. has intellectual property rights
relating to technology described in this document. In
particular, and without limitation, these intellectual
property rights may include one or more of the U.S. patents
listed at http://www.sun.com/patents and one or more
additional patents or pending patent applications in the
U.S. and other countries.

This document and the technology which it describes are
distributed under licenses restricting their use, copying,

distribution, and decompilation. No part of this document
may be reproduced in any form by any means without prior
written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is
copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Solaris, Java, J2EE,

JavaServer Pages, Enterprise JavaBeans and the Java Coffee
Cup logo are trademarks or registered trademarks of Sun
Microsystems, Inc. in the U.S. and other countries.

Federal Acquisitions: Commercial Software - Government Users
```

```
Subject to Standard License Terms and Conditions.
</xsd:documentation>
    </xsd:annotation>
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[

This is the XML Schema for the EJB 3.0 deployment descriptor.

The deployment descriptor must be named "META-INF/ejb-jar.xml" in
the EJB's jar file. All EJB deployment descriptors must indicate
the ejb-jar schema by using the Java EE namespace:

http://java.sun.com/xml/ns/Java EE
and by indicating the version of the schema by
using the version element as shown below:
<ejb-jar xmlns="http://java.sun.com/xml/ns/Java EE"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/Java EE
http://java.sun.com/xml/ns/Java EE/ejb-jar_3_0.xsd"

version="3.0">

...
</ejb-jar>

The instance documents may indicate the published version of
the schema using the xsi:schemaLocation attribute for the
Java EE namespace with the following location:

http://java.sun.com/xml/ns/Java EE/ejb-jar_3_0.xsd
]]>
</xsd:documentation>
</xsd:annotation>
<xsd:annotation>
    <xsd:documentation>

The following conventions apply to all Java EE
```

```
deployment descriptor elements unless indicated otherwise.

- In elements that specify a pathname to a file within the
same JAR file, relative filenames (i.e., those not
starting with "/") are considered relative to the root of
the JAR file's namespace. Absolute filenames (i.e., those
starting with "/") also specify names in the root of the
JAR file's namespace. In general, relative names are
preferred. The exception is .war files where absolute
names are preferred for consistency with the Servlet API.
</xsd:documentation>
</xsd:annotation>
<xsd:include schemaLocation="Java EE_5.xsd" />
<!--
**************************************************
>
<xsd:element name="ejb-jar" type="Java EE:ejb-jarType">
<xsd:annotation>
<xsd:documentation>

This is the root of the ejb-jar deployment descriptor.
</xsd:documentation>
</xsd:annotation>
<xsd:key name="ejb-name-key">
<xsd:annotation>
<xsd:documentation>

The ejb-name element contains the name of an enterprise
bean. The name must be unique within the ejb-jar file.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:enterprise-beans/*"/>
<xsd:field xpath="Java EE:ejb-name"/>
</xsd:key>
<xsd:keyref name="ejb-name-references"

refer="Java EE:ejb-name-key">
<xsd:annotation>
<xsd:documentation>
```

```
The keyref indicates the references from
relationship-role-source must be to a specific ejb-name
defined within the scope of enterprise-beans element.
</xsd:documentation>
</xsd:annotation>
<xsd:selector
xpath=".//Java EE:ejb-relationship-role/Java EE:relationship-role-source"/>
<xsd:field
xpath="Java EE:ejb-name"/>
</xsd:keyref>
<xsd:key name="role-name-key">
<xsd:annotation>
<xsd:documentation>

A role-name-key is specified to allow the references
from the security-role-refs.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:assembly-descriptor/Java EE:security-role"/>
<xsd:field xpath="Java EE:role-name"/>
</xsd:key>
<xsd:keyref name="role-name-references"

refer="Java EE:role-name-key">
<xsd:annotation>
<xsd:documentation>

The keyref indicates the references from
security-role-ref to a specified role-name.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:enterprise-beans/*/Java EE:security-role-ref"/>
<xsd:field xpath="Java EE:role-link"/>
</xsd:keyref>
</xsd:element>
<!--
**************************************************
>
<xsd:complexType name="activation-config-propertyType">
<xsd:annotation>
```

```
<xsd:documentation>

The activation-config-propertyType contains a name/value
configuration property pair for a message-driven bean.

The properties that are recognized for a particular
message-driven bean are determined by the messaging type.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="activation-config-property-name"

type="Java EE:xsdStringType">
<xsd:annotation>
<xsd:documentation>

The activation-config-property-name element contains
the name for an activation configuration property of
a message-driven bean.

For JMS message-driven beans, the following property
names are recognized: acknowledgeMode,

messageSelector, destinationType, subscriptionDurability
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="activation-config-property-value"

type="Java EE:xsdStringType">
<xsd:annotation>
<xsd:documentation>

The activation-config-property-value element
contains the value for an activation configuration
property of a message-driven bean.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
```

```
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="activation-configType">
<xsd:annotation>
<xsd:documentation>

The activation-configType defines information about the
expected configuration properties of the message-driven bean
in its operational environment. This may include information
about message acknowledgement, message selector, expected
destination type, etc.

The configuration information is expressed in terms of
name/value configuration properties.

The properties that are recognized for a particular
message-driven bean are determined by the messaging type.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="activation-config-property"

type="Java EE:activation-config-propertyType"

maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
```

```
>
<xsd:complexType name="application-exceptionType">
<xsd:annotation>
<xsd:documentation>

The application-exceptionType declares an application
exception. The declaration consists of:

- the exception class. When the container receives
an exception of this type, it is required to
forward this exception as an applcation exception
to the client regardless of whether it is a checked
or unchecked exception.

- an optional rollback element. If this element is
set to true, the container must rollback the current
transaction before forwarding the exception to the
client.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="exception-class"

type="Java EE:fully-qualified-classType"/>
<xsd:element name="rollback"

type="Java EE:true-falseType"

minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="around-invokeType">
<xsd:annotation>
<xsd:documentation>

The around-invoke type specifies a method on a
```

```
class to be called during the around invoke portion of an
ejb invocation. Note that each class may have only one
around invoke method and that the method may not be
overloaded.

If the around-invoke element is missing then
the class defining the callback is assumed to be the
interceptor class or component class in scope at the
location in the descriptor in which the around invoke
definition appears.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="class"

type="Java EE:fully-qualified-classType"

minOccurs="0"/>
<xsd:element name="method-name"

type="Java EE:java-identifierType"/>
</xsd:sequence>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="assembly-descriptorType">
<xsd:annotation>
<xsd:documentation>

The assembly-descriptorType defines
application-assembly information.

The application-assembly information consists of the
following parts: the definition of security roles, the
definition of method permissions, the definition of
transaction attributes for enterprise beans with
container-managed transaction demarcation, the definition
of interceptor bindings, a list of
methods to be excluded from being invoked, and a list of
```

```
exception types that should be treated as application exceptions.

All the parts are optional in the sense that they are
omitted if the lists represented by them are empty.

Providing an assembly-descriptor in the deployment
descriptor is optional for the ejb-jar file producer.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="security-role"

type="Java EE:security-roleType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="method-permission"

type="Java EE:method-permissionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="container-transaction"

type="Java EE:container-transactionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="interceptor-binding"

type="Java EE:interceptor-bindingType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="message-destination"
```

```
type="Java EE:message-destinationType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="exclude-list"

type="Java EE:exclude-listType"

minOccurs="0"/>
<xsd:element name="application-exception"

type="Java EE:application-exceptionType"

minOccurs="0"

maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="cmp-fieldType">
<xsd:annotation>
<xsd:documentation>

The cmp-fieldType describes a container-managed field. The
cmp-fieldType contains an optional description of the field,

and the name of the field.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"
```

```
maxOccurs="unbounded"/>
<xsd:element name="field-name"

type="Java EE:java-identifierType">
<xsd:annotation>
<xsd:documentation>

The field-name element specifies the name of a
container managed field.

The name of the cmp-field of an entity bean with
cmp-version 2.x must begin with a lowercase
letter. This field is accessed by methods whose
names consists of the name of the field specified by
field-name in which the first letter is uppercased,

prefixed by "get" or "set".

The name of the cmp-field of an entity bean with
cmp-version 1.x must denote a public field of the
enterprise bean class or one of its superclasses.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="cmp-versionType">
<xsd:annotation>
<xsd:documentation>

The cmp-versionType specifies the version of an entity bean
with container-managed persistence. It is used by
cmp-version elements.

The value must be one of the two following:
```

```
1.x
2.x
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:string">
<xsd:enumeration value="1.x"/>
<xsd:enumeration value="2.x"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="cmr-field-typeType">
<xsd:annotation>
<xsd:documentation>

The cmr-field-type element specifies the class of a
collection-valued logical relationship field in the entity
bean class. The value of an element using cmr-field-typeType
must be either: java.util.Collection or java.util.Set.
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:string">
<xsd:enumeration value="java.util.Collection"/>
<xsd:enumeration value="java.util.Set"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="cmr-fieldType">
<xsd:annotation>
<xsd:documentation>

The cmr-fieldType describes the bean provider's view of
a relationship. It consists of an optional description, and
```

```
the name and the class type of a field in the source of a
role of a relationship. The cmr-field-name element
corresponds to the name used for the get and set accessor
methods for the relationship. The cmr-field-type element is
used only for collection-valued cmr-fields. It specifies the
type of the collection that is used.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="cmr-field-name"

type="Java EE:string">
<xsd:annotation>
<xsd:documentation>

The cmr-field-name element specifies the name of a
logical relationship field in the entity bean
class. The name of the cmr-field must begin with a
lowercase letter. This field is accessed by methods
whose names consist of the name of the field
specified by cmr-field-name in which the first
letter is uppercased, prefixed by "get" or "set".
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="cmr-field-type"

type="Java EE:cmr-field-typeType"

minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

```
<!--
**************************************************
>
<xsd:complexType name="container-transactionType">
<xsd:annotation>
<xsd:documentation>

The container-transactionType specifies how the container
must manage transaction scopes for the enterprise bean's
method invocations. It defines an optional description, a
list of method elements, and a transaction attribute. The
transaction attribute is to be applied to all the specified
methods.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="method"

type="Java EE:methodType"

maxOccurs="unbounded"/>
<xsd:element name="trans-attribute"

type="Java EE:trans-attributeType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="ejb-classType">
<xsd:annotation>
<xsd:documentation>
```

```
<![CDATA[

The ejb-classType contains the fully-qualified name of the
enterprise bean's class. It is used by ejb-class elements.

Example:
<ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>

]]>
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:fully-qualified-classType"/>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="ejb-jarType">
<xsd:annotation>
<xsd:documentation>

The ejb-jarType defines the root element of the EJB
deployment descriptor. It contains
- an optional description of the ejb-jar file
- an optional display name
- an optional icon that contains a small and a large
icon file name
- structural information about all included
enterprise beans that is not specified through
annotations
- structural information about interceptor classes
- a descriptor for container managed relationships,

if any.

- an optional application-assembly descriptor
- an optional name of an ejb-client-jar file for the
ejb-jar.
</xsd:documentation>
```

```
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="Java EE:descriptionGroup"/>
<xsd:element name="enterprise-beans"

type="Java EE:enterprise-beansType"

minOccurs="0"/>
<xsd:element name="interceptors"

type="Java EE:interceptorsType"

minOccurs="0"/>
<xsd:element name="relationships"

type="Java EE:relationshipsType"

minOccurs="0">
<xsd:unique name="relationship-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-relation-name contains the name of a
relation. The name must be unique within
relationships.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:ejb-relation"/>
<xsd:field xpath="Java EE:ejb-relation-name"/>
</xsd:unique>
</xsd:element>
<xsd:element name="assembly-descriptor"

type="Java EE:assembly-descriptorType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

Providing an assembly-descriptor in the deployment
```

```
descriptor is optional for the ejb-jar file
producer.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="ejb-client-jar"

type="Java EE:pathType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>
<![CDATA[

The optional ejb-client-jar element specifies a JAR
file that contains the class files necessary for a
client program to access the
enterprise beans in the ejb-jar file.

Example:
<ejb-client-jar>employee_service_client.jar
</ejb-client-jar>

]]>
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="version"

type="Java EE:dewey-versionType"

fixed="3.0"

use="required">
<xsd:annotation>
<xsd:documentation>

The version specifies the version of the
EJB specification that the instance document must
```

```
comply with. This information enables deployment tools
to validate a particular EJB Deployment
Descriptor with respect to a specific version of the EJB
schema.
</xsd:documentation>
</xsd:annotation>
</xsd:attribute>
<xsd:attribute name="metadata-complete" type="xsd:boolean">
<xsd:annotation>
<xsd:documentation>

The metadata-complete attribute defines whether this
deployment descriptor and other related deployment
descriptors for this module (e.g., web service
descriptors) are complete, or whether the class
files available to this module and packaged with
this application should be examined for annotations
that specify deployment information.

If metadata-complete is set to "true", the deployment
tool must ignore any annotations that specify deployment
information, which might be present in the class files
of the application.

If metadata-complete is not specified or is set to
"false", the deployment tool must examine the class
files of the application for annotations, as
specified by the specifications.
</xsd:documentation>
</xsd:annotation>
</xsd:attribute>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="ejb-nameType">
<xsd:annotation>
<xsd:documentation>
<![CDATA[
```

```
The ejb-nameType specifies an enterprise bean's name. It is
used by ejb-name elements. This name is assigned by the
ejb-jar file producer to name the enterprise bean in the
ejb-jar file's deployment descriptor. The name must be
unique among the names of the enterprise beans in the same
ejb-jar file.

There is no architected relationship between the used
ejb-name in the deployment descriptor and the JNDI name that
the Deployer will assign to the enterprise bean's home.

The name for an entity bean must conform to the lexical
rules for an NMTOKEN.

Example:
<ejb-name>EmployeeService</ejb-name>

]]>
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:xsdNMTOKENType"/>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="ejb-relationType">
<xsd:annotation>
<xsd:documentation>

The ejb-relationType describes a relationship between two
entity beans with container-managed persistence. It is used
by ejb-relation elements. It contains a description; an
optional ejb-relation-name element; and exactly two
relationship role declarations, defined by the
ejb-relationship-role elements. The name of the
relationship, if specified, is unique within the ejb-jar
file.
```

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="ejb-relation-name"

type="Java EE:string"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The ejb-relation-name element provides a unique name
within the ejb-jar file for a relationship.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="ejb-relationship-role"

type="Java EE:ejb-relationship-roleType"/>
<xsd:element name="ejb-relationship-role"

type="Java EE:ejb-relationship-roleType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="ejb-relationship-roleType">
<xsd:annotation>
<xsd:documentation>
<![CDATA[
```

```
The ejb-relationship-roleType describes a role within a
relationship. There are two roles in each relationship.

The ejb-relationship-roleType contains an optional
description; an optional name for the relationship role; a
specification of the multiplicity of the role; an optional
specification of cascade-delete functionality for the role;

the role source; and a declaration of the cmr-field, if any,

by means of which the other side of the relationship is
accessed from the perspective of the role source.

The multiplicity and role-source element are mandatory.

The relationship-role-source element designates an entity
bean by means of an ejb-name element. For bidirectional
relationships, both roles of a relationship must declare a
relationship-role-source element that specifies a cmr-field
in terms of which the relationship is accessed. The lack of
a cmr-field element in an ejb-relationship-role specifies
that the relationship is unidirectional in navigability and
the entity bean that participates in the relationship is
"not aware" of the relationship.

Example:
<ejb-relation>
<ejb-relation-name>Product-LineItem</ejb-relation-name>
<ejb-relationship-role>
<ejb-relationship-role-name>product-has-lineitems
</ejb-relationship-role-name>
<multiplicity>One</multiplicity>
<relationship-role-source>
<ejb-name>ProductEJB</ejb-name>
</relationship-role-source>
</ejb-relationship-role>
</ejb-relation>

]]>
</xsd:documentation>
```

```
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="ejb-relationship-role-name"

type="Java EE:string"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The ejb-relationship-role-name element defines a
name for a role that is unique within an
ejb-relation. Different relationships can use the
same name for a role.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="multiplicity"

type="Java EE:multiplicityType"/>
<xsd:element name="cascade-delete"

type="Java EE:emptyType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The cascade-delete element specifies that, within a
particular relationship, the lifetime of one or more
entity beans is dependent upon the lifetime of
another entity bean. The cascade-delete element can
only be specified for an ejb-relationship-role
```

```
element contained in an ejb-relation element in
which the other ejb-relationship-role
element specifies a multiplicity of One.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="relationship-role-source"

type="Java EE:relationship-role-sourceType"/>
<xsd:element name="cmr-field"

type="Java EE:cmr-fieldType"

minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="enterprise-beansType">
<xsd:annotation>
<xsd:documentation>

The enterprise-beansType declares one or more enterprise
beans. Each bean can be a session, entity or message-driven
bean.
</xsd:documentation>
</xsd:annotation>
<xsd:choice maxOccurs="unbounded">
<xsd:element name="session"

type="Java EE:session-beanType">
<xsd:unique name="session-ejb-local-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-ref-name element contains the name of
an EJB reference. The EJB reference is an entry in
the component's environment and is relative to the
```

```
java:comp/env context. The name must be unique within
the component.

It is recommended that name be prefixed with "ejb/".
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:ejb-local-ref"/>
<xsd:field xpath="Java EE:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="session-ejb-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-ref-name element contains the name of an EJB
reference. The EJB reference is an entry in the
component's environment and is relative to the
java:comp/env context. The name must be unique
within the component.

It is recommended that name is prefixed with "ejb/".
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:ejb-ref"/>
<xsd:field xpath="Java EE:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="session-resource-env-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>

The resource-env-ref-name element specifies the name
of a resource environment reference; its value is
the environment entry name used in the component
code. The name is a JNDI name relative to the
java:comp/env context and must be unique within an
component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:resource-env-ref"/>
<xsd:field xpath="Java EE:resource-env-ref-name"/>
</xsd:unique>
```

```
<xsd:unique name="session-message-destination-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>

The message-destination-ref-name element specifies the name
of a message destination reference; its value is
the message destination reference name used in the component
code. The name is a JNDI name relative to the
java:comp/env context and must be unique within an
component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:message-destination-ref"/>
<xsd:field xpath="Java EE:message-destination-ref-name"/>
</xsd:unique>
<xsd:unique name="session-res-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The res-ref-name element specifies the name of a
resource manager connection factory reference. The name
is a JNDI name relative to the java:comp/env context.

The name must be unique within an component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:resource-ref"/>
<xsd:field xpath="Java EE:res-ref-name"/>
</xsd:unique>
<xsd:unique name="session-env-entry-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The env-entry-name element contains the name of a
component's environment entry. The name is a JNDI
name relative to the java:comp/env context. The
name must be unique within an component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:env-entry"/>
```

```
<xsd:field xpath="Java EE:env-entry-name"/>
</xsd:unique>
</xsd:element>
<xsd:element name="entity"

type="Java EE:entity-beanType">
<xsd:unique name="entity-ejb-local-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-ref-name element contains the name of
an EJB reference. The EJB reference is an entry in
the component's environment and is relative to the
java:comp/env context. The name must be unique within
the component.

It is recommended that name be prefixed with "ejb/".
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:ejb-local-ref"/>
<xsd:field xpath="Java EE:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-ejb-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-ref-name element contains the name of an EJB
reference. The EJB reference is an entry in the
component's environment and is relative to the
java:comp/env context. The name must be unique
within the component.

It is recommended that name is prefixed with "ejb/".
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:ejb-ref"/>
<xsd:field xpath="Java EE:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-resource-env-ref-uniqueness">
<xsd:annotation>
```

```
<xsd:documentation>

The resource-env-ref-name element specifies the name
of a resource environment reference; its value is
the environment entry name used in the component
code. The name is a JNDI name relative to the
java:comp/env context and must be unique within an
component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:resource-env-ref"/>
<xsd:field xpath="Java EE:resource-env-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-message-destination-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>

The message-destination-ref-name element specifies the name
of a message destination reference; its value is
the message destination reference name used in the component
code. The name is a JNDI name relative to the
java:comp/env context and must be unique within an
component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:message-destination-ref"/>
<xsd:field xpath="Java EE:message-destination-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-res-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The res-ref-name element specifies the name of a
resource manager connection factory reference. The name
is a JNDI name relative to the java:comp/env context.

The name must be unique within an component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:resource-ref"/>
```

```
<xsd:field xpath="Java EE:res-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-env-entry-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The env-entry-name element contains the name of a
component's environment entry. The name is a JNDI
name relative to the java:comp/env context. The
name must be unique within an component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:env-entry"/>
<xsd:field xpath="Java EE:env-entry-name"/>
</xsd:unique>
</xsd:element>
<xsd:element name="message-driven"

type="Java EE:message-driven-beanType">
<xsd:unique name="messaged-ejb-local-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-ref-name element contains the name of
an EJB reference. The EJB reference is an entry in
the component's environment and is relative to the
java:comp/env context. The name must be unique within
the component.

It is recommended that name be prefixed with "ejb/".
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:ejb-local-ref"/>
<xsd:field xpath="Java EE:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="messaged-ejb-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-ref-name element contains the name of an EJB
```

```
reference. The EJB reference is an entry in the
component's environment and is relative to the
java:comp/env context. The name must be unique
within the component.

It is recommended that name is prefixed with "ejb/".
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:ejb-ref"/>
<xsd:field xpath="Java EE:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="messaged-resource-env-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>

The resource-env-ref-name element specifies the name
of a resource environment reference; its value is
the environment entry name used in the component
code. The name is a JNDI name relative to the
java:comp/env context and must be unique within an
component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:resource-env-ref"/>
<xsd:field xpath="Java EE:resource-env-ref-name"/>
</xsd:unique>
<xsd:unique name="messaged-message-destination-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>

The message-destination-ref-name element specifies the name
of a message destination reference; its value is
the message destination reference name used in the component
code. The name is a JNDI name relative to the
java:comp/env context and must be unique within an
component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:message-destination-ref"/>
<xsd:field xpath="Java EE:message-destination-ref-name"/>
```

```
</xsd:unique>
<xsd:unique name="messaged-res-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The res-ref-name element specifies the name of a
resource manager connection factory reference. The name
is a JNDI name relative to the java:comp/env context.

The name must be unique within an component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:resource-ref"/>
<xsd:field xpath="Java EE:res-ref-name"/>
</xsd:unique>
<xsd:unique name="messaged-env-entry-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The env-entry-name element contains the name of a
component's environment entry. The name is a JNDI
name relative to the java:comp/env context. The
name must be unique within an component.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="Java EE:env-entry"/>
<xsd:field xpath="Java EE:env-entry-name"/>
</xsd:unique>
</xsd:element>
</xsd:choice>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="entity-beanType">
<xsd:annotation>
<xsd:documentation>

The entity-beanType declares an entity bean. The declaration
```

```
consists of:

- an optional description
- an optional display name
- an optional icon element that contains a small and a large
icon file name
- a unique name assigned to the enterprise bean
in the deployment descriptor
- an optional mapped-name element that can be used to provide
vendor-specific deployment information such as the physical
jndi-name of the entity bean's remote home interface. This
element is not required to be supported by all implementations.

Any use of this element is non-portable.

- the names of the entity bean's remote home
and remote interfaces, if any
- the names of the entity bean's local home and local
interfaces, if any
- the entity bean's implementation class
- the optional entity bean's persistence management type. If
this element is not specified it is defaulted to Container.

- the entity bean's primary key class name
- an indication of the entity bean's reentrancy
- an optional specification of the
entity bean's cmp-version
- an optional specification of the entity bean's
abstract schema name
- an optional list of container-managed fields
- an optional specification of the primary key
field
- an optional declaration of the bean's environment
entries
- an optional declaration of the bean's EJB
references
- an optional declaration of the bean's local
EJB references
- an optional declaration of the bean's web
service references
```

```
- an optional declaration of the security role
references
- an optional declaration of the security identity
to be used for the execution of the bean's methods
- an optional declaration of the bean's
resource manager connection factory references
- an optional declaration of the bean's
resource environment references
- an optional declaration of the bean's message
destination references
- an optional set of query declarations
for finder and select methods for an entity
bean with cmp-version 2.x.

The optional abstract-schema-name element must be specified
for an entity bean with container-managed persistence and
cmp-version 2.x.

The optional primkey-field may be present in the descriptor
if the entity's persistence-type is Container.

The optional cmp-version element may be present in the
descriptor if the entity's persistence-type is Container. If
the persistence-type is Container and the cmp-version
element is not specified, its value defaults to 2.x.

The optional home and remote elements must be specified if
the entity bean cmp-version is 1.x.

The optional home and remote elements must be specified if
the entity bean has a remote home and remote interface.

The optional local-home and local elements must be specified
if the entity bean has a local home and local interface.

Either both the local-home and the local elements or both
the home and the remote elements must be specified.

The optional query elements must be present if the
persistence-type is Container and the cmp-version is 2.x and
```

```
query methods other than findByPrimaryKey have been defined
for the entity bean.

The other elements that are optional are "optional" in the
sense that they are omitted if the lists represented by them
are empty.

At least one cmp-field element must be present in the
descriptor if the entity's persistence-type is Container and
the cmp-version is 1.x, and none must not be present if the
entity's persistence-type is Bean.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="Java EE:descriptionGroup"/>
<xsd:element name="ejb-name"

type="Java EE:ejb-nameType"/>
<xsd:element name="mapped-name"

type="Java EE:xsdStringType"

minOccurs="0"/>
<xsd:element name="home"

type="Java EE:homeType"

minOccurs="0"/>
<xsd:element name="remote"

type="Java EE:remoteType"

minOccurs="0"/>
<xsd:element name="local-home"

type="Java EE:local-homeType"

minOccurs="0"/>
<xsd:element name="local"
```

```
type="Java EE:localType"

minOccurs="0"/>
<xsd:element name="ejb-class"

type="Java EE:ejb-classType"/>
<xsd:element name="persistence-type"

type="Java EE:persistence-typeType"/>
<xsd:element name="prim-key-class"

type="Java EE:fully-qualified-classType">
<xsd:annotation>
<xsd:documentation>

The prim-key-class element contains the
fully-qualified name of an
entity bean's primary key class.

If the definition of the primary key class is
deferred to deployment time, the prim-key-class
element should specify java.lang.Object.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="reentrant"

type="Java EE:true-falseType">
<xsd:annotation>
<xsd:documentation>

The reentrant element specifies whether an entity
bean is reentrant or not.

The reentrant element must be one of the two
following: true or false
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="cmp-version"
```

```
type="Java EE:cmp-versionType"

minOccurs="0"/>
<xsd:element name="abstract-schema-name"

type="Java EE:java-identifierType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The abstract-schema-name element specifies the name
of the abstract schema type of an entity bean with
cmp-version 2.x. It is used in EJB QL queries.

For example, the abstract-schema-name for an entity
bean whose local interface is
com.acme.commerce.Order might be Order.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="cmp-field"

type="Java EE:cmp-fieldType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="primkey-field"

type="Java EE:string"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The primkey-field element is used to specify the
name of the primary key field for an entity with
container-managed persistence.
```

```
The primkey-field must be one of the fields declared
in the cmp-field element, and the type of the field
must be the same as the primary key type.

The primkey-field element is not used if the primary
key maps to multiple container-managed fields
(i.e. the key is a compound key). In this case, the
fields of the primary key class must be public, and
their names must correspond to the field names of
the entity bean class that comprise the key.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:group ref="Java EE:jndiEnvironmentRefsGroup"/>
<xsd:element name="security-role-ref"

type="Java EE:security-role-refType"

minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="security-identity"

type="Java EE:security-identityType"

minOccurs="0"/>
<xsd:element name="query"

type="Java EE:queryType"

minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
************************************************
>
<xsd:complexType name="exclude-listType">
<xsd:annotation>
<xsd:documentation>
```

```
The exclude-listType specifies one or more methods which
the Assembler marks to be uncallable.

If the method permission relation contains methods that are
in the exclude list, the Deployer should consider those
methods to be uncallable.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="method"

type="Java EE:methodType"

maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="init-methodType">
<xsd:sequence>
<xsd:element name="create-method"

type="Java EE:named-methodType"/>
<xsd:element name="bean-method"

type="Java EE:named-methodType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
```

```
>
<xsd:complexType name="interceptor-bindingType">
<xsd:annotation>
<xsd:documentation>

The interceptor-bindingType element describes the binding of
interceptor classes to beans within the ejb-jar.

It consists of :

- An optional description.

- The name of an ejb within the ejb-jar or the wildcard value "*",

which is used to define interceptors that are bound to all
beans in the ejb-jar.

- A list of interceptor classes that are bound to the contents of
the ejb-name element or a specification of the total ordering
over the interceptors defined for the given level and above.

- An optional exclude-default-interceptors element. If set to true,

specifies that default interceptors are not to be applied to
a bean-class and/or business method.

- An optional exclude-class-interceptors element. If set to true,

specifies that class interceptors are not to be applied to
a business method.

- An optional set of method elements for describing the name/params
of a method-level interceptor.

Interceptors bound to all classes using the wildcard syntax
"*" are default interceptors for the components in the ejb-jar.

In addition, interceptors may be bound at the level of the bean
class (class-level interceptors) or business methods (method-level
interceptors ).
```

```
The binding of interceptors to classes is additive. If interceptors
are bound at the class-level and/or default-level as well as the
method-level, both class-level and/or default-level as well as
method-level will apply.

There are four possible styles of the interceptor element syntax :

1.
<interceptor-binding>
<ejb-name>*</ejb-name>
<interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>

Specifying the ejb-name as the wildcard value "*" designates
default interceptors (interceptors that apply to all session and
message-driven beans contained in the ejb-jar).

2.
<interceptor-binding>
<ejb-name>EJBNAME</ejb-name>
<interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>

This style is used to refer to interceptors associated with the
specified enterprise bean(class-level interceptors).

3.
<interceptor-binding>
<ejb-name>EJBNAME</ejb-name>
<interceptor-class>INTERCEPTOR</interceptor-class>
<method>
<method-name>METHOD</method-name>
</method>
</interceptor-binding>

This style is used to associate a method-level interceptor with
the specified enterprise bean. If there are multiple methods
with the same overloaded name, the element of this style refers
to all the methods with the overloaded name. Method-level
```

```
interceptors can only be associated with business methods of the
bean class. Note that the wildcard value "*" cannot be used
to specify method-level interceptors.

4.
<interceptor-binding>
<ejb-name>EJBNAME</ejb-name>
<interceptor-class>INTERCEPTOR</interceptor-class>
<method>
<method-name>METHOD</method-name>
<method-params>
<method-param>PARAM-1</method-param>
<method-param>PARAM-2</method-param>

...
<method-param>PARAM-N</method-param>
</method-params>
</method>
</interceptor-binding>

This style is used to associate a method-level interceptor with
the specified method of the specified enterprise bean. This
style is used to refer to a single method within a set of methods
with an overloaded name. The values PARAM-1 through PARAM-N
are the fully-qualified Java types of the method's input parameters
(if the method has no input arguments, the method-params element
contains no method-param elements). Arrays are specified by the
array element's type, followed by one or more pair of square
brackets (e.g. int[][]).
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="ejb-name"
```

```
type="Java EE:string"/>
<xsd:choice>
<xsd:element name="interceptor-class"

type="Java EE:fully-qualified-classType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="interceptor-order"

type="Java EE:interceptor-orderType"

minOccurs="1"/>
</xsd:choice>
<xsd:element name="exclude-default-interceptors"

type="Java EE:true-falseType"

minOccurs="0"/>
<xsd:element name="exclude-class-interceptors"

type="Java EE:true-falseType"

minOccurs="0"/>
<xsd:element name="method"

type="Java EE:named-methodType"

minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="interceptor-orderType">
<xsd:annotation>
<xsd:documentation>
```

```
The interceptor-orderType element describes a total ordering
of interceptor classes.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="interceptor-class"

type="Java EE:fully-qualified-classType"

minOccurs="1"

maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="interceptorType">
<xsd:annotation>
<xsd:documentation>

The interceptorType element declares information about a single
interceptor class. It consists of :

- An optional description.

- The fully-qualified name of the interceptor class.

- An optional list of around invoke methods declared on the
interceptor class and/or its super-classes.

- An optional list environment dependencies for the interceptor
class and/or its super-classes.

- An optional list of post-activate methods declared on the
interceptor class and/or its super-classes.

- An optional list of pre-passivate methods declared on the
```

```
interceptor class and/or its super-classes.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="interceptor-class"

type="Java EE:fully-qualified-classType"/>
<xsd:element name="around-invoke"

type="Java EE:around-invokeType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:group ref="Java EE:jndiEnvironmentRefsGroup"/>
<xsd:element name="post-activate"

type="Java EE:lifecycle-callbackType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="pre-passivate"

type="Java EE:lifecycle-callbackType"

minOccurs="0"

maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
```

```
**********************************************
>
<xsd:complexType name="interceptorsType">
<xsd:annotation>
<xsd:documentation>

The interceptorsType element declares one or more interceptor
classes used by components within this ejb-jar. The declaration
consists of :

- An optional description.

- One or more interceptor elements.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="interceptor"

type="Java EE:interceptorType"

maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**********************************************
>
<xsd:complexType name="message-driven-beanType">
<xsd:annotation>
<xsd:documentation>

The message-driven element declares a message-driven
bean. The declaration consists of:
```

```
- an optional description
- an optional display name
- an optional icon element that contains a small and a large
icon file name.

- a name assigned to the enterprise bean in
the deployment descriptor
- an optional mapped-name element that can be used to provide
vendor-specific deployment information such as the physical
jndi-name of destination from which this message-driven bean
should consume. This element is not required to be supported
by all implementations. Any use of this element is non-portable.

- the message-driven bean's implementation class
- an optional declaration of the bean's messaging
type
- an optional declaration of the bean's timeout method.

- the optional message-driven bean's transaction management
type. If it is not defined, it is defaulted to Container.

- an optional declaration of the bean's
message-destination-type
- an optional declaration of the bean's
message-destination-link
- an optional declaration of the message-driven bean's
activation configuration properties
- an optional list of the message-driven bean class and/or
superclass around-invoke methods.

- an optional declaration of the bean's environment
entries
- an optional declaration of the bean's EJB references
- an optional declaration of the bean's local EJB
references
- an optional declaration of the bean's web service
references
- an optional declaration of the security
identity to be used for the execution of the bean's
```

```
methods
- an optional declaration of the bean's
resource manager connection factory
references
- an optional declaration of the bean's resource
environment references.

- an optional declaration of the bean's message
destination references
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="Java EE:descriptionGroup"/>
<xsd:element name="ejb-name"

type="Java EE:ejb-nameType"/>
<xsd:element name="mapped-name"

type="Java EE:xsdStringType"

minOccurs="0"/>
<xsd:element name="ejb-class"

type="Java EE:ejb-classType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The ejb-class element specifies the fully qualified name
of the bean class for this ejb. It is required unless
there is a component-defining annotation for the same
ejb-name.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="messaging-type"

type="Java EE:fully-qualified-classType"
```

```
minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The messaging-type element specifies the message
listener interface of the message-driven bean.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="timeout-method"

type="Java EE:named-methodType"

minOccurs="0"/>
<xsd:element name="transaction-type"

type="Java EE:transaction-typeType"

minOccurs="0"/>
<xsd:element name="message-destination-type"

type="Java EE:message-destination-typeType"

minOccurs="0"/>
<xsd:element name="message-destination-link"

type="Java EE:message-destination-linkType"

minOccurs="0"/>
<xsd:element name="activation-config"

type="Java EE:activation-configType"

minOccurs="0"/>
<xsd:element name="around-invoke"

type="Java EE:around-invokeType"

minOccurs="0"
```

```
maxOccurs="unbounded"/>
<xsd:group ref="Java EE:jndiEnvironmentRefsGroup"/>
<xsd:element name="security-identity"

type="Java EE:security-identityType"

minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="method-intfType">
<xsd:annotation>
<xsd:documentation>

The method-intf element allows a method element to
differentiate between the methods with the same name and
signature that are multiply defined across the home and
component interfaces (e.g, in both an enterprise bean's
remote and local interfaces or in both an enterprise bean's
home and remote interfaces, etc.); the component and web
service endpoint interfaces, and so on. The Local applies to
both local component interface and local business interface.

Similarly, Remote applies to both remote component interface
and the remote business interface.

The method-intf element must be one of the following:

Home
Remote
LocalHome
Local
ServiceEndpoint
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:string">
```

```
<xsd:enumeration value="Home"/>
<xsd:enumeration value="Remote"/>
<xsd:enumeration value="LocalHome"/>
<xsd:enumeration value="Local"/>
<xsd:enumeration value="ServiceEndpoint"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="method-nameType">
<xsd:annotation>
<xsd:documentation>

The method-nameType contains a name of an enterprise
bean method or the asterisk (*) character. The asterisk is
used when the element denotes all the methods of an
enterprise bean's client view interfaces.
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:string"/>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="method-paramsType">
<xsd:annotation>
<xsd:documentation>

The method-paramsType defines a list of the
fully-qualified Java type names of the method parameters.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="method-param"

type="Java EE:java-typeType"
```

```
minOccurs="0"

maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>

The method-param element contains a primitive
or a fully-qualified Java type name of a method
parameter.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="method-permissionType">
<xsd:annotation>
<xsd:documentation>

The method-permissionType specifies that one or more
security roles are allowed to invoke one or more enterprise
bean methods. The method-permissionType consists of an
optional description, a list of security role names or an
indicator to state that the method is unchecked for
authorization, and a list of method elements.

The security roles used in the method-permissionType
must be defined in the security-role elements of the
deployment descriptor, and the methods must be methods
defined in the enterprise bean's business, home, component
and/or web service endpoint interfaces.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
```

```
type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:choice>
<xsd:element name="role-name"

type="Java EE:role-nameType"

maxOccurs="unbounded"/>
<xsd:element name="unchecked"

type="Java EE:emptyType">
<xsd:annotation>
<xsd:documentation>

The unchecked element specifies that a method is
not checked for authorization by the container
prior to invocation of the method.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:choice>
<xsd:element name="method"

type="Java EE:methodType"

maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="methodType">
<xsd:annotation>
<xsd:documentation>
<![CDATA[
```

```
The methodType is used to denote a method of an enterprise
bean's business, home, component, and/or web service endpoint
interface, or, in the case of a message-driven bean, the
bean's message listener method, or a set of such
methods. The ejb-name element must be the name of one of the
enterprise beans declared in the deployment descriptor; the
optional method-intf element allows to distinguish between a
method with the same signature that is multiply defined
across the business, home, component, and/or web service
endpoint nterfaces; the method-name element specifies the
method name; and the optional method-params elements identify
a single method among multiple methods with an overloaded
method name.

There are three possible styles of using methodType element
within a method element:

1.
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>*</method-name>
</method>

This style is used to refer to all the methods of the
specified enterprise bean's business, home, component,

and/or web service endpoint interfaces.

2.
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>METHOD</method-name>
</method>

This style is used to refer to the specified method of
the specified enterprise bean. If there are multiple
methods with the same overloaded name, the element of
this style refers to all the methods with the overloaded
name.
```

```
3.
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>METHOD</method-name>
<method-params>
<method-param>PARAM-1</method-param>
<method-param>PARAM-2</method-param>

...
<method-param>PARAM-n</method-param>
</method-params>
</method>

This style is used to refer to a single method within a
set of methods with an overloaded name. PARAM-1 through
PARAM-n are the fully-qualified Java types of the
method's input parameters (if the method has no input
arguments, the method-params element contains no
method-param elements). Arrays are specified by the
array element's type, followed by one or more pair of
square brackets (e.g. int[][]). If there are multiple
methods with the same overloaded name, this style refers
to all of the overloaded methods.

Examples:

Style 1: The following method element refers to all the
methods of the EmployeeService bean's business, home,

component, and/or web service endpoint interfaces:
<method>
<ejb-name>EmployeeService</ejb-name>
<method-name>*</method-name>
</method>

Style 2: The following method element refers to all the
create methods of the EmployeeService bean's home
interface(s).
<method>
<ejb-name>EmployeeService</ejb-name>
```

```
<method-name>create</method-name>
</method>

Style 3: The following method element refers to the
create(String firstName, String LastName) method of the
EmployeeService bean's home interface(s).
<method>
<ejb-name>EmployeeService</ejb-name>
<method-name>create</method-name>
<method-params>
<method-param>java.lang.String</method-param>
<method-param>java.lang.String</method-param>
</method-params>
</method>

The following example illustrates a Style 3 element with
more complex parameter types. The method
foobar(char s, int i, int[] iar, mypackage.MyClass mycl,

mypackage.MyClass[][] myclaar) would be specified as:
<method>
<ejb-name>EmployeeService</ejb-name>
<method-name>foobar</method-name>
<method-params>
<method-param>char</method-param>
<method-param>int</method-param>
<method-param>int[]</method-param>
<method-param>mypackage.MyClass</method-param>
<method-param>mypackage.MyClass[][]</method-param>
</method-params>
</method>

The optional method-intf element can be used when it becomes
necessary to differentiate between a method that is multiply
defined across the enterprise bean's business, home, component,

and/or web service endpoint interfaces with the same name and
signature. However, if the same method is a method of both the
local business interface, and the local component interface,
```

```
the same attribute applies to the method for both interfaces.

Likewise, if the same method is a method of both the remote
business interface and the remote component interface, the same
attribute applies to the method for both interfaces.

For example, the method element
<method>
<ejb-name>EmployeeService</ejb-name>
<method-intf>Remote</method-intf>
<method-name>create</method-name>
<method-params>
<method-param>java.lang.String</method-param>
<method-param>java.lang.String</method-param>
</method-params>
</method>

can be used to differentiate the create(String, String)

method defined in the remote interface from the
create(String, String) method defined in the remote home
interface, which would be defined as
<method>
<ejb-name>EmployeeService</ejb-name>
<method-intf>Home</method-intf>
<method-name>create</method-name>
<method-params>
<method-param>java.lang.String</method-param>
<method-param>java.lang.String</method-param>
</method-params>
</method>

and the create method that is defined in the local home
interface which would be defined as
<method>
<ejb-name>EmployeeService</ejb-name>
<method-intf>LocalHome</method-intf>
<method-name>create</method-name>
<method-params>
<method-param>java.lang.String</method-param>
```

```
<method-param>java.lang.String</method-param>
</method-params>
</method>

The method-intf element can be used with all three Styles
of the method element usage. For example, the following
method element example could be used to refer to all the
methods of the EmployeeService bean's remote home interface
and the remote business interface.
<method>
<ejb-name>EmployeeService</ejb-name>
<method-intf>Home</method-intf>
<method-name>*</method-name>
</method>

]]>
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="ejb-name"

type="Java EE:ejb-nameType"/>
<xsd:element name="method-intf"

type="Java EE:method-intfType"

minOccurs="0">
</xsd:element>
<xsd:element name="method-name"

type="Java EE:method-nameType"/>
<xsd:element name="method-params"

type="Java EE:method-paramsType"
```

```
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
************************************************
>
<xsd:complexType name="multiplicityType">
<xsd:annotation>
<xsd:documentation>

The multiplicityType describes the multiplicity of the
role that participates in a relation.

The value must be one of the two following:

One
Many
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:string">
<xsd:enumeration value="One"/>
<xsd:enumeration value="Many"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
************************************************
>
<xsd:complexType name="named-methodType">
<xsd:sequence>
<xsd:element name="method-name"

type="Java EE:string"/>
<xsd:element name="method-params"

type="Java EE:method-paramsType"

minOccurs="0"/>
```

```
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="persistence-typeType">
<xsd:annotation>
<xsd:documentation>

The persistence-typeType specifies an entity bean's persistence
management type.

The persistence-type element must be one of the two following:

Bean
Container
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:string">
<xsd:enumeration value="Bean"/>
<xsd:enumeration value="Container"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="query-methodType">
<xsd:annotation>
<xsd:documentation>
<![CDATA[

The query-method specifies the method for a finder or select
query.

The method-name element specifies the name of a finder or select
method in the entity bean's implementation class.
```

```
Each method-param must be defined for a query-method using the
method-params element.

It is used by the query-method element.

Example:
<query>
<description>Method finds large orders</description>
<query-method>
<method-name>findLargeOrders</method-name>
<method-params></method-params>
</query-method>
<ejb-ql>

SELECT OBJECT(o) FROM Order o
WHERE o.amount > 1000
</ejb-ql>
</query>

]]>
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="method-name"

type="Java EE:method-nameType"/>
<xsd:element name="method-params"

type="Java EE:method-paramsType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="queryType">
<xsd:annotation>
<xsd:documentation>

The queryType defines a finder or select
```

```
query. It contains
- an optional description of the query
- the specification of the finder or select
method it is used by
- an optional specification of the result type
mapping, if the query is for a select method
and entity objects are returned.

- the EJB QL query string that defines the query.

Queries that are expressible in EJB QL must use the ejb-ql
element to specify the query. If a query is not expressible
in EJB QL, the description element should be used to
describe the semantics of the query and the ejb-ql element
should be empty.

The result-type-mapping is an optional element. It can only
be present if the query-method specifies a select method
that returns entity objects. The default value for the
result-type-mapping element is "Local".
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType" minOccurs="0"/>
<xsd:element name="query-method"

type="Java EE:query-methodType"/>
<xsd:element name="result-type-mapping"

type="Java EE:result-type-mappingType"

minOccurs="0"/>
<xsd:element name="ejb-ql"

type="Java EE:xsdStringType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

```
<!--
**************************************************
>
<xsd:complexType name="relationship-role-sourceType">
<xsd:annotation>
<xsd:documentation>

The relationship-role-sourceType designates the source of a
role that participates in a relationship. A
relationship-role-sourceType is used by
relationship-role-source elements to uniquely identify an
entity bean.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="ejb-name"

type="Java EE:ejb-nameType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="relationshipsType">
<xsd:annotation>
<xsd:documentation>

The relationshipsType describes the relationships in
which entity beans with container-managed persistence
participate. The relationshipsType contains an optional
description; and a list of ejb-relation elements, which
specify the container managed relationships.
```

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="ejb-relation"

type="Java EE:ejb-relationType"

maxOccurs="unbounded">
<xsd:unique name="role-name-uniqueness">
<xsd:annotation>
<xsd:documentation>

The ejb-relationship-role-name contains the name of a
relationship role. The name must be unique within
a relationship, but can be reused in different
relationships.
</xsd:documentation>
</xsd:annotation>
<xsd:selector
xpath=".//Java EE:ejb-relationship-role-name"/>
<xsd:field
xpath="."/>
</xsd:unique>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="remove-methodType">
<xsd:sequence>
<xsd:element name="bean-method"
```

```
type="Java EE:named-methodType"/>
<xsd:element name="retain-if-exception"

type="Java EE:true-falseType"

minOccurs="1"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="result-type-mappingType">
<xsd:annotation>
<xsd:documentation>

The result-type-mappingType is used in the query element to
specify whether an abstract schema type returned by a query
for a select method is to be mapped to an EJBLocalObject or
EJBObject type.

The value must be one of the following:

Local
Remote
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:string">
<xsd:enumeration value="Local"/>
<xsd:enumeration value="Remote"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="security-identityType">
<xsd:annotation>
```

```
<xsd:documentation>

The security-identityType specifies whether the caller's
security identity is to be used for the execution of the
methods of the enterprise bean or whether a specific run-as
identity is to be used. It contains an optional description
and a specification of the security identity to be used.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"

type="Java EE:descriptionType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:choice>
<xsd:element name="use-caller-identity"

type="Java EE:emptyType">
<xsd:annotation>
<xsd:documentation>

The use-caller-identity element specifies that
the caller's security identity be used as the
security identity for the execution of the
enterprise bean's methods.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="run-as"

type="Java EE:run-asType"/>
</xsd:choice>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
```

```
>
<xsd:complexType name="session-beanType">
<xsd:annotation>
<xsd:documentation>

The session-beanType declares an session bean. The
declaration consists of:

- an optional description
- an optional display name
- an optional icon element that contains a small and a large
icon file name
- a name assigned to the enterprise bean
in the deployment description
- an optional mapped-name element that can be used to provide
vendor-specific deployment information such as the physical
jndi-name of the session bean's remote home/business interface.

This element is not required to be supported by all
implementations. Any use of this element is non-portable.

- the names of all the remote or local business interfaces,

if any
- the names of the session bean's remote home and
remote interfaces, if any
- the names of the session bean's local home and
local interfaces, if any
- the name of the session bean's web service endpoint
interface, if any
- the session bean's implementation class
- the session bean's state management type
- an optional declaration of the session bean's timeout method.

- the optional session bean's transaction management type.

If it is not present, it is defaulted to Container.

- an optional list of the session bean class and/or
superclass around-invoke methods.
```

```
- an optional declaration of the bean's
environment entries
- an optional declaration of the bean's EJB references
- an optional declaration of the bean's local
EJB references
- an optional declaration of the bean's web
service references
- an optional declaration of the security role
references
- an optional declaration of the security identity
to be used for the execution of the bean's methods
- an optional declaration of the bean's resource
manager connection factory references
- an optional declaration of the bean's resource
environment references.

- an optional declaration of the bean's message
destination references
The elements that are optional are "optional" in the sense
that they are omitted when if lists represented by them are
empty.

Either both the local-home and the local elements or both
the home and the remote elements must be specified for the
session bean.

The service-endpoint element may only be specified if the
bean is a stateless session bean.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="Java EE:descriptionGroup"/>
<xsd:element name="ejb-name"

type="Java EE:ejb-nameType"/>
<xsd:element name="mapped-name"

type="Java EE:xsdStringType"
```

```
minOccurs="0"/>
<xsd:element name="home"

type="Java EE:homeType"

minOccurs="0"/>
<xsd:element name="remote"

type="Java EE:remoteType"

minOccurs="0"/>
<xsd:element name="local-home"

type="Java EE:local-homeType"

minOccurs="0"/>
<xsd:element name="local"

type="Java EE:localType"

minOccurs="0"/>
<xsd:element name="business-local"

type="Java EE:fully-qualified-classType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="business-remote"

type="Java EE:fully-qualified-classType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="service-endpoint"

type="Java EE:fully-qualified-classType"

minOccurs="0">
```

```
<xsd:annotation>
<xsd:documentation>

The service-endpoint element contains the
fully-qualified name of the enterprise bean's web
service endpoint interface. The service-endpoint
element may only be specified for a stateless
session bean. The specified interface must be a
valid JAX-RPC service endpoint interface.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="ejb-class"

type="Java EE:ejb-classType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The ejb-class element specifies the fully qualified name
of the bean class for this ejb. It is required unless
there is a component-defining annotation for the same
ejb-name.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="session-type"

type="Java EE:session-typeType"

minOccurs="0"/>
<xsd:element name="timeout-method"

type="Java EE:named-methodType"

minOccurs="0"/>
<xsd:element name="init-method"

type="Java EE:init-methodType"
```

```
minOccurs="0"

maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>

The init-method element specifies the mappings for
EJB 2.x style create methods for an EJB 3.0 bean.

This element can only be specified for stateful
session beans.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="remove-method"

type="Java EE:remove-methodType"

minOccurs="0"

maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>

The remove-method element specifies the mappings for
EJB 2.x style remove methods for an EJB 3.0 bean.

This element can only be specified for stateful
session beans.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="transaction-type"

type="Java EE:transaction-typeType"

minOccurs="0"/>
<xsd:element name="around-invoke"
```

```
type="Java EE:around-invokeType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:group ref="Java EE:jndiEnvironmentRefsGroup"/>
<xsd:element name="post-activate"

type="Java EE:lifecycle-callbackType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="pre-passivate"

type="Java EE:lifecycle-callbackType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="security-role-ref"

type="Java EE:security-role-refType"

minOccurs="0"

maxOccurs="unbounded">
</xsd:element>
<xsd:element name="security-identity"

type="Java EE:security-identityType"

minOccurs="0">
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
************************************************
>
```

```
<xsd:complexType name="session-typeType">
<xsd:annotation>
<xsd:documentation>

The session-typeType describes whether the session bean is a
stateful session or stateless session. It is used by
session-type elements.

The value must be one of the two following:

Stateful
Stateless
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:string">
<xsd:enumeration value="Stateful"/>
<xsd:enumeration value="Stateless"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="trans-attributeType">
<xsd:annotation>
<xsd:documentation>

The trans-attributeType specifies how the container must
manage the transaction boundaries when delegating a method
invocation to an enterprise bean's business method.

The value must be one of the following:

NotSupported
Supports
Required
RequiresNew
Mandatory
Never
```

```
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="Java EE:string">
<xsd:enumeration value="NotSupported"/>
<xsd:enumeration value="Supports"/>
<xsd:enumeration value="Required"/>
<xsd:enumeration value="RequiresNew"/>
<xsd:enumeration value="Mandatory"/>
<xsd:enumeration value="Never"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="transaction-typeType">
<xsd:annotation>
<xsd:documentation>

The transaction-typeType specifies an enterprise bean's
transaction management type.

The transaction-type must be one of the two following:

Bean
Container
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="JavaEE:string">
<xsd:enumeration value="Bean"/>
<xsd:enumeration value="Container"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
</xsd:schema>
```

17.1.10 application-client_5_0.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://java.sun.com/xml/ns/j2ee"
    xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified"
version="5.0">
    <xsd:annotation>
        <xsd:documentation>

@(#)application_5_0.xsds 1.15 01/04/05
</xsd:documentation>
    </xsd:annotation>
    <xsd:annotation>
        <xsd:documentation>

Copyright 2003-2005 Sun Microsystems, Inc.

4150 Network Circle
Santa Clara, California 95054
U.S.A
All rights reserved.

Sun Microsystems, Inc. has intellectual property rights
relating to technology described in this document. In
particular, and without limitation, these intellectual
property rights may include one or more of the U.S. patents
listed at http://www.sun.com/patents and one or more
additional patents or pending patent applications in the
U.S. and other countries.

This document and the technology which it describes are
distributed under licenses restricting their use, copying,

distribution, and decompilation. No part of this document
may be reproduced in any form by any means without prior
written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is
```

```
copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Solaris, Java, J2EE,

JavaServer Pages, Enterprise JavaBeans and the Java Coffee
Cup logo are trademarks or registered trademarks of Sun
Microsystems, Inc. in the U.S. and other countries.

Federal Acquisitions: Commercial Software - Government Users
Subject to Standard License Terms and Conditions.
</xsd:documentation>
    </xsd:annotation>
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[

This is the XML Schema for the application 5.0 deployment
descriptor. The deployment descriptor must be named
"META-INF/application.xml" in the application's ear file.

All application deployment descriptors must indicate
the application schema by using the J2EE namespace:

http://java.sun.com/xml/ns/j2ee
and indicate the version of the schema by
using the version element as shown below:
<application xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/application_5_0.xsd"

version="5.0">

...
</application>

The instance documents may indicate the published version of
the schema using the xsi:schemaLocation attribute for J2EE
```

```
namespace with the following location:

http://java.sun.com/xml/ns/j2ee/application_5_0.xsd
]]>
</xsd:documentation>
</xsd:annotation>
<xsd:annotation>
    <xsd:documentation>

The following conventions apply to all J2EE
deployment descriptor elements unless indicated otherwise.

- In elements that specify a pathname to a file within the
same JAR file, relative filenames (i.e., those not
starting with "/") are considered relative to the root of
the JAR file's namespace. Absolute filenames (i.e., those
starting with "/") also specify names in the root of the
JAR file's namespace. In general, relative names are
preferred. The exception is .war files where absolute
names are preferred for consistency with the Servlet API.
</xsd:documentation>
</xsd:annotation>
<xsd:include schemaLocation="j2ee_5_0.xsd" />
<!--
**************************************************
>
<xsd:element name="application" type="j2ee:applicationType">
<xsd:annotation>
<xsd:documentation>

The application element is the root element of a J2EE
application deployment descriptor.
</xsd:documentation>
</xsd:annotation>
<xsd:unique name="context-root-uniqueness">
<xsd:annotation>
<xsd:documentation>

The context-root element content must be unique
in the ear.
```

```
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:module/j2ee:web"/>
<xsd:field xpath="j2ee:context-root"/>
</xsd:unique>
<xsd:unique name="security-role-uniqueness">
<xsd:annotation>
<xsd:documentation>

The security-role-name element content
must be unique in the ear.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="j2ee:security-role"/>
<xsd:field xpath="j2ee:role-name"/>
</xsd:unique>
</xsd:element>
<!--
**************************************************
>
<xsd:complexType name="applicationType">
<xsd:annotation>
<xsd:documentation>

The applicationType defines the structure of the
application.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="j2ee:descriptionGroup"/>
<xsd:element name="module"

type="j2ee:moduleType"

maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>

The application deployment descriptor must have one
module element for each J2EE module in the
```

```
application package. A module element is defined
by moduleType definition.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="security-role"

type="j2ee:security-roleType"

minOccurs="0"

maxOccurs="unbounded"/>
<xsd:element name="library-directory"

type="j2ee:pathType"

minOccurs="0"

maxOccurs="1">
<xsd:annotation>
<xsd:documentation>

The library-directory element specifies the pathname
of a directory within the application package, relative
to the top level of the application package. All files
named "*.jar" in this directory must be made available
in the class path of all components included in this
application package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="version"

type="j2ee:dewey-versionType"

fixed="5.0"

use="required">
<xsd:annotation>
```

```
<xsd:documentation>

The required value for the version is 5.0.
</xsd:documentation>
</xsd:annotation>
</xsd:attribute>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="moduleType">
<xsd:annotation>
<xsd:documentation>

The moduleType defines a single J2EE module and contains a
connector, ejb, java, or web element, which indicates the
module type and contains a path to the module file, and an
optional alt-dd element, which specifies an optional URI to
the post-assembly version of the deployment descriptor.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:choice>
<xsd:element name="connector"

type="j2ee:pathType">
<xsd:annotation>
<xsd:documentation>

The connector element specifies the URI of a
resource adapter archive file, relative to the
top level of the application package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="ejb"

type="j2ee:pathType">
<xsd:annotation>
```

```
<xsd:documentation>

The ejb element specifies the URI of an ejb-jar,

relative to the top level of the application
package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="java"

type="j2ee:pathType">
<xsd:annotation>
<xsd:documentation>

The java element specifies the URI of a java
application client module, relative to the top
level of the application package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="web"

type="j2ee:webType"/>
</xsd:choice>
<xsd:element name="alt-dd"

type="j2ee:pathType"

minOccurs="0">
<xsd:annotation>
<xsd:documentation>

The alt-dd element specifies an optional URI to the
post-assembly version of the deployment descriptor
file for a particular J2EE module. The URI must
specify the full pathname of the deployment
descriptor file relative to the application's root
directory. If alt-dd is not specified, the deployer
must read the deployment descriptor from the default
```

```
location and file name required by the respective
component specification.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!--
**************************************************
>
<xsd:complexType name="webType">
<xsd:annotation>
<xsd:documentation>

The webType defines the web-uri and context-root of
a web application module.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="web-uri"

type="j2ee:pathType">
<xsd:annotation>
<xsd:documentation>

The web-uri element specifies the URI of a web
application file, relative to the top level of the
application package.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="context-root"

type="j2ee:string">
<xsd:annotation>
<xsd:documentation>

The context-root element specifies the context root
of a web application.
```

```
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
</xsd:schema>
```

17.1.11 apusic-application_5_0.dtd

```
<!--

This is the XML DTD for the Apusic Application Configuration information.

>
<!ELEMENT apusic-application (module*, mail-session*,
security-role*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT module (connector | ejb | java | web)>
<!ATTLIST module uri CDATA #IMPLIED>
<!ELEMENT connector (resourceadapter+)>
<!ELEMENT resourceadapter (jndi-name, property*,
principal-map*)>
<!ATTLIST resourceadapter idle-timeout CDATA "300"

min-pool-size CDATA "5"

max-pool-size CDATA "30">
<!ELEMENT principal-map (description?, server-principal+,
connector-principal)>
<!ELEMENT server-principal (#PCDATA)>
<!ELEMENT connector-principal EMPTY>
<!ATTLIST connector-principal username CDATA #REQUIRED password CDATA
#REQUIRED>
<!ELEMENT ejb ((entity | session | message-driven)*,
relationship-mapping*, cmp-resource?)>
<!ELEMENT entity (jndi-name?, local-jndi-name?, cmp?,
is-modified-method-name?,
```

```
ejb-ref*, ejb-local-ref*, resource-ref*, resource-env-ref*)>
<!ATTLIST entity ejb-name CDATA #REQUIRED
pool-size CDATA #IMPLIED
cache-size CDATA #IMPLIED
transaction-timeout CDATA #IMPLIED
concurrency-strategy CDATA #IMPLIED
force-refresh (True|False) "False"

delay-updates (True|False) "True"

load-for-update (True|False) "False"

cmp11-promotion CDATA #IMPLIED
expiration-time CDATA #IMPLIED>
<!ELEMENT session (jndi-name?, local-jndi-name?, ejb-ref*,
ejb-local-ref*,

resource-ref*, resource-env-ref*)>
<!ATTLIST session ejb-name CDATA #REQUIRED
pool-size CDATA #IMPLIED
cache-size CDATA #IMPLIED
session-timeout CDATA #IMPLIED
transaction-timeout CDATA #IMPLIED>
<!ELEMENT message-driven (message-driven-destination, ejb-ref*,
ejb-local-ref*,

resource-ref*, resource-env-ref*)>
<!ATTLIST message-driven
ejb-name CDATA #REQUIRED
pool-size CDATA #IMPLIED>
<!ELEMENT jndi-name (#PCDATA)>
<!ELEMENT local-jndi-name (#PCDATA)>
<!ELEMENT cmp (jdbc)>
<!ELEMENT jdbc (datasource-name?, table-name?, auto-create-table?,
auto-generate-key?,

field-mapping*, load-unit*, finder*)>
<!ELEMENT datasource-name (#PCDATA)>
<!ELEMENT table-name (#PCDATA)>
```

```
<!ELEMENT auto-create-table EMPTY>
<!ELEMENT auto-generate-key EMPTY>
<!ELEMENT field-mapping (field-name, column-name)>
<!ELEMENT field-name (#PCDATA)>
<!ELEMENT column-name (#PCDATA)>
<!ELEMENT load-unit (unit-name, (cmp-field | cmr-field)+)>
<!ELEMENT unit-name (#PCDATA)>
<!ELEMENT cmp-field (#PCDATA)>
<!ELEMENT cmr-field (#PCDATA)>
<!ELEMENT finder (description?, method, sql-clause)>
<!ELEMENT method (description?, method-intf?, method-name,
method-params?)>
<!ELEMENT method-intf (#PCDATA)>
<!ELEMENT method-name (#PCDATA)>
<!ELEMENT method-params (method-param*)>
<!ELEMENT method-param (#PCDATA)>
<!ELEMENT sql-clause (#PCDATA)>
<!ELEMENT is-modified-method-name (#PCDATA)>
<!ELEMENT relationship-mapping (ejb-relation-name, table-name?,
auto-create-table?,

source-role?, sink-role?)>
<!ELEMENT ejb-relation-name (#PCDATA)>
<!ELEMENT source-role (field-mapping*)>
<!ELEMENT sink-role (field-mapping*)>
<!ELEMENT cmp-resource (jndi-name)>
<!ELEMENT message-driven-destination (connection-factory-name?,

destination-name, auto-create-destination?,

client-id?, durable-subscription-name?)>
<!ELEMENT connection-factory-name (#PCDATA)>
<!ELEMENT destination-name (#PCDATA)>
<!ELEMENT auto-create-destination EMPTY>
<!ELEMENT client-id (#PCDATA)>
<!ELEMENT durable-subscription-name (#PCDATA)>
<!ELEMENT web (context-root?, ejb-ref*, ejb-local-ref*,
resource-ref*, resource-env-ref*)>
<!ELEMENT context-root (#PCDATA)>
<!ELEMENT java (ejb-ref*, resource-ref*, resource-env-ref*)>
```

```
<!ELEMENT ejb-ref (ejb-ref-name, jndi-name)>
<!ELEMENT ejb-local-ref (ejb-ref-name, jndi-name)>
<!ELEMENT ejb-ref-name (#PCDATA)>
<!ELEMENT resource-ref (res-ref-name, jndi-name)>
<!ELEMENT res-ref-name (#PCDATA)>
<!ELEMENT resource-env-ref (resource-env-ref-name, jndi-name)>
<!ELEMENT resource-env-ref-name (#PCDATA)>
<!ELEMENT mail-session (description?, jndi-name, property*)>
<!ELEMENT property EMPTY>
<!ATTLIST property name CDATA #REQUIRED value CDATA #REQUIRED>
<!ELEMENT security-role (role-name, (principal | group)+)>
<!ELEMENT role-name (#PCDATA)>
<!ELEMENT principal (#PCDATA)>
<!ELEMENT group (#PCDATA)>
```

18 部分VIII 管理控制台使用指南

金蝶Apusic应用服务器提供统一管理控制台，通过可视化界面进行操作。

默认情况下，允许localhost可以访问，不能进行远程访问。如果需要远程其他地址访问，则需要配置系统参数 `-Dcom.apusic.webtool.allowHosts=172.24.5.1`，值1 `72.24.5.1` 是可以访问管控的ip地址，多个地址用逗号分隔,也可以在 `vm.options` 文件中写： `com.apusic.webtool.allowHosts=172.24.5.6` 。

如果是三员分立版的管理控制台，配置系统参数 `-Dcom.apusic.admin.allowHosts=172.24.5.1` ，在 `vm.options` 文件中写： `com.apusic.admin.allowHosts=172.24.5.1` 。

18.1 进入Apusic管理监控台

18.1.1 进入到登录页面

首先，打开一个浏览器，推荐IE或者firefox。

浏览器要求：能支持FlashPlayer10.0以上的浏览器，如IE8+、FireFox4+、Chrome12+，IE6由于FlashPlayer插件机制过久，暂不能支持。建议使用Chrome15以上版本。

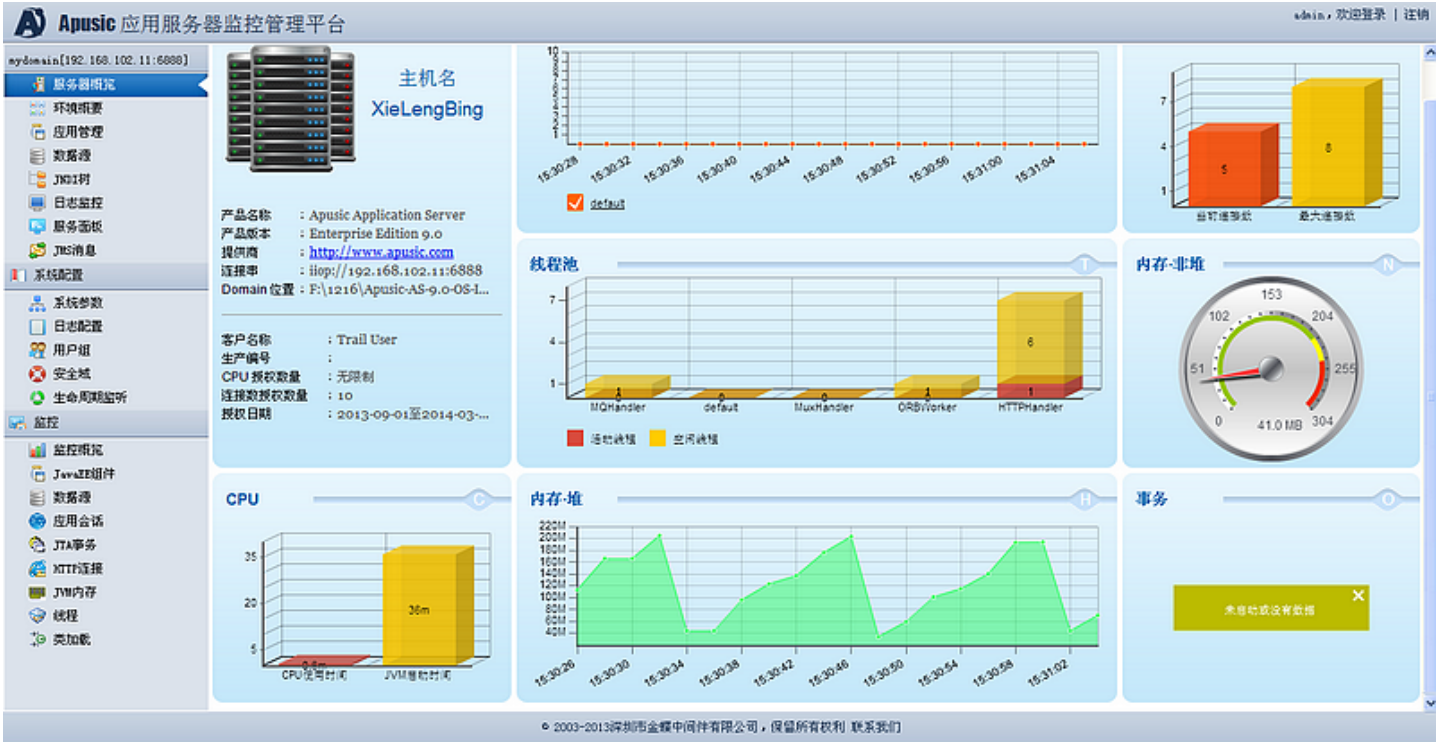
然后，输入以下URL：http://serverIP:serverPort/admin/。其中ServerIP和ServerPort根据实际情况调整，如 http://11.129.2.18:6888/admin/进入到管理监控台登录页面：



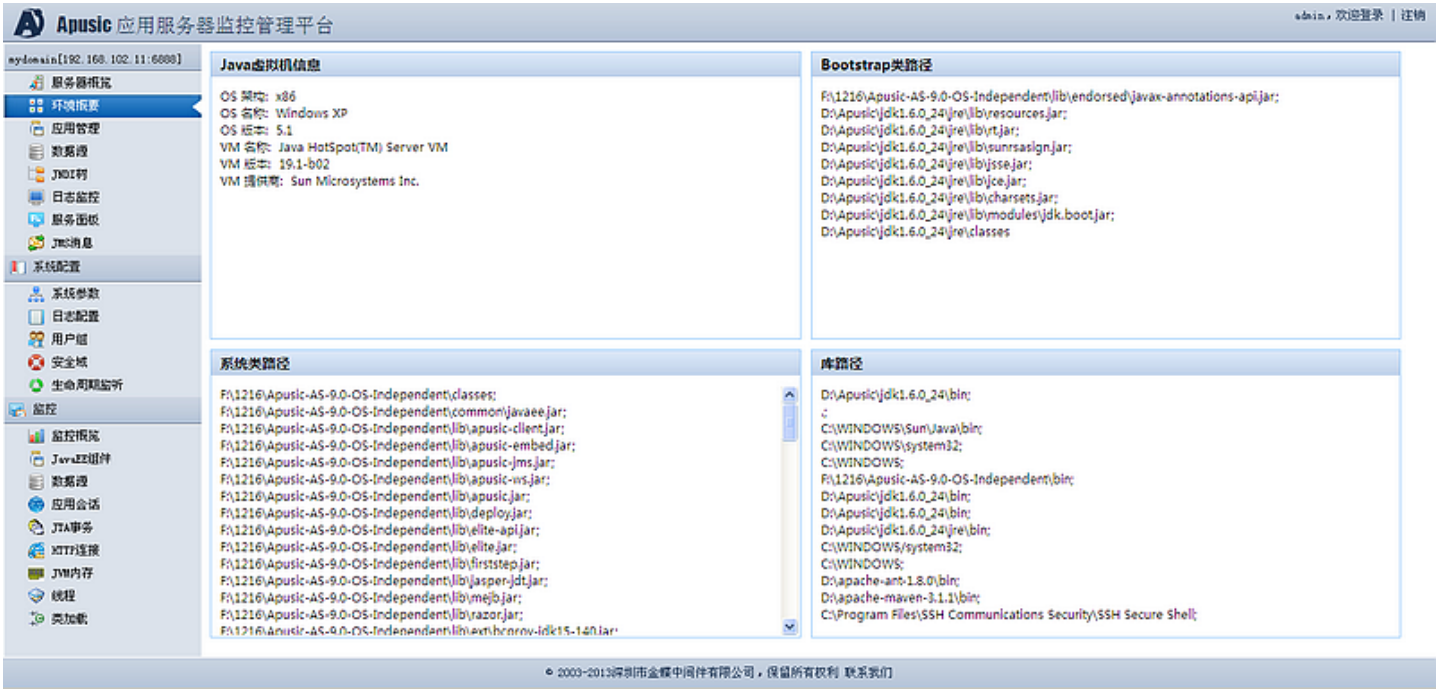
18.1.2 登录到管理监控首页

输入用户名“admin”、密码（首次启动服务器输入的密码），点击“登录”， 进入到“服务器概览“界面：

此界面显示的信息有主机的基本信息，虚拟机的内存使用信息、线程信息和当前连接数，CPU和JVM使用时间，应用的用户数等。其中事务的监控要在监控选项中开启监控。



在Apusic应用服务器监控管理平台树状结构图下点击"环境概要"，进入到系统环境概要界面信息下：



在此界面下显示的是应用服务器的所在主机以及本身所加载的类的具体信息，具体包括Java虚拟机信息、系统类路径等。

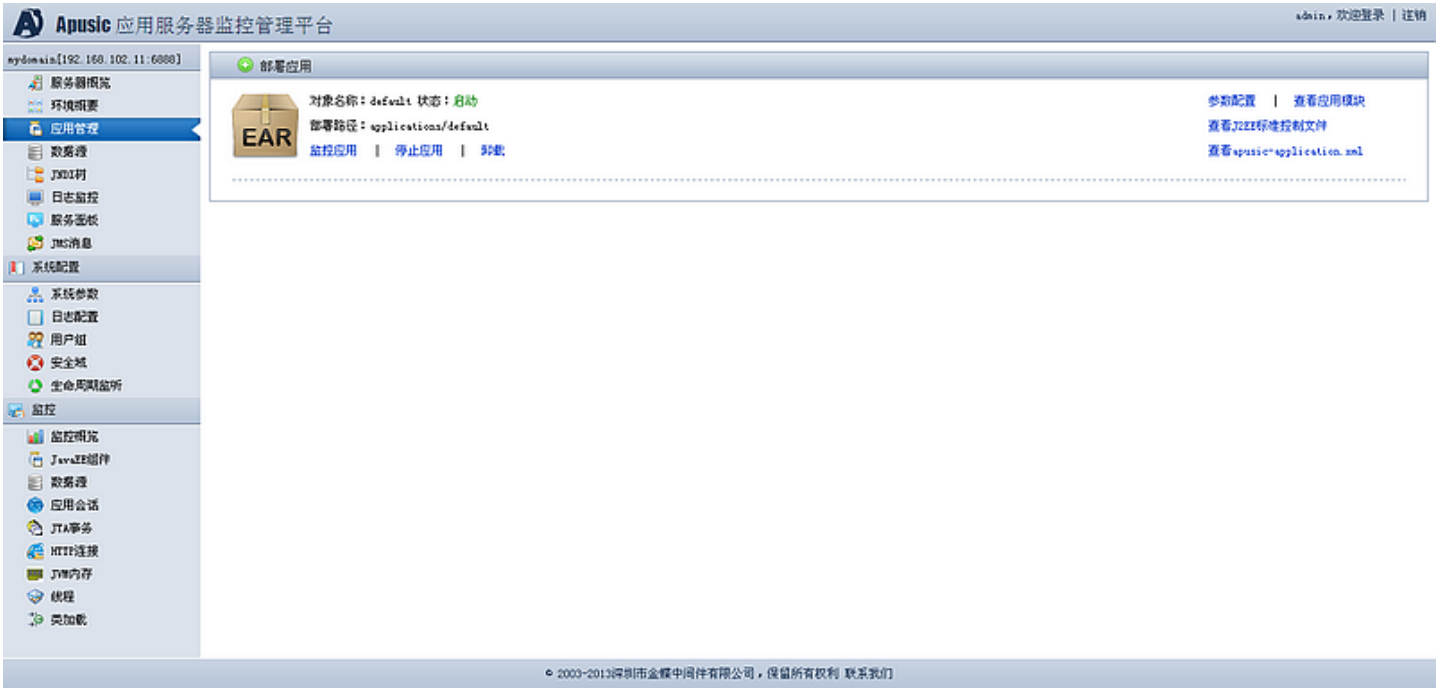
18.2 应用管理

18.2.1 部署J2EE应用

Apusic应用服务器支持两种应用部署方式，具体内容如下：

1. 将相应的J2EE应用文件（文件夹）拷贝到$DOMAIN_HOME/applications文件夹下，J2EE应用会被自动部署。
2. 直接配置Server.xml来进行应用部署，请参考"配置Web应用"。
3. 通过Apusic Web Admin Console部署J2EE应用。

首先，打开Apusic应用服务器监控管理平台"应用管理"操作界面。



然后，点击"部署应用"按钮进入到应用部署信息填写界面。



信息填写界面提供对应用的配置项，以下配置项根据实际配置的需要来填写。

属性名称	用途说明
应用名称	部署应用所填写的名称，可以和应用名相同，也可以自定义；作为标识整个应用的主键信息，必须唯一。
部署方式	可以选择部署服务器上文件系统下的应用和本地的应用。
上传应用包	上传应用时的路径。
应用基础上下文	改变应用的访问路径。举例说明，部署应用名称为test，访问应用URL为：http://serverIP:serverPort/test/，应用基础上下文配置为/test1后保存。则应用的访问URL变为：http://serverIP:serverPort/test1/ 。
启动类型	分为自动和手动，自动时，启动服务器即自动启动应用。
加载顺序	填写为数字，最小则优先加载。
是否全局session	将session属性改为全局。
虚拟主机	配置应用访问时的DNS上配置了域名到apusic地址的配对的虚拟主机，访问时通过虚拟主机来访问。
允许访问ip列表	对访问应用的IP进允许限制。允许IP地址列表格式：192.168.6.5：表示IP地址 192.168.6.5；192.168.6.10-50：表示192.168.6.10到192.168.6.50之间的所有 IP；192.168.6.*：表示192.168.6.0到192.168.6.255之间的所有IP；多个IP地址之间用英文逗号分隔。
启动类型	禁止访问ip列表：对访问应用的IP进行禁止限制。禁止IP地址列表格式：192.168.6.5：表示IP地址 192.168.6.5；192.168.6.10-50：表示192.168.6.10到192.168.6.50之间的所有 IP；192.168.6.*：表示192.168.6.0到192.168.6.255之间的所有IP；多个IP地址之间用英文逗号分隔。

最后，点击"完成"按钮，完成J2EE应用的部署。

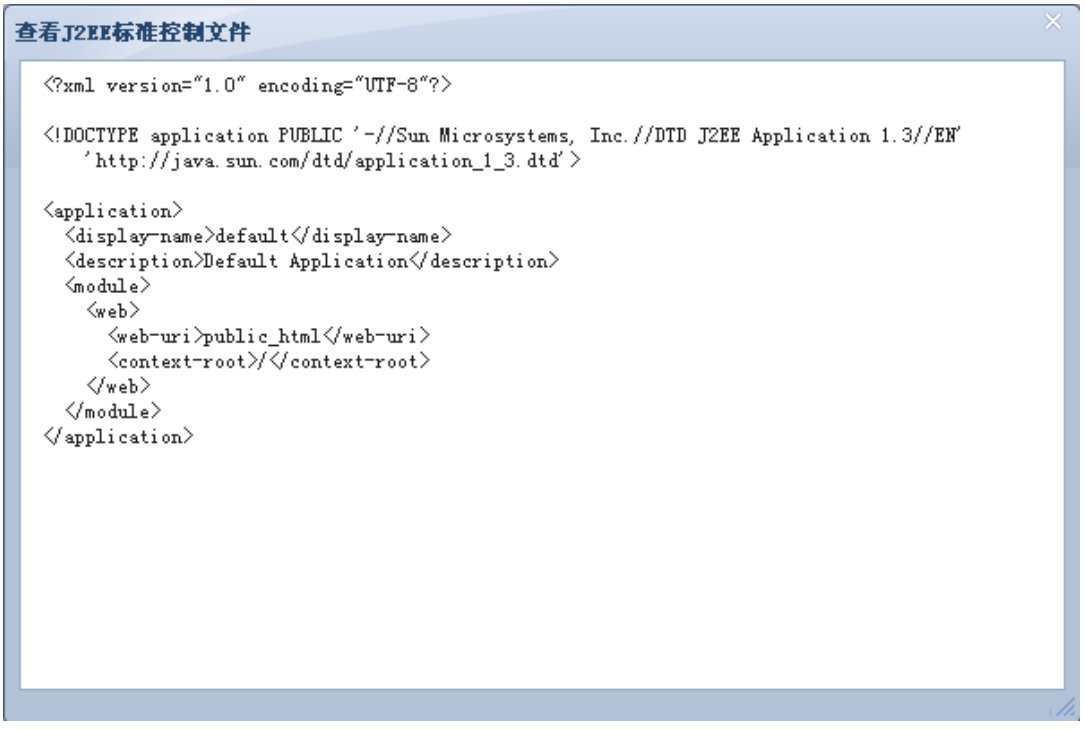
如果需要重新填写部署应用信息，则点击"重置"按钮即可。

18.2.2 管理J2EE应用

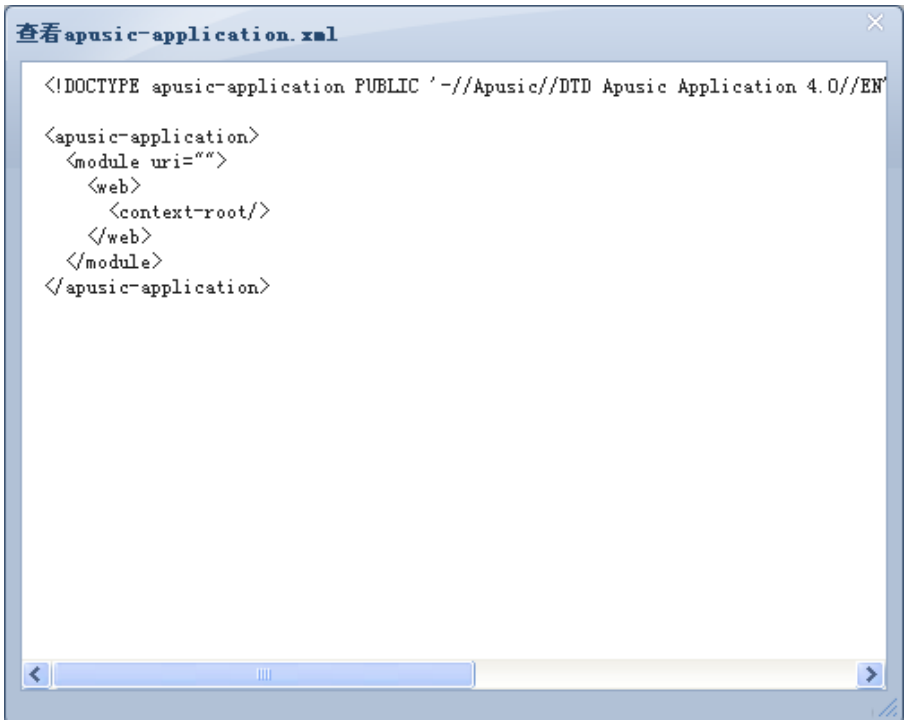
1. 打开Apusic应用服务器监控管理平台"应用管理"操作界面，选择J2EE应用后点击"参数按钮"对J2EE进行属性更新。



2. 点击"查看J2EE标准控制文件"按钮，进入到标准控制文件查看界面下：



3. 点击"查看apusic-application.xml"按钮，进入到apusic-application.xml文件查看界面下：



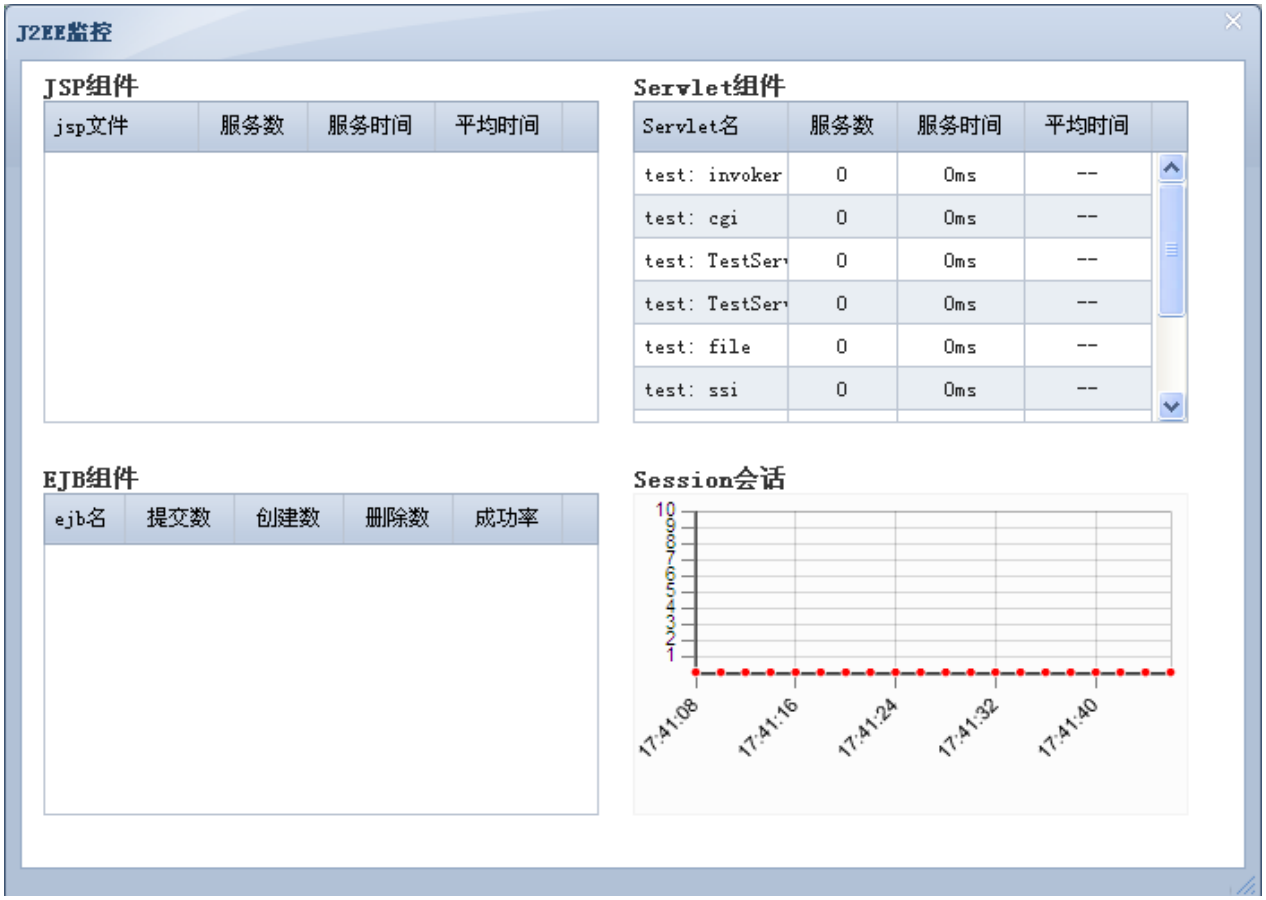
4. 点击"停止应用"按钮，在提示信息中点击"是"，停止J2EE应用。再次点击启动，启动应用。

5. 点击"查看应用模块"按钮，在弹出的应用模块界面下可以看到应用的web模块，EJB模块等，并可以直接点击web模块先的"访问应用"链接来访问整个应用。

6. 点击"卸载"按钮，在提示信息中点击"是"，卸载J2EE应用。

18.2.3 监控J2EE应用

打开Apusic应用服务器监控管理平台"应用管理"操作界面，选择J2EE后点击"监控应用"按钮查看对J2EE组件的监控。



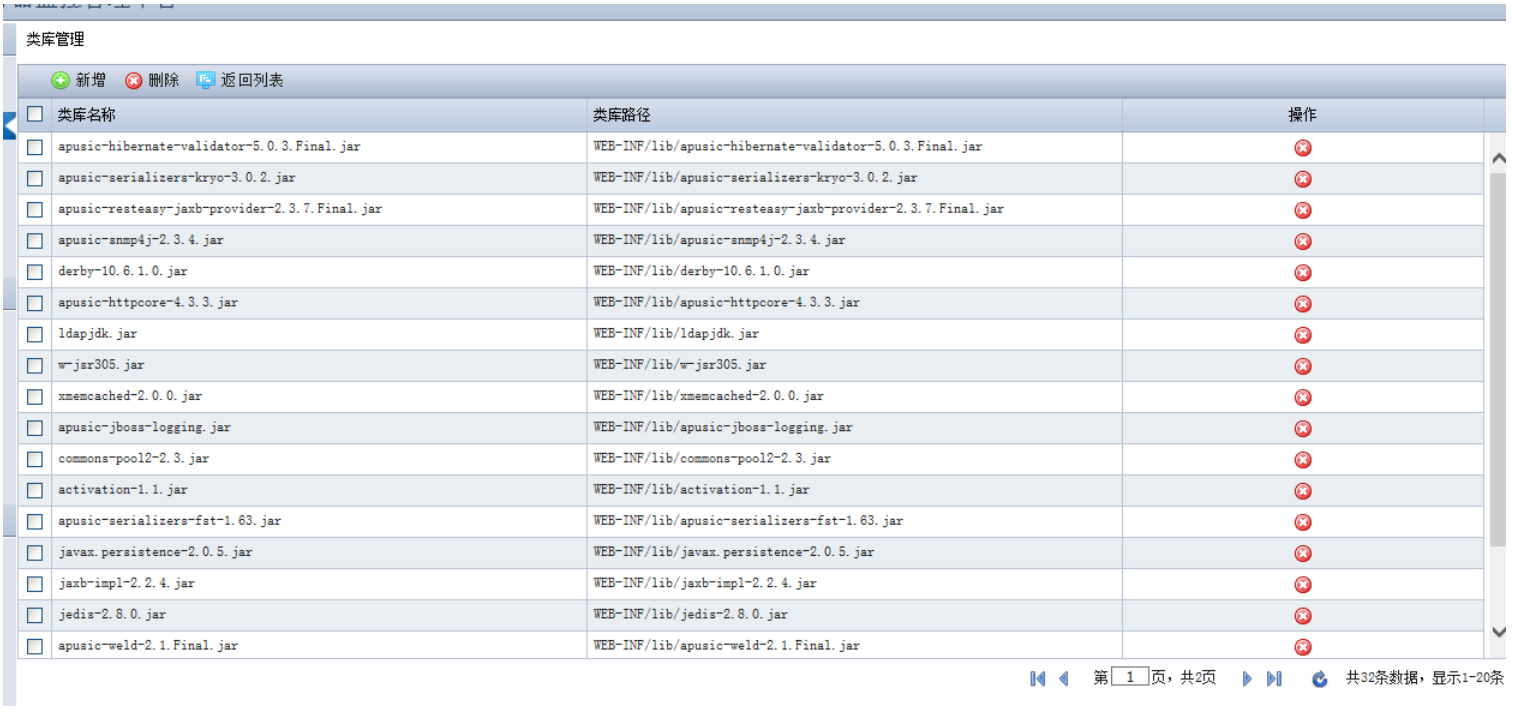
在此，我们可以监控到J2EE应用的JSP组件，Servlet组件，EJB组件以及Session会话的具体使用情况。

18.2.4 类库管理

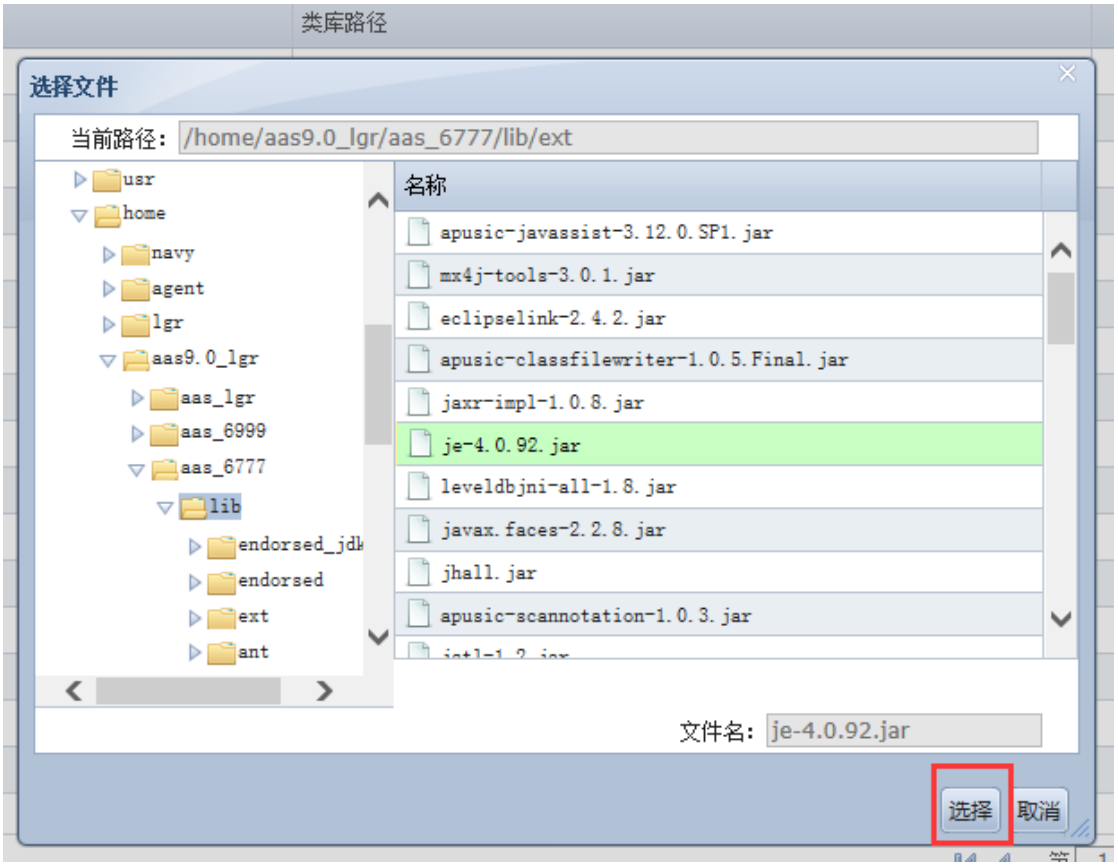
1. 打开Apusic应用服务器监控管理平台"应用管理"操作界面，选择某个应用后点击"类库管理"按钮对该应用的类库进行操作。



2. 进入类库管理界面，可以对类库进行增加删除等操作。



3. 点击“新增”按钮，进入到新增类库窗口，选择好需要的文件，点击“选择”按钮。



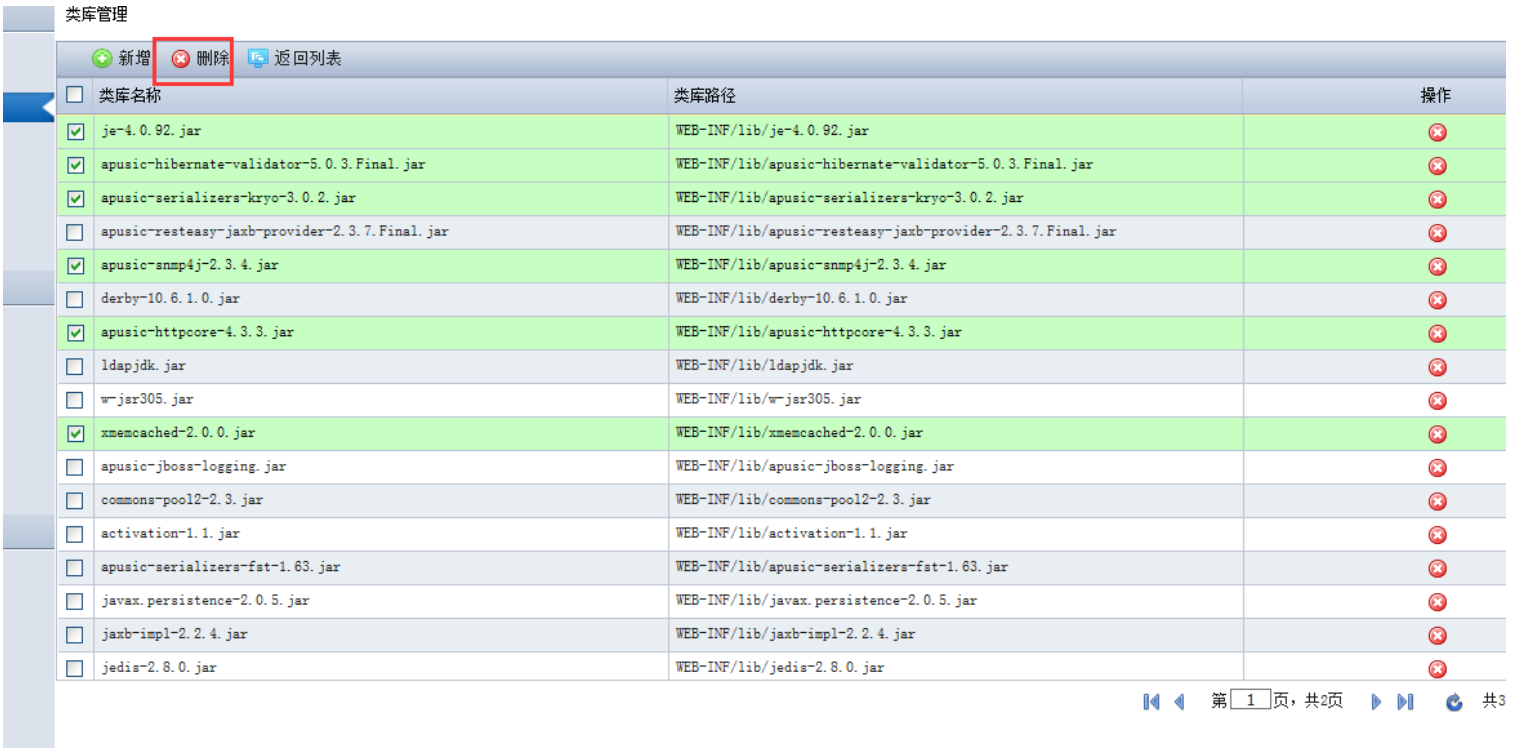
4. 新增成功后在类库管理列表中显示



5. 删除单个类库：选中某个类库，点击后面的删除图标按钮，删除该类库



6. 批量删除类库：勾选需要删除的类库，点击左上方的删除按钮图标进行删除

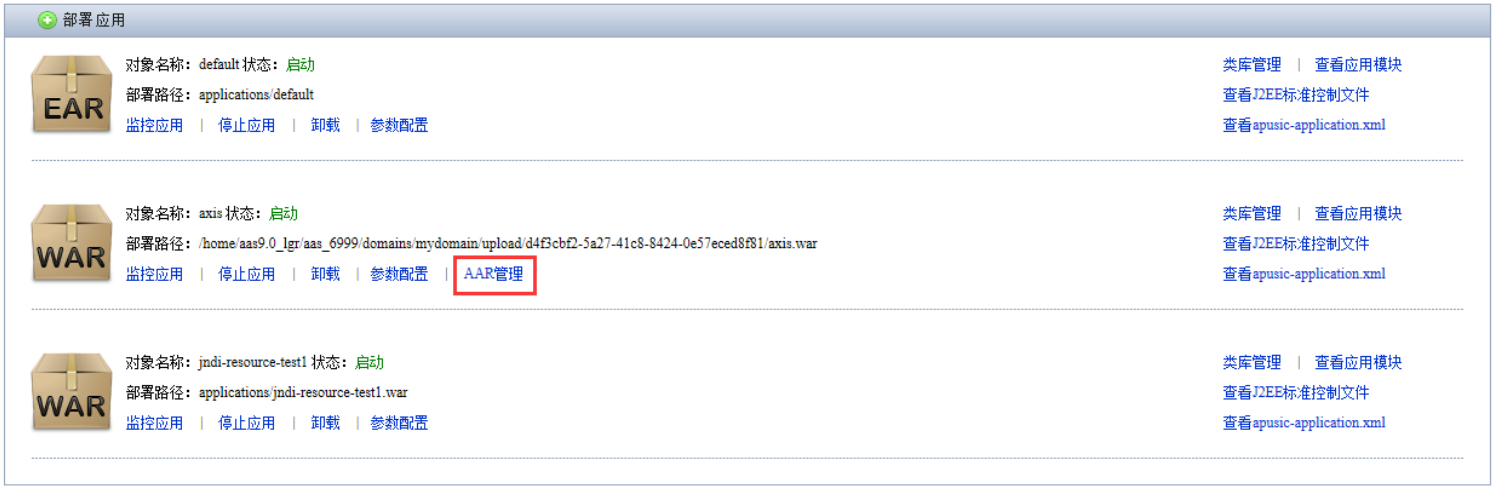


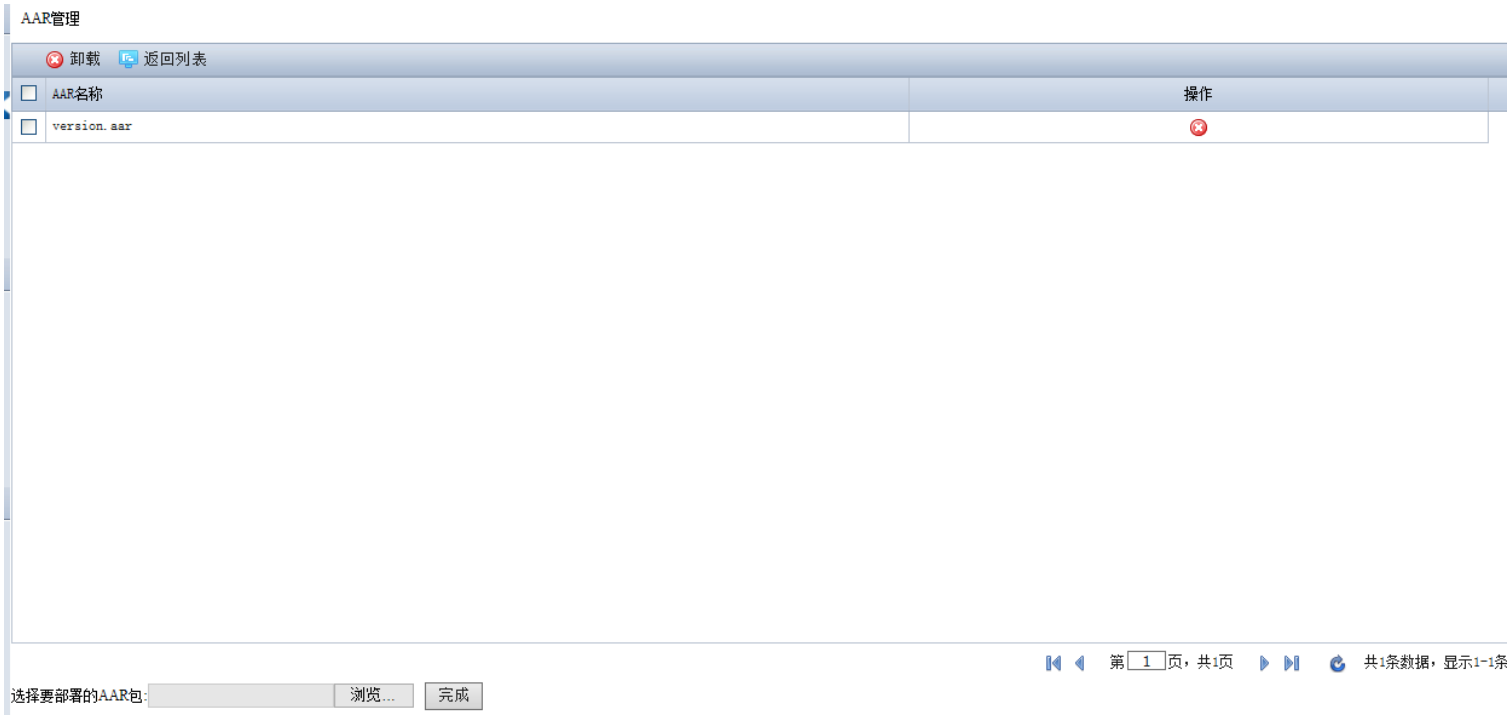
7. 点击“返回列表”的图标，返回到应用管理的界面。

18.2.5 AAR管理

1. 部署一个axis应用

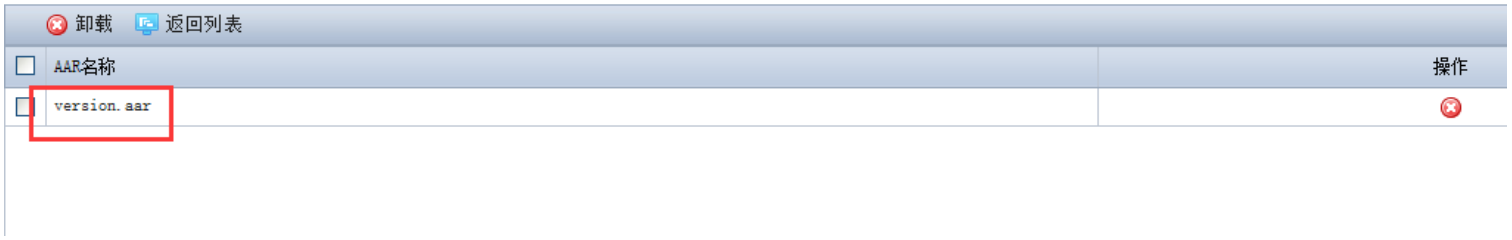
2. 在该axis应用那点击“AAR管理”按钮进入到AAR管理界面





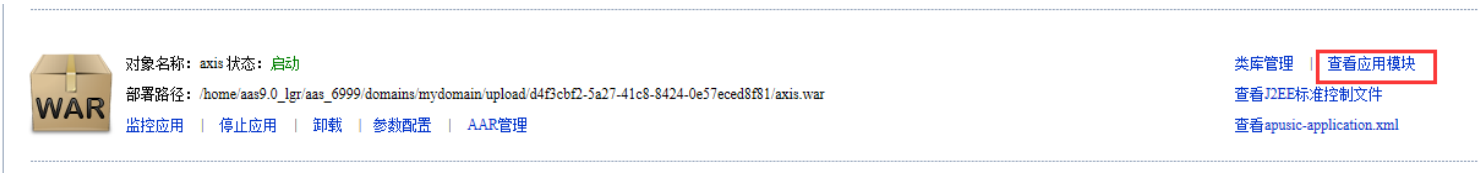
3. 新增AAR包:在界面左下角点击"浏览"，选择一个AAR包，点击"完成"按钮部署成功

4. 部署成功后可以在界面上查看到该AAR包。

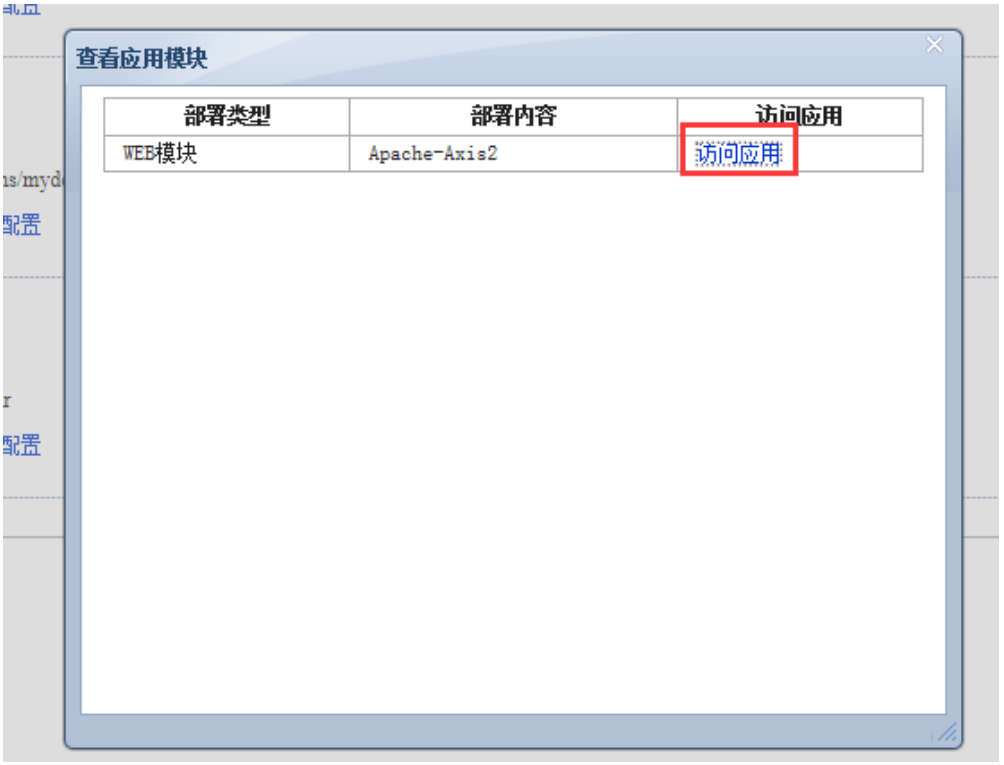


5. 查看AAR包内容的正确性：

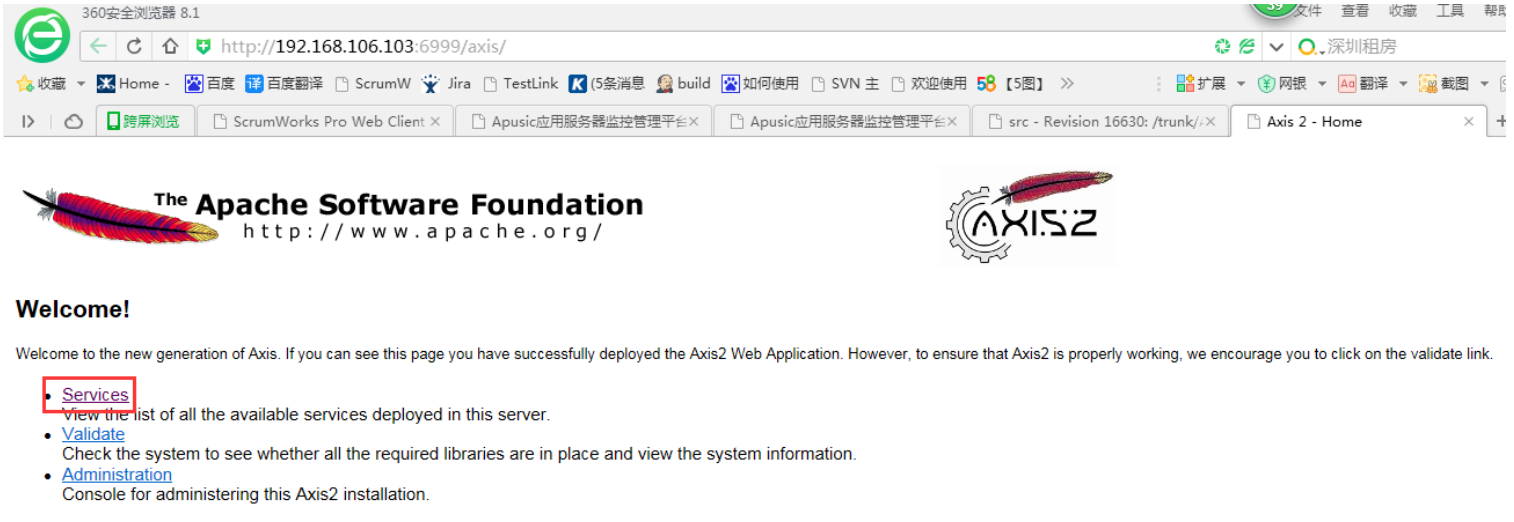
点击返回列表，回到应该管理界面，在axis应用那点击"查看应用模块"按钮



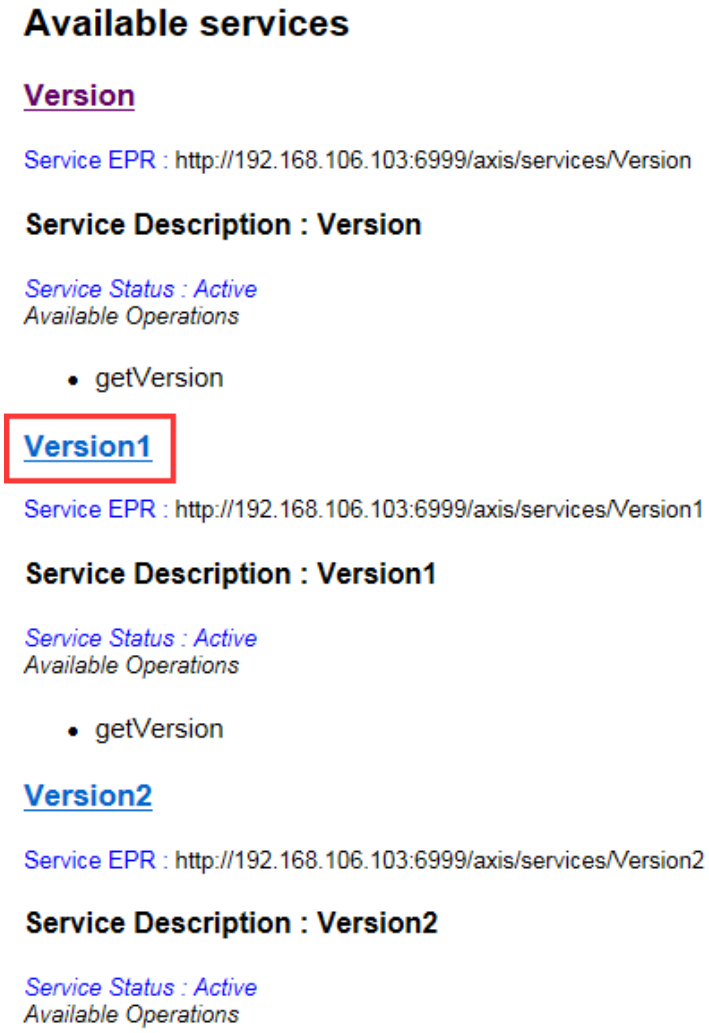
点击"访问应用"



点击“Service”按钮

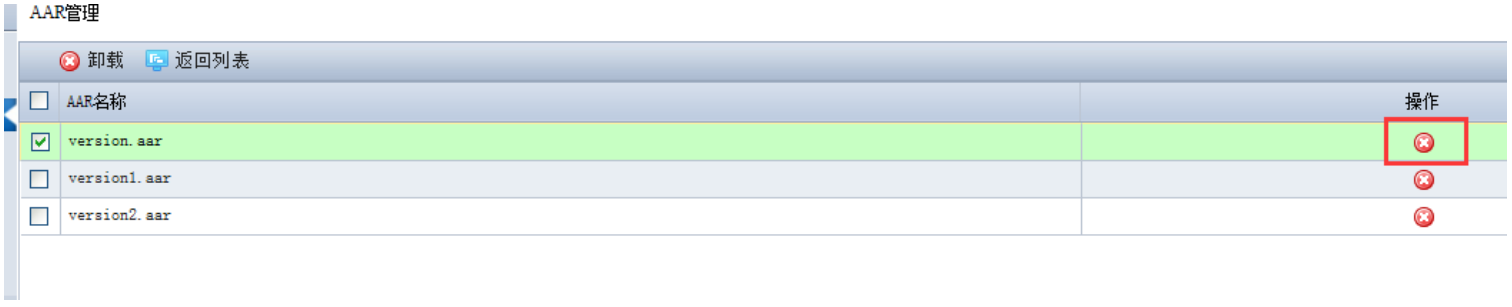


在列表中点击对应的aar包查看内容是否正确

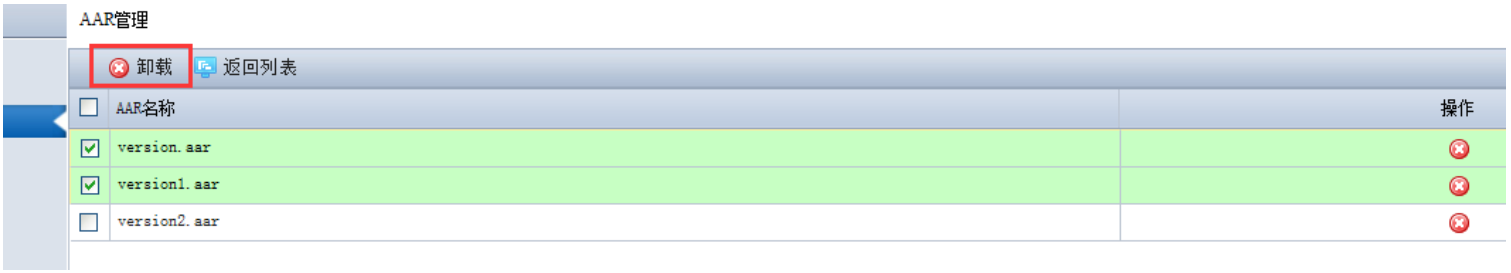


6. 删除AAR包：

单个删除：选择需要删除的AAR包，点击后面的删除图标按钮进行删除



批量删除：选择需要删除的aar包，点击左上角的删除按钮进行删除



7. 查看是否成功删除：回到应用管理界面，点击查看应用模块，访问应用，点击“Service“查看

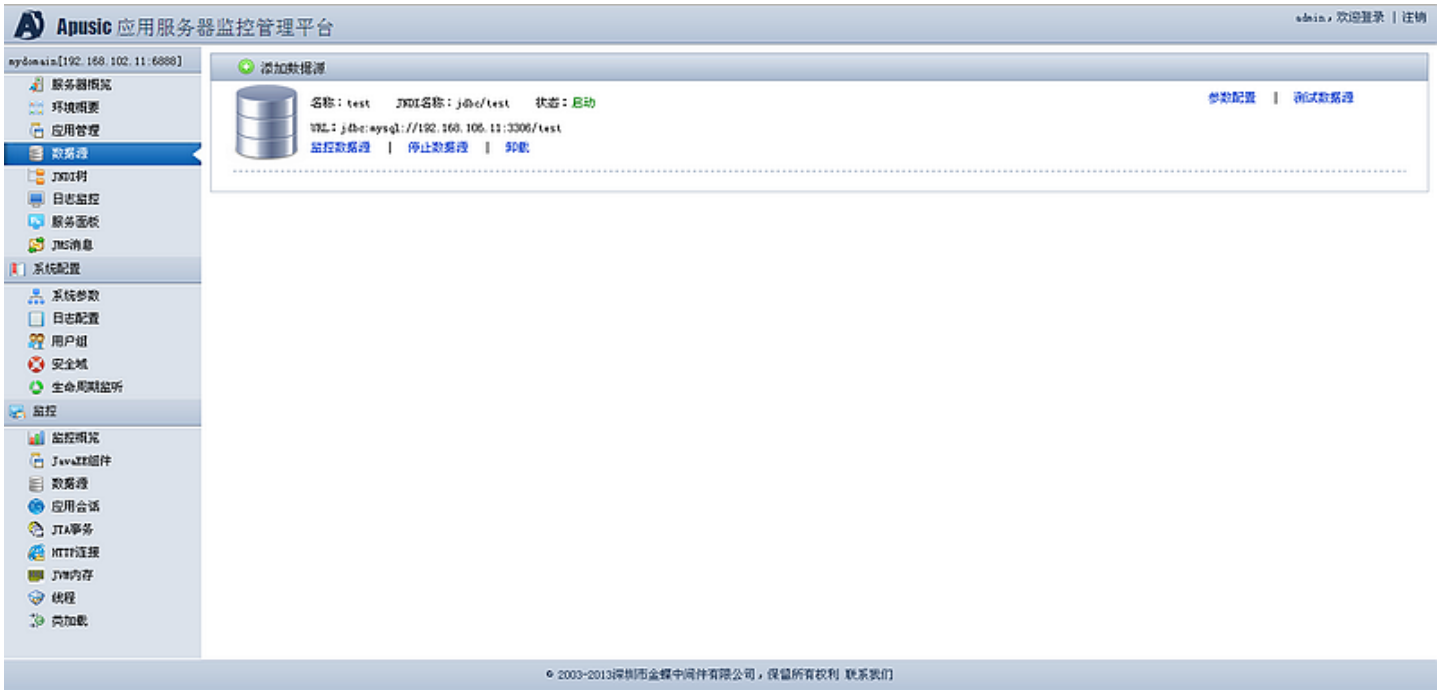


18.3 数据源管理

在数据源中存储了所有建立数据库连接的信息。就象通过指定文件名你可以在文件系统中找到文件一样，通过提供正确的数据源名称，你可以找到相应的数据库连接。

18.3.1 新增数据源

首先，打开Apusic应用服务器监控管理平台“数据源”操作界面：



然后，点击"添加数据源"按钮，填写数据源信息，并添加jar包：



如上图所示，以下配置项根据所需已存在数据库信息来填写。

属性名称	用途说明
基本信息	填写的数据源的名称，选择数据库类型以及驱动。
连接属性	填写的数据源的数据库信息，包括数据库名，主机，端口，用户名及密码等。

驱动属性	选择数据源所需要的驱动类。

18.3.2 配置和测试数据源

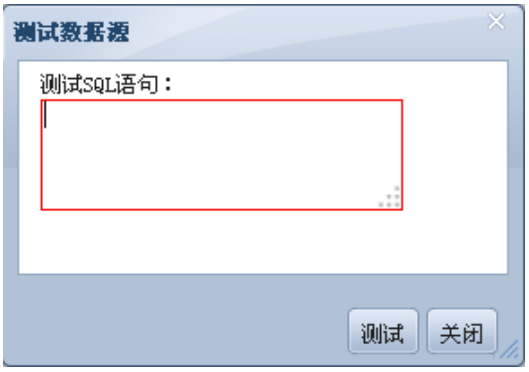
首先，打开管理监控台“数据源”操作界面，选择数据源后点击“参数配置”。



然后，修改数据源配置信息，在此除数据源名称不可更改外，其余信息均可修改，包括远程访问的用户和角色等。

最后，点击“完成”按钮，完成数据源的配置修改

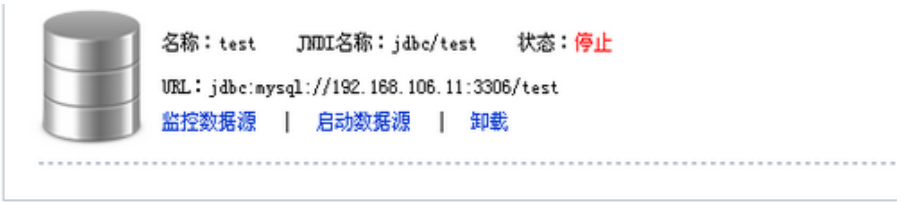
选定数据源点击“测试”按钮，在“测试连接”下的输入框输入测试SQL语句如“select*from table1”，点击测试按钮。



除此之外，也可以在数据源配置界面下进行测试，具体在此不做描述，大致与上相同。

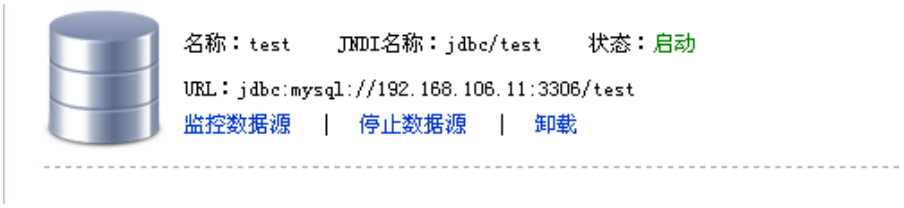
18.3.3 停止和启动数据源

1. 选定相应数据源，点击“停止数据源”按钮，在提示信息中点击“是”。



数据源状态变为“停止”，颜色变为灰色。

2. 点击“启动数据源”按钮，启动数据源。

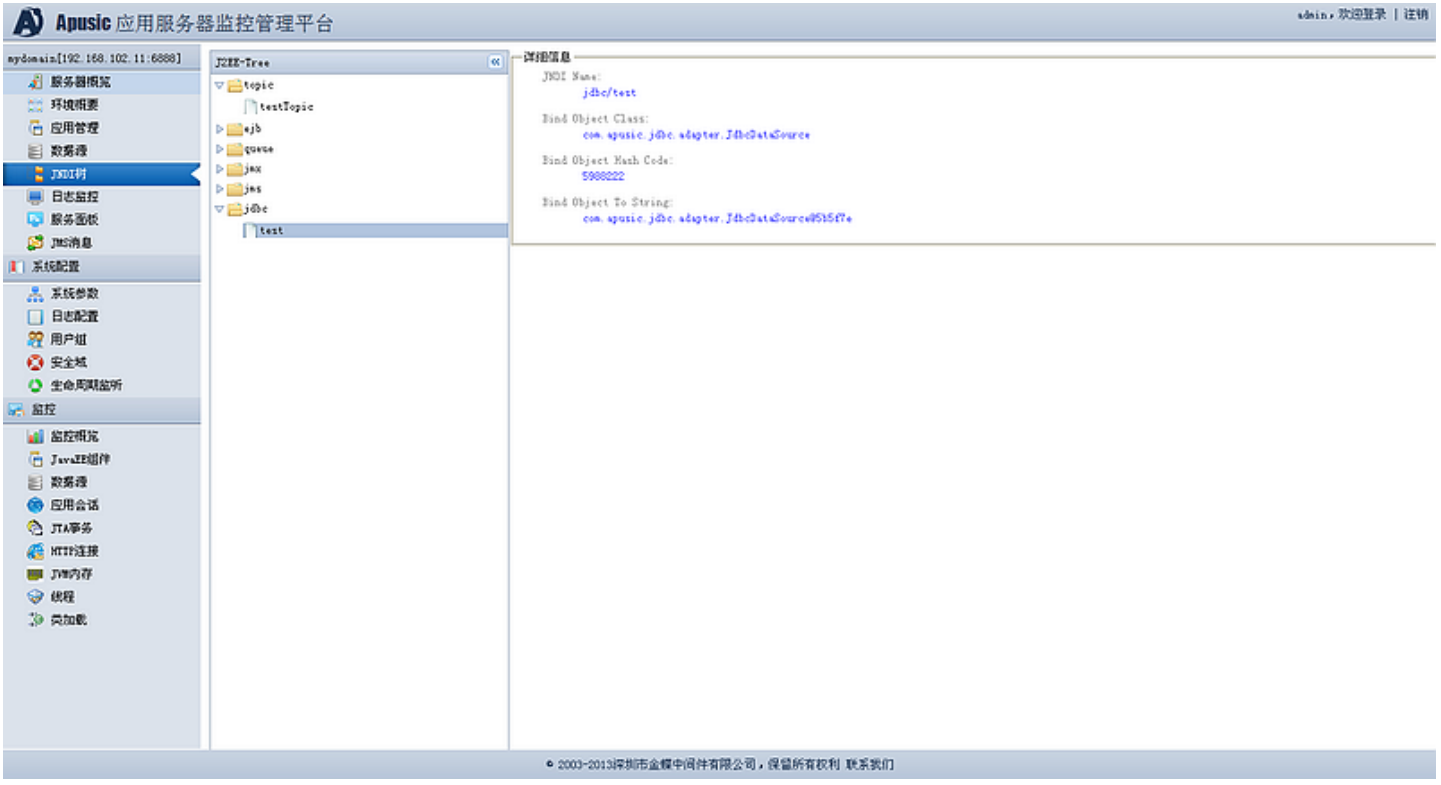


数据源状态变为“启动”，颜色变亮。

18.4 JNDI树

18.4.1 查看JNDI树

在JNDI树下可以查看系统的J2EE-Tree，其中包括topic，ejb,queue，JDBC等。具体如下图所示：

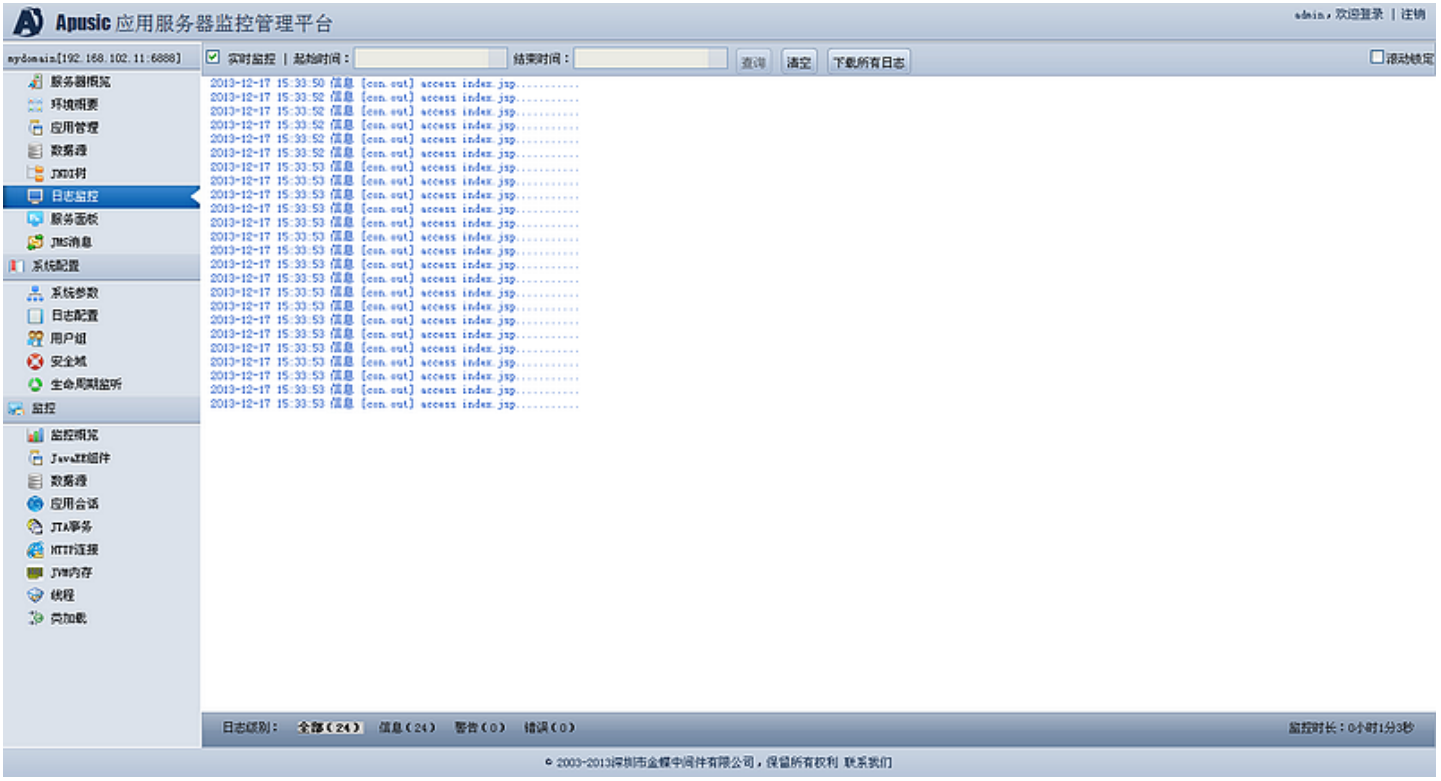


18.5 日志监控

提供日志信息的实时监控或者分时间段查看，同时提供日志信息的级别分类和日志下载功能。

18.5.1 日志监控

打开Apusic应用服务器监控管理平台，进入到"日志监控"操作界面下：



在此可以实时监控日志信息，或者根据时间段来查看日志信息等。

实时监控：勾选实时监控时，右下角的监控时长在打开页面时候或者勾选时候开始计时。若在实时监控下用户对服务器或者应用等操作产生日志则会即时打印（系统默认是实时监控）。

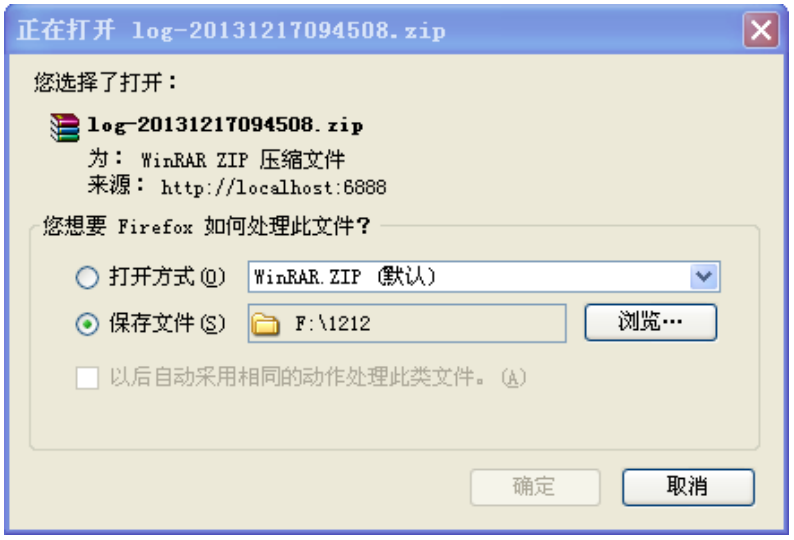
清空： 点击清空按钮，则将页面显示的所有日志做清空操作。

滚动锁定：将日志滚动锁定，不随新日志出现而移动到日志底端。

日志级别按钮：在左下方存在不同颜色标明的日志级别按钮，分别为all,debug,warn,error。分别点击相应按钮会筛选相应级别的日志信息。

18.5.2 下载日志

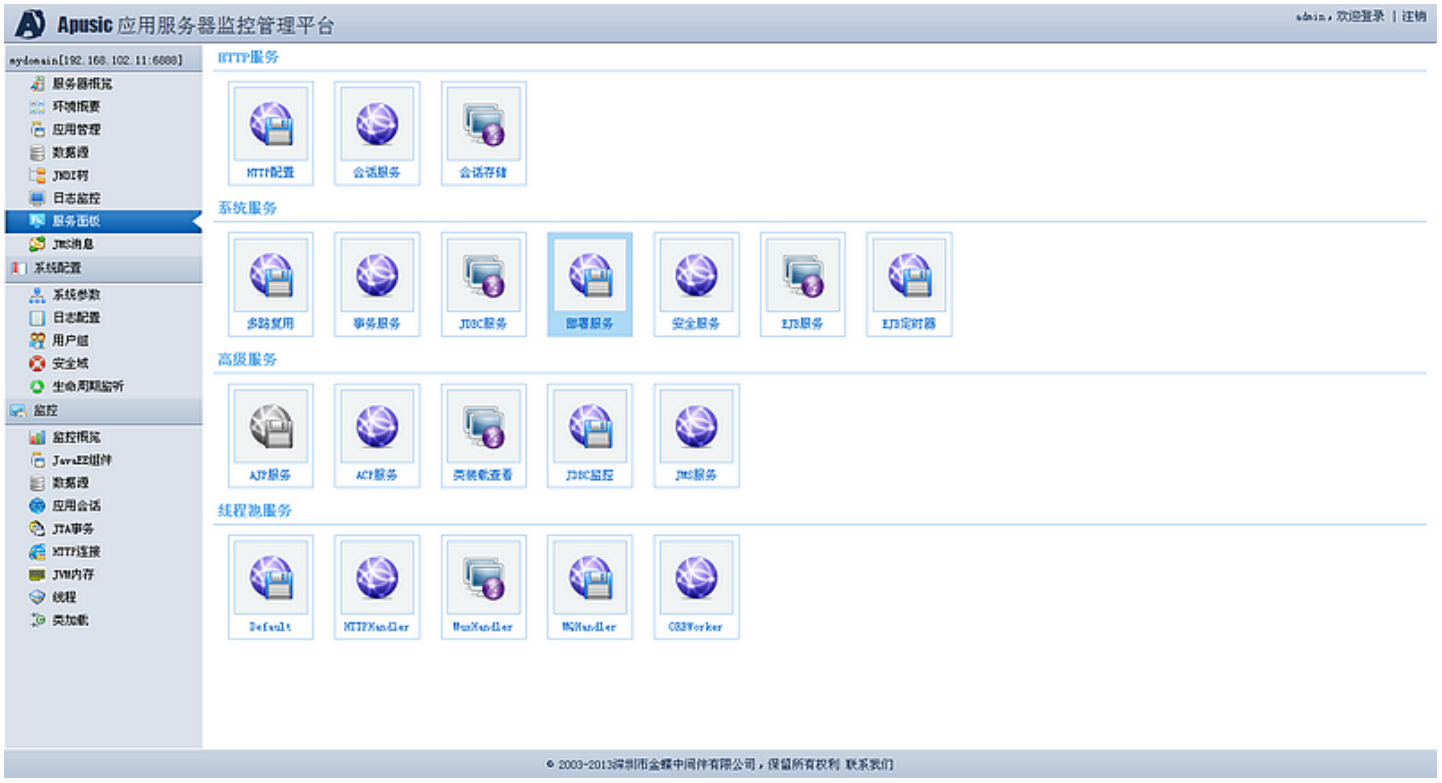
打开Apusic应用服务器监控管理平台，进入到"日志监控">"下载所有日志"操作界面下：



解压缩后打开文件即可查看服务器的所有日志信息。

18.6 服务面板配置

打开Apusic应用服务器监控管理平台，进入到“服务面板”操作界面下：

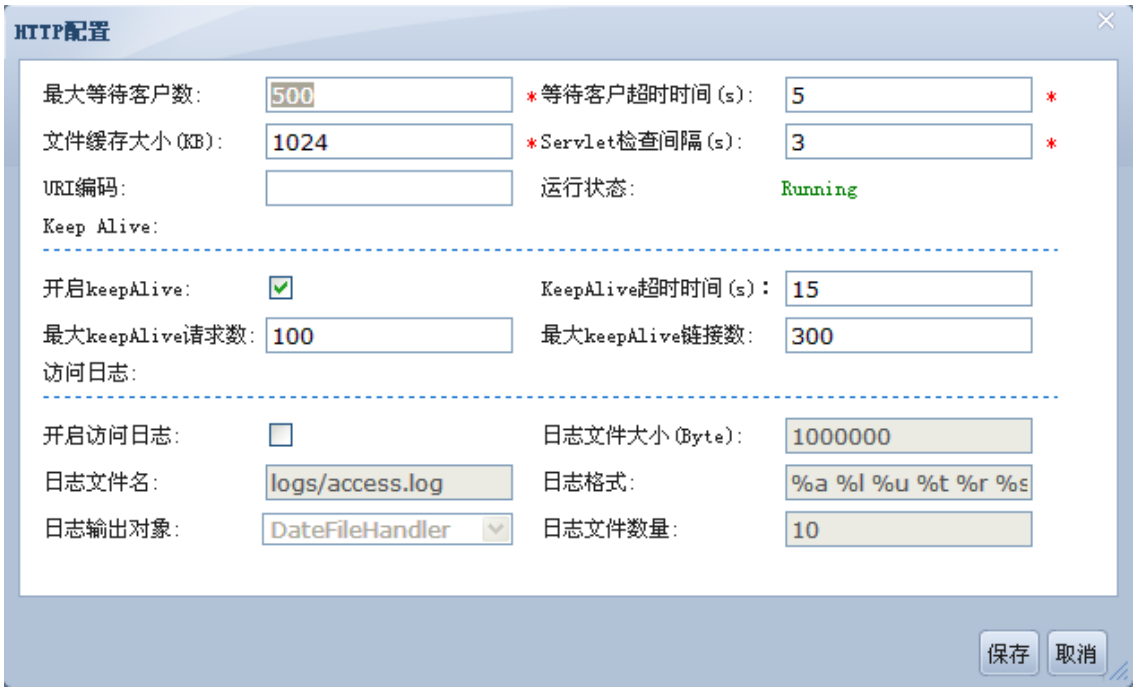


点击每一个图标均会进入到相应的配置界面，在每一个配置页面下可以修改相应apusic服务的配置。

18.6.1 HTTP服务配置

1. HTTP配置

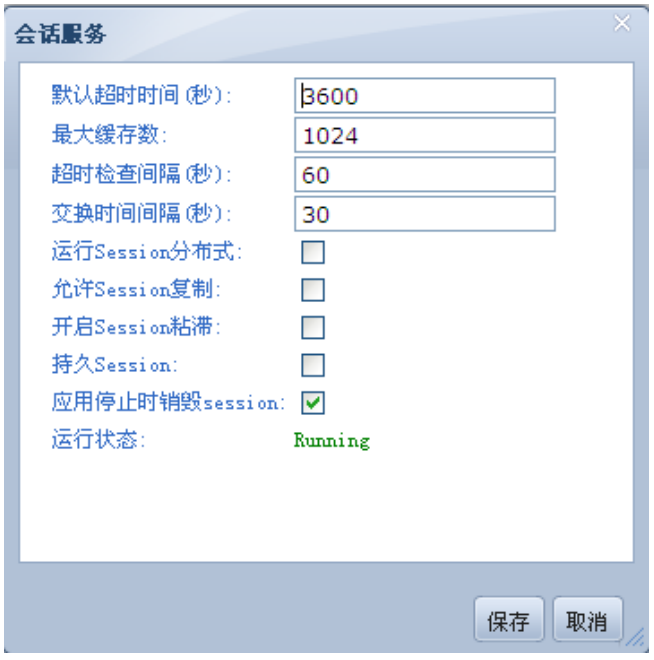
首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“HTTP服务”>“HTTP配置”操作界面下：



然后修改相应配置项，然后点击"保存"完成修改。在HTTP配置下，可以配置Keep Alive和访问日志等。

2. 会话服务配置

首先，打开Apusic应用服务器监控管理平台，进入到"服务面板">"HTTP服务">"会话服务"操作界面下：



然后，修改相应配置项，然后点击"保存"完成修改。

3. 会话存储服务配置

首先，打开Apusic应用服务器监控管理平台，进入到"服务面板">"HTTP服务">"会话存储服务"操作界面下：



存储方式分为存储到文件，存储到DB，SQL实现和Cache实现这四种。其中，存储到文件是使用AAS实现，存储到DB使用berkeley DB存储，SQL实现使用关系型数据库存储，Cache则使用使用MemoryBase存储。修改选择相应配置项，然后点击“保存”完成修改。

18.6.2 系统服务配置

1. 多路复用服务配置

首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“系统服务”>“多路复用”操作界面下：



然后，修改多路复用的配置以及开启SSL并配置。修改相应配置项，然后点击“保存”完成修改。

2. 事务服务配置

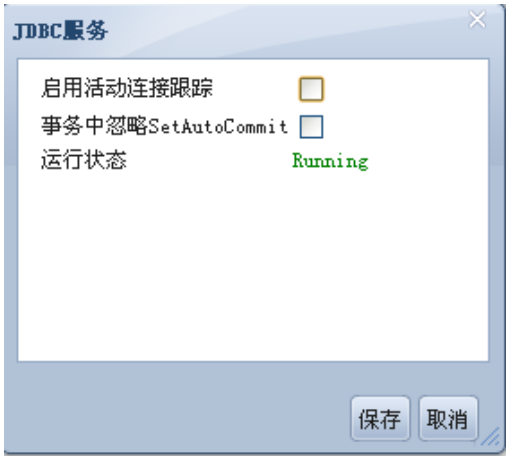
首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“系统服务”>“事务服务图标”操作界面下：



然后，修改相应配置项，然后点击“保存”完成修改。

3. JDBC服务配置

首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“系统服务”>“JDBC服务”操作界面下：



然后，修改相应配置项，然后点击“保存”完成修改。

4. 部署服务配置

首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“系统服务”>“部署服务”操作界面下：



然后，修改相应配置项，然后点击“保存”完成修改。

5. 安全服务配置

首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“系统服务”>“安全服务”操作界面下：



然后，修改相应配置项，然后点击“保存”完成修改。

6. EJB服务配置

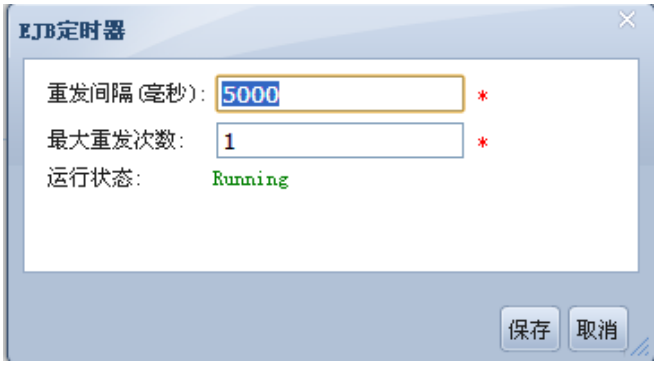
首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“系统服务”>“EJB服务”操作界面下：



然后，对EJB服务的配置项进行更改，修改相应配置项，然后点击“保存”完成修改。

7. EJB定时器服务配置

首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“系统服务”>“EJB定时器服务”操作界面下：



然后，修改相应配置项，然后点击“保存”完成修改。

18.6.3 高级服务配置

1. AJP服务配置

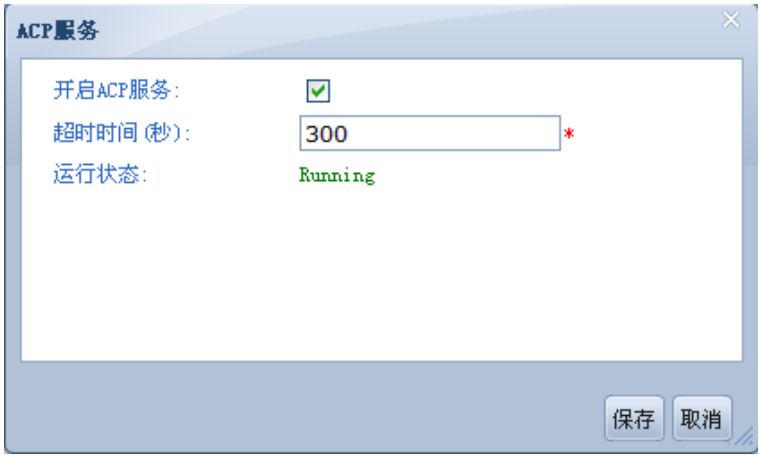
首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“高级服务”>“AJP服务”操作界面下：



然后，修改相应配置项，然后点击“保存”完成修改。

2. ACP服务配置

首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“高级服务”>“ACP服务”操作界面下：



开启ACP服务并提交更新，然后，修改相应配置项，然后点击“保存”完成修改。

3. 类装载查看服务配置

首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“高级服务”>“类装载查看”操作界面下：

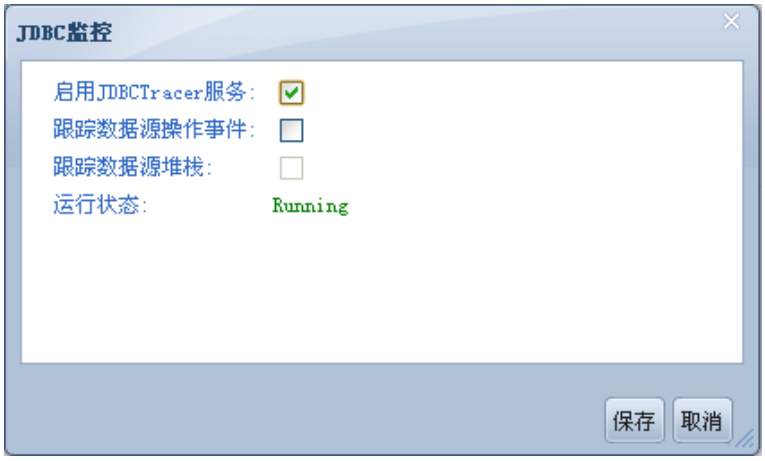


然后修改相应配置项，然后点击“保存”完成修改。

启用类加载查看器功能需要在“启用ClassLoaderViewer”后打勾。

4. JDBC监控服务配置

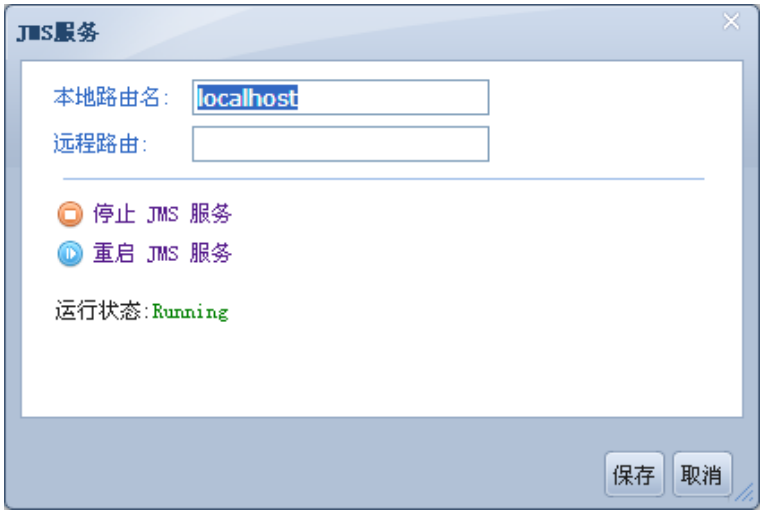
首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“高级服务”>“JDBC监控”操作界面下：



然后点击“保存”完成修改。启用JDBCTracer功能需要在“启用JDBCTracer”后打勾修改相应配置项

5. JMS服务配置

首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“高级服务”>“JMS服务”操作界面下：



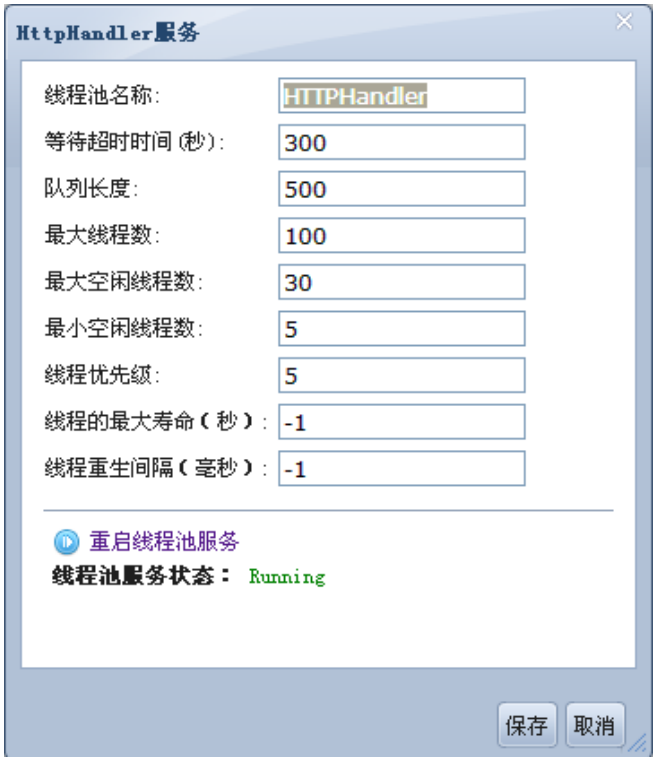
然后点击“保存”完成修改。也可以对JMS服务进行启动，停止，重启操作。

18.6.4 线程池服务

Apusic 应用服务器使用了五种池：default、MuxHandler、HTTPHandler、MQHandler和ORBHandler。

1. HTTPHandler服务配置

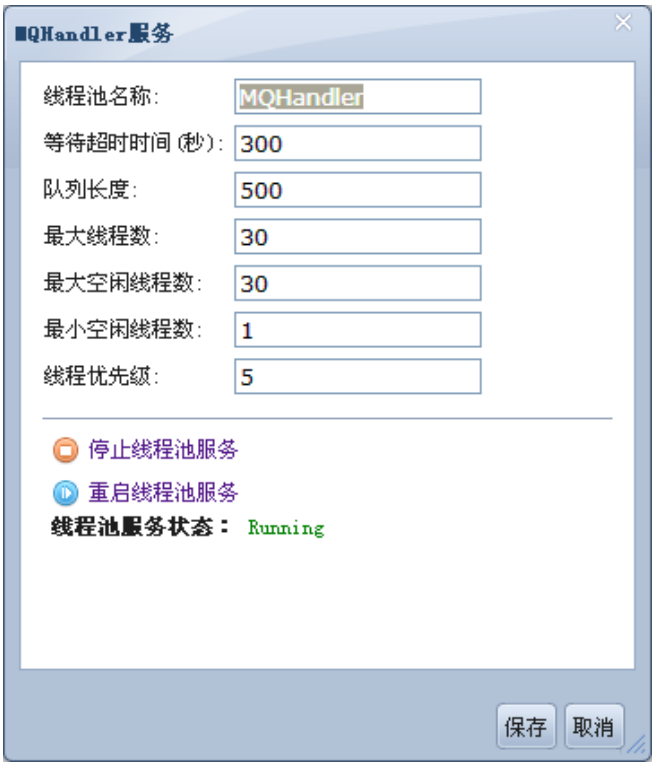
首先，打开Apusic应用服务器监控管理平台，进入到“服务面板”>“线程池服务”>“HTTPHandler”操作界面下：



HTTPHandler是HTTP服务的线程池。修改HTTPHandler线程池相应配置项或者对HTTPHandler服务进行重启操作，然后点击“保存”完成配置修改。

2. JMSHandler服务配置

打开Apusic应用服务器监控管理平台，进入到“服务面板”>“线程池服务”>“MQHandler”操作界面下：



MQHandler是MQ服务的线程池。修改相应配置项，也可以对MQHander服务进行启动，停止，重启操作，然后点击“保存”完成修改。

3. ORBWorker服务配置

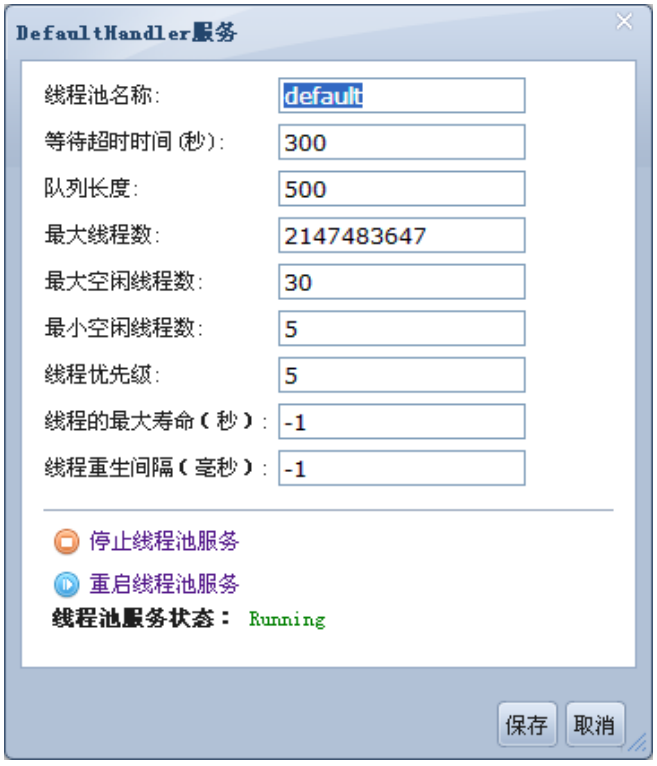
打开Apusic应用服务器监控管理平台，进入到“服务面板”>“线程池服务”>“ORBWorker”操作界面下：



ORBWorker是ORB服务的线程池。修改其相应配置项，也可以对ORBWorker重启操作，然后点击“保存”完成修改。

4. Default服务配置

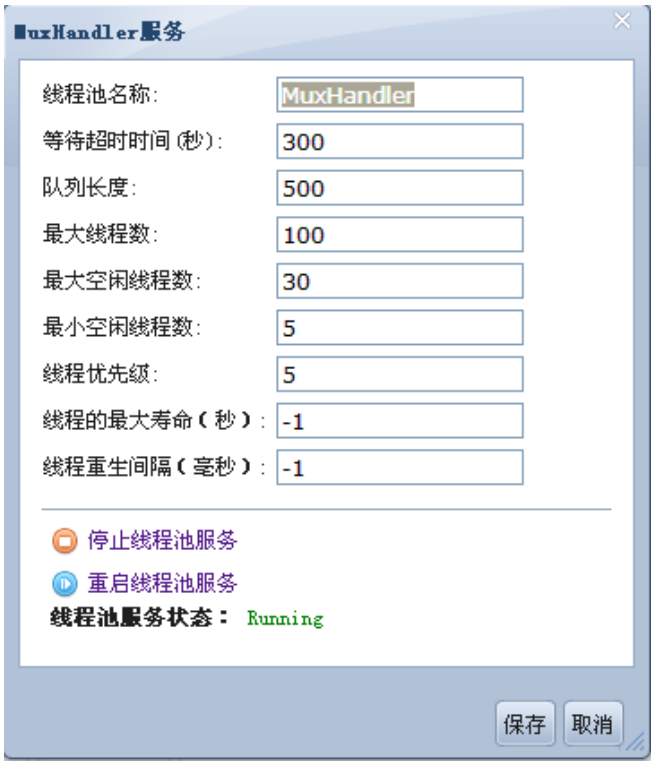
打开Apusic应用服务器监控管理平台，进入到“服务面板”>“线程池服务”>“Default”操作界面下：



Default线程池是Apusic服务器默认的线程池。修改其相应配置项，也可以对Default线程池进行启动，停止，重启操作，然后点击“保存”完成修改。

5. MuxHandler服务配置

打开Apusic应用服务器监控管理平台，进入到“服务面板”>“线程池服务”>“MuxHandler”操作界面下：

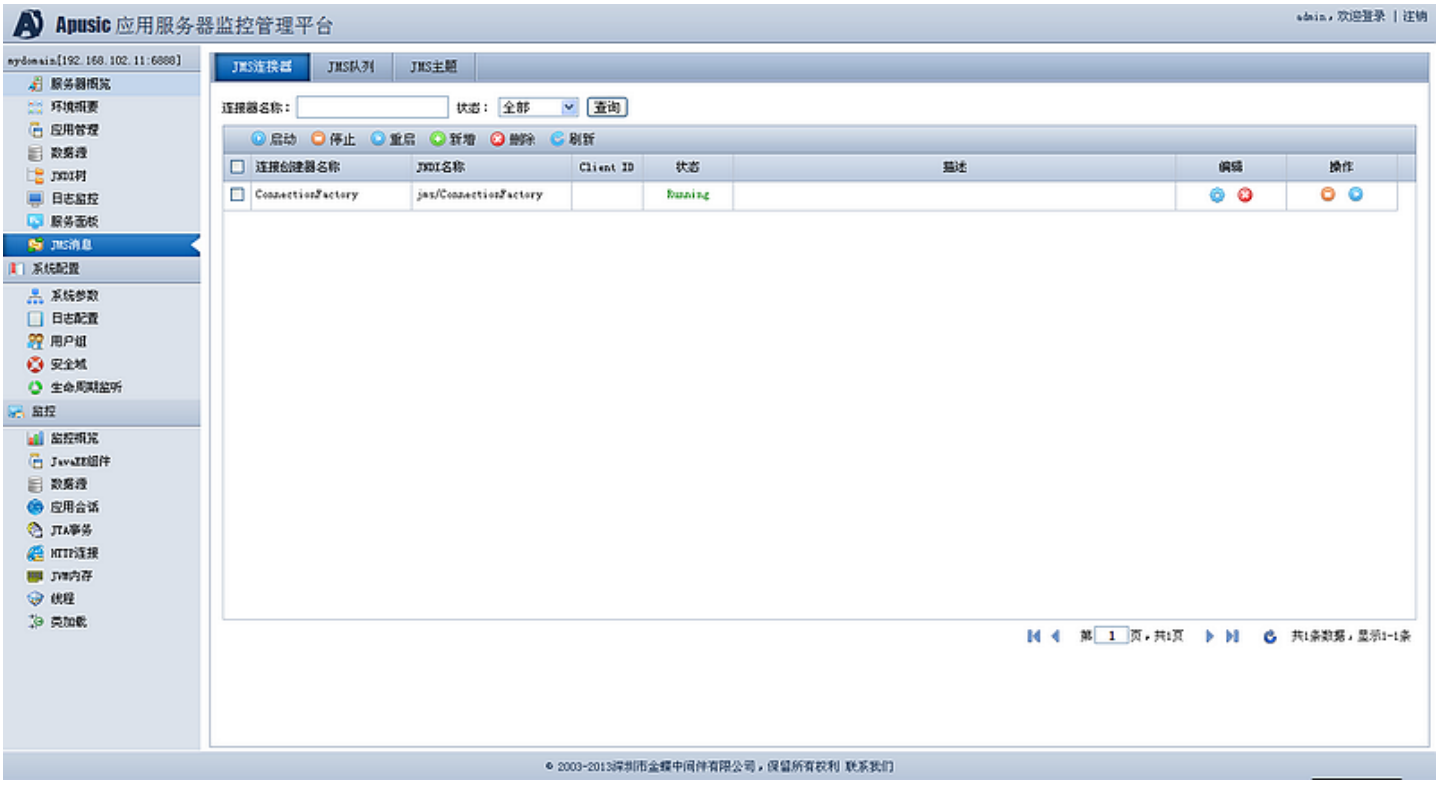


MuxHandler是多路复用的线程池，主要是将请求向不同的服务，如：HTTP、JMS、ORB等服务上分配。修改相应配置项，也可以对MuxHandler服务进行启动，停止，重启操作，然后点击“保存”完成修改。

18.7 JMS消息服务配置

18.7.1 管理JMS消息服务

打开Apusic应用服务器监控管理平台，“JMS消息”操作界面:



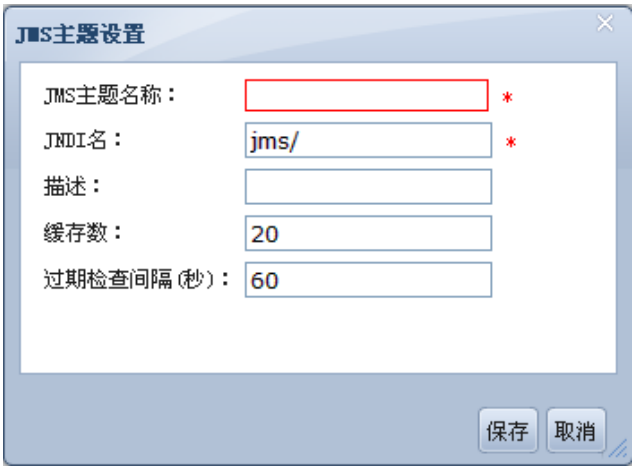
图"JMS消息"操作界面

新增JMS连接器：打开Apusic应用服务器监控管理平台，点击"JMS消息"> "JMS 连接器"，点击"新增"：



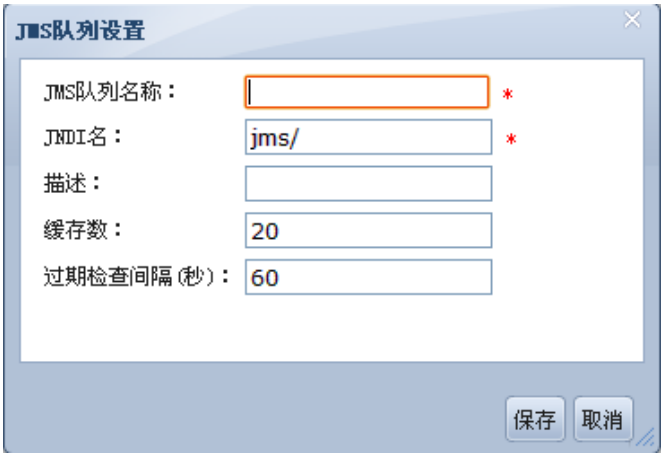
填写JMS连接器信息后（其中连接器名称，JNDI名称为必填项），点击"保存"按钮完成连接器的创建。

新增JMS主题：打开Apusic应用服务器监控管理平台，点击"JMS消息"> "JMS 主题"，点击"新增"：



填写JMS主题信息后（其中主题名称、缓存数、过期检查间隔为必填项，缓存数默认为20，过期检查间隔为60），点击"确定"按钮完成连接器的创建。

新增JMS队列的：打开Apusic应用服务器监控管理平台，点击"JMS消息"> "JMS 队列"，点击"新增"：



填写JMS队列信息后（其中队列名称、缓存数、过期检查间隔为必填项，缓存数默认为20，过期检查间隔为60），点击"确定"按钮完成连接器的创建。

18.7.2 配置JMS服务

配置JMS 连接器

首先,打开Apusic应用服务器监控管理平台，点击"JMS消息"> "JMS 连接器"，进入到JMS操作界面下：

然后，选择相应连接器后点击"编辑"下的"设置属性"按钮：



在修改JMS连接器界面下，可以对连接器的基本信息和高级属性进行相应的修改，点击“确定”完成配置修改。

配置“JMS 主题

首先,打开Apusic应用服务器监控管理平台，点击“JMS消息”> “JMS 主题”，进入到JMS操作界面下：

然后，选择相应主题后点击“编辑”下的“设置属性”按钮：

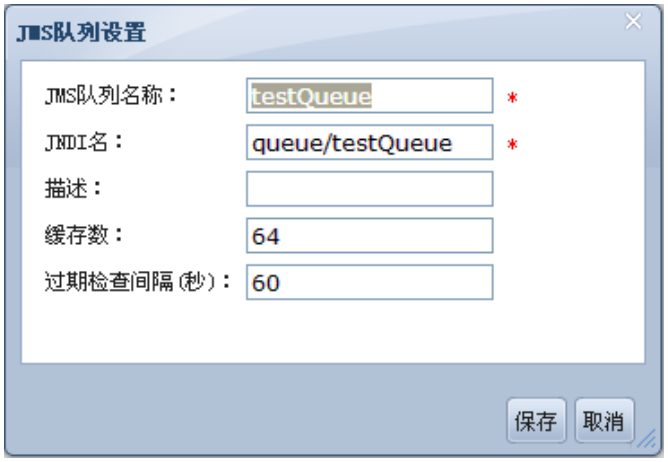


在修改JMS主题界面下，可以对主题的基本信息进行相应的修改，点击“确定”完成配置修改。

配置JMS 队列

首先,打开Apusic应用服务器监控管理平台，点击“JMS消息”> “JMS 队列”，进入到JMS操作界面下：

然后，选择相应连接器后点击“编辑”下的“设置属性”按钮：



在修改JMS队列界面下，可以对队列的基本信息进行相应的修改，点击“确定”完成配置修改。

18.8 系统配置

18.8.1 系统参数配置

1. JVM内存参数修改：

打开Apusic应用服务器监控管理平台，进入到“系统配置”>“系统参数”操作界面下：



可以对Apusic服务器所在的java虚拟机的内存参数进行修改，恢复默认值等操作 。

2. Apusic参数配置：Apusic服务器在启动时可以设置自定义参数。参数分为两种类型，一种是定义在启动脚本中，另一种定义在web.xml文件中。另外也存在一些参数既可以定义在启动脚本中，也可以定义在web.xml中。

1. 添加Apusic参数

打开Apusic应用服务器监控管理平台，进入到“系统配置”>“系统参数”>“Apusic参数”>“新增”到新增Apusic参数界面下：



选择“参数类型”、“参数名称”，填写参数值后点击“确定”增加Apusic参数。

2. 修改Apusic参数

打开Apusic应用服务器监控管理平台，进入到“系统配置”>“系统参数”>“Apusic参数”>“操作”>“修改此Apusic参数”按钮到修改Apusic参数界面下：



选定相应Apusic参数修改参数值后点击“确定”，完成相应的修改。

3. 删除Apusic参数

首先，打开Apusic应用服务器监控管理平台，进入到“系统配置”>“系统参数”>“Apusic参数”下，选定相应的参数后点击“删除”按钮来删除选定的Apusic参数或者通过点击选定Apusic参数下的“操作”>“删除此Apusic参数”按钮来进行删除。

然后，在弹出的信息中点击“是”，删除相应的参数。

4. 刷新Apusic参数

打开Apusic应用服务器监控管理平台，进入到“系统配置”>“系统参数”>“Apusic参数”>“刷新”来刷新Apusic参数值。

18.8.2 日志配置

Apusic服务器可以配置监控台输出，日期文件输出，日志文件输出，端口输出日志。

1. 修改全局日志配置：

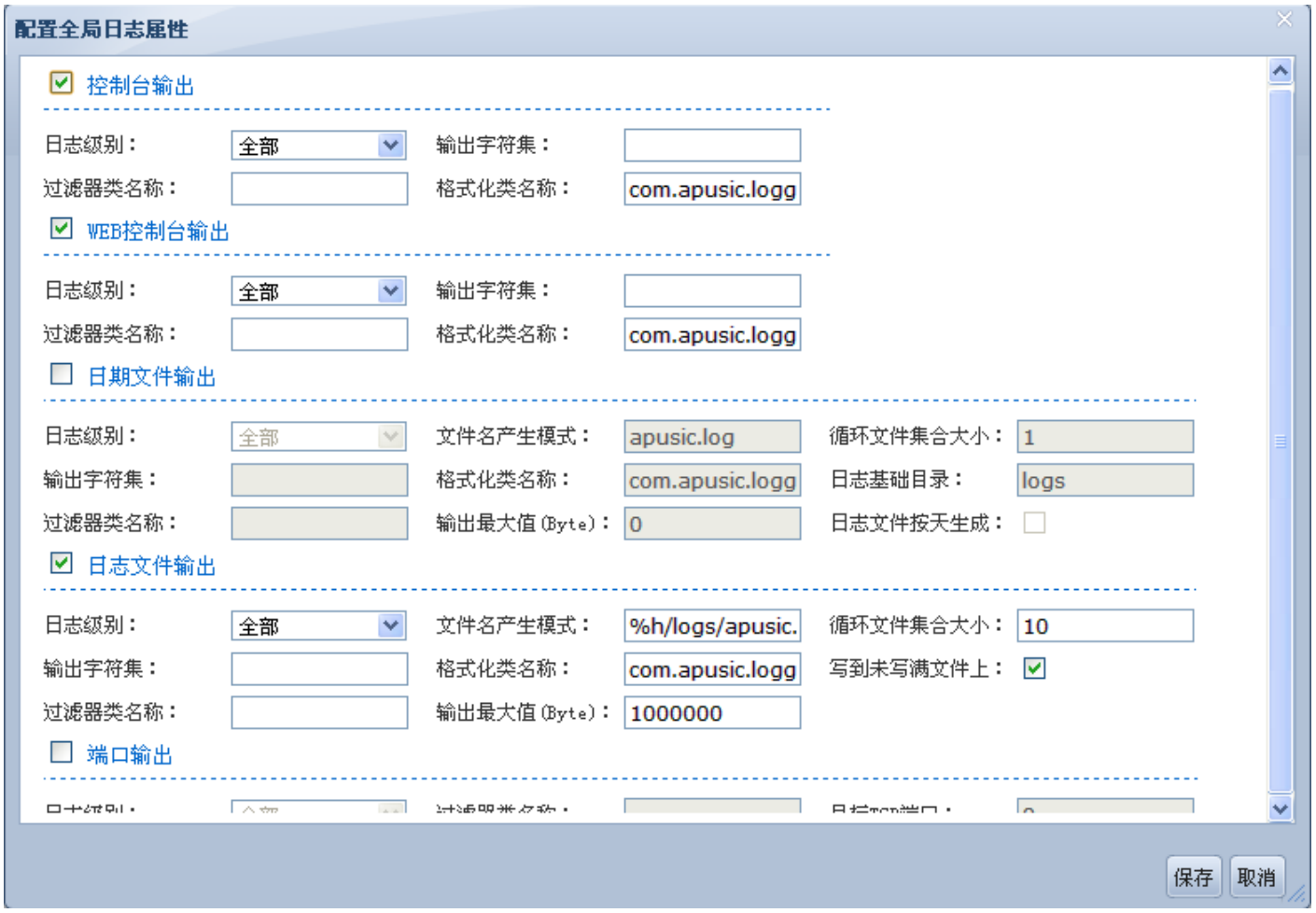
首先，打开Apusic应用服务器监控管理平台，进入到“系统配置”>“日志”>进入到全局日志设置界面：



然后，修改全局日志设置相应信息后点击“保存”按钮。

2. 配置全局日志属性

首先，打开Apusic应用服务器监控管理平台，进入到“系统配置”>“日志”>“配置全局日志属性”进入到配置全局日志属性界面：



然后，选择相应的信息修改后点击“保存”。

3．配置日志信息

(a)新增日志

首先，打开Apusic应用服务器监控管理平台，进入到“系统配置”> “日志”>“新增”进入到配置日志信息界面：



填写相应信息后点击提交完成新日志的配置增加。

(b)修改日志

首先，打开Apusic应用服务器监控管理平台，进入到“系统配置”>“日志”>“操作”下的“编辑”按钮进入到配置日志信息界面：



然后，修改选定日志名的配置信息后点击提交，完成日志配置信息的修改。

(c)删除日志

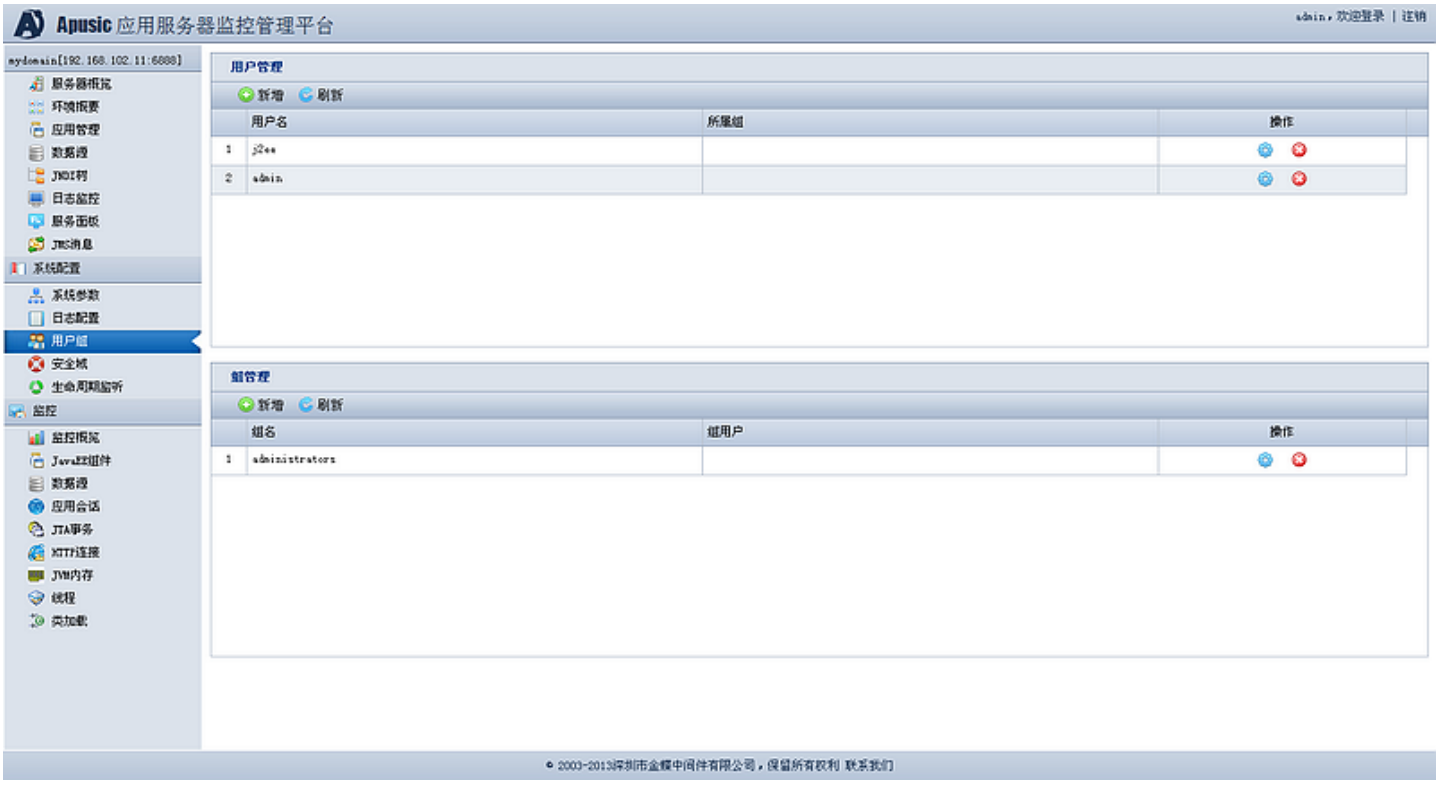
打开Apusic应用服务器监控管理平台，进入到“系统配置”>“日志”操作界面下，然后选定要删除的日志点击“操作”下的“删除”按钮，在提示信息中点击“是”。

18.8.3 用户组

1. 用户管理

(a)新增用户

首先，打开Apusic应用服务器监控管理平台，进入到“系统配置”> “用户组”下操作界面：



然后，点击新增用户



最后，填写信息后点击"保存"完成新增用户。

(b)修改用户

首先，打开Apusic应用服务器监控管理平台，进入到"系统配置"> "用户组"下操作界面，选定要修改的用户下的操作列表下的"修改此用户"按钮：

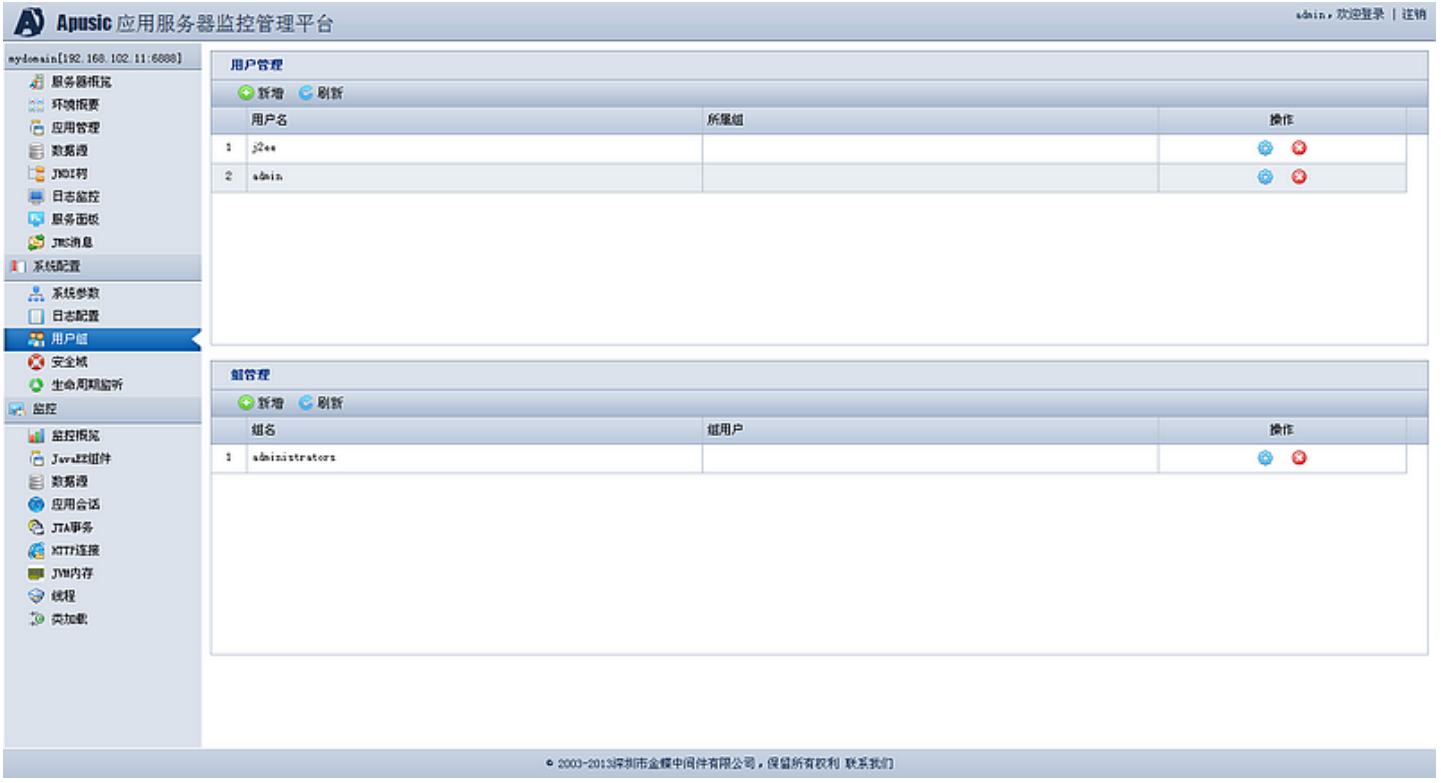


然后，修改相应信息后点击“保存”后完成修改。

2. 组管理

(a)新增组

首先，打开Apusic应用服务器监控管理平台，进入到“系统配置”> “用户组”下操作界面：



然后，点击组管理下的“新增组”按钮，进入到新增组操作界面下：



最后，填写组信息后点击“保存”后完成组的新建增加。

(b)修改组

首先，打开Apusic应用服务器监控管理平台，进入到“系统配置”> “用户组”下操作界面，选定组列表下的要修改的组后点击其操作列下的“修改此组”按钮：

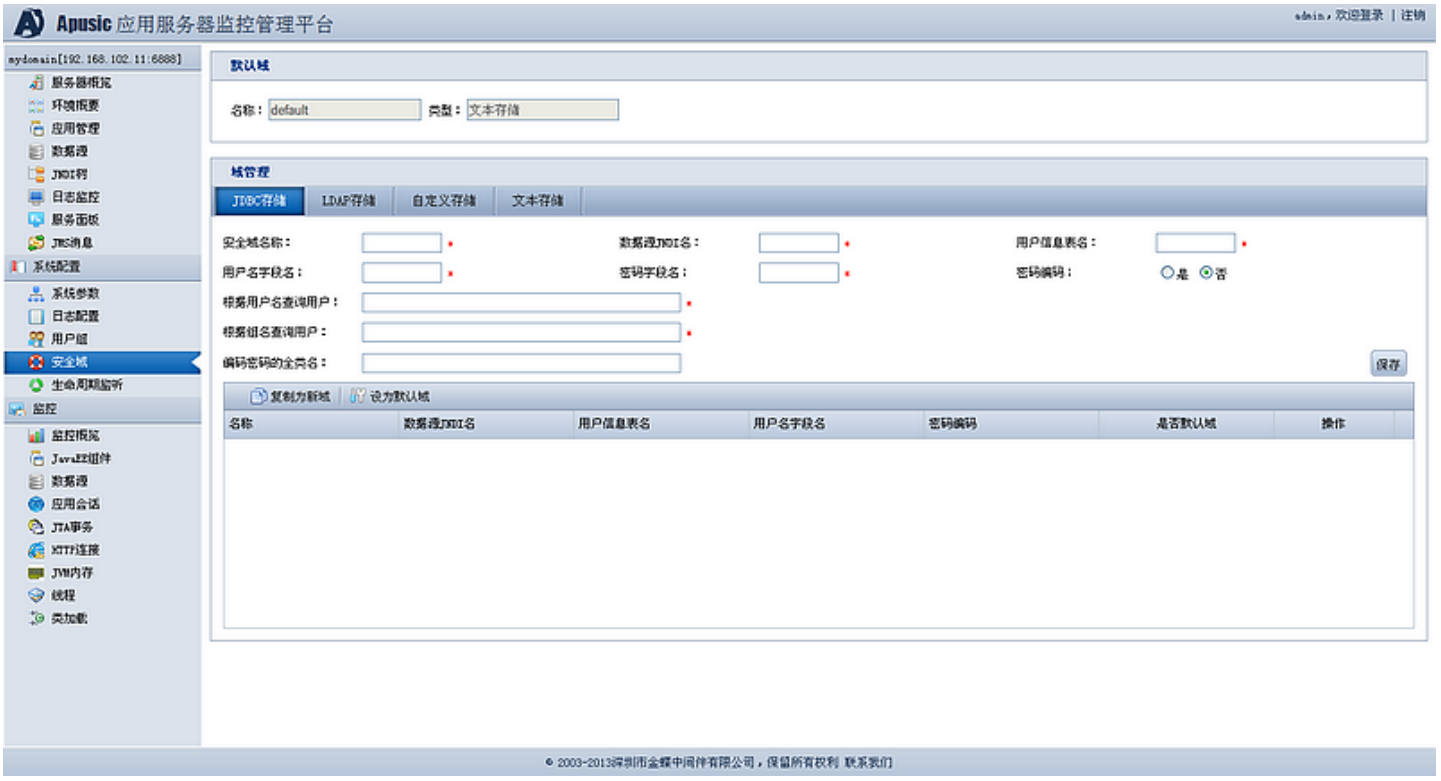


然后，选择要修改的用户列表后点击“保存”按钮后完成组的修改。

18.8.4 安全域参数配置

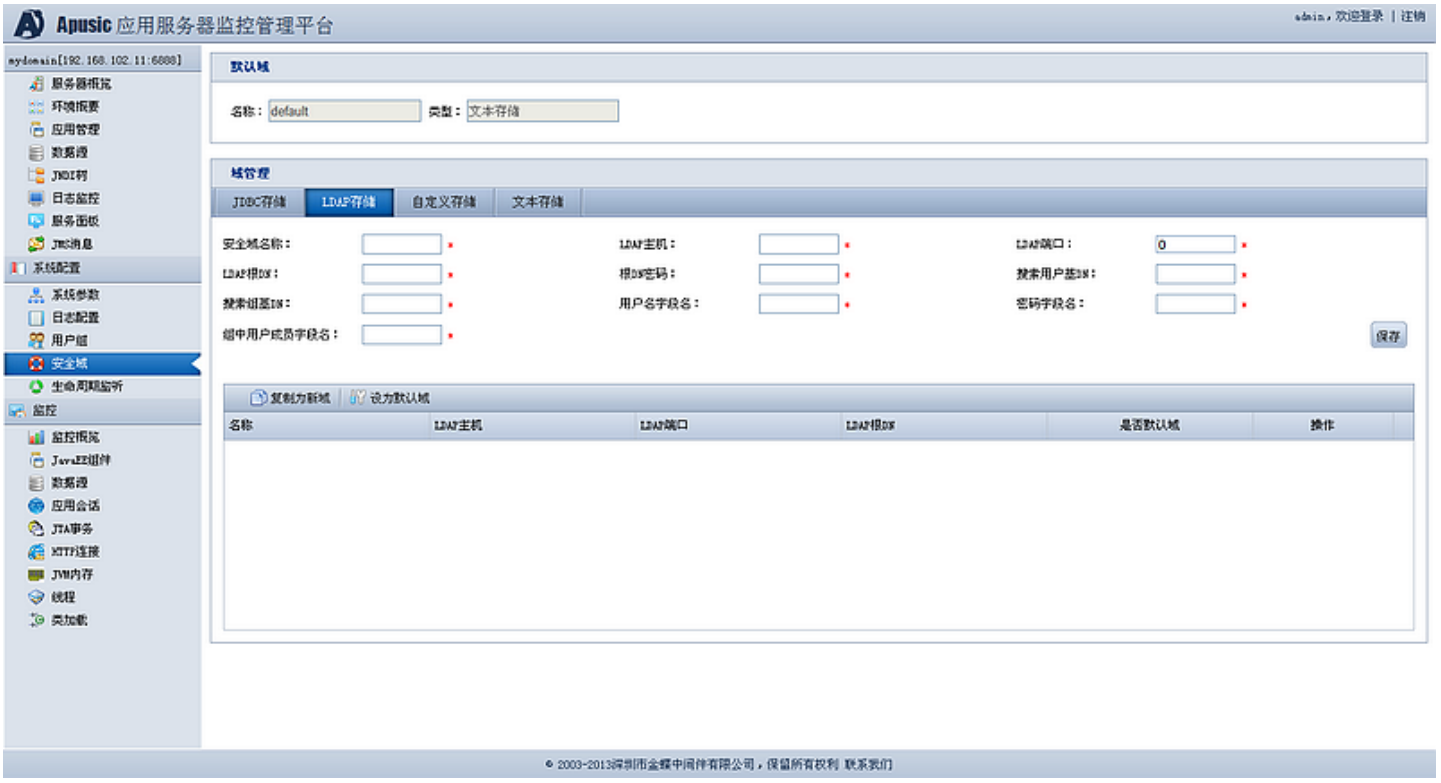
Apusic 应用服务器内置了四种安全提供程序实现，包含文件、JDBC以及LDAP存储，自定义存储，以方便用户根据自身业务进行选择。

1. JDBC存储配置：打开Apusic应用服务器监控管理平台，进入到“系统配置”>“安全域”>域管理>“JDBC存储”操作界面下：



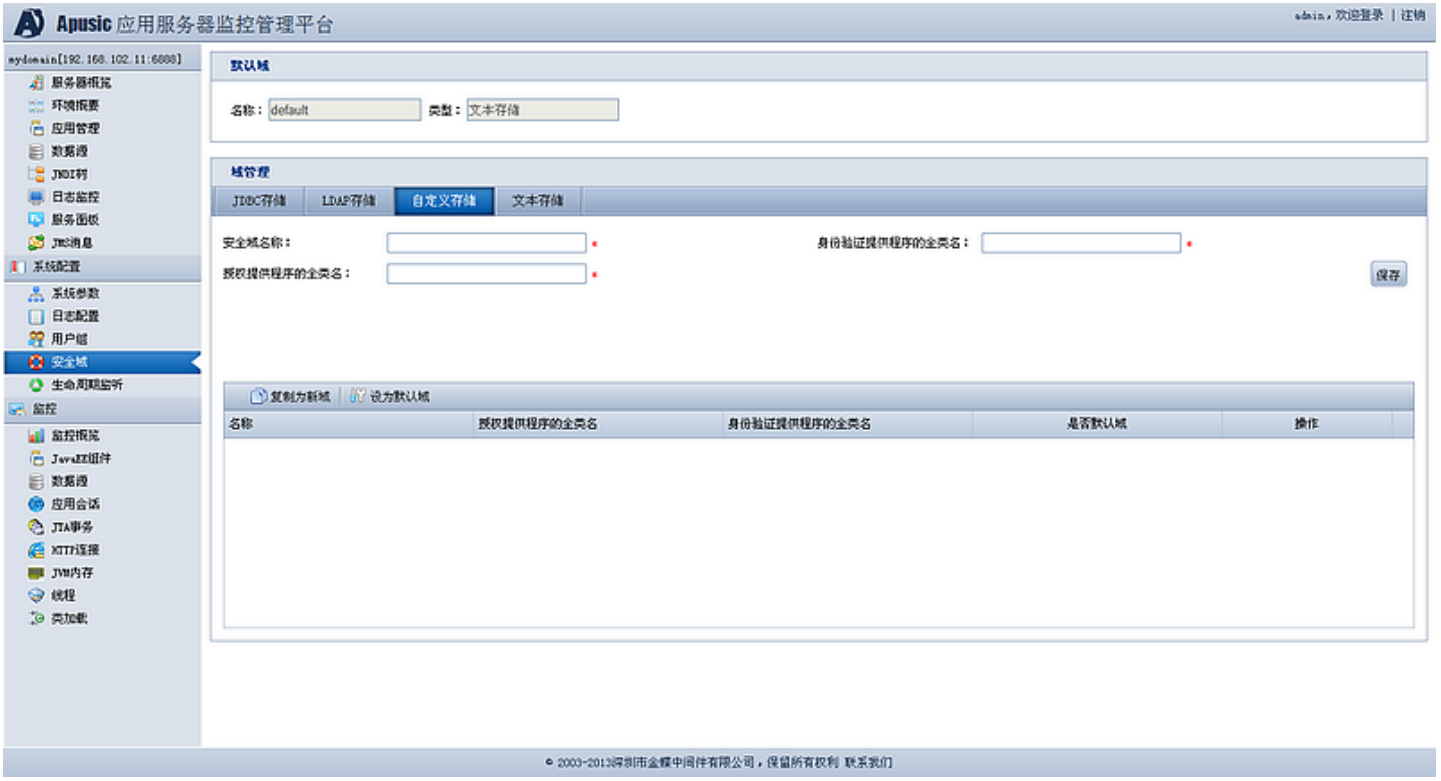
在此，我们可以对JDBC存储下的域进行新增，复制，设为默认域等操作。

2. LDAP存储配置：打开Apusic应用服务器监控管理平台，进入到“系统配置”>“安全域”>域管理>“LDAP存储”操作界面下：



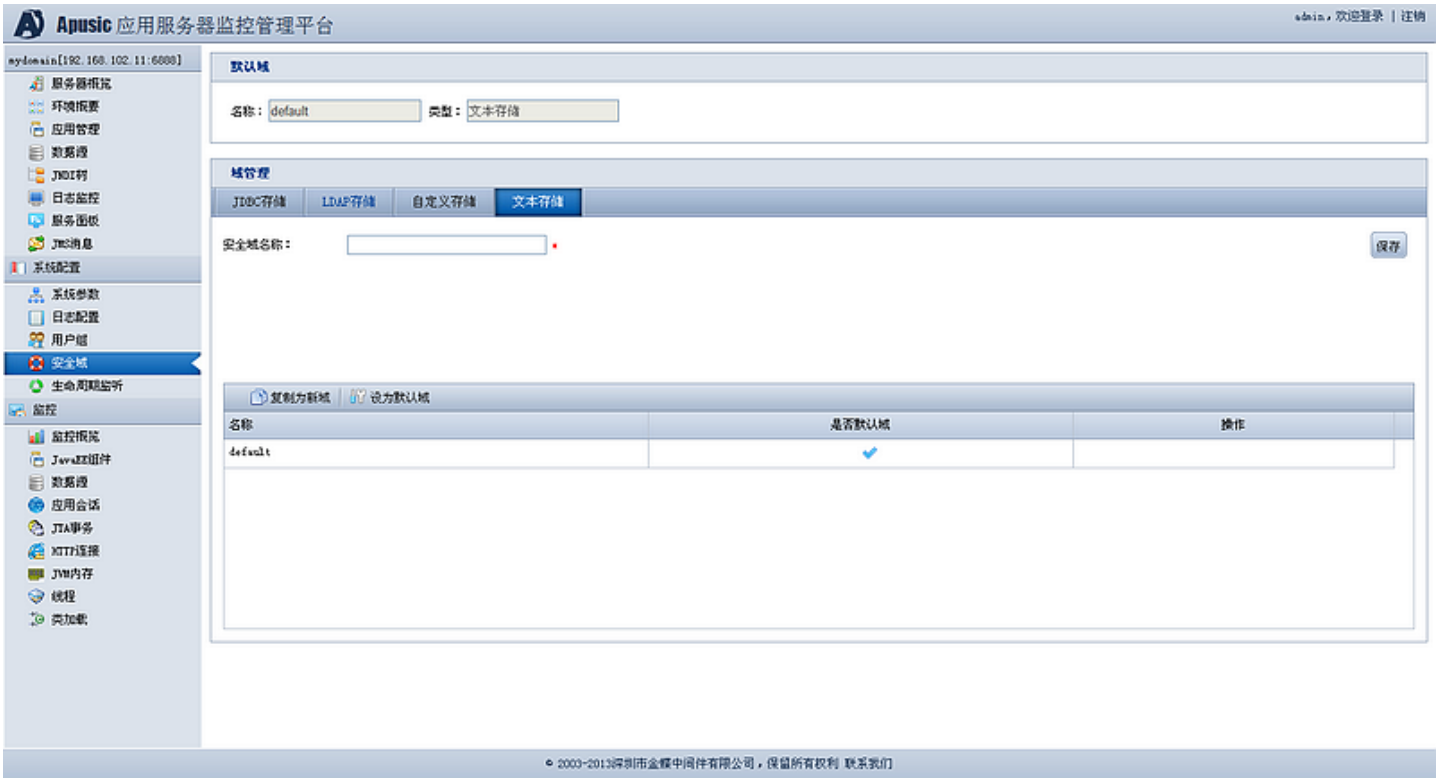
在此，我们可以对LDAP存储下的域进行新增，复制，设为默认域等操作。

3. 自定义存储配置：打开Apusic应用服务器监控管理平台，进入到“系统配置”>“安全域”>域管理>“自定义存储”操作界面下：



在此，我们可以对自定义存储下的域进行新增，复制，设为默认域等操作。

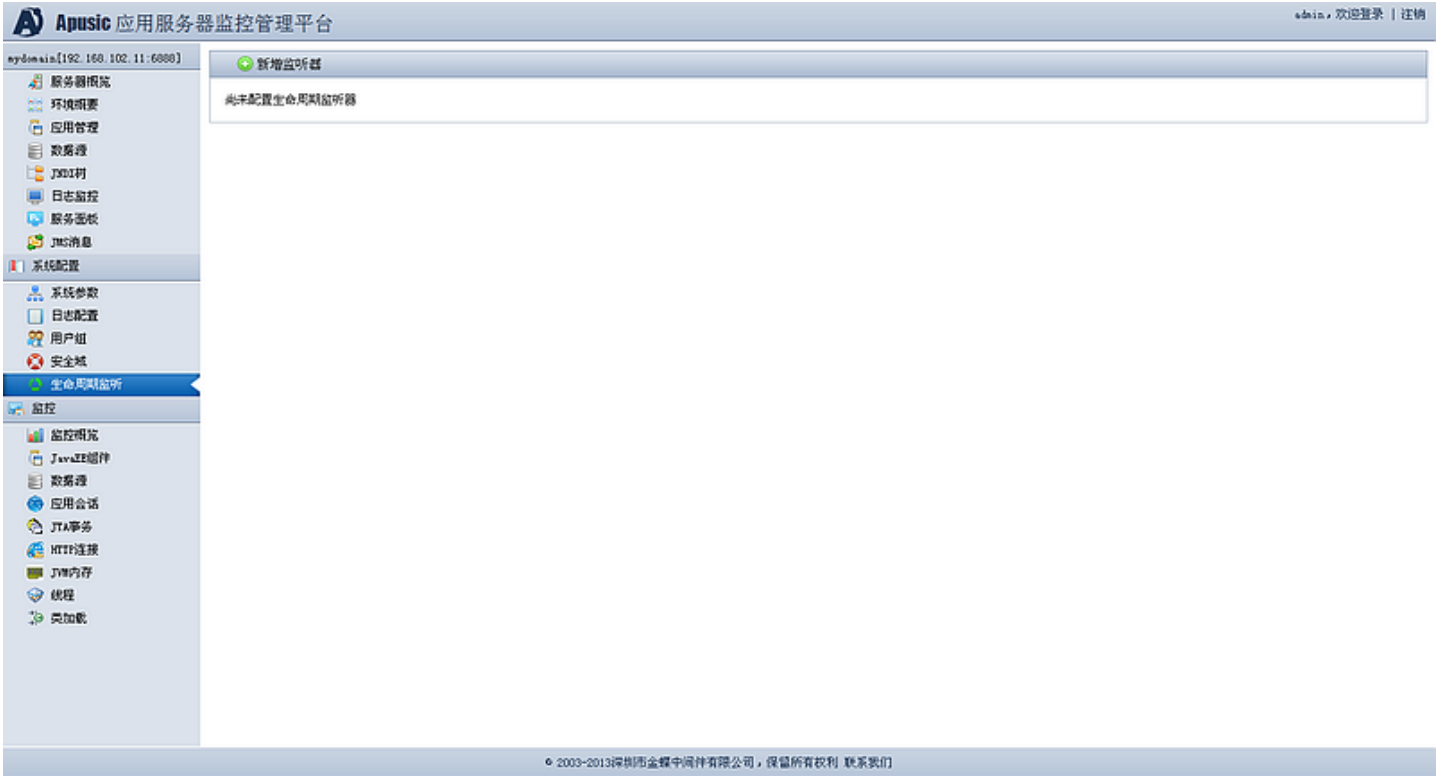
4. 文本存储配置：打开Apusic应用服务器监控管理平台，进入到“系统配置”>“安全域”>域管理>“文本存储”操作界面下：



在此，我们可以对文本存储下的域进行新增，复制，设为默认域等操作。

18.8.5 生命周期监听

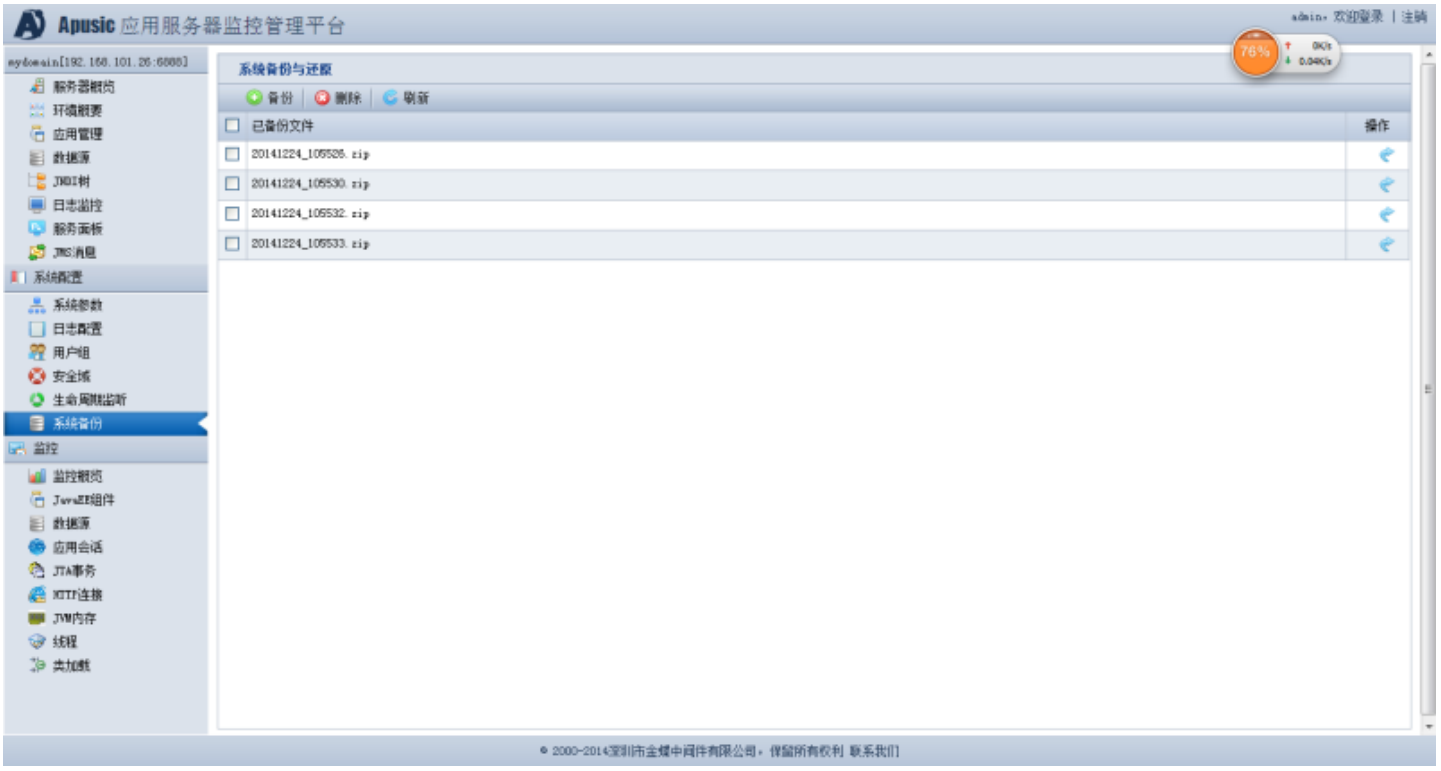
打开Apusic应用服务器监控管理平台，进入到“系统配置”>“生命周期监听”操作界面下：



在此，我们可以对生命周期类进行新增，删除，查询等操作。需要注意的是包含生命周期监听类的jar包，需要放置到系统类加载路径下，如%DOMAIN_HOME%/lib或者%APUSIC_HOME%/lib或者%APUSIC_HOME%/sp下。

18.8.6 系统备份

打开Apusic应用服务器监控管理平台，进入到"系统配置">"系统备份"操作界面下：



在此，我们可以对apusic应用服务器进行备份，备份删除，备份还原等操作。

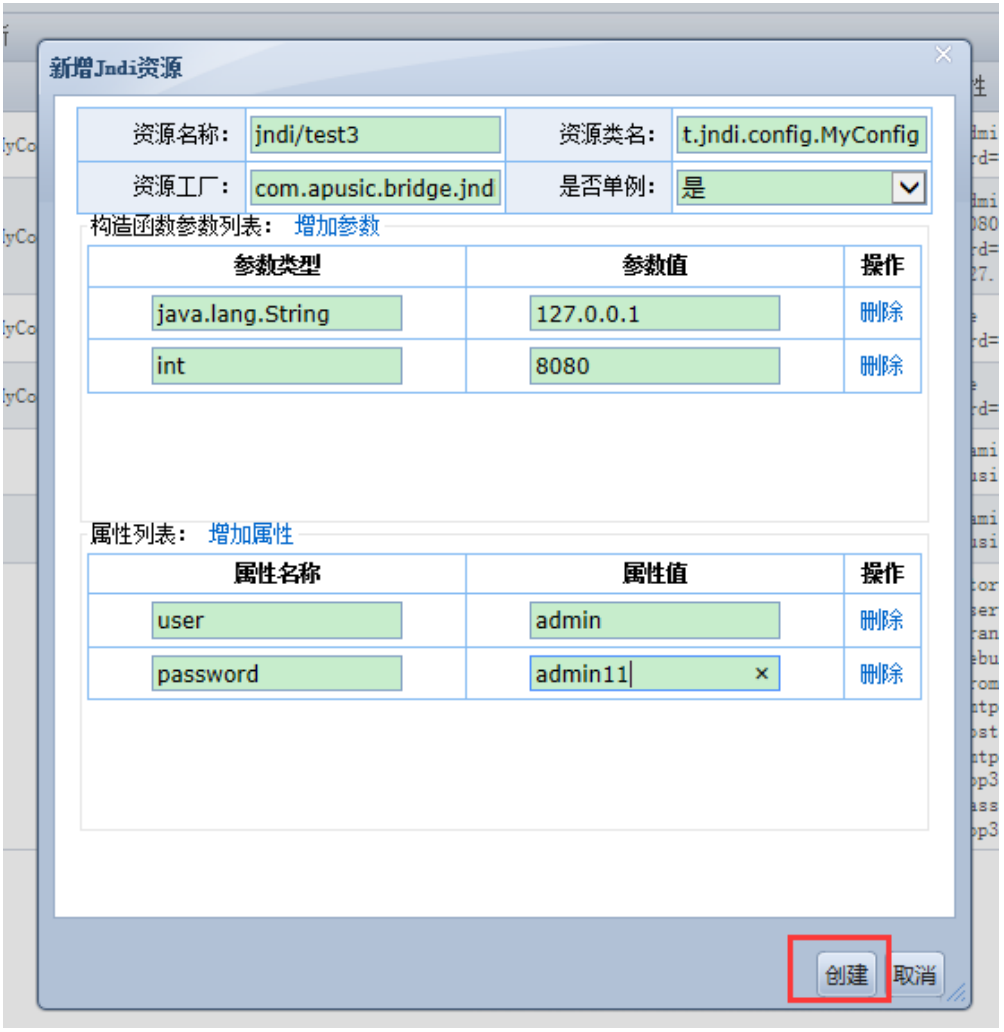
18.8.7 JNDI资源

打开Apusic应用服务器监控管理平台，进入到"系统配置">"JNDI资源"操作界面下：



1. 新增jndi资源：点击“新增JNDI资源”，弹出新增界面，输入相应的内容，点击“创建”按钮创建资源。





字段说明：

资源名称：资源的唯一标识，必须唯一。

资源类型：当前资源对应的类。

资源工厂：默认com.apusic.bridge.jndi.JNDICustomResourceFactory，不需要修改。

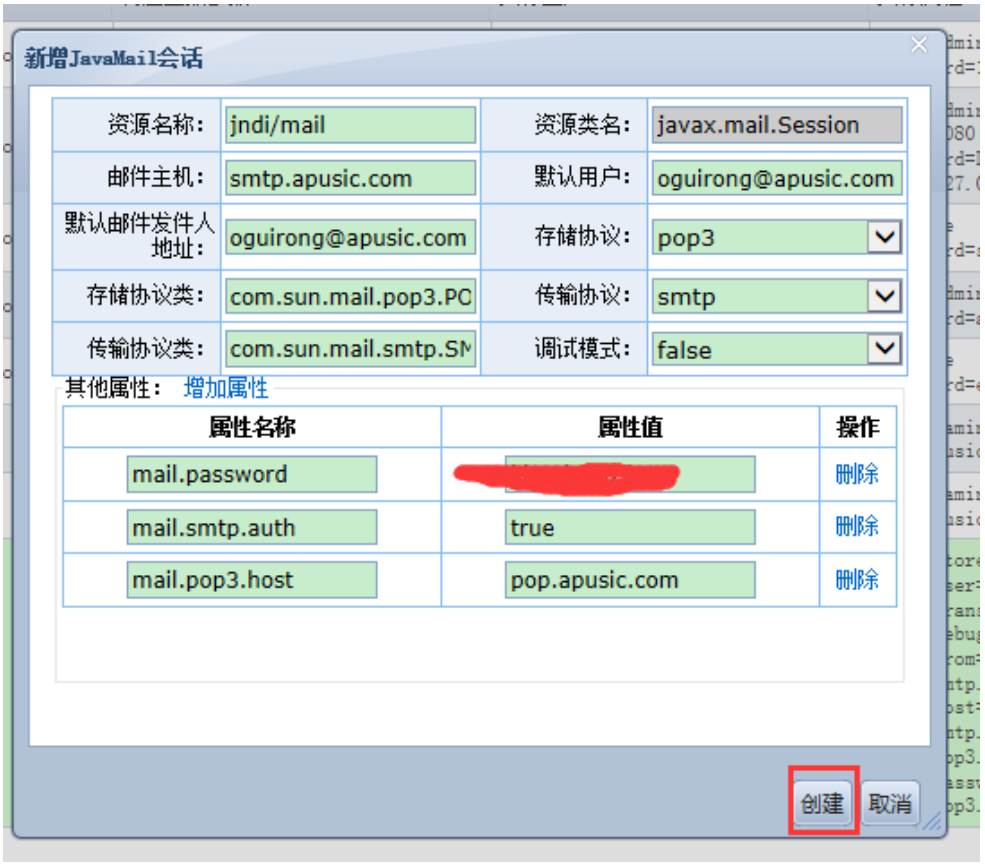
是否单例：是否是单实例。：单列（是），每次获取资源的实例（Bind Object Hash Code）是一致的，共享同一个实例，（修改后才会重新生成实例）

单列（否）：每次获取资源的实例都是不同的

构造函数参数列表：创建该对象需要的参数，注意参数的顺序。

属性列表：其他属性列表

2. 新增JavaMail会话：点击"新增JavaMail会话"按钮，弹出新增界面，输入相应内容，点击"创建按钮"创建会话资源。



字段说明：

资源名称：资源唯一标识。

资源类型：不能修改，只能为javax.mail.Session

邮件主机：邮件服务器的域名或者地址

默认用户：一般为邮箱的用户名

默认邮件发送地址：邮箱的用户名

存储协议：pop3或者imap，用于获取邮件的协议

存储协议类：根据存储协议不同而不同，默认即可

传输协议：发送邮件协议，smtp

传输协议类：不需要修改，默认即可

调试模式：选择true可在控制台看到邮件发送过程或接收邮件过程

其他属性：可添加自定义属性。（其他属性可自行添加）

mail.pop3.ssl.enable （true/false）是否启用ssl，启用和不启用mai.pop3.port的值会不一样

mail.smtp.port发送邮件端口

mail.password登录邮箱的密码

mail.smtp.password登录邮箱的密码，与上面的区别是作用范围不一样

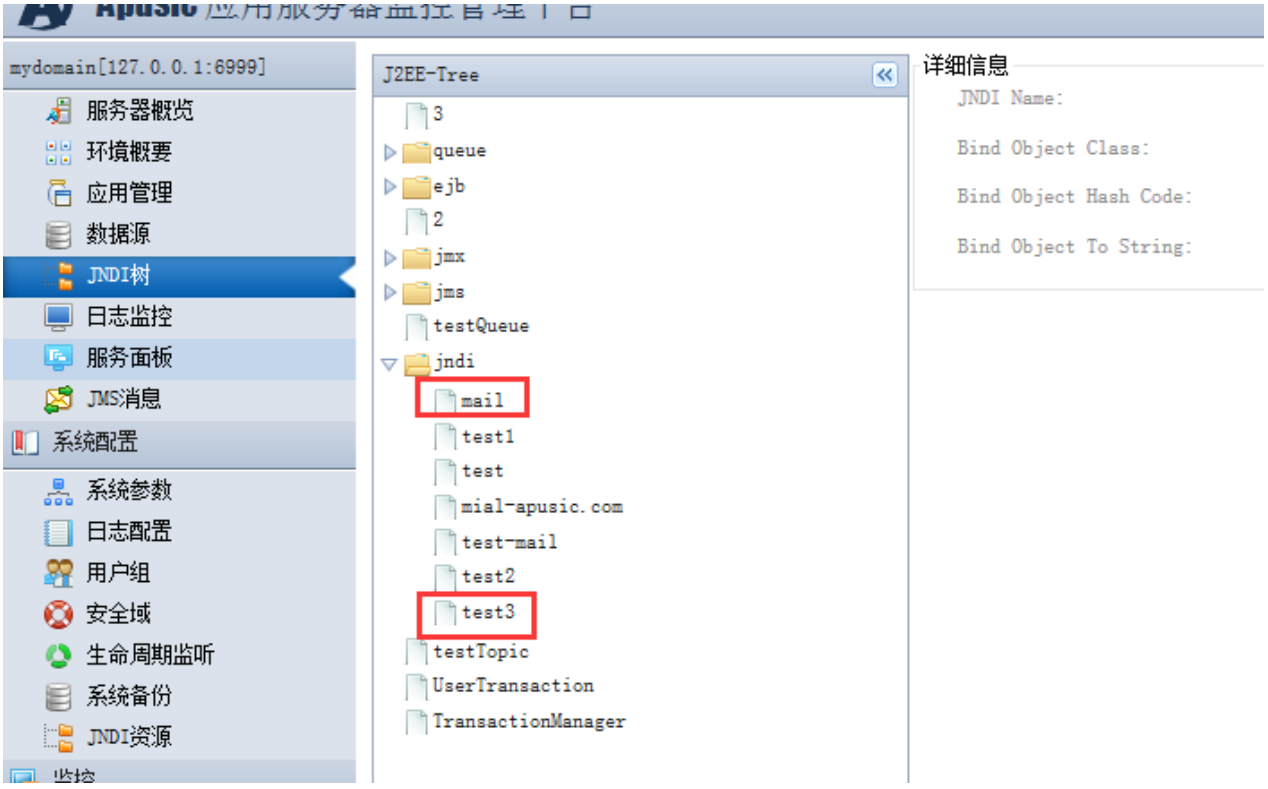
mail.smtp.auth (true/false)是否需要验证，有些邮箱不需要验证用户名和密码

mail.smtp.ssl.enable (true/false)是否启用ssl，针对smtp协议

mail.pop3.host 邮箱服务器地址(接收邮件)

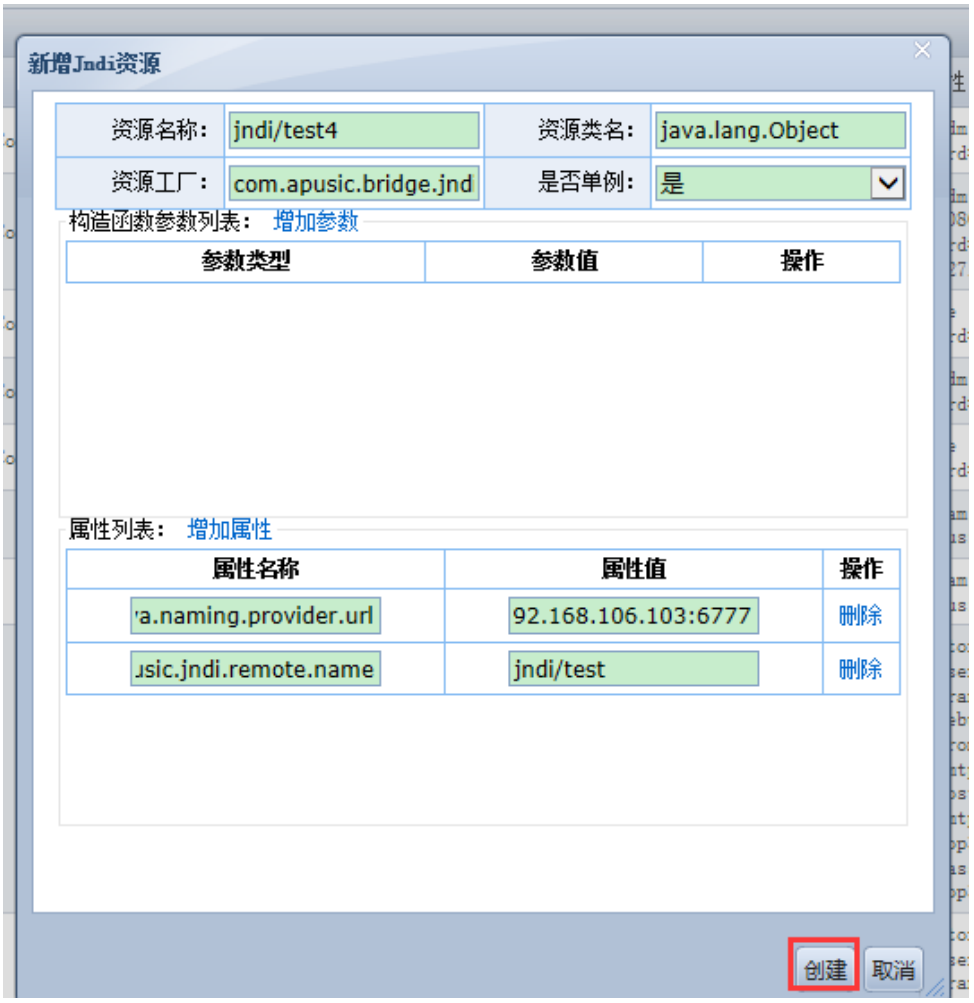
mail.pop3.port 邮箱服务器端口

3. 在JNDI树查看刚刚新增的资源实例



4. 新建JNDI外部资源：JNDI外部资源主要是通过本机的jndi名称，引用外部的jndi资源(引用的外部资源可以为普通的jndi资源，或者JavaMail会话)

点击"新增JNDI资源"按钮，输入相应的内容，点击"创建"



字段说明：

资源名称：jndi资源名称，必须唯一

资源类型：必须为java.lang.Object

资源工厂：默认即可

是否单例：默认即可，每次都会从远程机器上获取

构造函数参数列表：不需要添加

属性列表：必须添加如下属性

java.naming.provider.url 远程机器上的jndi资源地址：如iiop://127.0.0.1:6777

com.apusic.jndi.remote.name 引用远程机器上的jndi资源名称:如jndi/test

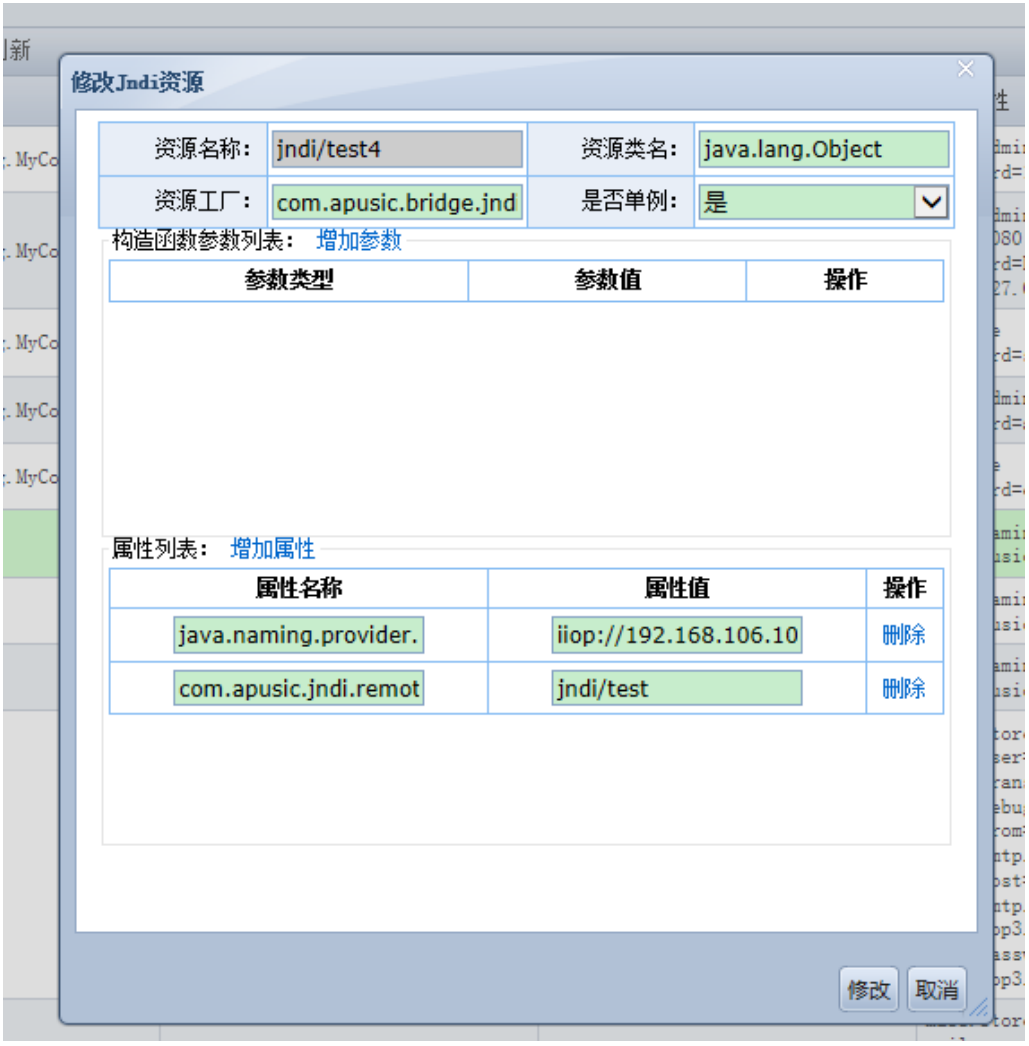
其他属性列表可选参数说明：

用于定义JNDI初始表现行为的系统属性

属性名称	含义	对应Context常量属性
java.naming.factory.initial	服务提供者提供的InitialContext实现工厂	INITIAL_CONTEXT_FACTORY
java.naming.provider.url	初始化服务提供者的URL，以配置初始上下文	PROVIDER_URL
java.naming.factory.object	用于创建对象的工厂名字，该对象用于名字到对象的映射	OBJECT_FACTORIES
java.naming.factory.state	用于创建JNDI状态对象的工厂名字	STATE_FACTORIES
java.naming.security.authentication	使用的认证类型，有none、simple和stong三种取值	SECURITY_AUTHENTICATION
java.naming.security.principal	用于识别连接服务的用户名	SECURITY_PRINCIPAL
java.naming.security.credentials	指定需要进行认证的信息	SECURITY_CREDENTIALS
java.naming.security.protocol	描述后台服务所用的协议	SECURITY_PROTOCOL
java.naming.factory.url.pkgs		URL_PKG_PREFIXES
java.naming.dns.url		DNS_URL
java.naming.authoritative		AUTHORITATIVE
java.naming.batchsize		BATCHSIZE
java.naming.referral		REFERRAL
java.naming.language		LANGUAGE
java.naming.applet		APPLET

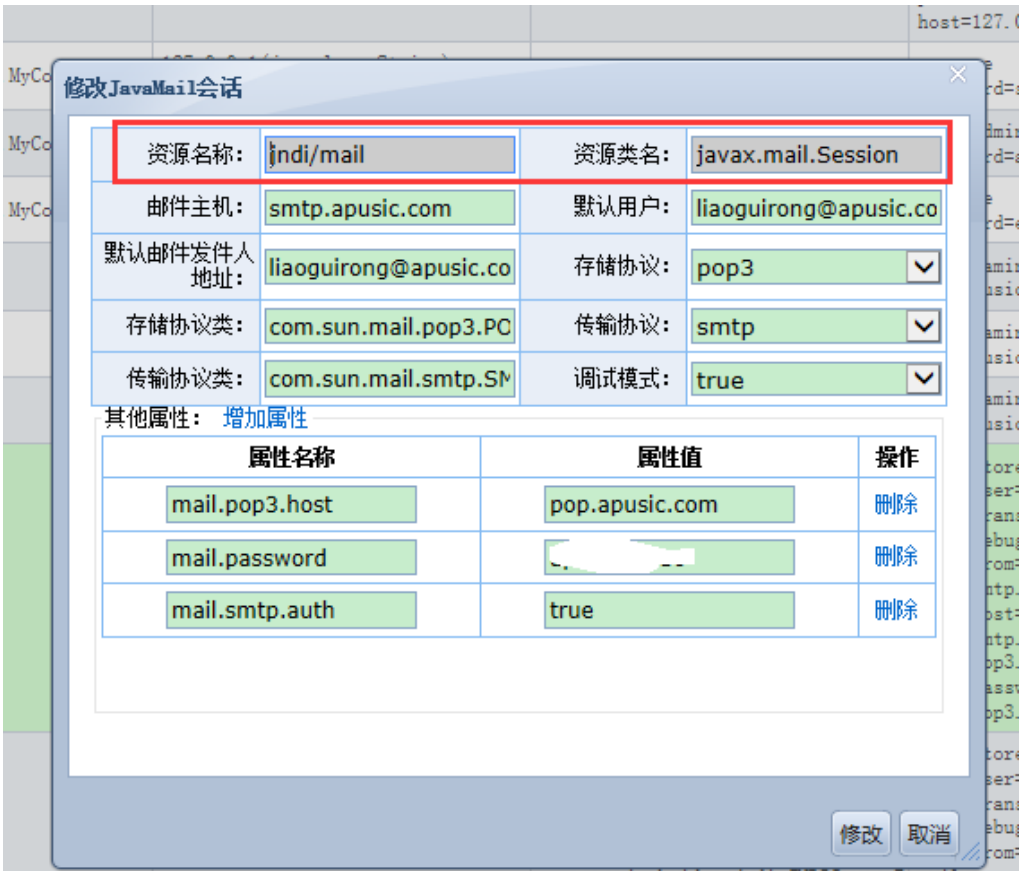
5. 修改资源：选择某个资源，点击后面的修改图标，进入修改界面

1	3	com.test.jndi.config.MyConfig	1234(int)	com.apusic.bridge.jndi.JNDICustomR	password=11	
2	jndi/test1	com.test.jndi.config.MyConfig	192.168.106.103(java.lang.String) 6999(int)	com.apusic.bridge.jndi.JNDICustomR	user=admin port=8080 password=L host=127.0.0.1	
3	2	com.test.jndi.config.MyConfig	127.0.0.1(java.lang.String) 44(int)	com.apusic.bridge.jndi.JNDICustomR	user=ee password=sss	
4	jndi/test3	com.test.jndi.config.MyConfig	127.0.0.1(java.lang.String) 8080(int)	com.apusic.bridge.jndi.JNDICustomR	user=admin password=admin11	
5	jndi/test2	com.test.jndi.config.MyConfig	132.132.132.132(java.lang.String) 1111(int)	com.apusic.bridge.jndi.JNDICustomR	user=ee password=ee	
6	jndi/test4	java.lang.Object		com.apusic.bridge.jndi.JNDICustomR	java.naming.provider.url=iiop://192.168.106. com.apusic.jndi.remote.name=jndi/test	
7	jndi/test	java.lang.Object		com.apusic.bridge.jndi.JNDICustomR	java.naming.provider.url=iiop://192.168.106. com.apusic.jndi.remote.name=jndi/test	
8	jndi/test-mail	java.lang.Object		com.apusic.bridge.jndi.JNDICustomR	java.naming.provider.url=iiop://192.168.106. com.apusic.jndi.remote.name=jndi/mail	
9	jndi/mail	javax.mail.Session		com.apusic.bridge.jndi.JNDICustomR	mail.store.protocol=pop3 mail.user=liaoguirong@apusic.com mail.transport.protocol=smtp mail.debug=true mail.from=liaoguirong@apusic.com mail.smtp.class=com.sun.mail.smtp.SMTPTransp mail.host=smtp.apusic.com mail.smtp.auth=true mail.pop3.class=com.sun.mail.pop3.POP3Store mail.password= mail.pop3.host=pop.apusic.com	
10	jndi/mial-apusic.com	javax.mail.Session		com.apusic.bridge.jndi.JNDICustomR	mail.store.protocol=pop3 mail.user=liaoguirong@apusic.com mail.transport.protocol=smtp mail.debug=false mail.from=liaoguirong@apusic.com mail.smtp.class=com.sun.mail.smtp.SMTPTransp mail.host=smtp.apusic.com mail.smtp.auth=true mail.pop3.host=pop.apusic.com mail.password= mail.pop3.class=com.sun.mail.pop3.POP3Store	



普通资源：资源名称不能修改

JavaMail资源：资源名称和资源类名不能修改。

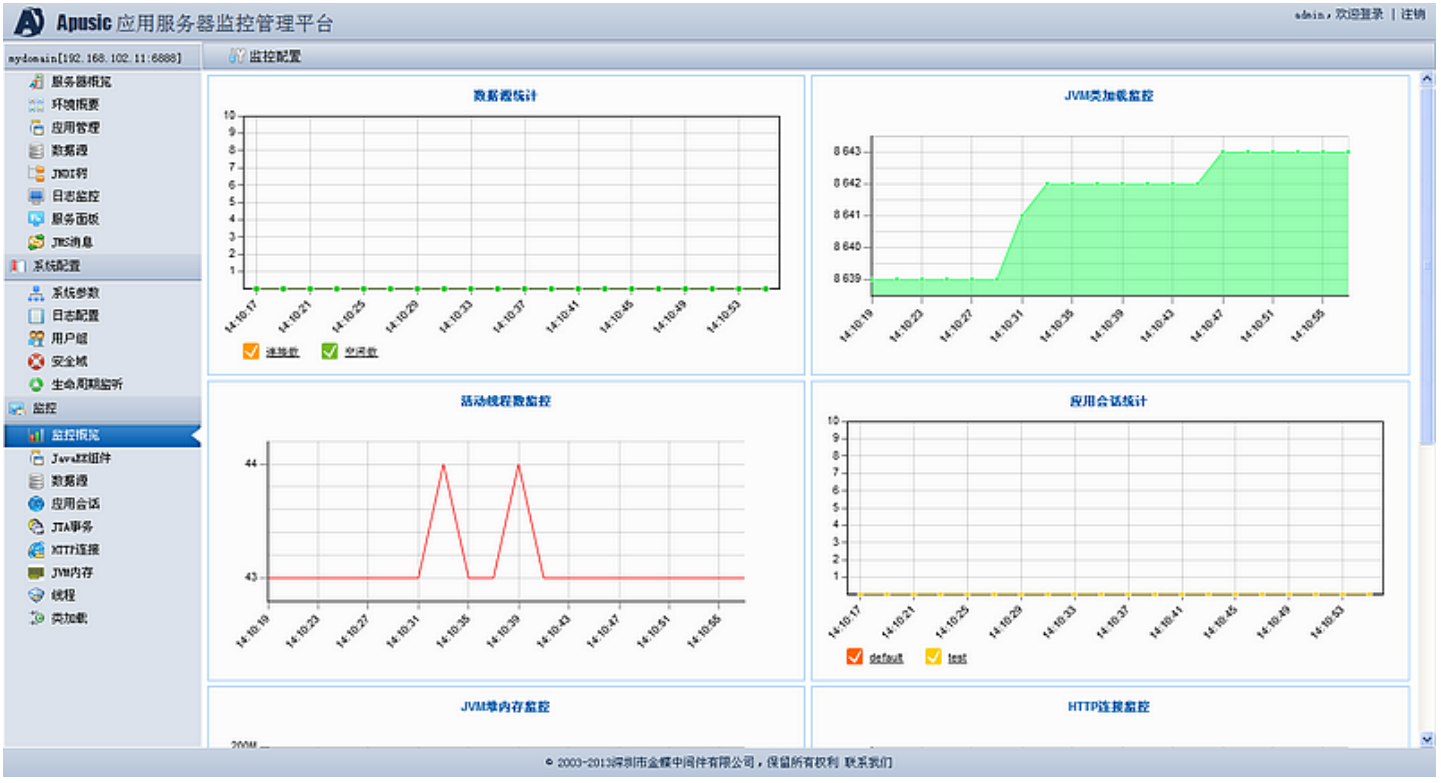


6. 删除资源：选择某个资源，点击后面的删除图标进行删除

1	3	com.test.jndi.config.MyConfig	1234(int)	com.apusic.bridge.jndi.JNDICustomR	password=11	
2	jndi/test1	com.test.jndi.config.MyConfig	192.168.106.103(java.lang.String) 6999(int)	com.apusic.bridge.jndi.JNDICustomR	user=admin port=8080 password=L host=127.0.0.1	
3	2	com.test.jndi.config.MyConfig	127.0.0.1(java.lang.String) 44(int)	com.apusic.bridge.jndi.JNDICustomR	user=ee password=sss	
4	jndi/test3	com.test.jndi.config.MyConfig	127.0.0.1(java.lang.String) 8080(int)	com.apusic.bridge.jndi.JNDICustomR	user=admin password=admin11	
5	jndi/test2	com.test.jndi.config.MyConfig	132.132.132.132(java.lang.String) 1111(int)	com.apusic.bridge.jndi.JNDICustomR	user=ee password=ee	
6	jndi/test4	java.lang.Object		com.apusic.bridge.jndi.JNDICustomR	java.naming.provider.url=iiop://192.168.106. com.apusic.jndi.remote.name=jndi/test	
7	jndi/test	java.lang.Object		com.apusic.bridge.jndi.JNDICustomR	java.naming.provider.url=iiop://192.168.106. com.apusic.jndi.remote.name=jndi/test	
8	jndi/test-mail	java.lang.Object		com.apusic.bridge.jndi.JNDICustomR	java.naming.provider.url=iiop://192.168.106. com.apusic.jndi.remote.name=jndi/mail	
9	jndi/mail	javax.mail.Session		com.apusic.bridge.jndi.JNDICustomR	mail.store.protocol=pop3 mail.user=liaoguirong@apusic.com mail.transport.protocol=smtp mail.debug=true mail.from=liaoguirong@apusic.com mail.smtp.class=com.sun.mail.smtp.SMTPTransp mail.host=smtp.apusic.com mail.smtp.auth=true mail.pop3.class=com.sun.mail.pop3.POP3Store mail.password= mail.pop3.host=pop.apusic.com	
10	jndi/mial-apusic.com	javax.mail.Session		com.apusic.bridge.jndi.JNDICustomR	mail.store.protocol=pop3 mail.user=liaoguirong@apusic.com mail.transport.protocol=smtp mail.debug=false mail.from=liaoguirong@apusic.com mail.smtp.class=com.sun.mail.smtp.SMTPTransp mail.host=smtp.apusic.com mail.smtp.auth=true mail.pop3.host=pop.apusic.com	

18.9 监控

打开Apusic应用服务器监控管理平台，进入到“监控”->”监控概览”操作界面下：



在此我们可以对配置的监控项进行总体的查看与监控。

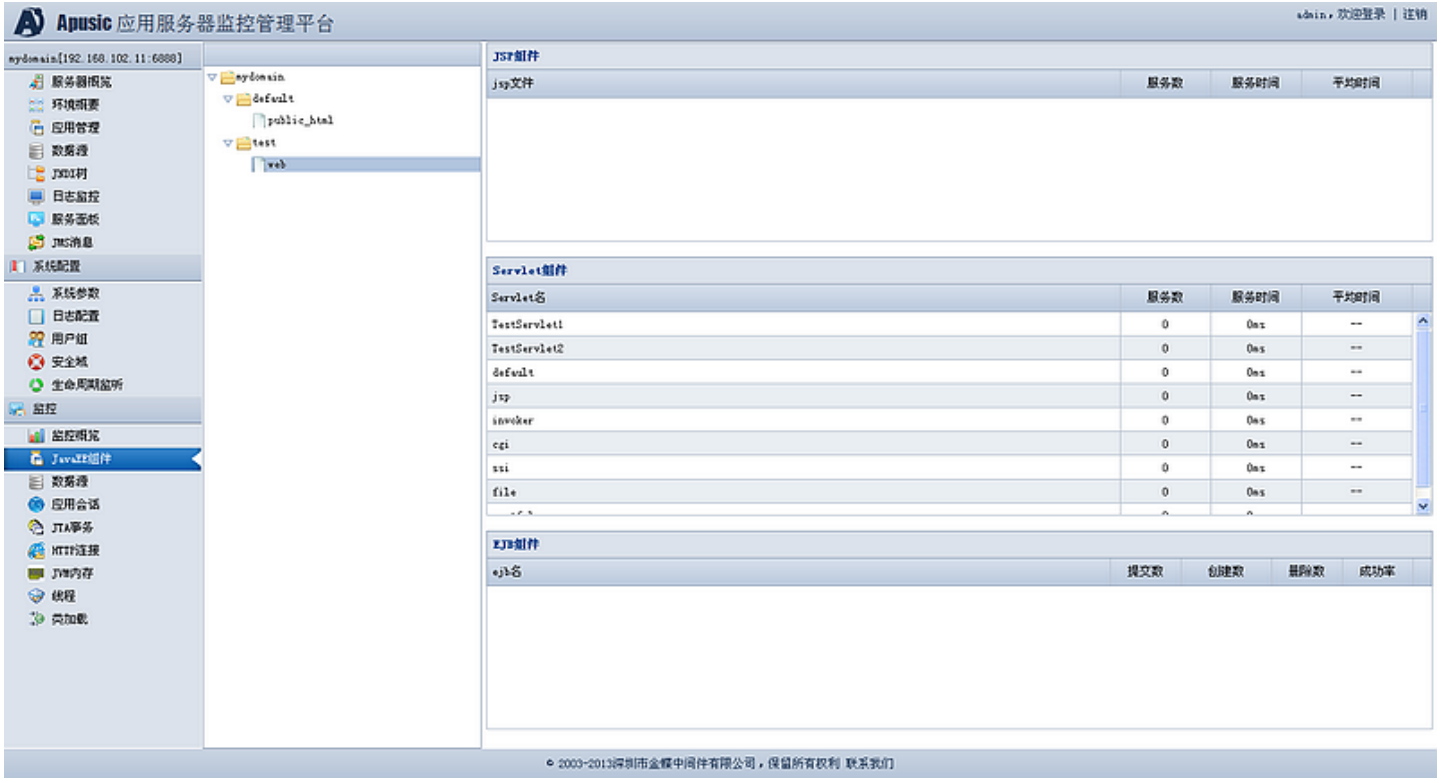
点击“监控配置”按钮，进入到监控挂件操作界面：



在此我们可以对挂件的每项监控或者全部监控挂件进行停止和启动，添加至概要以及搜索等操作。

18.9.1 javaEE组件监控

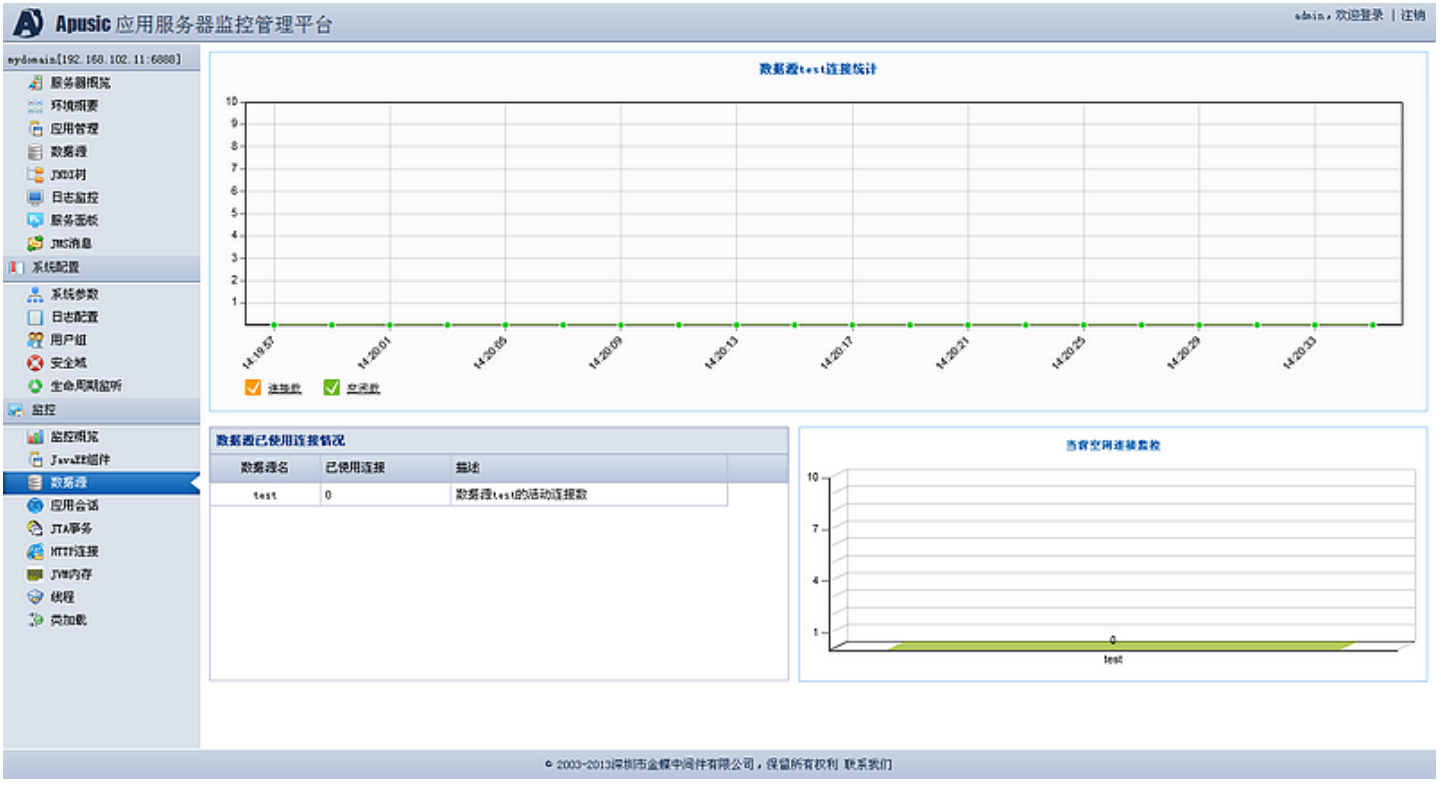
打开Apusic应用服务器监控管理平台，进入到"监控">"javaEE组件"操作界面下：



在此可以查看每一个应用下javaEE组件的服务数，服务时间，平均时间等。

18.9.2 数据源监控

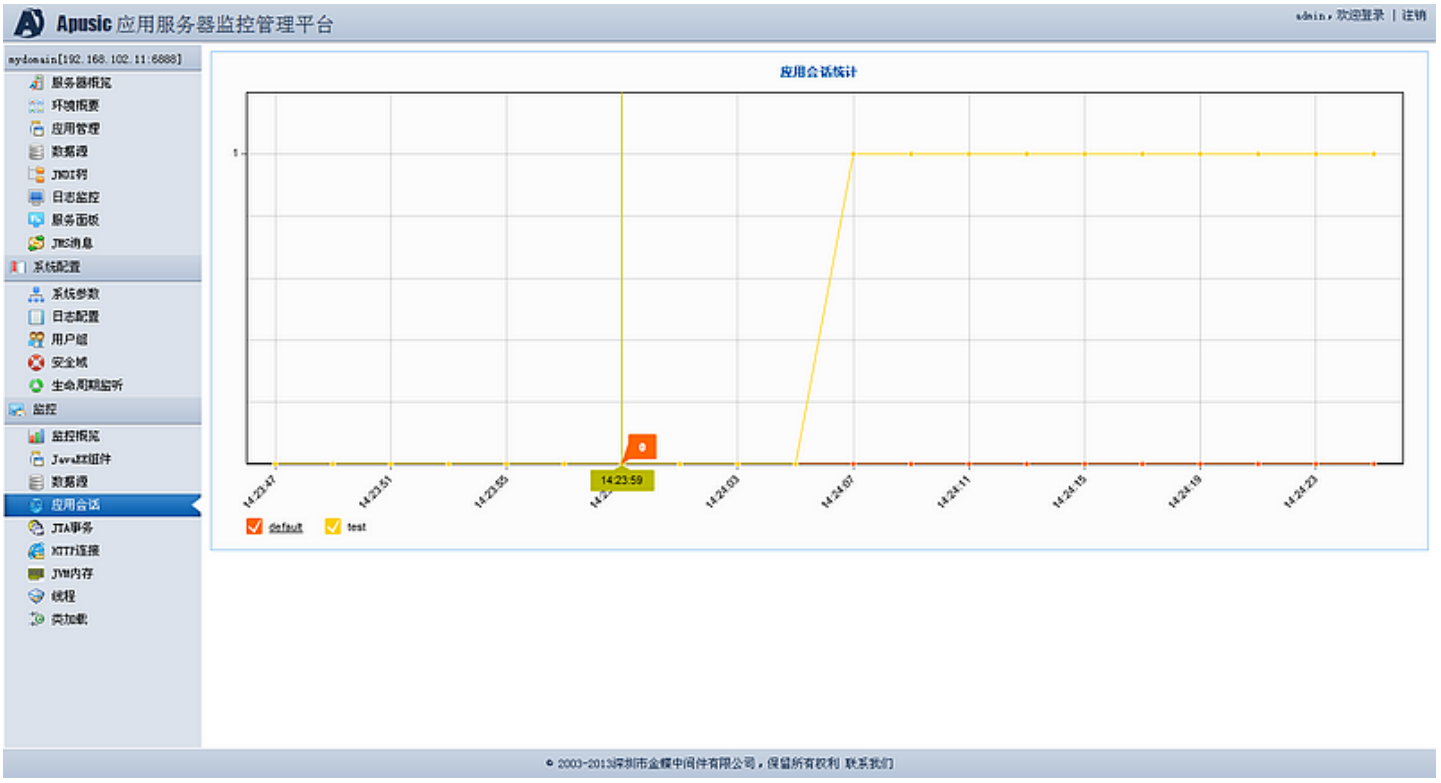
打开Apusic应用服务器监控管理平台，进入到“监控”>“数据源”操作界面下：



在此我们可以查看数据源总的链接数和空闲数以及每个数据源的链接数和空闲数和活动连接数。

18.9.3 应用会话监控

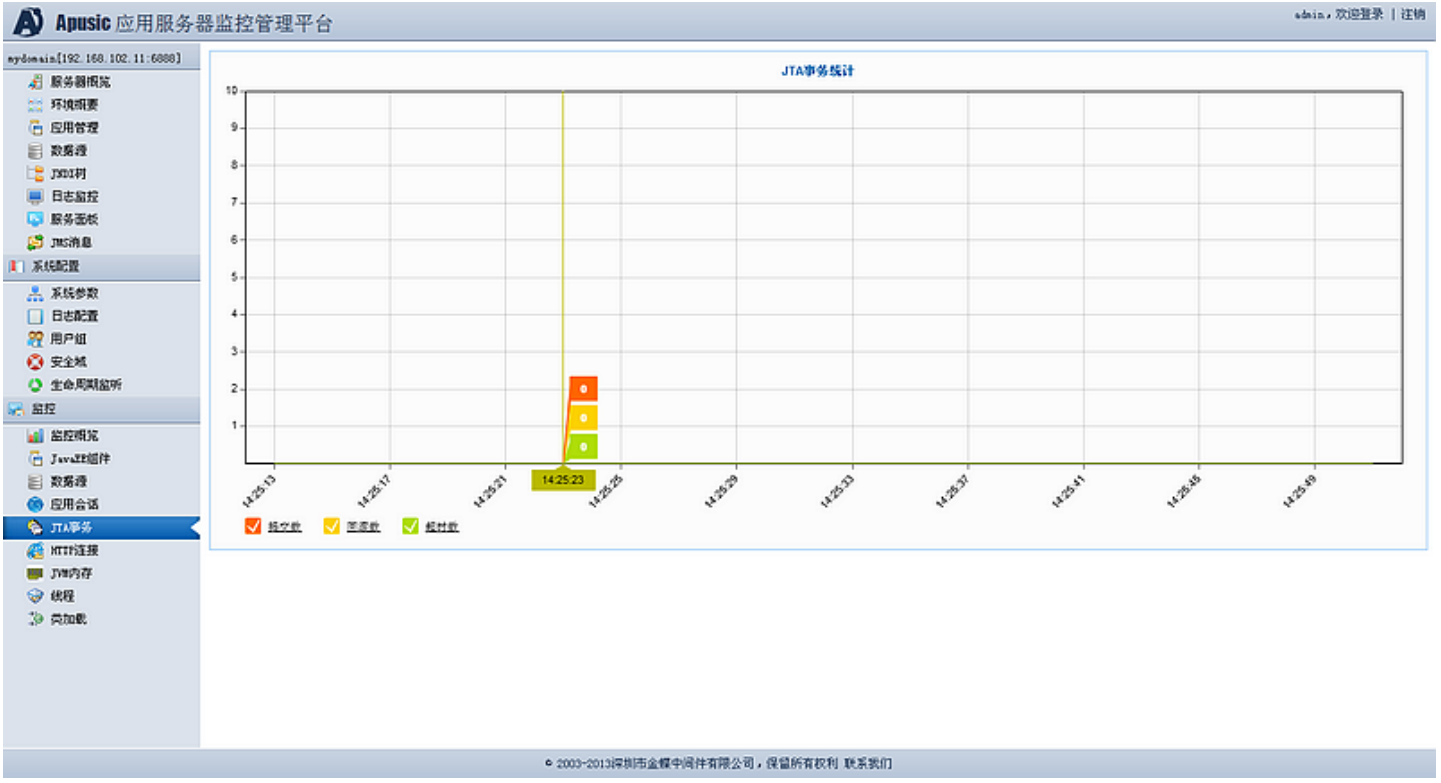
打开Apusic应用服务器监控管理平台，进入到“监控”>“应用会话”操作界面下：



在此我们可以从图表和比例图以及文字描述中监控到每个应用的会话连接数的变化和比例。

18.9.4 JTA事务监控

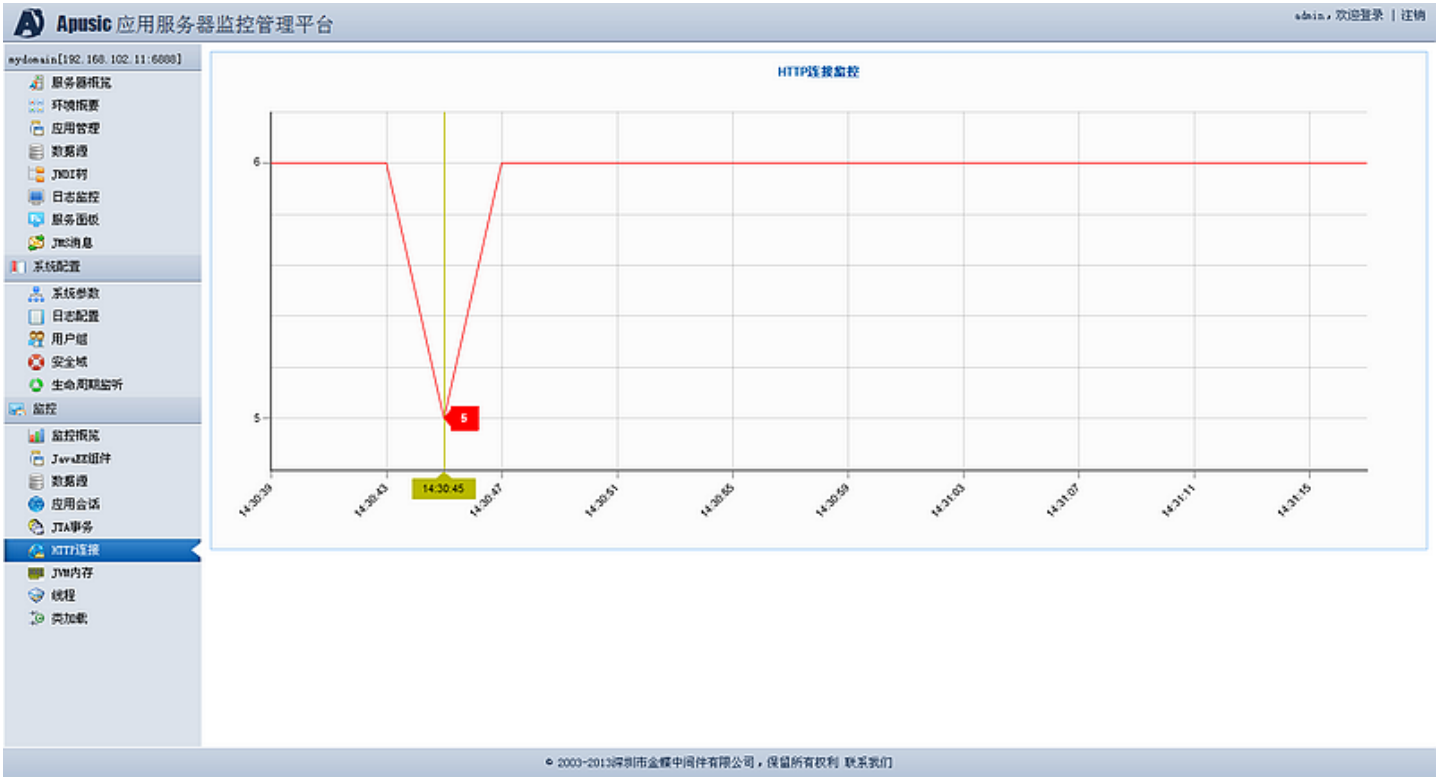
打开Apusic应用服务器监控管理平台，进入到"监控">"JTA事务"操作界面下：



在此我们可以监控到JTA事务的提交数，回滚数以及超时数随时间的变化等。

18.9.5 HTTP链接监控

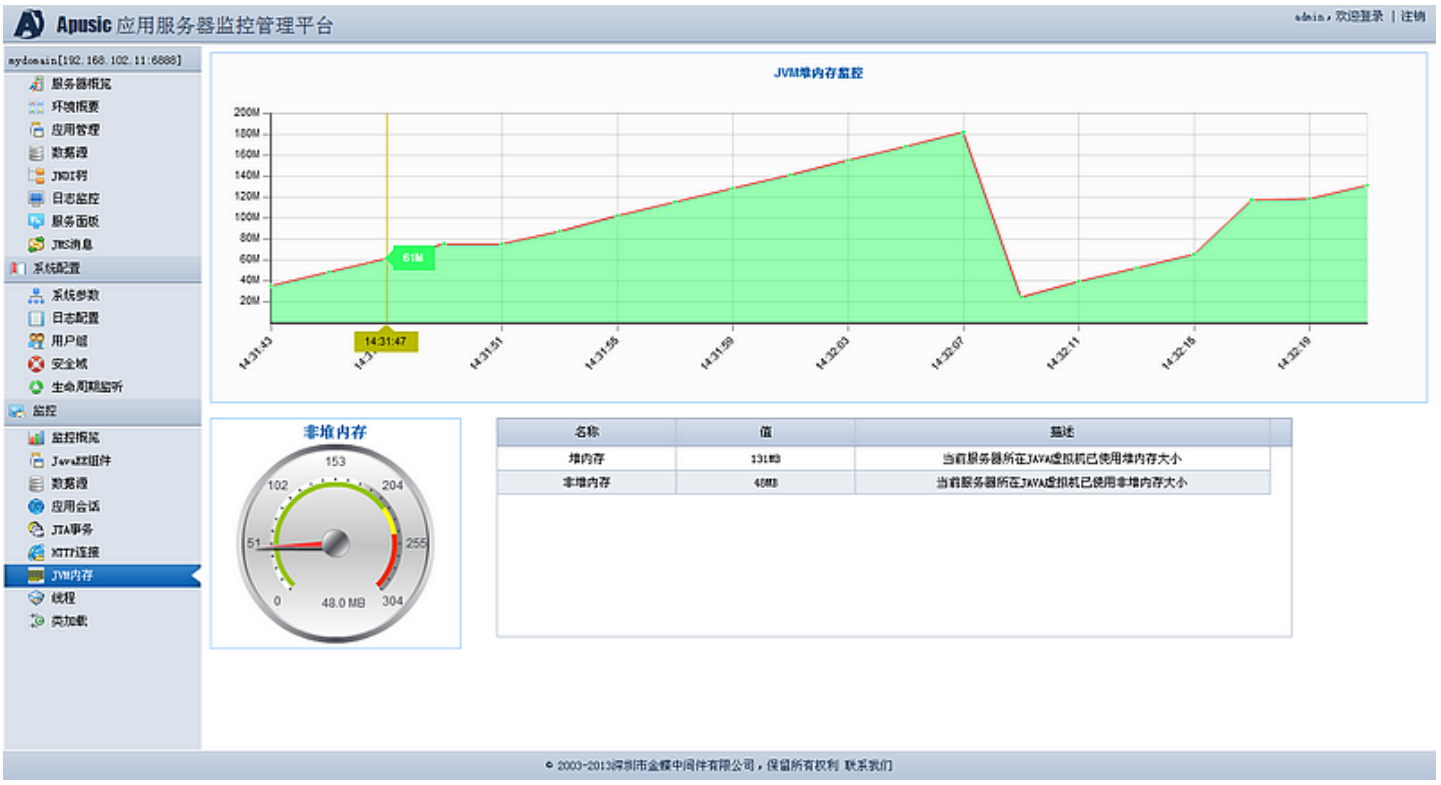
打开Apusic应用服务器监控管理平台，进入到“监控”>“HTTP链接监控”操作界面下：



在此我们可以监控HTTP链接数以及随时间变化的曲线图。

18.9.6 JVM内存监控

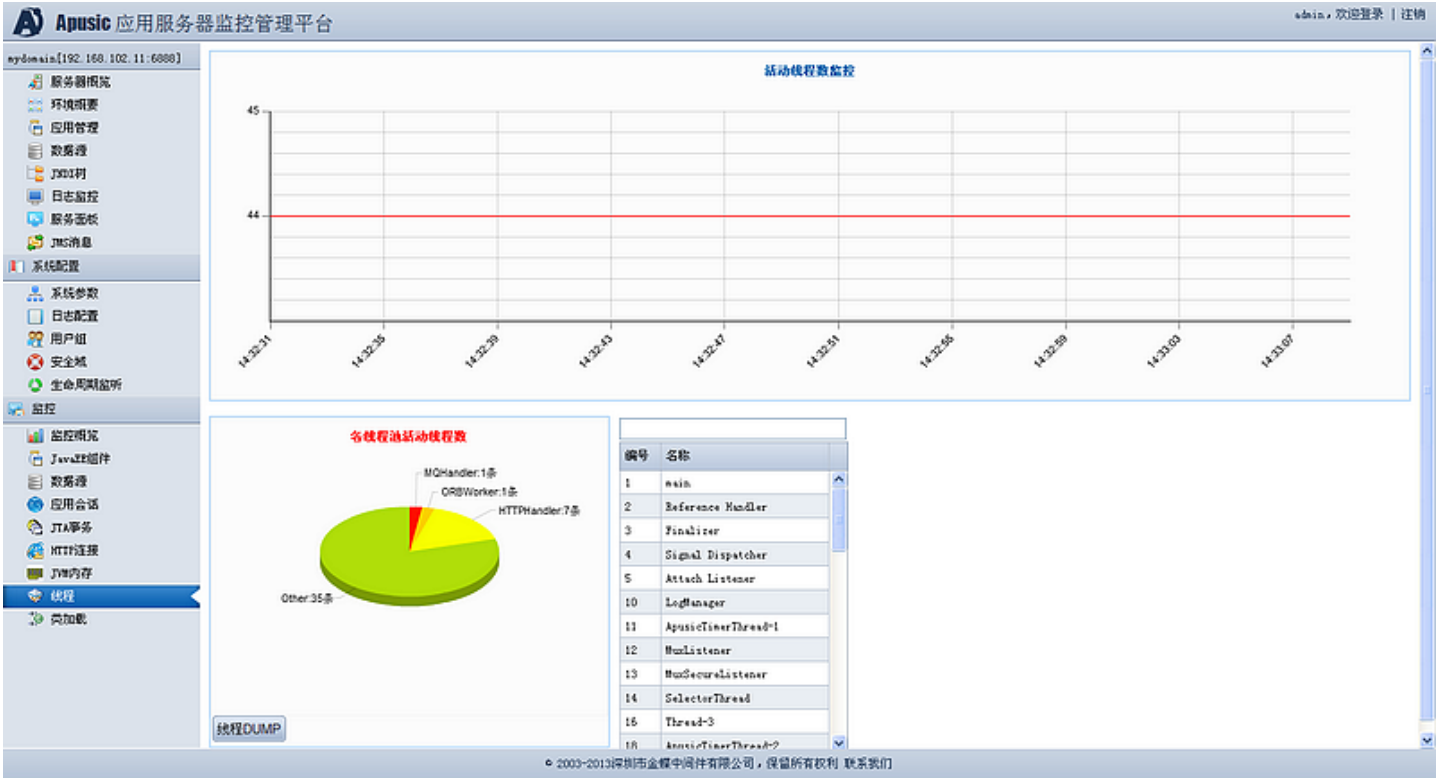
打开Apusic应用服务器监控管理平台，进入到“监控”>“JVM内存”操作界面下：



在此我们可以查看当前服务器所在java虚拟机已使用的堆内存的使用量及随时间变化的曲线图，还可以查看到java虚拟机内存使用的情况，包含堆和非堆。

18.9.7 线程监控

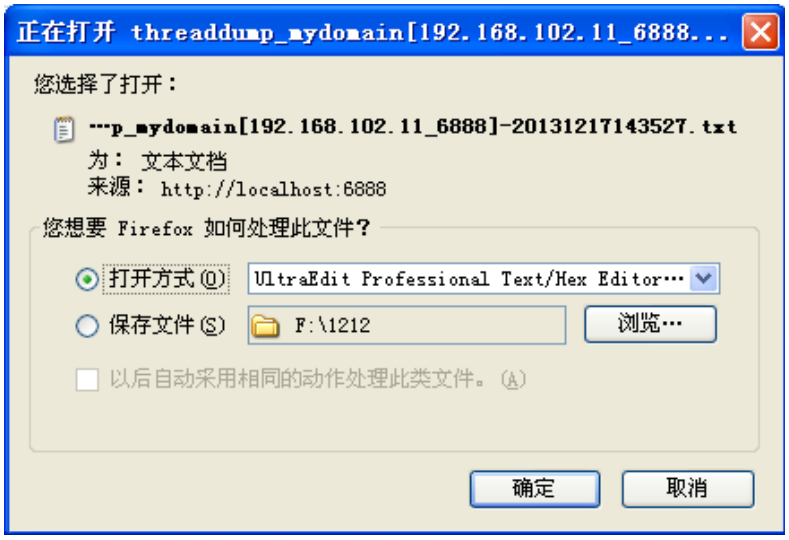
打开Apusic应用服务器监控管理平台，进入到“监控”>“线程”操作界面下：



在此我们可以监控当前服务器所在java虚拟机的活动线程数及其随时间的变化活动曲线。另外以饼状图的形式显示了线程池中各线程的比例。

18.9.8 线程DUMP

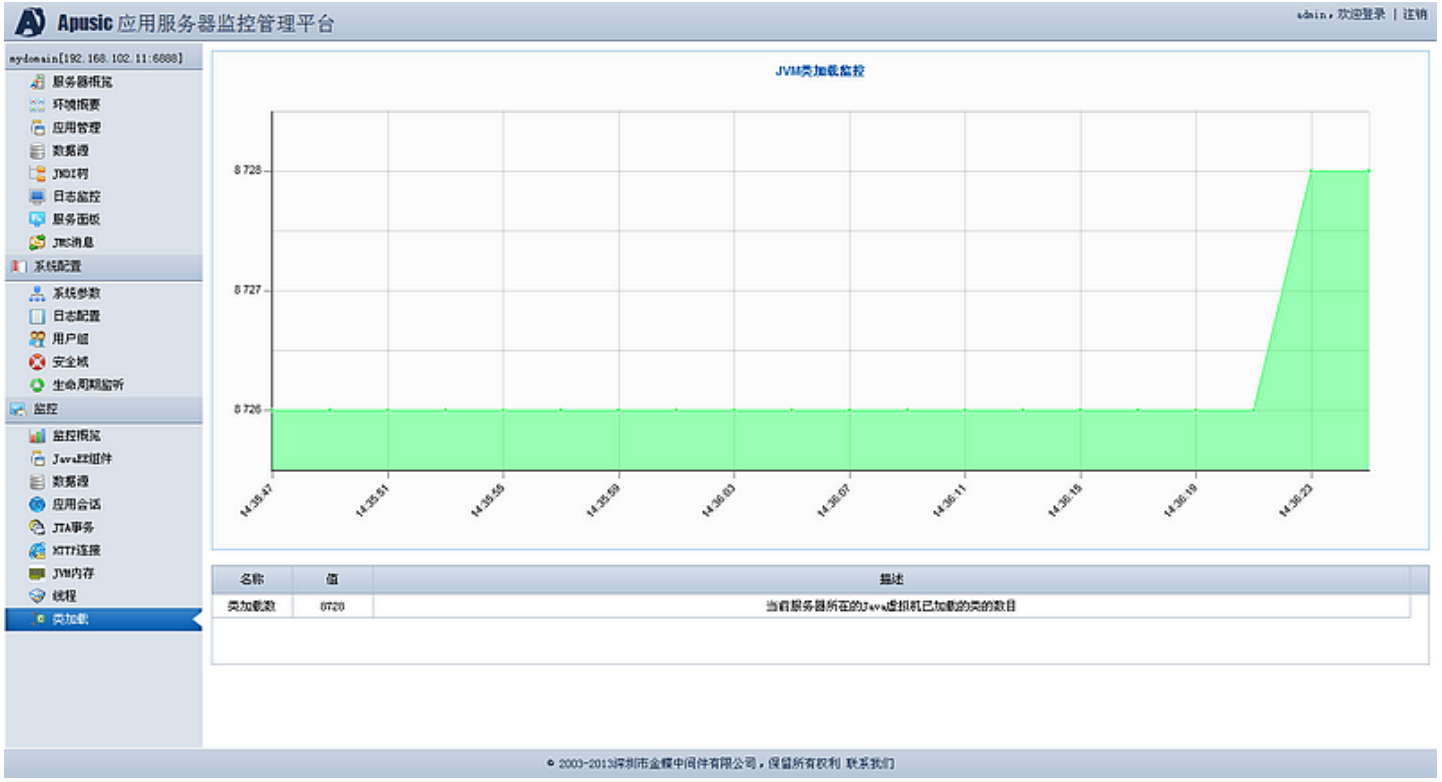
打开Apusic应用服务器监控管理平台，进入到"监控">"线程"操作界面下，点击"线程DUMP"按钮：



打开文件可以查看打印出所有线程的状态和调用堆栈的信息。

18.9.9 类加载监控

打开Apusic应用服务器监控管理平台，进入到"监控">"类加载"操作界面下：



在此界面下可以查看当前服务器所在java虚拟机已加载的类的数目以及其随时间的变化图。

18.10 三员分立

金蝶Apusic应用服务器支持三员分立功能，提供安全保密管理员、安全审计员、系统管理员三个角色。其中安全保密管理员主要管理角色管理、用户管理、密码策略等信息；安全审计员主要管理日志、操作审计信息；系统管理员主要管理应用部署、资源管理、配置管理等功能。

金蝶Apusic应用服务器默认带有管理员用户角色，安全保密管理员（security）默认用户为security；安全审计员（audit）默认用户为auditor；角色为系统管理员（system）默认用户为sysadmin。应用服务器中至少需要保留三员角色，每个角色至少需要有一个可用的用户。默认管理员用户的默认密码为 `1qazXSW@` 。初次登录需要更改密码。

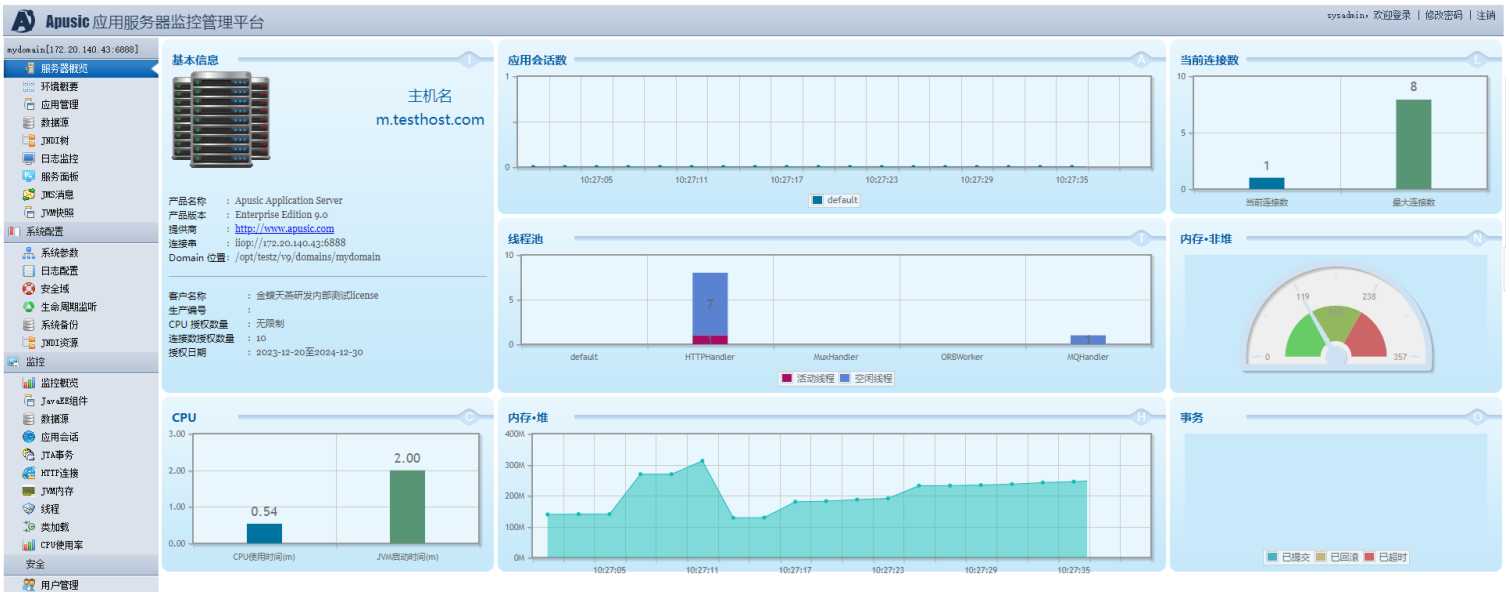
角色为安全保密管理员（security）的用户访问,如使用默认安全保密管理员用户 security登录，进入到系统界面如下:



角色为安全审计员（audit）的用户访问，如使用默认安全审计员用户auditor登录，进入到系统界面如下:



角色为系统管理员（system）的用户访问，如使用默认系统管理员用户sysadmin登录，进入到系统界面如下:



18.11 角色管理

安全管理员登录管控平台，可管理角色信息。

创建并管理角色，金蝶Apusic应用服务器自带角色包括 system,security,audit。

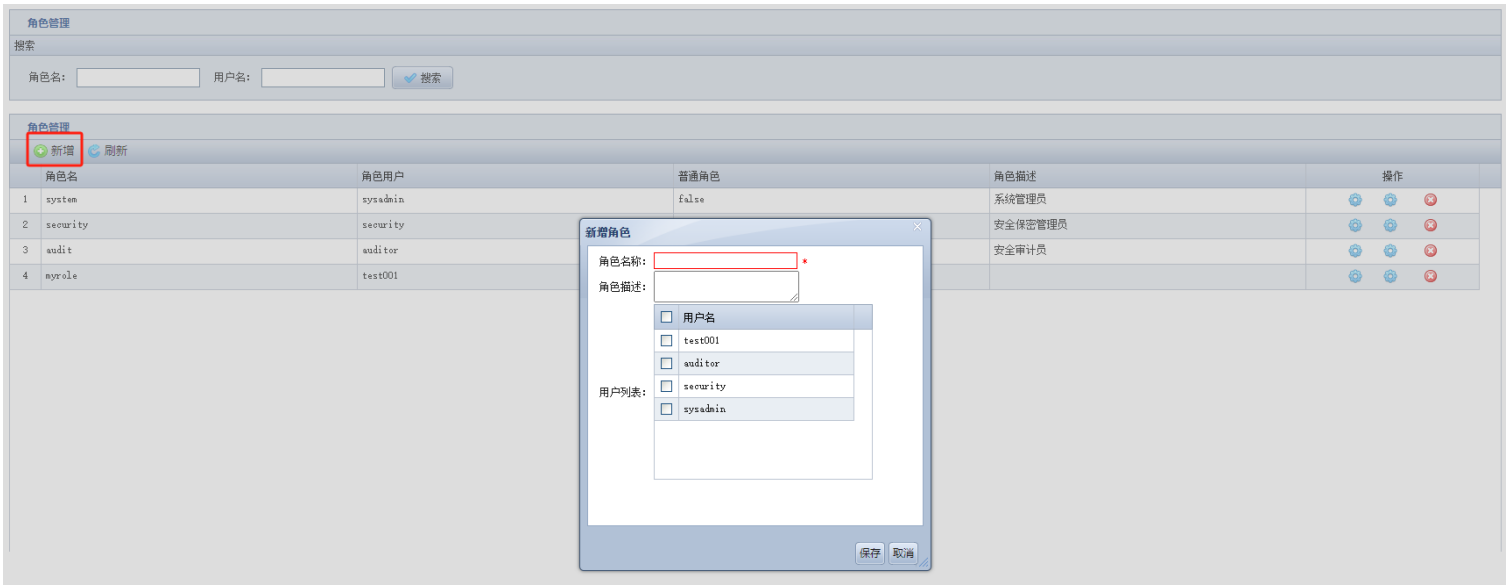
- **角色名**: 角色的名称。
- **角色用户**: 当前角色下的用户，角色 system,security,audit必须有一个以上的用户，且用户只能为一个角色，【用户管理】中设置"系统角色"将实时更新。
- **普通角色**: 当前角色是否为普通角色； 非普通角色非必要，不要删除。



18.11.1 新建角色

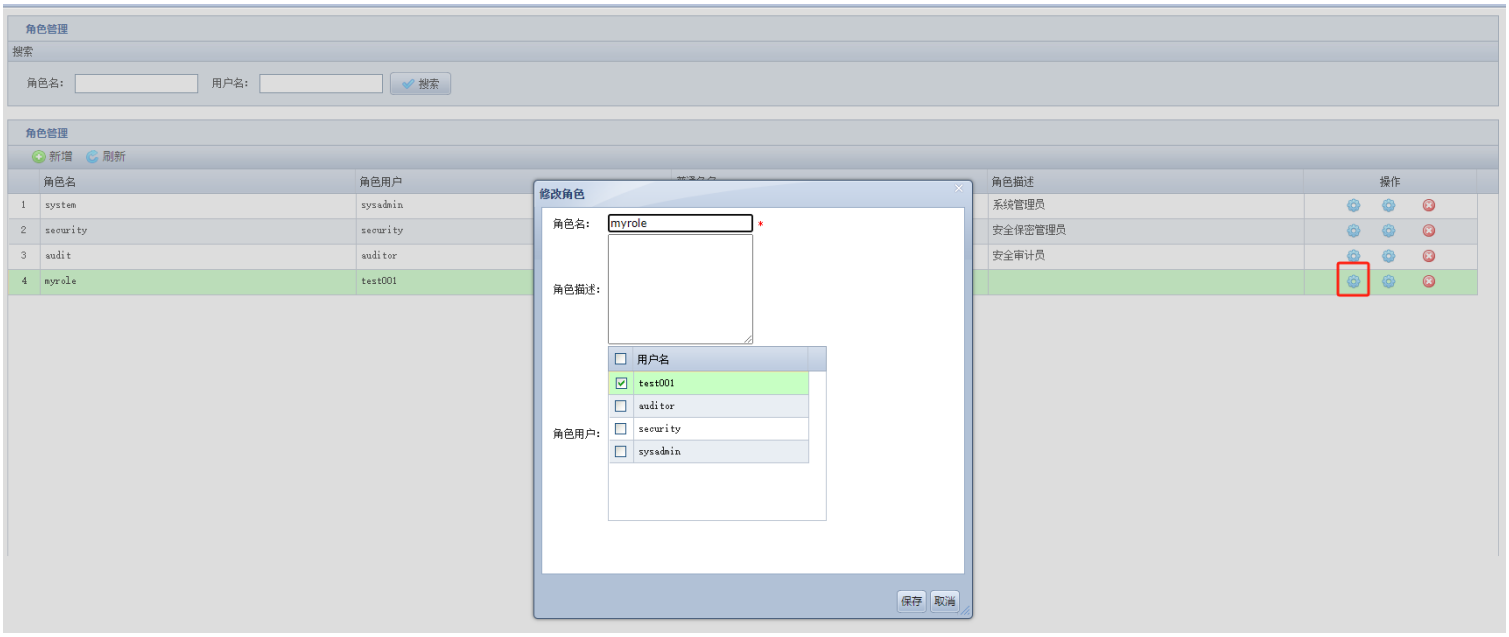
点击"新建"，进入新建系统角色页面。

- **角色名**: 角色的名称，只能包含字母数字,下划线,短横线或点字符，需要唯一。
- **角色描述**: 对角色进行说明。
- **用户列表**: 设置当前角色的用户，可以选择多选。



18.11.2 编辑角色

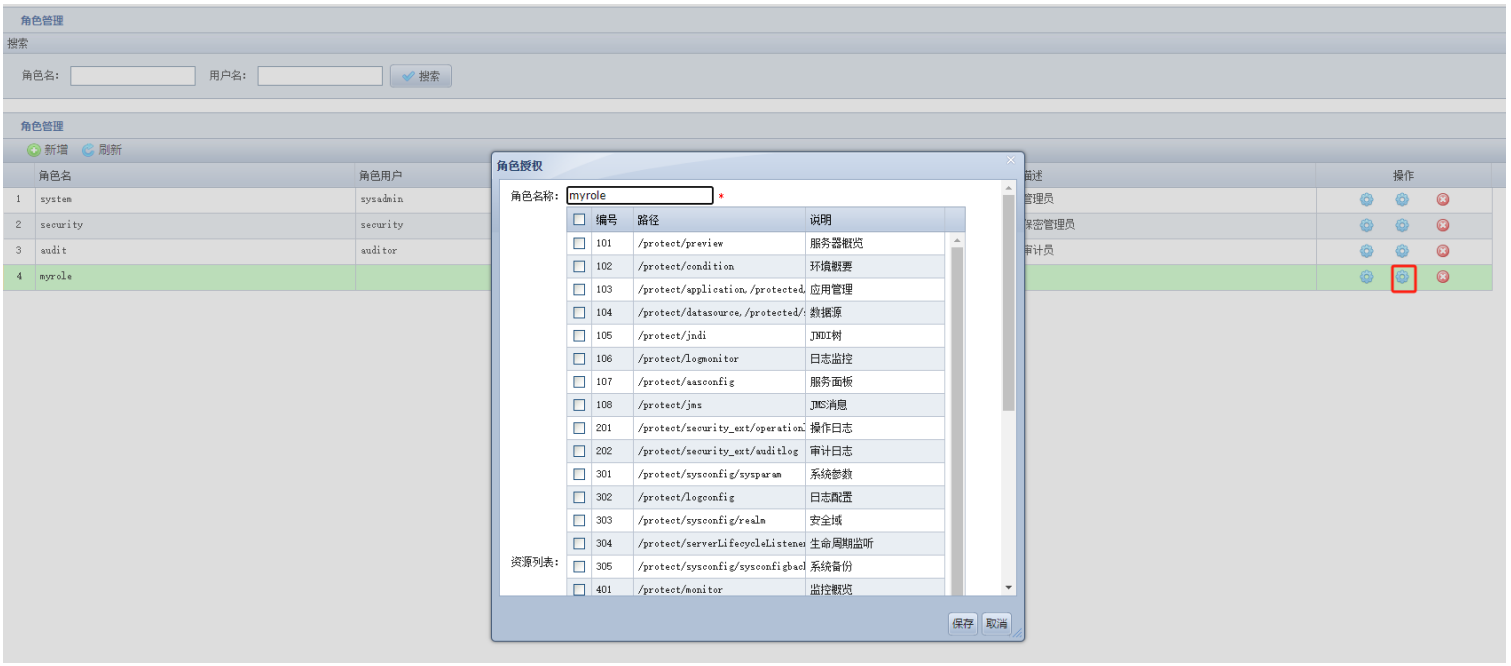
点击角色列表中的"操作"-"编辑角色"，将进入修改角色页面，可对角色信息进行编辑。修改后，用户需要重新登录管控平台才能生效。



18.11.3 角色授权

点击角色列表中的"操作"-"角色授权"，将进入角色授权页面，可为角色添加资源，建议选择"编号"101。修改后，用户需要重新登录管控平台才能生效。

需注意，非必要，不要修改默认角色的资源列表。



18.11.4 删除角色

点击角色列表中的"操作"-"删除"，确认删除后将会删除该角色信息。需注意，AAS 自带角色包括 system,security,audit，非必要不能删除。



18.12 用户管理

角色为安全保密员（security）的用户登录管控平台，可以对用户进行新增、修改删除等管理操作，还可以对用户进行安全属性的设置，如对用户进行锁定、禁用、IP 访问限制以及访问时间段限制等操作。

使用角色为安全保密管理员的用户登录管控，切换到用户管理界面。

在用户管理界面可以对用户进行添加、编辑和删除操作。

- **用户名**: 应用服务器的用户名称，需要唯一，需要使用用户名登录管控平台。
- **姓名**: 该用户的名字。

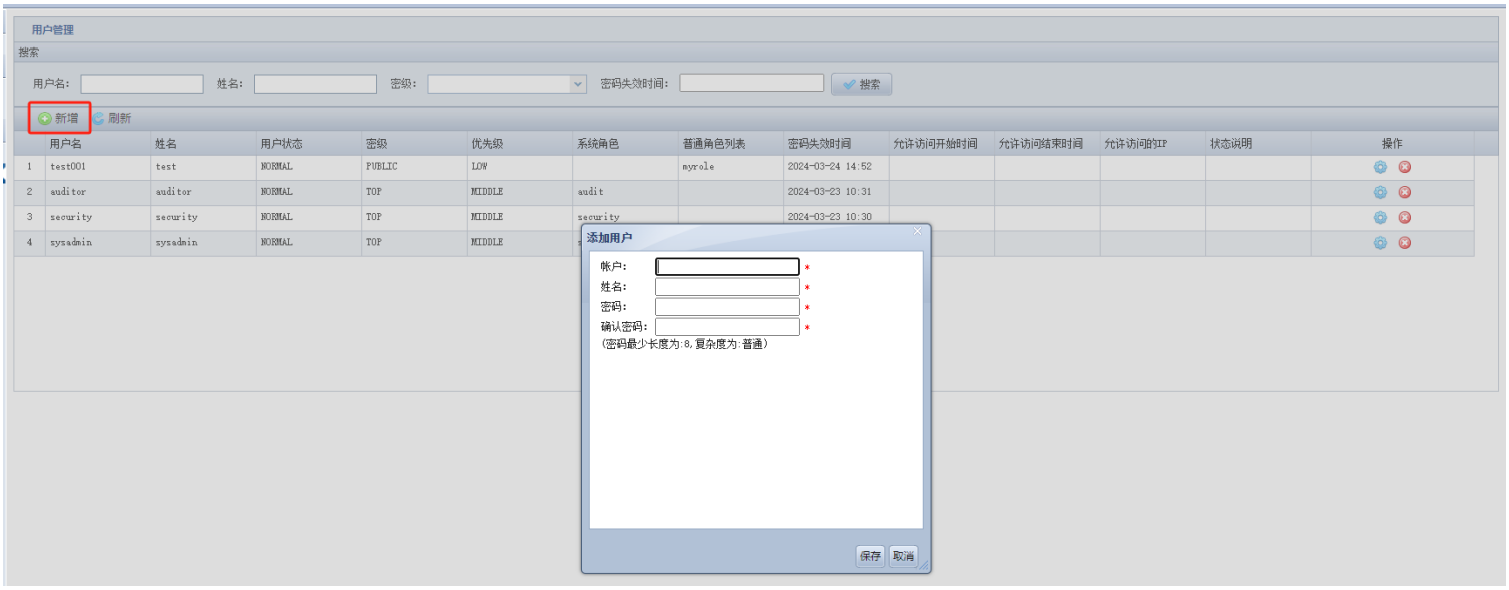
- **用户状态**: 显示用户的状态。NORMAL为正常状态；LOCK为锁定，锁定时间默认为15分钟；DISABLED为禁用，此时无法访问。
- **系统角色**: 该用户为系统角色，每个默认系统角色最少需要一个用户。
- **普通角色列表**: 该用户为普通角色。
- **密码失效时间**: 显示密码的失效时间，默认为初始化用户后 30 天，在【系统管理】中修改"密码失效时间"，该处的时间将会同步修改。
- **允许访问开始时间（HH: MM）**: 用户允许访问管控台的开始时间，格式为 HH: MM。
- **允许访问结束时间（HH: MM）**: 用户允许访问管控台的结束时间，格式为 HH: MM。
- **允许访问 IP**: 用户允许访问的 IP，即为浏览器所在的 IP。
- **密级**: 用户所属的密级。



18.12.1 新建用户

可点击"添加"，进入新建用户页面。

- **用户名**: 应用服务器的用户名称，名称最多可以包含 255 个字符, 并且只能包含字母数字,下划线,短横线或点字符，需要唯一。
- **角色名**: 选择用户的角色，不同角色有不同的权限。分别有 security、sysadmin、auditor、publish_role（表示新建、编辑等操作受限制的用户）。
- **新口令**: 设置用户的口令，【系统管理】中，"密码复杂度"设置完为"普通"时，密码必须包含字母、数字和特殊符号中的至少两种组合;"复杂"时，密码必须包含字母、数字和特殊符号。
- **确认新口令**: 需要输入与"新口令"一致。

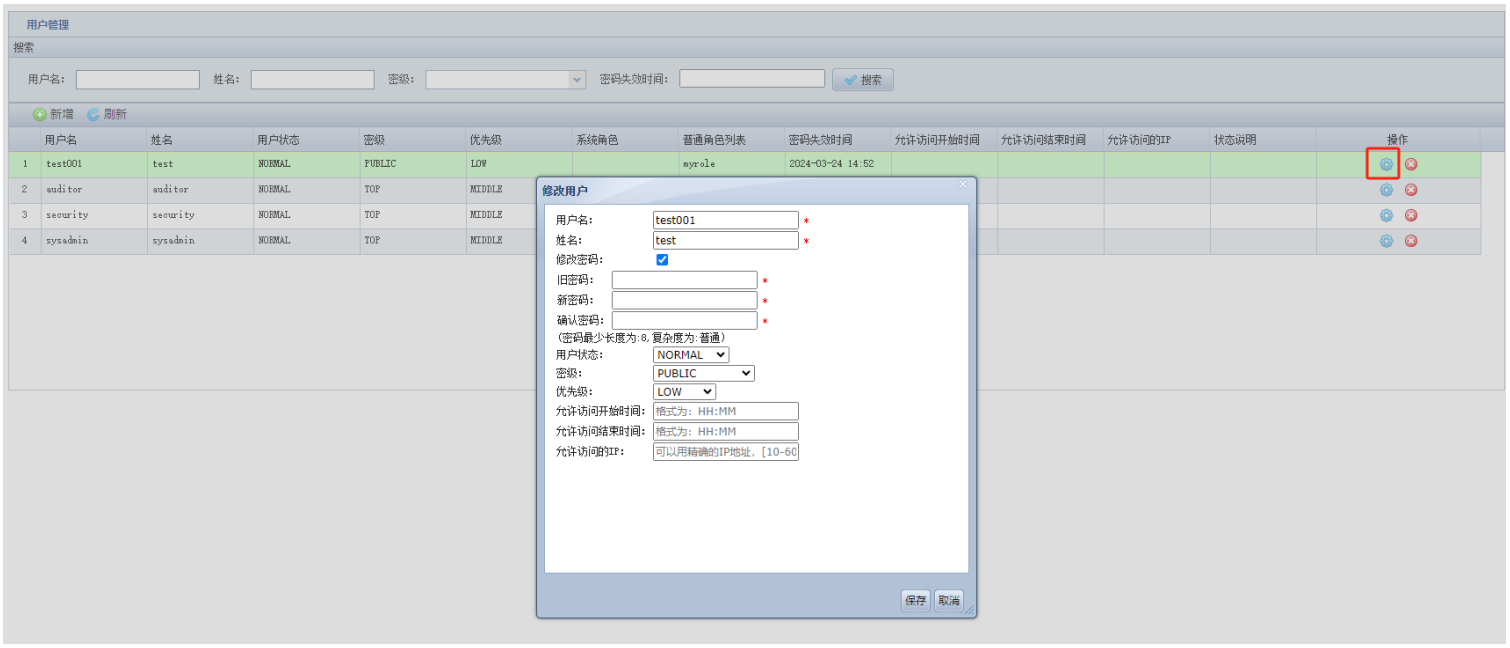


18.12.2 编辑用户信息

点击"用户管理"列表中的"操作"-"编辑用户"，进入对应用户的编辑页面。

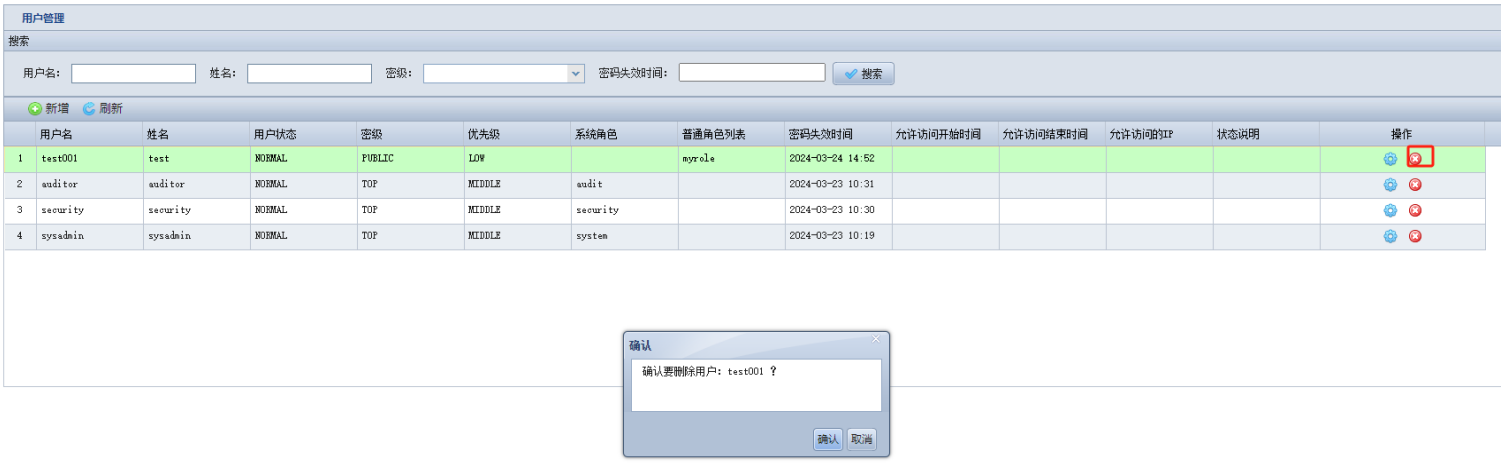
- **用户名**: 该用户的名称，不能编辑。
- **姓名**: 该用户的姓名。
- **用户状态**: 设置用户的状态，NORMAL 表示正常；LOCKED 表示锁定，默认为锁定 15 分钟；DISABLED 表示禁用。
- **系统角色**: 设置用户的角色。
- **修改密码**: 点击可以修改密码。
- **旧密码**: 修改密码时需要输入用户的旧密码。
- **新密码**: 设置用户的密码，【系统管理】中，"密码复杂度"设置完为"普通"时，密码必须包含字母、数字和特殊符号中的至少两种组合;"复杂"时，密码必须包含字母、数字和特殊符号。修改密码 5 次内不能重复。
- **确认新密码**: 需要输入与"新密码"一致。
- **允许访问开始时间（HH: MM）**: 用户允许访问管控台的开始时间，格式为 HH: MM。
- **允许访问结束时间（HH: MM）**: 用户允许访问管控台的结束时间，格式为 HH: MM。
- **允许访问 IP**: 对用户访问管控的 IP 进允许限制,即为浏览器所在的 IP 可以用精确的 IP 地址，[10-60]及*格式，多个 IP 使用英文逗号分隔。
- **密级**: 用户所属的密级，该用户只能访问同一等级或低等级的应用或资源。当前有秘密、机密、绝密三个等级，权限等级：秘密<机密<绝密。

> ***注意**: 编辑用户信息后，会同步刷新用户的 session 状态，处于登录中的会话将会失效，退出登录。*



18.12.3 删除用户信息

点击"用户管理"列表中的"操作"-"删除"，确认删除后将可删除该用户。需注意:系统必须有 system,security,audit分别对应的用户。

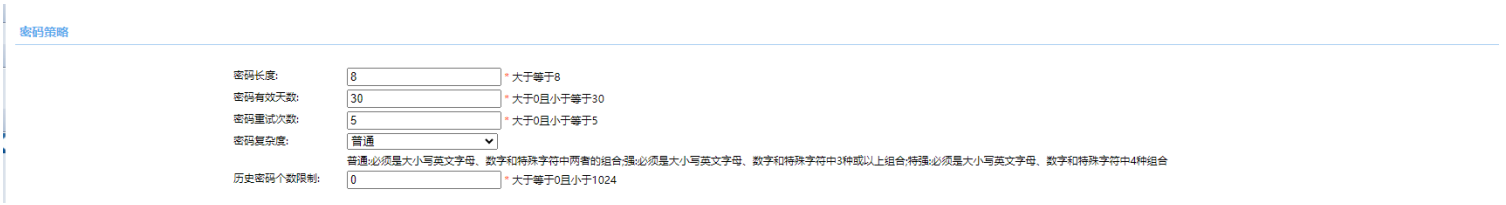


18.13 密码策略

安全管理员登录管控平台，可配置密码策略。

密码策略配置项主要总体配置用户的密码的长度复杂度等，具体属性如下:

- **密码长度**: 规定所有用户包括管理员的密码长度要求，必须大于等于设定值；默认为 8 个字符。
- **密码有效天数**: 规定密码从修改日开始的有效天数，超过有效天数则需要重新修改；默认为 30 天。
- **密码重试次数**: 用户登陆时允许重试密码的次数，超过此次数则锁定用户；默认为 5 次。
- **密码复杂度**: 普通，必须是大小写英文字母、数字和特殊字符中两者的组合；复杂，必须是大小写英文字母、数字和特殊字符中两者以上的组合；。
- **历史密码个数限制**: 修改用户密码时，限制多少次内密码不能重复。

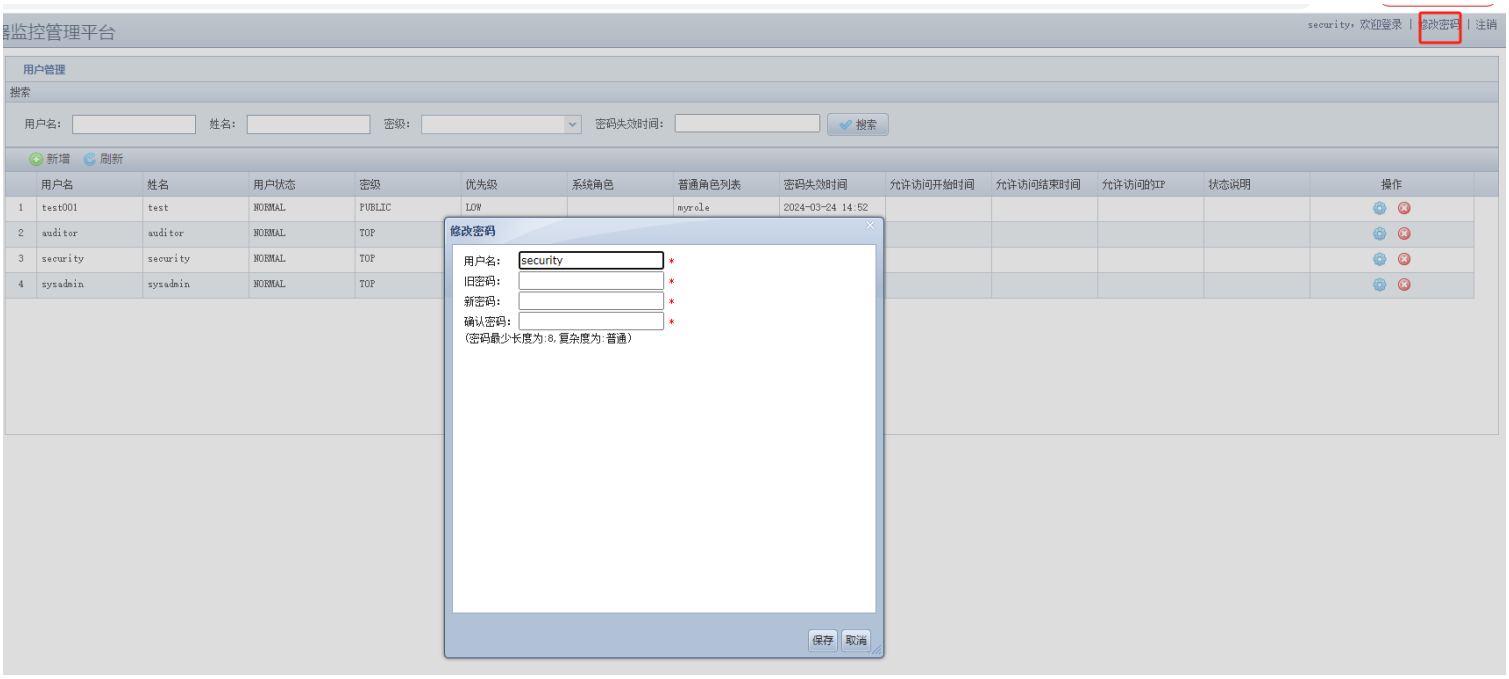


18.14 重置密码

用户重置密码可参考以下方法。

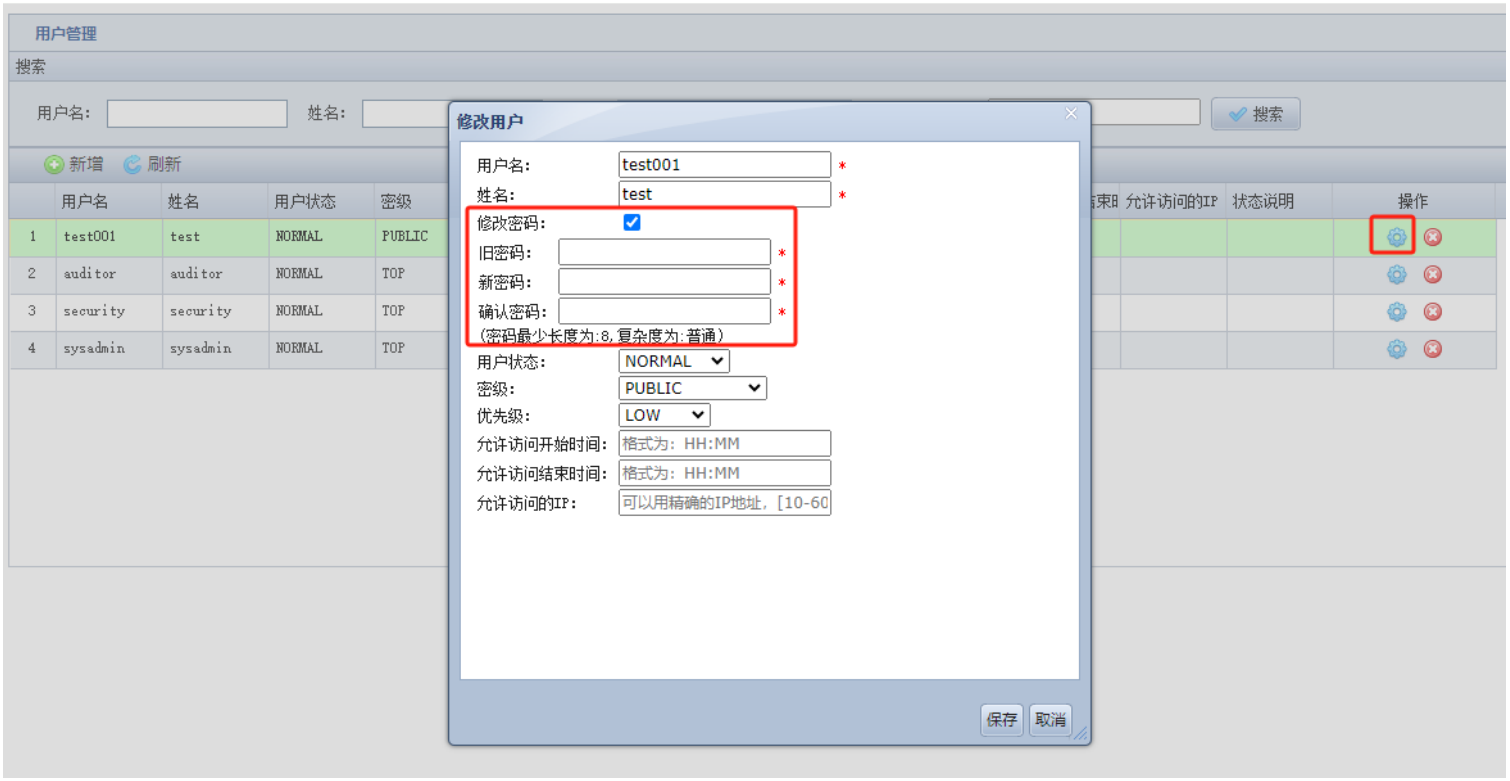
一、当前用户重置密码

当前用户登录管控平台，通过右上角"修改密码，进入【修改密码】页面，可自行修改密码。



二、通过安全管理员重置用户密码

安全管理员登录管控平台，进入【用户管理】，点击用户名称中的"操作，进入该用户编辑页面，可以为该用户设置密码。

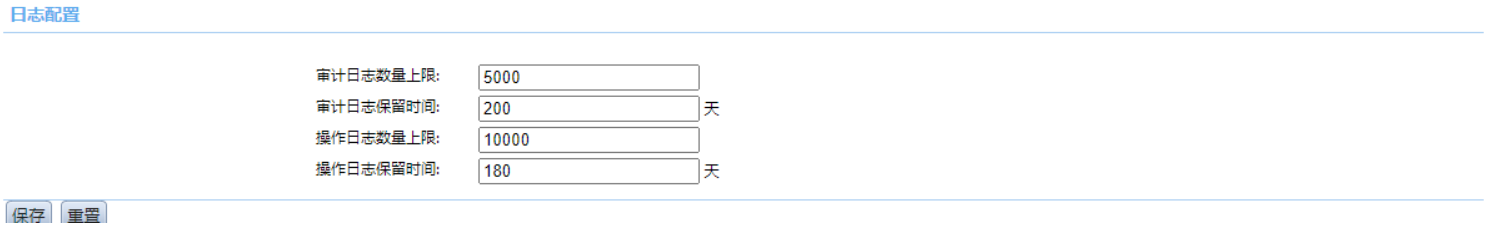


18.15 审计日志配置

安全管理员可管理审计日志和操作日志配置。

审计和操作配置项主要分别配置审计日志和操作日志的数量和保留时间等信息,具体配置属性如下:

- **审计日志数量上限**: 审计日志存储数据库里的数量限制，超过此限制则从最早的记录开始覆盖。
- **审计日志保留时间**: 审计日志记录保存时间，单位是天。
- **操作日志数量上限**: 操作日志存储数据库里的数量限制，超过此限制则从最早的记录开始覆盖。
- **操作日志保留时间**: 操作日志记录保存时间，单位是天。



18.16 日志审计

对系统重要的操作，敏感数据等操作或者一些非法操作等系统会生成相应的日志，审计员可以对这些日志进行审计。

18.16.1 操作日志

使用角色为审计管理员（audit）的用户，如默认审计管理员auditor，登录管控，切换到操作日志界面。



18.16.2 审计日志

使用角色为审计员的用户登录管控，切换到审计日志界面。



在审计日志界面可以做如下操作:

- 可以查看系统所有的审计日志，并且可以根据添加进行过滤查询。
- 点击【备份】按钮可以对审计日志进行备份操作，备份的日志文件存放在 `${APUSIC_HOME}/domain_name/store/audit` 目录下。

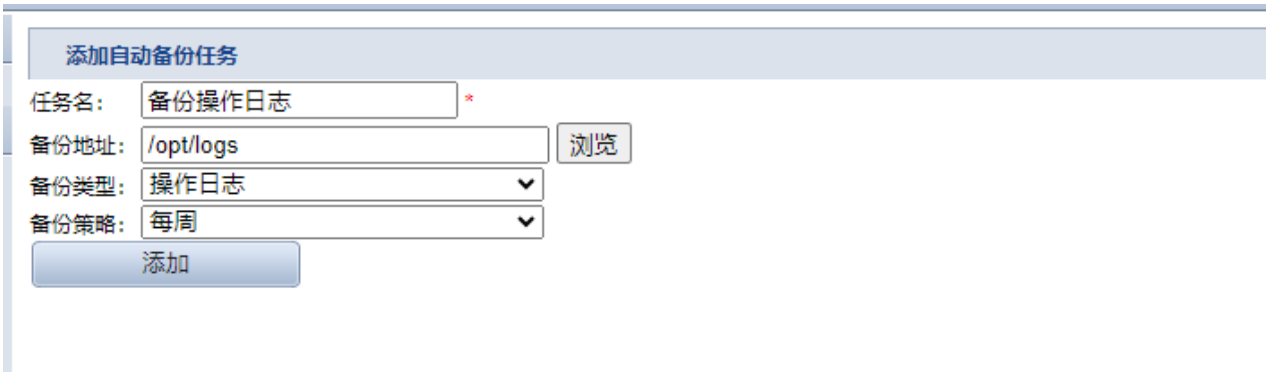
18.16.3 自动备份设置

使用角色为审计员的用户登录管控，切换到自动备份设置界面，可以设置自动备份策略。



点击"新增"，进入添加自动备份任务页面，输入：

- **任务名**: 定义该任务名称。
- **备份地址**: 选择该备份文件需要存放到的目录，该目录需要有权限。
- **备份类型**: 选择该任务的类型，操作日志或审计日志。
- **备份策略**: 设置自动备份的周期，每小时、每日、每周、每月、每年。



18.17 管控台转为普通应用

如果需要配置管控平台的应用属性，需要将管控平台作为普通应用程序部署到AAS中，再进行配置。操作如下：

1、拷贝 `${APUSIC_HOME}/lib` 目录下的 `webtool.war` 至 `${DOMAIN_HOME}/applications` 目录下。

2、启动AAS，等待管控平台应用 `webtool.war` 自动部署。

3、部署完成后，访问并登录管控平台。在【应用管理】模块可以看到有webtool应用，如果要配置属性，在该应用下配置即可；配置完成后需要重启AAS。

19 部分IX Apusic Http Server使用指南

19.1 Apusic Http Server启动和停止

通过%Apusic_Http_Server_HOME%/bin目录下的httpdctl控制脚本来启动、停止和重启Http Server。httpdctl脚本设置了正常运行httpd所必需的环境变量，然后调用httpd可执行文件。

	注意
	在linux中，1024以下的端口号为系统保留的端口。如果Http Server使用默认的80或者1024以下端口，那么启动Http Server需要root 权限以将它绑定到特权端口上。

如果启动过程一切正常，服务器将与终端分离并几乎立即出现命令行提示符。这表示服务器已经启动并开始运行。然后你就可以用你的浏览器去访问你的服务器来测试Http Server是否工作正常。

httpdctl脚本参数：

启动：	httpdctl start
立即停止：	httpdctl stop
平滑停止：	httpdctl graceful-stop
立即重启：	httpdctl restart
平滑重启：	httpdctl graceful

19.2 使用Apusic Http Server作为web代理

利用Apusic Http Server自身提供的mod_proxy_http模块就可以完成与Apusic应用服务器的集成，而且Apusic应用服务器端不需要做任何设置。mod_proxy_http提供代理HTTP请求的功能。支持HTTP/0.9,HTTP/1.0,HTTP/1.1标准。

代理配置：

需要在%Apusic_Http_Server_HOME%/conf/httpd.conf做如下配置：

```
...

LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
...

ProxyPass /${request_uri} http://192.168.6.119:6888/${request_uri}
```

```
...
```

其中${request_uri}为要代理的URI前缀。

19.3 使用Apusic Http Server作为负载均衡前置机

19.3.1 配置Apusic Http Server

编辑Apusic Http Server的配置文件%Apusic_Http_Server_HOME%/conf/httpd.conf，确保以下module：

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

处于加载状态。同时在文件末尾加入如下配置：

```
<VirtualHost *:80>

ProxyRequests off
ProxyPass / balancer://test/ #注意这里以"/"结尾
<Proxy balancer://test>

BalancerMember http://localhost:3888 loadfactor=1
BalancerMember http://localhost:4888 loadfactor=1
BalancerMember http://localhost:5888 loadfactor=1
BalancerMember http://localhost:6888 loadfactor=1
</Proxy>
</VirtualHost>
```

其中80端口是用户安装时配置的http协议监听端口，确认是否为80端口，可以查看配置文件中的Listen 选项。BalancerMember为后置机节点，后面的值为后置机的地址和端口。Loadfactor为负载权重。

当用户想使用会话粘滞（Session-Stick）,可以在ProxyPass/balancer://test/后面加入stickysession=JSESSIONID，同时在每个BalancerMember最后面加入route=RouteName；若用户想使用失效转移时还要在后面加入nofailover=off，。最终的配置形式如下：

```
<VirtualHost *:80>

ProxyRequests off
ProxyPass / balancer://test/ stickysession=JSESSIONID nofailover=off
```

```
<Proxy balancer://test>

BalancerMember http://localhost:3888 loadfactor=1 route=server0
BalancerMember http://localhost:4888 loadfactor=1 route=server1
BalancerMember http://localhost:5888 loadfactor=1 route=server2
BalancerMember http://localhost:6888 loadfactor=1 route=server3
</Proxy>
</VirtualHost>
```

19.3.2 配置Apusic应用服务器

19.3.2.1 配置高可用性

对于只需要负载均衡集群的用户，如果不需要配置高可用性，直接通过负载均衡器转发即可。如果需要负载均衡和Session高可用，则有两种配置方式。

1. session通过集群服务迁移的方式

为了使用集群，你需要在%DOMAIN_HOME%\config\apusic.conf中配置服务ClusterService，示例如下：

```
<SERVICE CLASS="com.apusic.cluster.ClusterService">
    <ATTRIBUTE NAME="ClusterName" VALUE="ApusicCluster" />
    <ATTRIBUTE NAME="LoadWeight" VALUE="100" />
    <ATTRIBUTE NAME="ServerName" VALUE="$DOMAIN_NAME" />
</SERVICE>
```

其中属性ClusterName为集群的名称，多个节点如果想加入到同一个集群中，ClusterName必须相同。ServerName为当前节点的名称，必须唯一，如果存在相同名称的节点，则后加入的节点会加不到集群中。默认的ServerName为当前Domain的名称，AutoReConnect属性，用于集群节点断开重连。

同时，你需要设定SessionService的Distributable和Replicable属性为True，示例如下：

```
<SERVICE CLASS="com.apusic.servlet.http.session.SessionService">
    <ATTRIBUTE NAME="DefaultSessionTimeout" VALUE="3600" />
    <ATTRIBUTE NAME="MaxSessionsInCache" VALUE="1024" />
    <ATTRIBUTE NAME="SessionInvalidateCheckInterval" VALUE="60" />
    <ATTRIBUTE NAME="Distributable" VALUE="True" />
    <ATTRIBUTE NAME="Replicable" VALUE="True" />
    <ATTRIBUTE NAME="SessionStick" VALUE="True" />
</SERVICE>
```

注意：在linux下以上配置都正确的情况下，出现节点不能正确加入集群，请设置etc/hosts文件的地址为局域网内部地址。

根据上面配置好集群之后要配置session的存储方式。在这里，session的存储方式只能是文件存储，关系数据库存储，嵌入式数据库存储中的一种。具体配置内容请参考请参考"配置Session存储"。

由于Apusic默认Session复制策略为配对复制的，如果用户想使用Session多点复制，需要设定ClusterService的配置属性：

```
<ATTRIBUTE NAME="ReplicationPolicy" VALUE="all" />
<!--ReplicationPolicy默认为"pair"，即配对复制。-->
```

在使用Session配对复制时，用户可以通过管理控制台，设置节点配对。也可以手动在ClusterService中增加属性SlaveServerName来指定其备份服务器，如下：

```
<ATTRIBUTE NAME="SlaveServerName" VALUE="serverName" />
```

其中value值为其他节点的ServerName。

2. 分布式session存储

这种方式只需要配置各个服务器的session存储统一为分布式存储即可。如何配置Session的分布式存储，请参考第 30.3 节"配置Session存储"。

19.3.2.2 配置Session粘滞

当用户想使用Session Stick时，不仅需要在apusic.conf文件中的SessionService中配置如下属性：

```
<ATTRIBUTE NAME="SessionStick" VALUE="true" />
```

还需要在Apusic应用服务器的vm.options(位置在%DOMAIN_HOME%/config目录下)文件中设置属性：

...

com.apusic.jvm.route=%RouteName%

...

其中，这里的%RouteName%为在第1步<Proxy balancer://test>中配置的与此Apusic应用服务器节点相对应的BalancerMember后面的route属性。

20 部分X Node Manager使用指南

20.1 概述

Apusic Node Manager（节点管理器）是一个节点管理平台，主要功能是管理和监控网络节点上受管服务器的进程，收集服务器运行的状态信息。Apusic Node Manager是一个可扩展的平台，可以通过插件的形式来扩展程序，增加新的功能只需开发Node Manager的插件，默认提供对Apusice Http server和Apusic Application Server的管理监控。

Apusic Node Manager为产品环境中的受管服务器提供的管理监控功能包括：

- 受管服务器的状态监控。
- 受管服务器在故障时自动重启。
- 对受管服务器进行远程和集中控制，包括启动和停止服务器。
- 受管服务器的运行数据采集。
- 辅助集群配置及应用部署。
- 修改受管服务器的配置。

一个Node Manager实例与具有唯一IP主机地址的逻辑或物理计算机系统对应，不能跨多台计算机。Node Manager会启动线程对该主机上的每个受管服务器进行监听，定时获取受管服务器的运行的状态。如果受管服务器进程意料外故障、失败或挂起状态，那么Node Manager能够自动重启受管服务器。

除了自动重启功能，Node Manager还具有智能控制功能，当集群需要增加应用服务器时，NodeManager 会自动启动其下配置的应用服务器实例，当集群压力达到可以释放应用服务器资源时，则会自动停止其管理的应用服务器。NodeManager对于其下已经启动的应用服务器会定时采集JVM信息及线程使用信息，用于统计运行压力情况。

Node Manager通过webService的形式向外发布接口，通过用户名与密码的方式来限制客户端对Node Manager Service的调用。从Apusic管理控制工具到Node Manager Service的通信也都是通过安全用户密码认证，保证通信的安全。

20.2 NodeManager启动与停止

NodeManager在各平台的启动与停止命令：

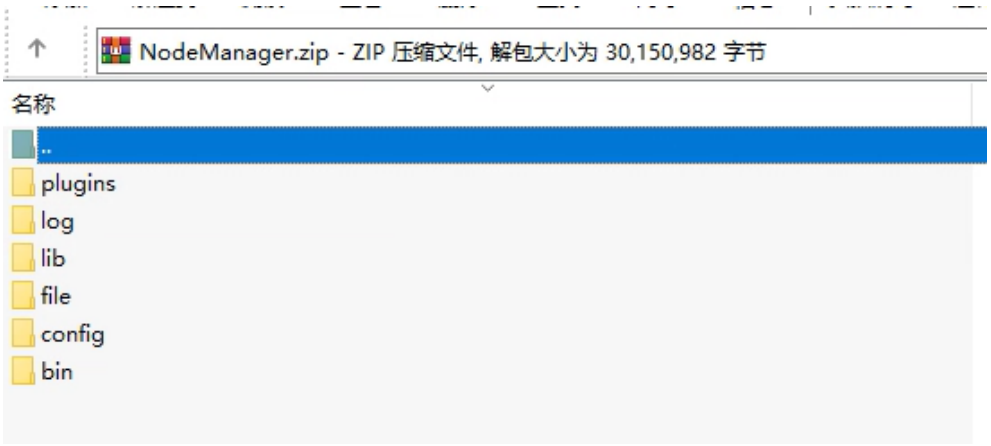
	Windows	Unix/Linux
启动	nodemanagerbinstartup.bat	nodemanagerbinstartup
停止	nodemanagerbinshutdown.bat	nodemanagerbinshutdown.sh

如果配置的端口不冲突，一台主机可以启动多个nodeManager。

20.3 NodeManager目录结构

20.3.1 目录结构

nodeManager的目录结构如下：



其中：

bin：目录下为启动nodeManager在各个平台的启动、停止脚本。

config：目录下是nodeManager的配置。

file：netty接收文件存放路径。

lib：目录下是具体实现的jar包。

log：目录下是日志信息。

plugins：目录下是开发的各种插件放置的位置。

plugins/aas-plugin目录是apusic application server插件的目录。

plugins/ahs-plugin目录是apusic http server Apache插件的目录。

plugins/nginx-plugin目录是apusic http server Nginx插件的目录。

20.4 主要的配置说明

20.4.1 Node Manager系统服务的配置文件

service.xml文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<services xmlns="http://www.apusic.com/aum/agent-service"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.apusic.com/aum/agent-service
http://www.apusic.com/aum/agent-service.xsd">
    <service class="com.apusic.agent.runtime.scheduler.SchedulerService"
```

```
name="SchedulerService">
        <attribute name="poolSize" value="10" />
    </service>
    <service class="com.apusic.agent.runtime.collector.CollectorManager"
name="CollectorManager">
        <attribute name="schedulerService" value="SchedulerService" />
    </service>
    <service class="com.apusic.agent.runtime.pool.ObjectPoolService"
name="ObjectPoolService">
    </service>
    <service class="com.apusic.agent.runtime.unit.UnitConfigManager"
name="UnitConfigManager">
    </service>
    <service class="com.apusic.agent.runtime.delegate.ObjectDelegateService"
name="ObjectDelegateService">
    </service>
    <service class="com.apusic.agent.runtime.jmx.JMXProtocolManager"
name="JMXProtocolManager">
        <attribute name="defaultProtocol" value="iiop" />
        <attribute name="supportProtocols" value="iiop,jmxmp" />
    </service>
    <service class="com.apusic.agent.runtime.ws.WebServiceManager"
name="WebServiceManager">
        <attribute name="webServicePort" value="9000" />
        <attribute name="isAuthentication" value="true" />
        <attribute name="userName" value="admin" />
        <attribute name="passWord" value="admin" />
    </service>
    <service class="com.apusic.agent.runtime.transport.DefaultDataSendPool"
name="DefaultDataSendPool">
    </service>
    <service
class="com.apusic.agent.runtime.transport.DefaultDataReceivePool"
name="DefaultDataReceivePool">
    </service>
    <service
class="com.apusic.agent.runtime.transport.netty.NettySendHandler"
name="NettySendHandler">
        <attribute name="dcReConnectJobTime" value="60" />
        <attribute name="dcReConnectTries" value="2" />
```

```
        <attribute name="dcReConnectWaitTime" value="5" />
        <attribute name="dcFilePreSize" value="10240" />
        <attribute name="dcClientfile" value="clientfile" />
        <attribute name="dcFilepath" value="file" />
    </service>
    <service
class="com.apusic.agent.runtime.transport.netty.NettyReceiveHandler"
name="NettyReceiveHandler">
        <attribute name="dispatcher" value="ActivityDispatcherService" />
    </service>
    <service
class="com.apusic.agent.runtime.transport.multicast.NMMulticastListner"
name="NMMulticastListner">
        <attribute name="multicastGroup" value="228.5.6.7" />
        <attribute name="multicastListenerPort" value="58889" />
        <attribute name="multicastTransportPort" value="58888" />
    </service>
    <service
class="com.apusic.agent.business.activity.ActivityDispatcherService"
name="ActivityDispatcherService">
    </service>
    <service class="com.apusic.agent.business.monitor.jvm.JvmMonitorService"
name="JvmMonitorService">
        <attribute name="dataSendPool" value="DefaultDataSendPool" />
        <attribute name="protocolManager" value="JMXProtocolManager" />
    </service>
    <service class="com.apusic.agent.business.monitor.event.EventsService"
name="EventsService">
    </service>
</services>
```

1. WebService管理器服务

```
<service class="com.apusic.agent.runtime.ws.WebServiceManager"
name="WebServiceManager">
    <attribute name="bindAddress" value="172.20.140.41" />
    <attribute name="webServicePort" value="9000" />
    <attribute name="isAuthentication" value="true" />
    <attribute name="userName" value="admin" />
```

```
    <attribute name="passWord" value="admin" />
</service>
```

属性	说明	默认值
webServicePort	指定webservice的监听端口	9000
bindAddress	指定监控的IP	字段未设置默认值，存在多网卡时可根据需要添加
isAuthentication	指定webservice是否启动访问认证	true
userName	指定webservice的访问认证用户名	admin
password	指定webservice的访问认证用户的密码	admin

2. 命令发送处理器服务

```
<service class="com.apusic.agent.runtime.transport.netty.NettySendHandler"
name="NettySendHandler">
    <attribute name="dcReConnectJobTime" value="60" />
    <attribute name="dcReConnectTries" value="2" />
    <attribute name="dcReConnectWaitTime" value="5" />
    <attribute name="dcFilePreSize" value="10240" />
    <attribute name="dcClientfile" value="clientfile" />
    <attribute name="dcFilepath" value="file" />
</service>
```

属性	说明	默认值
dcReConnectJobTime	指定和管理控制中心失去联系后重新连接的时间间隔，单位为秒	60秒
dcReConnectTries	指定和管理控制中心失去联系后重新连接重试的次数	2秒
dcReConnectWaitTime	指定每次和管理控制中心重新连接等待的时间，单位为秒	5秒
dcFilePreSize	指定对发送大文件时的分块大小，单位为KB	10240，即1M
dcClientfile	指定对发送文件进行备份的目录	
dcFilepath	指定对接收文件进行备份的目录	

3. 节点管理区多播服务

```
<service
class="com.apusic.agent.runtime.transport.multicast.NMMulticastListner"
name="NMMulticastListner">
```

```
    <attribute name="multicastGroup" value="228.5.6.7" />
    <attribute name="multicastListenerPort" value="58889" />
    <attribute name="multicastTransportPort" value="58888" />
</service>
```

属性	说明
multicastGroup	指定多播信息的IP地址
multicastListenerPort	指定多播信息监听端口
multicastTransportPort	指定多播信息发送端口

20.4.2 日志输出配置文件

logging.properties文件：

```
#reated by Apusic
#Fri Feb 24 15:05:06 CST 2012
java.util.logging.FileHandler.limit=1000000
java.util.logging.ConsoleHandler.formatter=com.apusic.agent.runtime.logger.Daem
handlers=java.util.logging.FileHandler,java.util.logging.ConsoleHandler
java.util.logging.FileHandler.count=10
java.util.logging.ConsoleHandler.level=ALL
java.util.logging.FileHandler.formatter=com.apusic.agent.runtime.logger.DaemonS
java.util.logging.FileHandler.pattern=log/log_%g.log
java.util.logging.FileHandler.encoding=utf-8
java.util.logging.FileHandler.level=ALL
java.util.logging.FileHandler.append=true
```

20.4.3 NodeManger插件配置文件

1. plugin-lists.xml

Plugins目录下有配置文件plugin-lists.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<plugins xmlns="http://www.apusic.com/aum/agent-pluginlist"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <plugin folder="aas-plugin" enable="true" />
    <plugin folder="ahs-plugin" enable="true" />
    <plugin folder="nginx-plugin" enable="false" />
</plugins>
```

默认是三个插件，一个是AAS插件，一个是Apache插件，一个是Nginx插件。

```
<plugin folder="aas-plugin" enable="true" />
```

是AAS插件配置，属性folder表示插件所在目录，默认是aas-plugin；enable表示是否启动该插件，enable为true表示启动插件，能够实现对Apusic Application Server实现管理和监控，enable为false则不开启插件。如果启动了该插件，则同时需要配置对应的插件目录中的plugin.xml文件，指定AAS的安装目录。

```
<plugin folder="ahs-plugin" enable="true" />
```

是Apache插件配置，属性folder表示插件所在目录，默认是ahs-plugin；enable表示是否启动该插件，enable为true表示启动插件，能够实现对Apusic Http Server实现管理，enable为false则不开启插件。如果启动了该插件，则同时需要配置对应的插件目录中的plugin.xml文件，指定Apache的启动脚本及配置文件路径。

```
<plugin folder="nginx-plugin" enable="false" />
```

是Nginx插件配置，属性folder表示插件所在目录，默认是Nginx-plugin；enable表示是否启动该插件，enable为true表示启动插件，enable为false则不开启插件。如果启动了该插件，则同时需要配置对应的插件目录中的plugin.xml文件，指定Nginx的启动脚本及配置文件路径。

注：同一个NodeManager只能启动一个负载均衡器插件

2. aas-plugin/plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="aas" provider="Apusic.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <attributes>
        <attribute name="apusicHome" value="F:/v8/Apusic-AS-8.0-OS-
Independent" />
        <!--
<attribute name="startCommand" value="startapusic.cmd" />
<attribute name="stopCommand" value="stopapusic.cmd" />
-->

<attribute name="scriptCharset" value="gbk" />
<attribute name="startTimeout" value="100000" />
<attribute name="shutdownMonitorInterval" value="60000" />
<attribute name="stateMonitorInterval" value="60000" />
</attributes>
<controller class="com.apusic.agent.plugin.aas.controller.AASController" />
```

```
<extension-point id="com.apusic.agent.ws" name="webService"

processor="com.apusic.agent.extension.ws.WebServiceProcessor"></extension-
point>
<extension point="com.apusic.agent.ws">
<webService
class="com.apusic.agent.plugin.aas.ws.impl.DomainAdminServiceImpl"

/>
<webService
class="com.apusic.agent.plugin.aas.ws.impl.ScriptAdminServiceImpl"

/>
</extension>
<extension-point id="com.apusic.agent.domain" name="JVM"

processor="com.apusic.agent.plugin.aas.processor.MonitorDomainProcessor">
</extension-point>
<extension point="com.apusic.agent.domain">
</extension>
<extension-point id="com.apusic.agent.collector" name="Collector"

processor="com.apusic.agent.extension.collector.CollectorProcessor">
</extension-point>
<extension point="com.apusic.agent.collector">
<collector name="MemoryCollector" displayName="MemoryCollector"

interval="15000"

class="com.apusic.agent.business.monitor.jvm.collector.MemoryCollector">
<metric name="HeapMemoryUsage" dataType="long"

displayName="HeapMemoryUsage"

description="" collectionType="dynamic" units="k" />
</collector>
<collector name="ThreadingCollector" displayName="ThreadingCollector"

interval="15000"
```

```
class="com.apusic.agent.business.monitor.jvm.collector.ThreadingCollector">
<metric name="BusyThreadCount" dataType="long"

displayName="BusyThreadCount"

description="" collectionType="dynamic" units="unit" />
<metric name="MaxThreads" dataType="long" displayName="MaxThreads"

description="" collectionType="dynamic" units="unit" />
</collector>
<collector name="SessionCollector" displayName="SessionCollector"

interval="15000"

class="com.apusic.agent.business.monitor.jvm.collector.SessionCollector">
<metric name="Stats" dataType="long" displayName="Current session
Count"

description="" collectionType="dynamic" units="unit" />
</collector>
</extension>
</plugin>
```

关键属性介绍:

属性	说明	默认值
apusicHome属性	指定插件监控的AAS的安装目录	
startCommand	指定插件监控的AAS的启动脚本名称，可以不填写。在windows平台下，默认为startapusic.cmd，其他平台都为startapusic	startapusic.cmd
stopCommand	指定插件监控的AAS的停止目录脚本名称，可以不填写。在windows平台下，默认为stopapusic.cmd，其他平台都为stopapusic	stopapusic.cmd
scriptCharset	指定启动和停止脚本的编码方式	gbk
startTimeout	指定执行启动和停止脚本的超时时间	
shutdownMonitorInterval	指定AHS故障检测时间间隔，单位为毫秒	60秒
stateMonitorInterval	指定获取AHS活动状态的时间间隔，单位为毫秒	60秒
interval	各个收集器(collector)的interval属性指定收集数据的间隔，单位为毫秒	15秒

服务中的其他属性不建议进行修改。

3. ahs-plugin/plugin.xml 应用服务器安装包AAS-V9.0-BM.zip自带龙芯和飞腾环境的Apache插件，可直接使用。示例中使用自带的龙芯环境的Apache插件

注：自带的负载均衡器需要解压在/opt/AAS/ 目录下

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="apache" xmlns="http://www.apusic.com/acp/agent-plugin"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.apusic.com/acp/agent-plugin
http://www.apusic.com/acp/agent-plugin.xsd">
    <attributes>
        <attribute name="operationFile"
value="/opt/AAS/alb/ahs/mips64/bin/apachectl" />
<attribute name="configFile"
value="/opt/AAS/alb/ahs/mips64/conf/httpd.conf" />
<attribute name="shutdownMonitorInterval" value="60000" />
<attribute name="stateMonitorInterval" value="60000" />
</attributes>
<controller
class="com.apusic.agent.plugin.apache.controller.ApacheLoadBalancerController"

/>
<extension-point id="com.apusic.agent.ws" name="webService Config"

processor="com.apusic.agent.extension.ws.WebServiceProcessor"></extension-
point>
<extension point="com.apusic.agent.ws">
<webService class="com.apusic.agent.plugin.apache.ws.ApacheProxyImpl"
/>
</extension>
</plugin>
```

属性	说明
operationFile	指定Apache的启动脚本文件
configFile	指定Apache的配置文件
shutdownMonitorInterval	指定Apache故障检测时间间隔，单位为毫秒，默认值为60秒
stateMonitorInterval	指定获取Apache活动状态的时间间隔，单位为毫秒，默认值为60秒

4. nginx-plugin/plugin.xml 应用服务器安装包AAS-V9.0-BM.zip自带龙芯和飞腾环境的Nginx插件，可直接使用。示例中使用自带的飞腾环境的Nginx插件。

注：自带的负载均衡器需要解压在/opt/AAS/ 目录下

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="loadbalance" xmlns="http://www.apusic.com/acp/agent-plugin"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.apusic.com/acp/agent-plugin
http://www.apusic.com/acp/agent-plugin.xsd">
    <attributes>
        <attribute name="operationFile"
value="/opt/AAS/alb/nhs/arm64/sbin/nginx" />
        <attribute name="configFile"
value="/opt/AAS/alb/nhs/arm64/conf/nginx.conf" />
        <attribute name="shutdownMonitorInterval" value="60000" />
        <attribute name="stateMonitorInterval" value="60000" />
    </attributes>
    <controller
class="com.apusic.agent.plugin.nginx.controller.NginxLoadBalancerController"
/>
    <extension-point id="com.apusic.agent.ws" name="webService Config"
        processor="com.apusic.agent.extension.ws.WebServiceProcessor">
    </extension-point>
    <extension point="com.apusic.agent.ws">
        <webService class="com.apusic.agent.plugin.ws.NginxProxyImpl" />
    </extension>
</plugin>
```

属性	说明
operationFile	指定Nginx的启动脚本文件
configFile	指定Nginx的配置文件
shutdownMonitorInterval	指定Nginx故障检测时间间隔，单位为毫秒，默认值为60秒
stateMonitorInterval	指定获取ANginx活动状态的时间间隔，单位为毫秒，默认值为60秒

21 部分XI 动态集群管理

21.1 概述

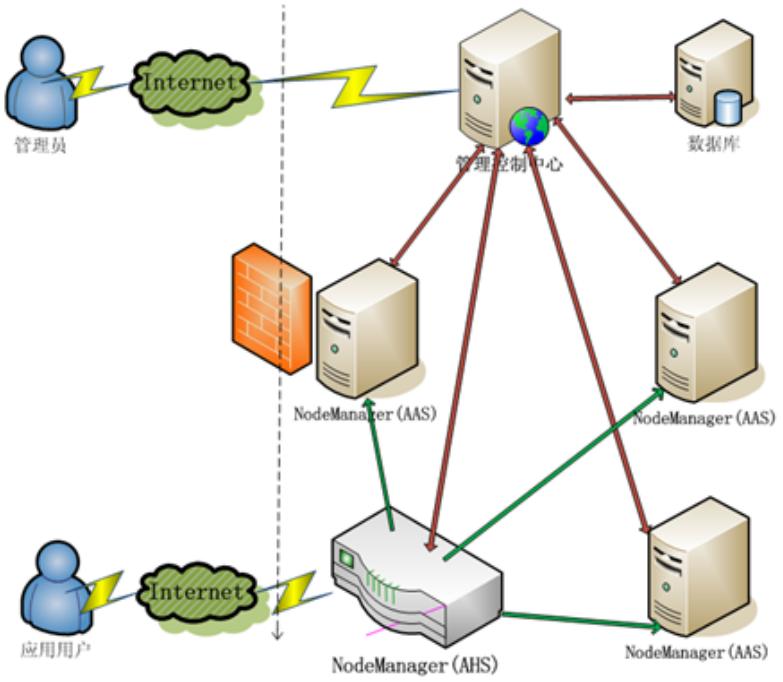
21.1.1 前言

动态集群管理系统包含管理控制中心（Console）及节点管理器（NodeManager）两个部分管理控制中心负责保存数据、提供基础服务以及运行其他定时任务，以及整个系统的管理界面，包括监控信息的展示等人机交互信息；节点管理器则主要用于协助部署集群和应用、获取监控数据和对托管的AAS和AHS 进行生命周期管理。

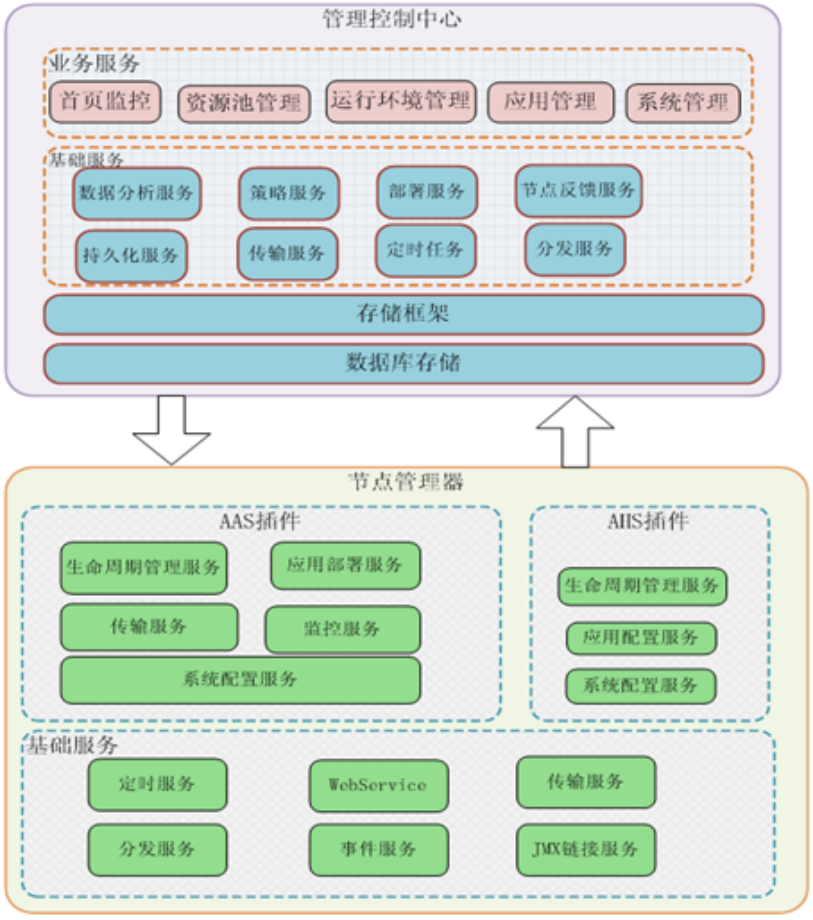
通过动态集群管理系统，能够实现AAS与AHS资源（Apache和Nginx）的统一管理、集群应用的集中部署以及能够为集群应用配置运行策略，实现按需使用AAS资源，达到充分利用有限资源，提高集群应用服务质量的目的。

21.1.1.1 系统网络拓扑图

从下面网络图看到，管理控制中心独立于节点管理器进行部署，运行环境为AAS；节点管理器作为独立的应用程序，它部署在运行有AAS或者AHS的机器上，实现对AAS和AHS的管理与监控。节点管理器管理的AAS或AHS，就是动态集群管理系统的资源，它通过管理控制中心的资源池管理功能进行注册，注册后就可以为集群应用提供服务。



21.1.1.2 系统架构图



21.2 节点管理器

21.2.1 NodeManager启动与停止

详情请参考

21.2.2 NodeManager目录结构及配置文件详细说明

详情请参考

21.3 管理控制中心

21.3.1 概述

管理控制中心是一个基于数据的服务平台，管理网络上各节点的交互，保存服务器运行的状态等信息；同是它也是用户管理操作的主要入口，是整合各个节点管理器的纽带。

管理控制中心主要功能包括负责整个系统的管理界面；保存系统相关信息，如节点信息、集群信息、应用基本信息等；保存节点管理器收集到的各个集群和应用的性能数据；分析性能数据和实施集群策略等。

21.3.2 安装与配置

21.3.2.1 部署“管理控制中心”应用包

1. 把“管理控制中心”应用包install-CloudAdmin.war放到应用服务器某个可用域下的applications目录，比如：放到服务器mydomain域下的applications目录。
2. 进入应用服务器域下的bin目录，比如服务器bin目录，执行 startas mydomain 启动管控中心服务。或进入服务器mydomain域下的bin目录，运行启动脚本startapusic就启动了管控中心服务。通过http://ip:port/cloudadmin即可访问。

21.3.2.2 “管理控制中心”个性化配置

1. applicationContext-service.xml配置文件

该配置文件在产品包目录WEB-INF/apusic-workbench/plugins/workbench-as-datacenter/resources中

21.3.2.2.1 部署机器为多网卡，设置指定与NODEMANGER交互的端口

可以修改配置文件中id为dsConfig的Bean属性multicastLocalHost进行配置。

该属性在产品包目录WEB-INF/apusic-workbench/plugins/workbench-as-datacenter/resources/applicationContext-service.xml 中

```
<bean id="dsConfig"
    class="com.apusic.datacenter.runtime.datacenter.DataCenterConfig">
    <property name="filePreSize" value="10240" />
    <property name="nettyPort" value="9001" />
    <property name="multicastGroup" value="228.5.6.7"></property>
    <property name="multicastListenerPort" value="58888"></property>
    <property name="multicastTransportPort" value="58889"></property>
    <property name="multicastLocalHost" value="172.20.140.40"></property>
    <property name="nettyExpireTime" value="40" />
    <property name="sendFilePath" value="send" />
    <property name="receiveFilePath" value="receive" />
    <property name="backupPath" value="template" />
    <property name="osCachePath" value="oscache" />
    <property name="checkTable" value="true" />
    <property name="clusterExpireTime" value="180000" />
    <property name="appExpireTime" value="180000" />
</bean>
```

21.3.2.2.2 数据库配置

管理控制中心使用数据库对系统信息及监控数据进行保存，默认情况下，使用derby内置数据库进行存储，也可以配置使用其他的外置数据库，配置方式可以使用dbcp连接池以及外接数据源的方式，可以通过配置文件中id为ds的Bean进行配置。

1. dbcp配置方式

以连接mysql数据库为例子，则示范代码类似:

```
<bean id="ds" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close" abstract="false" scope="singleton">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://192.168.100.60:3306/console?
useUnicode=true&characterEncoding=utf-8" />
    <property name="maxActive" value="100" />
    <property name="maxIdle" value="10" />
```

```
    <property name="maxWait" value="1000" />
    <property name="username" value="root" />
    <property name="password" value="123456" />
    <property name="initialSize" value="5" />
</bean>
```

2. 外接数据源的方式

在AAS中配置好数据源，如JNDI名称为jdbc/cloudadmin,则示范代码为：

```
<bean id="ds" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/cloudadmin" />
    <property name="lookupOnStartup" value="false" />
    <property name="cache" value="true" />
    <property name="resourceRef" value="true" />
    <property name="expectedType" value="javax.sql.DataSource" />
    <property name="proxyInterface" value="javax.sql.DataSource" />
</bean>
```

21.3.2.2.3 数据库创建脚本

管理控制中心支持多种关系型数据库：

Derby数据库，Derby10.5及Derby10.9版本通过测试

Mysql数据库，Mysql5.0版本通过测试

Oracle数据库，Oracle 10g版本通过测试

SQLServer数据库，SQLServer2000版本通过测试

对于不同数据库创建脚本在产品包下的WEB-INF/apusic-workbench/plugins/workbench-as-datacenter /resource目录，derby及mysql使用DB_CREATE.sql脚本，oracle使用DB_CREATE_ORACLE.sql脚本，sqlserver使用DB_CREATE_SQLSERVER.sql脚本。

21.3.3 动态集群管理监控台

推荐使用IE7或以上、firefox或者google chrome浏览器。

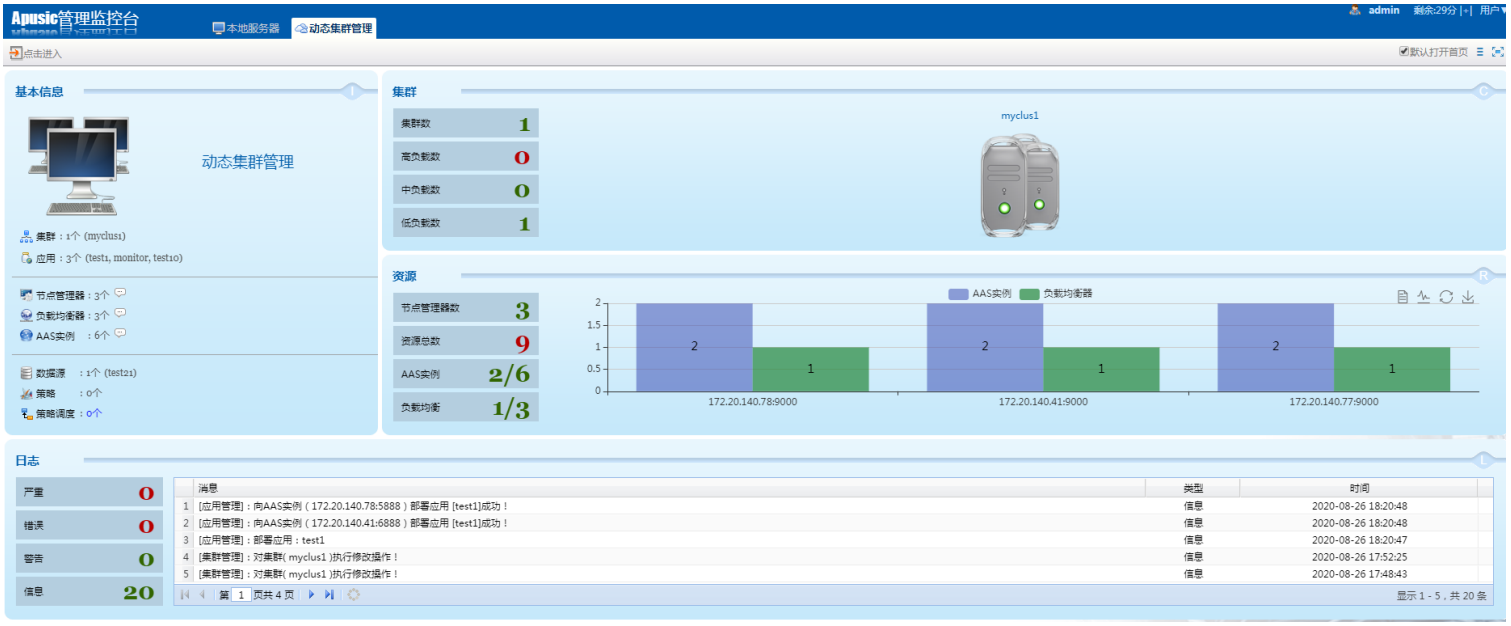
21.3.3.1 进入到登录页面

打开一个浏览器，输入以下URL：http://serverIP:serverPort/cloudadmin。其中ServerIP和ServerPort根据实际情况调整，如http://localhost:6888/cloudadmin/进入到管理监控台登录页面：



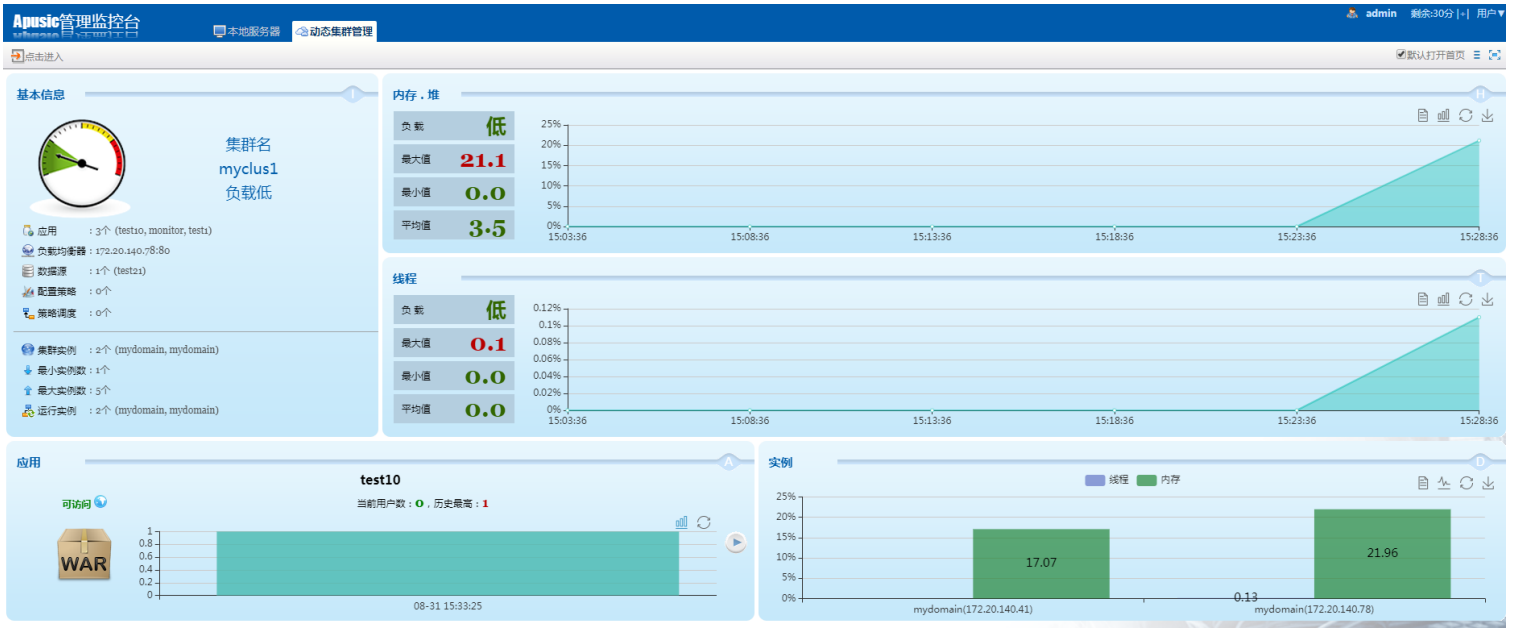
21.3.3.2 登录到动态集群管理监控首页

输入用户名“admin”，密码“admin”【如有更改请相应调整】，点击“登录”，进入动态集群管理监控台:



此界面显示的信息有“集群基本信息”、“集群列表”、“资源使用率”和“系统日志”。

在集群列表中选定一个集群，点击集群图标，进入到集群监控页面:



21.3.3.3 进入动态集群管理

登录后点击“动态集群管理”进入到动态集群管理功能介绍界面，在此界面下介绍了动态集群管理的各个功能。



21.3.3.4 资源池

21.3.3.4.1 资源管理

该模块管理的每一个资源对应着一个Apusic Node Manager，通过它间接管理和监控AAS或Apusic HttpServer资源，Apusic HttpServer一般作为应用的负载均衡器。一个资源可以对应一个apusic应用服务器和一个负载均衡器，也可以只管理其中的一个。



一. 新增资源

(1). 手动新增资源

填写资源的相关属性，如主机IP、端口等，资源可能是AAS资源和负载均衡器资源中的一个或者全部。同时，也可以通过填写主机属性信息更加清晰地标识这个资源的相关物理信息，如：使用的操作系统，CPU，内存等。点击“新增资源”按钮，进入到新增资源界面，如下：



具体配置信息如下：

属性名称	用途说明
别名	自定义的资源的名称，同时也用来唯一标识该资源。
主机IP	运行Apusic NodeManager所在机器的IP地址。
端口	Apusic NodeManager运行所监听的端口号。

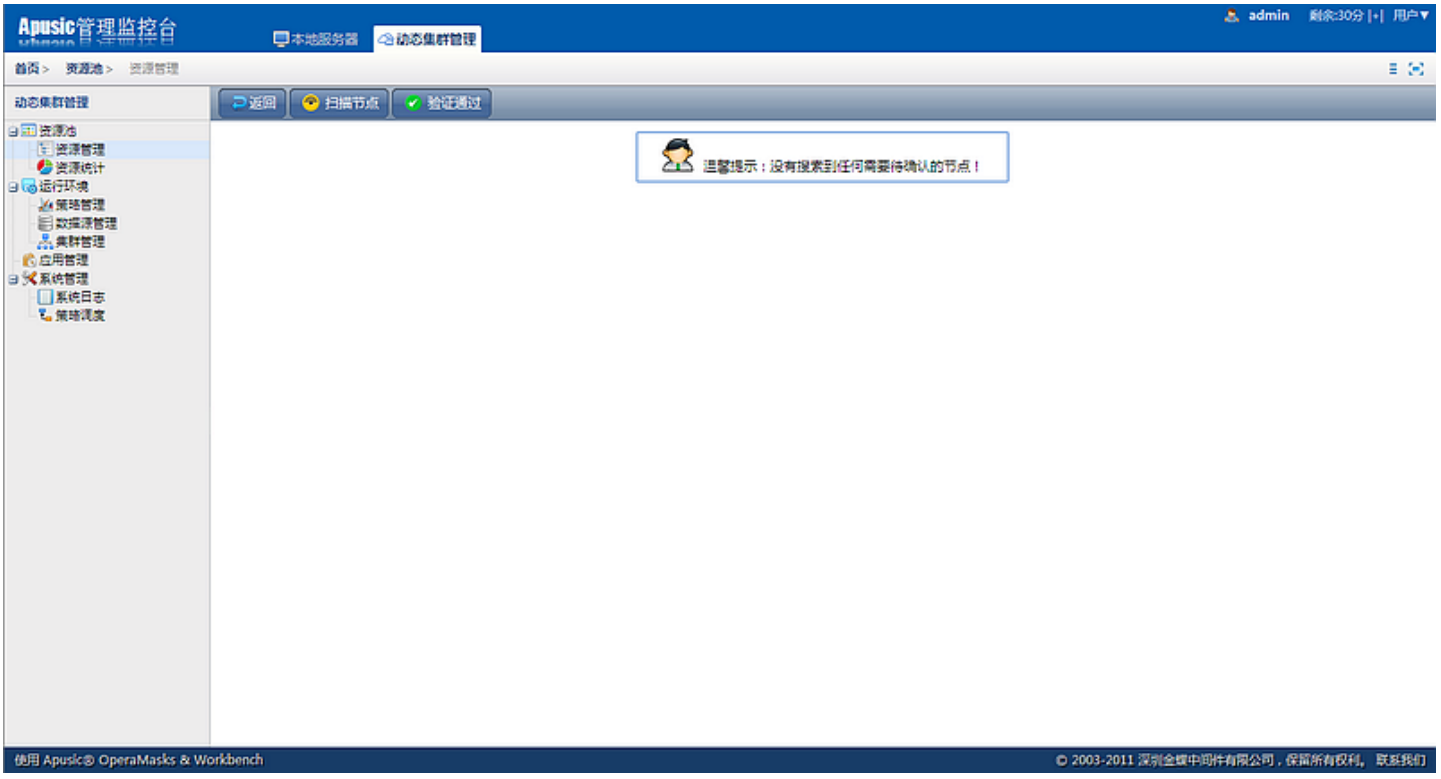
用户名	访问Apusic NodeManager服务的用户名，需要和其配置文件设置一致。
密码	访问Apusic NodeManager服务的密码，需要和其配置文件设置一致。
操作系统	运行Apusic NodeManager所在的操作系统，可以通过“自动获取”得到。
CPU	运行Apusic NodeManager所在的主机所用CPU情况，可以通过“自动获取”获取到CPU核数，更详细的信息可以手动补充。
内存	运行Apusic NodeManager所在的主机所用内存信息。
位置	运行Apusic NodeManager所在的主机所在机房位置。
信息	运行Apusic NodeManager所在的主机的其他备注信息。
用户名	访问Apusic Node Manager服务的用户名，需要和其配置文件设置一致。

信息填写完成后，点击“确定”按钮，新增资源成功，该资源会在未用资源列表中显示。

(2). **自动搜索**

动态集群管理控制台支持自动搜索资源并添加到资源池中，这里要求节点管理器配置文件中NMMulticastListner服务中的multicastTransportPort属性与管理控制台war包中的端口一致。

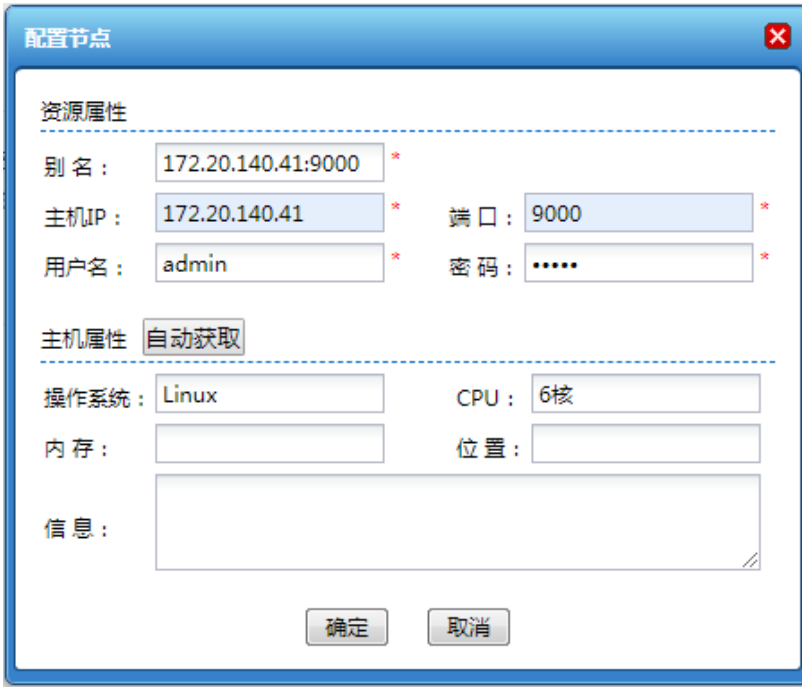
点击“自动搜索”，进入到扫描节点界面



点击“扫描节点”，然后选择要加入到资源管理的资源，点击“验证通过”完成。

二. **配置资源**

选定资源，点击"配置"按钮，则打开配置资源界面，修改除主机IP，端口之外的其他信息，修改后，点击"确定"按钮，修改资源完成；

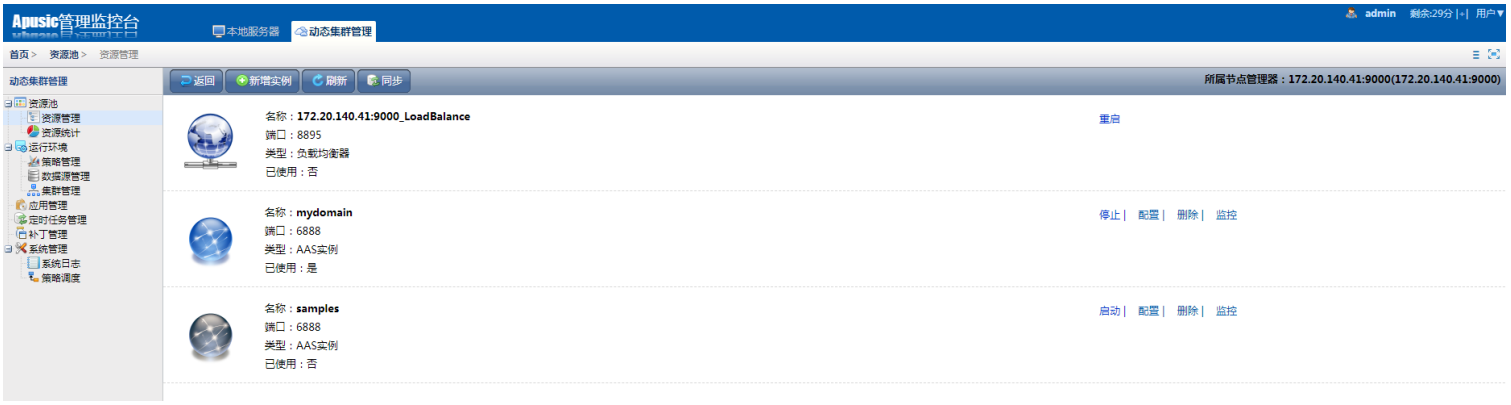


三. **删除资源**

选定资源，点击"删除"按钮，在提示信息中点击"是"，则完成资源删除。如果资源下的实例已经加入到集群中，则无法删除资源。

四. **实例管理**

点击"实例管理"按钮，打开"实例管理"界面，如下：



在实例管理界面下，用户可以对资源管理的apusic服务器实例和负载均衡器进行管理。

1. 负载均衡器的操作

如果资源管理的实例有负载均衡器，则可以对负载均衡器进行启动，重启操作。

2. apusic实例的启动，停止

如果资源管理的实例有apusic实例，则可以对域进行启动，停止，删除操作。

如果域已经加入到集群中，则不能进行停止启动和删除的操作。

3. apusic实例的监控

如果资源管理的实例有apusic实例，点击"监控"链接到单机版的监控。

4. apusic实例配置

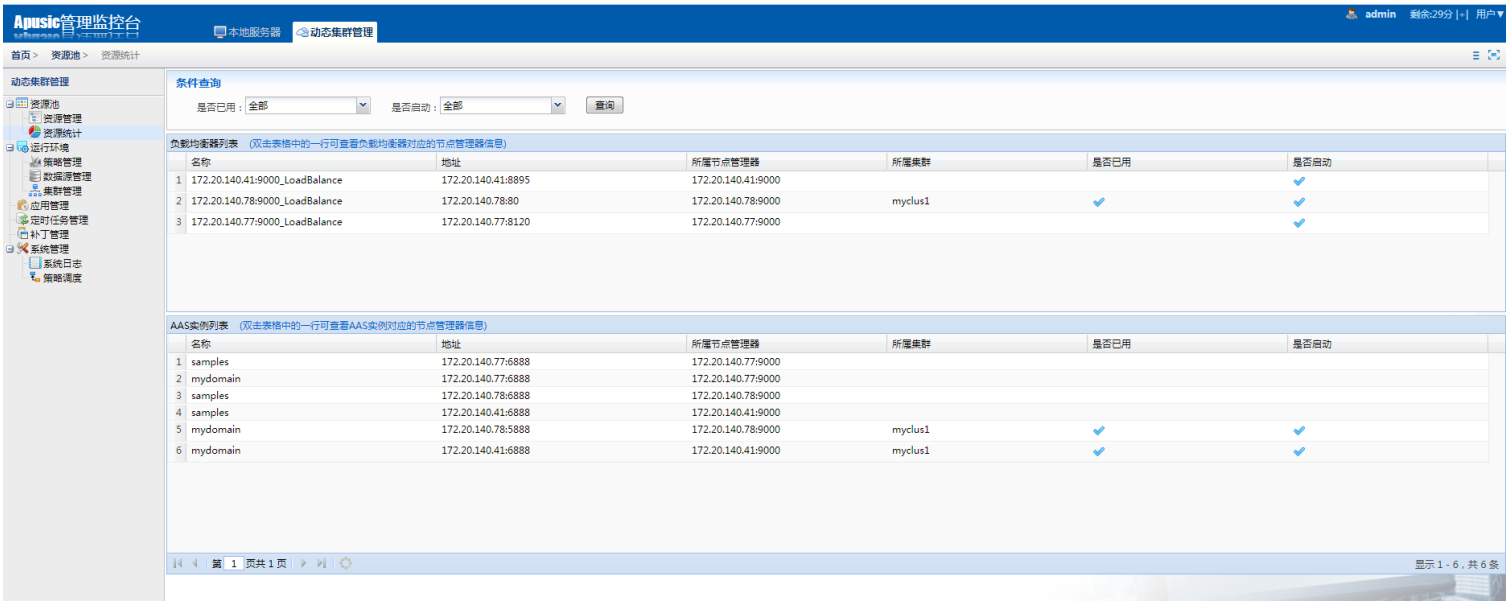
如果资源管理的实例有apusic实例，则可以对域进行密码的配置。

5. 资源同步

如果手动更改了负载均衡器或者管理的apusic实例的配置，则在重启节点管理器后进入到实例管理下可以对资源进行同步操作，以保证系统管理的资源与实际状态一致。

21.3.3.4.2 资源统计

点击"资源统计"进入到AAS和AHS资源的统计界面。在此界面下用户可以查看AHS和AAS实例的运行状态以及使用状态。



21.3.3.5 运行环境

21.3.3.5.1 策略管理

定制应用可以使用的策略。策略定义了一些条件及满足该条件时要采取的措施，即要执行的反馈动作。该版本中，可以定义的条件包括线程和内存的使用情况；反馈的动作包括增加资源节点和减少资源节点。

打开动态集群管理功能导航树，点击"策略管理"菜单，进入策略管理界面，如下：



一. 新增策略

通过定义条件和反馈动作等信息，新增加一个策略。

点击“策略管理”界面的“新增策略”按钮，打开“新增策略”窗口，如下：



具体配置信息如下：

1. 名称：用户自定义的策略名称，作为唯一标识策略的属性，必须唯一，不能为空，且新增后不能修改；
2. 描述：用户对该策略的一些详细说明和描述；
3. 策略类型和条件：策略类型设定了一些条件分类。它设定了一些条件，如：内存，线程超过或低于指定值；
4. 反馈模式：包含自动模式和手动模式，如果是自动模式，则达到设定的条件，系统自动执行反馈动作；如果是手动模式，则反馈动作由用户手动触发，该模式在该版本没有提供。
5. 反馈动作：集群应用达到指定条件时采取的措施，如：动态新增节点，即当集群应用达到指定的条件时，自动从资源池中获取空闲的资源添加到应用中，从而减少集群应用负载压力；

点击“确定”按钮，即可成功新增一个策略。

二. 配置策略

提供对策略的一些配置信息进行修改的功能。其中，“策略名称”和“策略类型”不能修改。

点击“策略管理”界面的”配置策略”按钮，打开“配置策略”窗口，如下：



点击“确定”按钮，即可成功修改当前选中的策略。

三. 删除策略

选定一个策略，点击“策略管理”界面的“删除”按钮：



在打开的确定窗口中点击“是”即可成功删除选中的策略，如果该策略已经被集群使用，则不能删除。

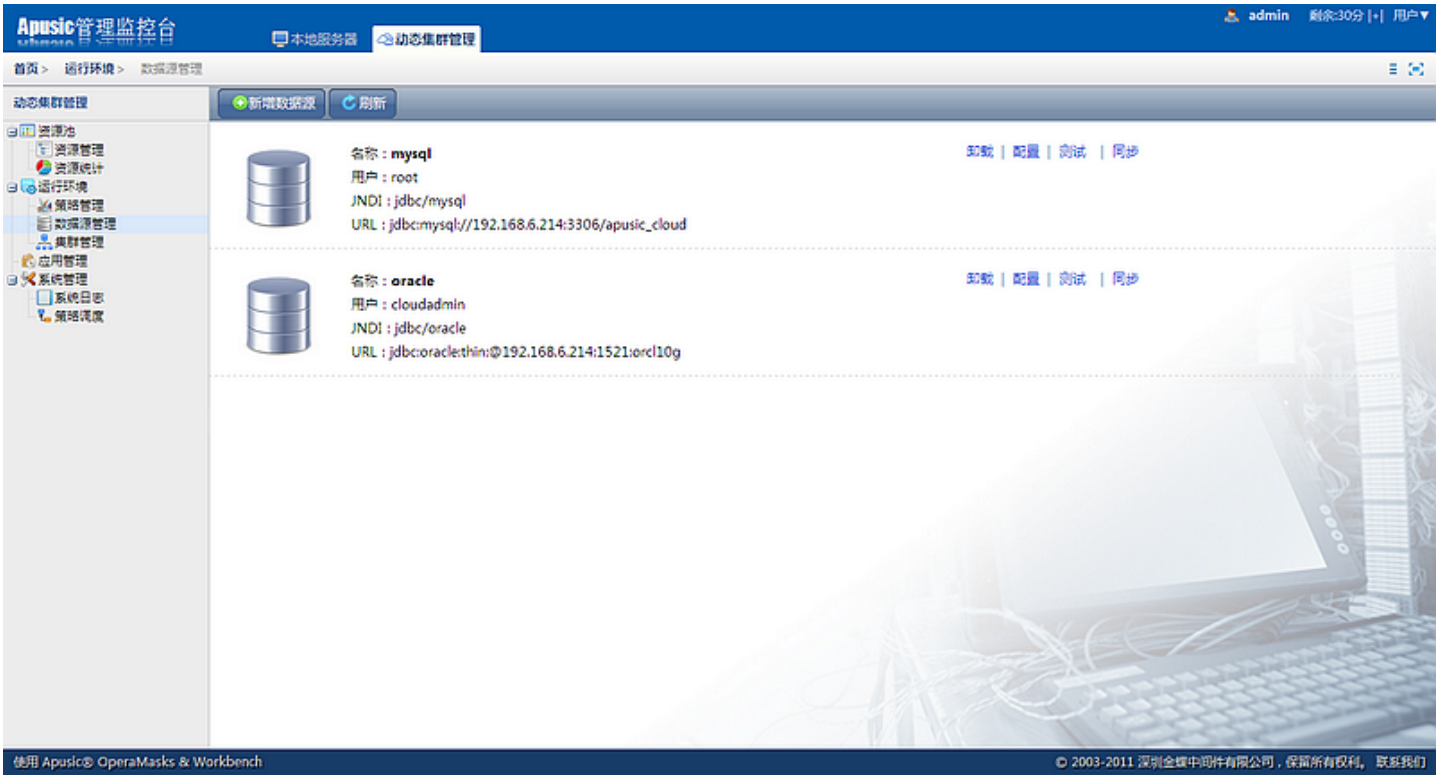
21.3.3.5.2 数据源管理

通过该功能实现对应用中使用到的数据源进行统一配置和管理。

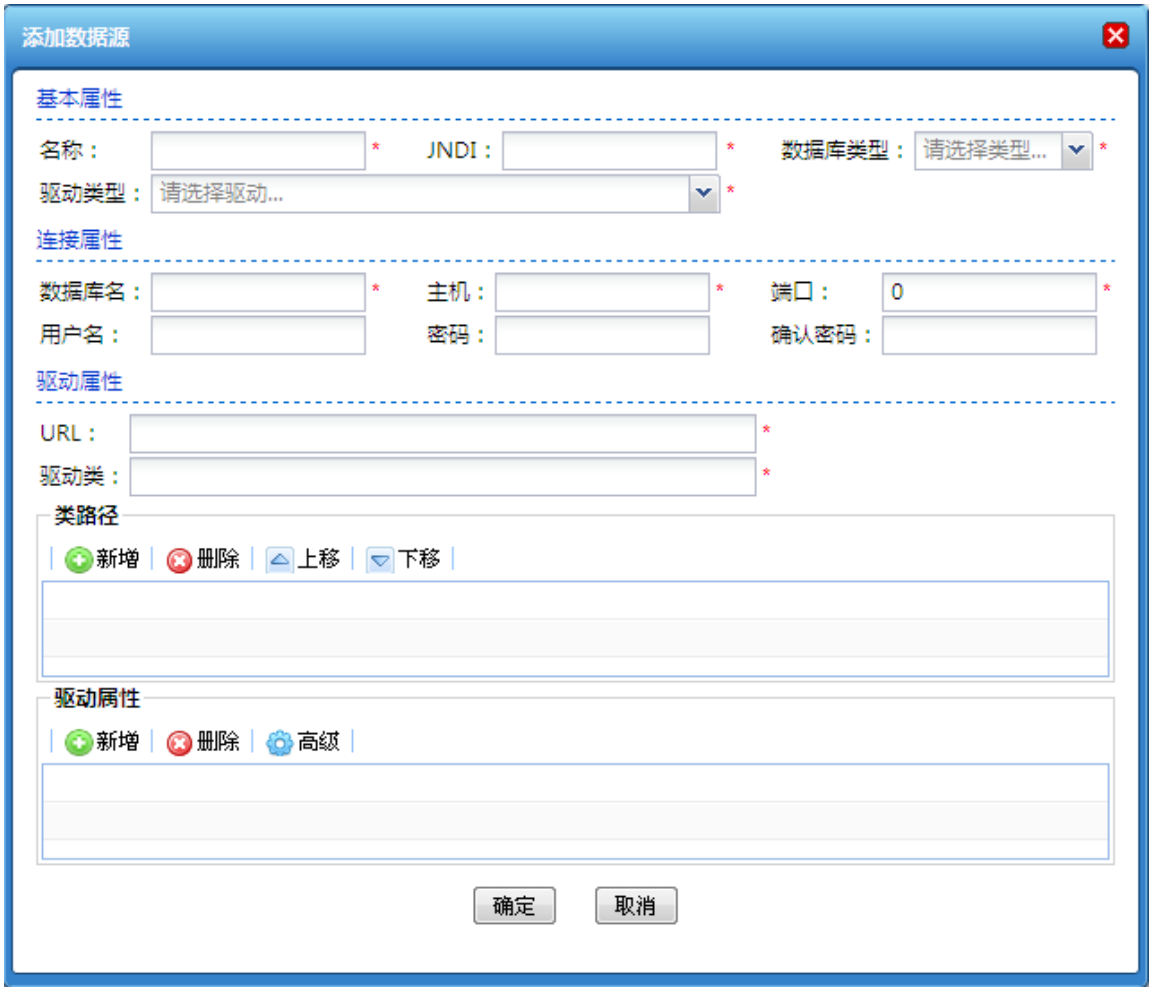
一. 新增数据源

新增一个数据源，部署应用时可以把新增的数据源添加到应用中。

打开动态集群管理功能导航树，点击“数据源管理”菜单，进入操作界面，如下：



然后，点击“新增数据源”按钮，填写数据源信息，并添加相关的数据源驱动jar包



基本属性：要添加的数据源的基本信息，具体包括：

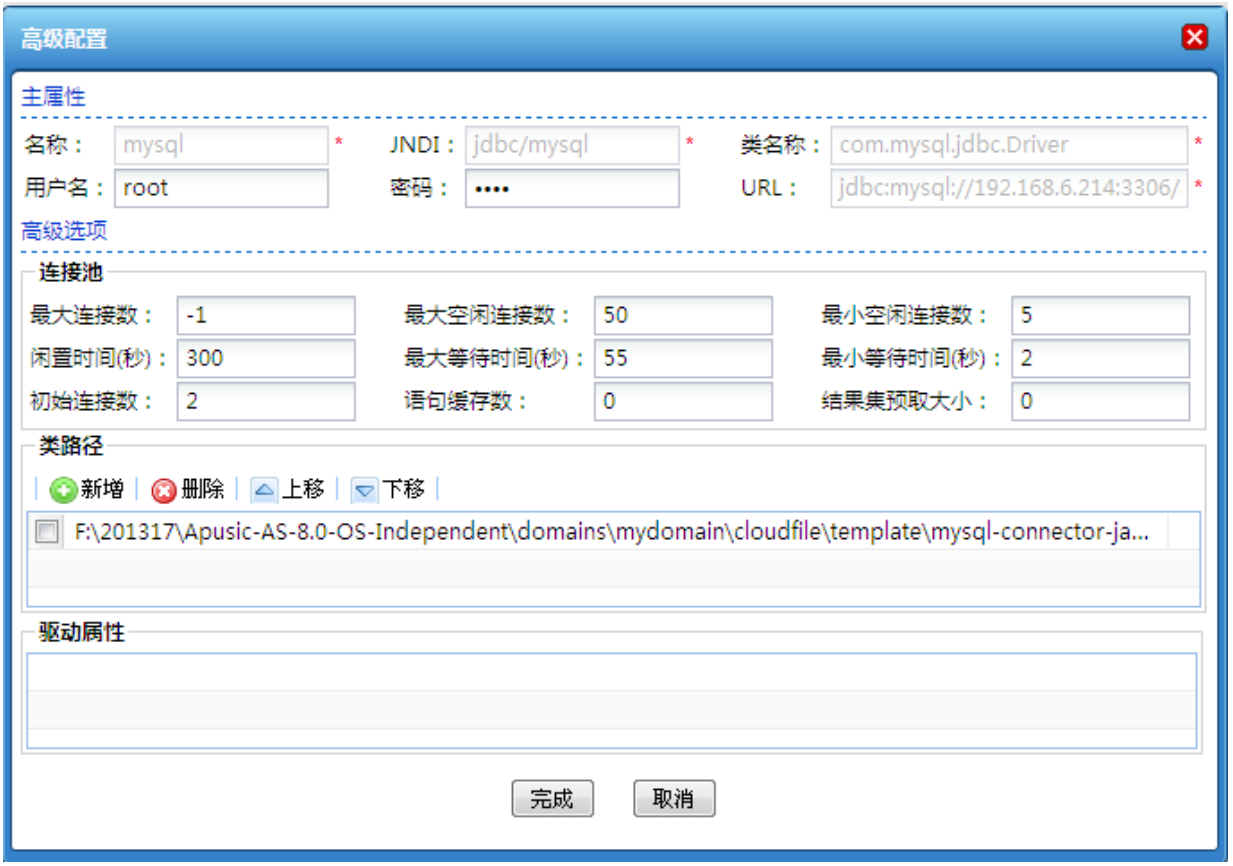
1. 名称：用户自定义的数据源名称，作为唯一标识数据源的字段；
2. JNDI: 数据源所对应的JNDI名称；
3. 数据库类型：下拉供用户选择，包括常用的数据库：mysql，Oracle，DB2等；
4. 连接属性：应用连接数据源时必须具备的属性，指定了要连接的数据库所在的主机，端口，用户名，密码和数据库；
5. 驱动属性：根据上述填写的信息构造的JAVA连接该数据源时需要的URL地址和驱动类；驱动的jar包，相关的驱动属性；

信息填写完成后，点击确认按钮

二. **配置数据源**

提供用户配置数据源的一些基本信息，主要配置连接池的基本属性和用户名，密码等。

打开动态集群管理功能导航树，点击"数据源管理"菜单，进入操作界面，选定数据源后点击"配置"，打开"配置数据源"界面，如下：



三. **测试数据源**

测试该数据源是否配置正确和连通可用。

选定数据源，点击“测试”，打开测试窗口，如下：

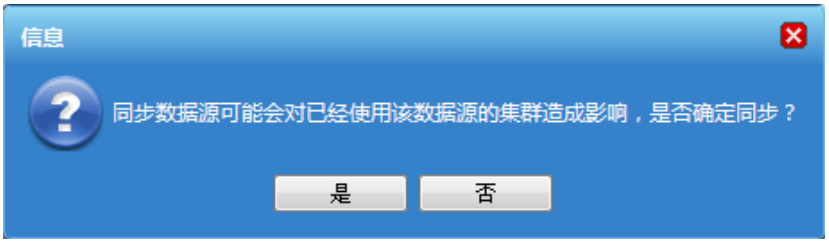


在测试SQL语句输入框输入测试SQL语句，如在Oracle中输入select* from dual,单击“测试”按钮后，会提示测试结果。如果测试成功，则表明配置的数据源信息无误。

四. **数据源同步**

如果数据源已经被集群使用，则修改数据源后要对数据源进行同步，以保证每个集群节点下的apusic域使用的数据源与管理控制台数据源中的一致。

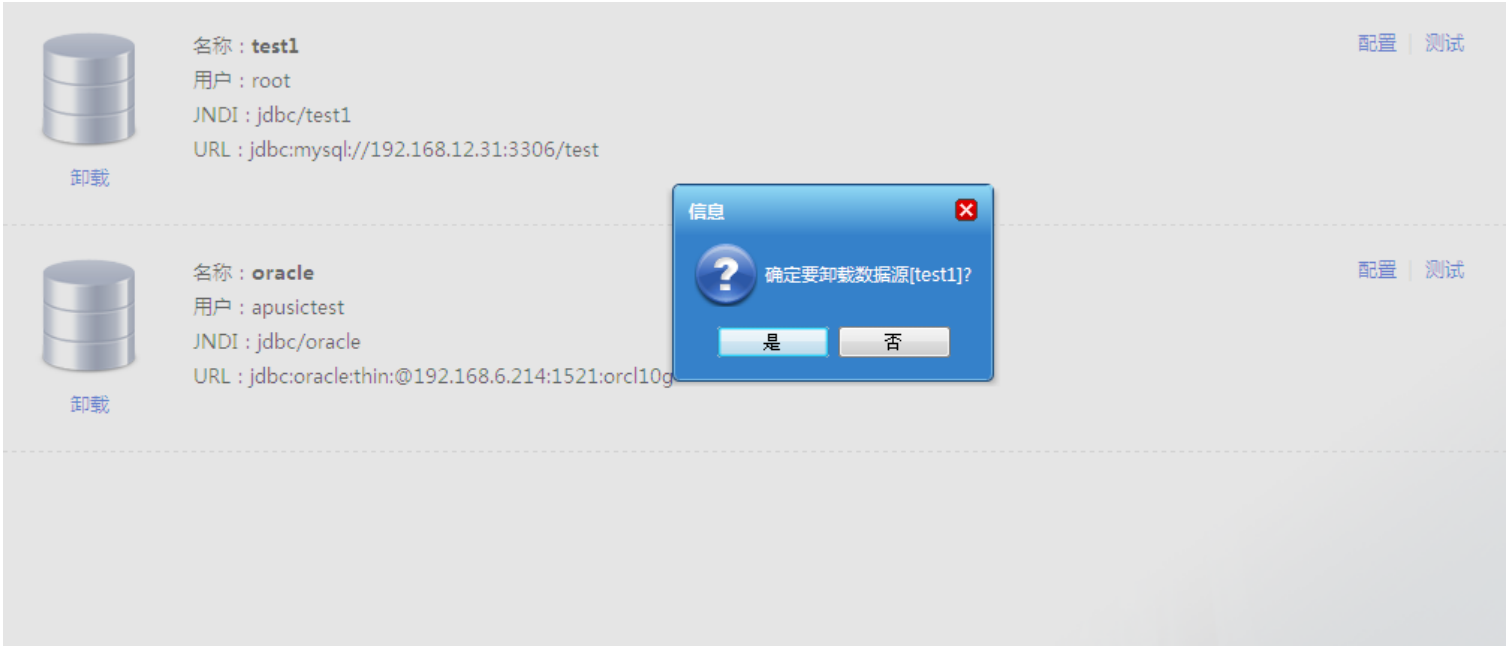
选中要同步的数据源点击同步，在弹出的确认窗口中点击“是”，开始数据源的同步。



五. **卸载数据源**

把数据库源信息从记录删除，已经被使用的数据源不能删除。

选定数据源，点击“卸载”按钮，在提示信息中点击“是”，则完成了数据源的卸载。

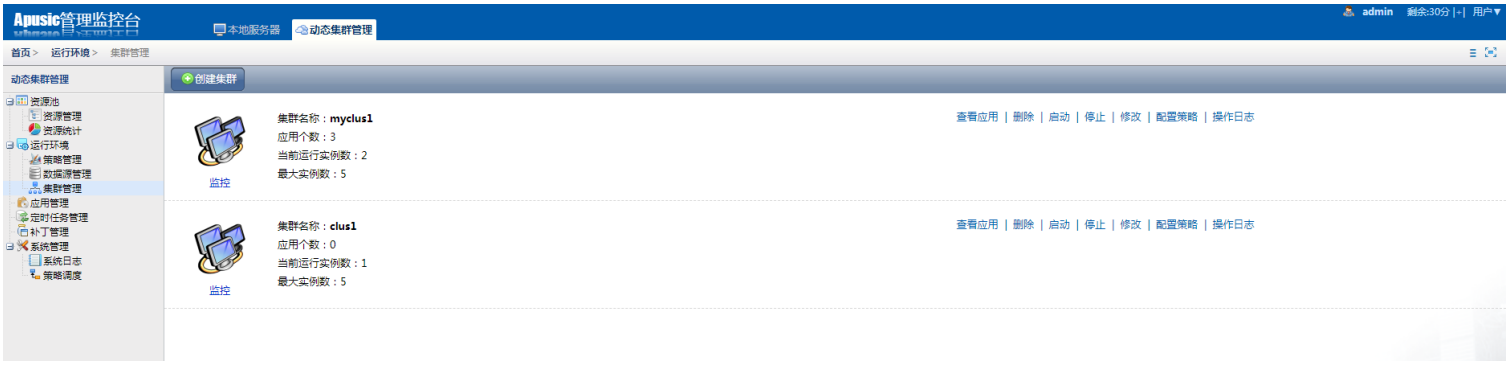


21.3.3.5.3 集群管理

一. **集群列表**

打开动态集群管理功能导航树，点击“集群管理”菜单，进入到集群列表界面下。

用户可以在集群列表下对系统中的所有集群进行管理



二. 新建集群

打开动态集群管理功能导航树，点击“集群管理”菜单下的“创建集群”按钮，进入到“集群创建”配置流程界面，整个配置流程分为：1、配置基本信息，2、选择资源，3、选择属性，4、确认信息，5、进行部署。具体配置过程如下:

第一步：填写要部署的应用的基本信息，具体包括：

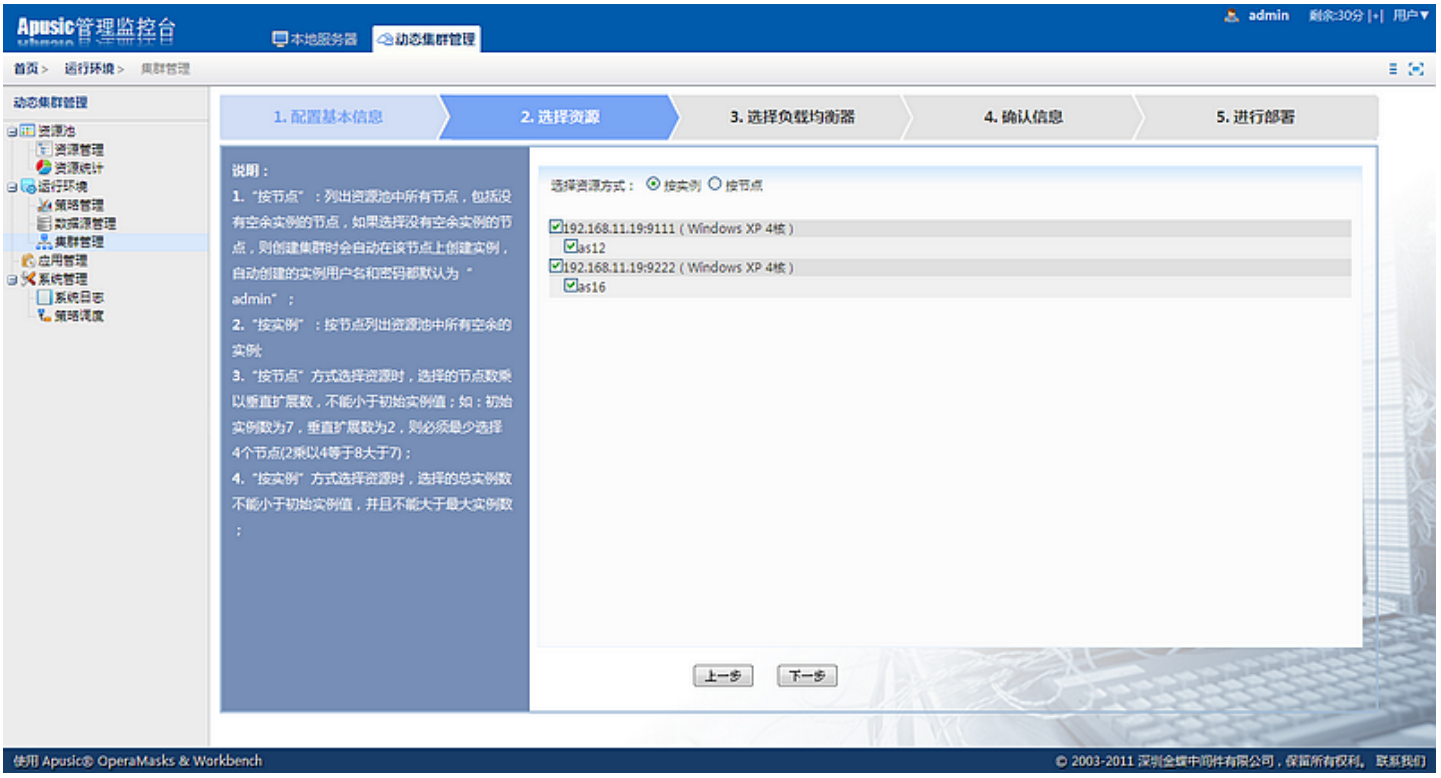
属性名称	用途说明
集群名称	作为整个应用的标识，必须唯一。
初始实例数	集群部署应用后默认运行的最低实例数。
最大实例数	集群部署应用后通过策略能动态调整到的最大实例数。
重要级别	是区分集群重要性的唯一参数，重要性越高，优先级越高，在多个集群都需要增加实例时，重要性高的集群优先增加实例。
限制垂直扩展	勾选则集群支持垂直扩展，不勾选默认不支持垂直扩展。
垂直扩展数	集群在一个节点上最多支持运行的实例数。



点击“下一步”，进入下一个配置界面。

第二步：选择资源

用户可以在资源列表中选择集群所需要的域，部署集群支持域部署和节点部署；选择节点部署时，如果节点下的域数量小于初始实例数，则会自动创建域到节点下。

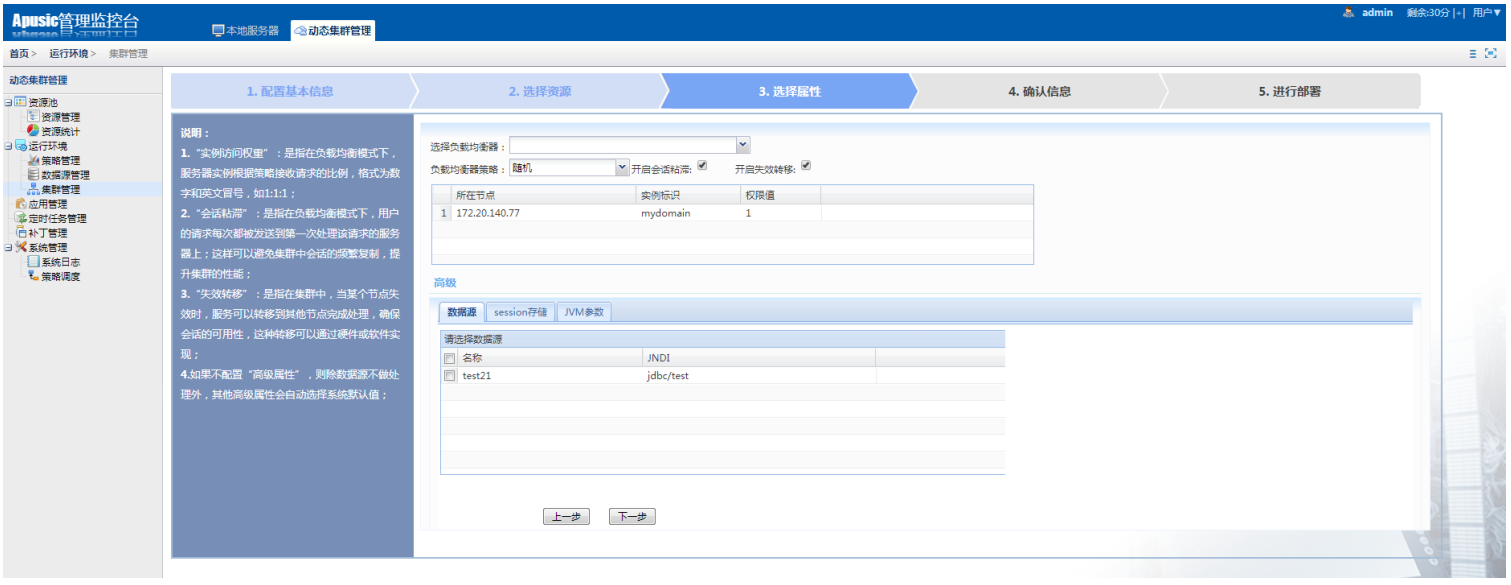


点击“下一步”，进入下一个配置界面

第三步:选择属性

用户可以选择集群所使用的负载均衡器，同时可以配置集群的高级属性。

该步非必填



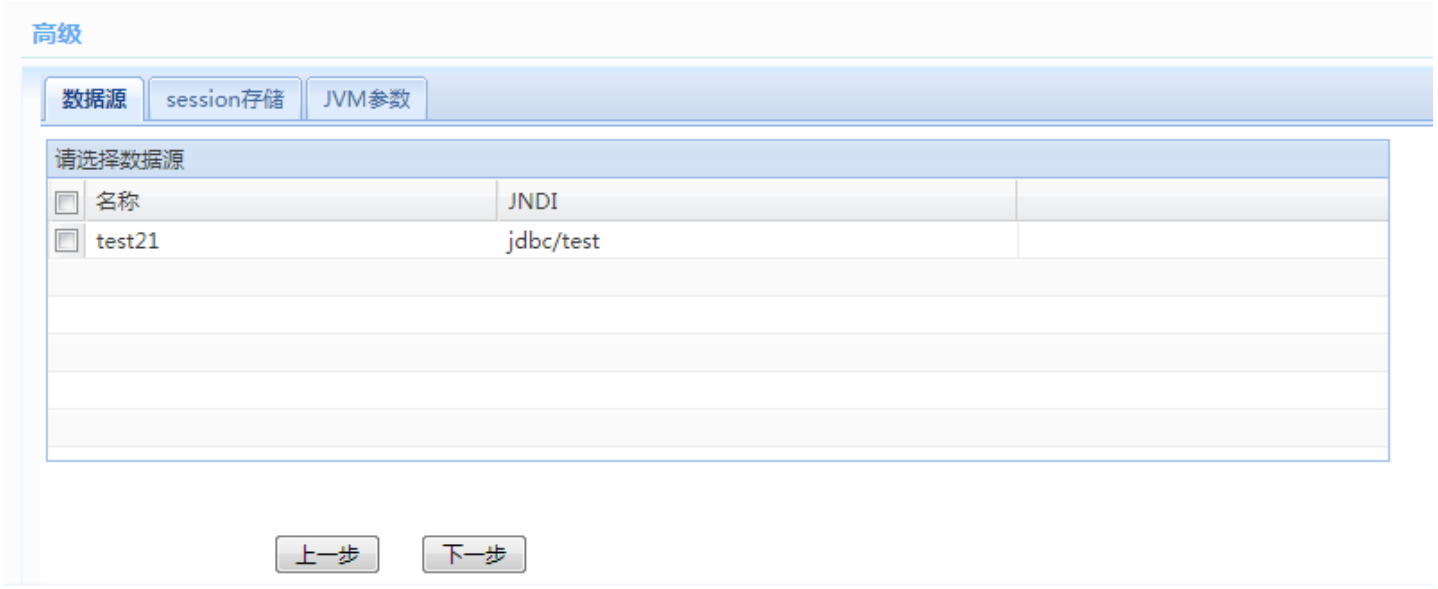
具体配置如下：

属性名称	用途说明

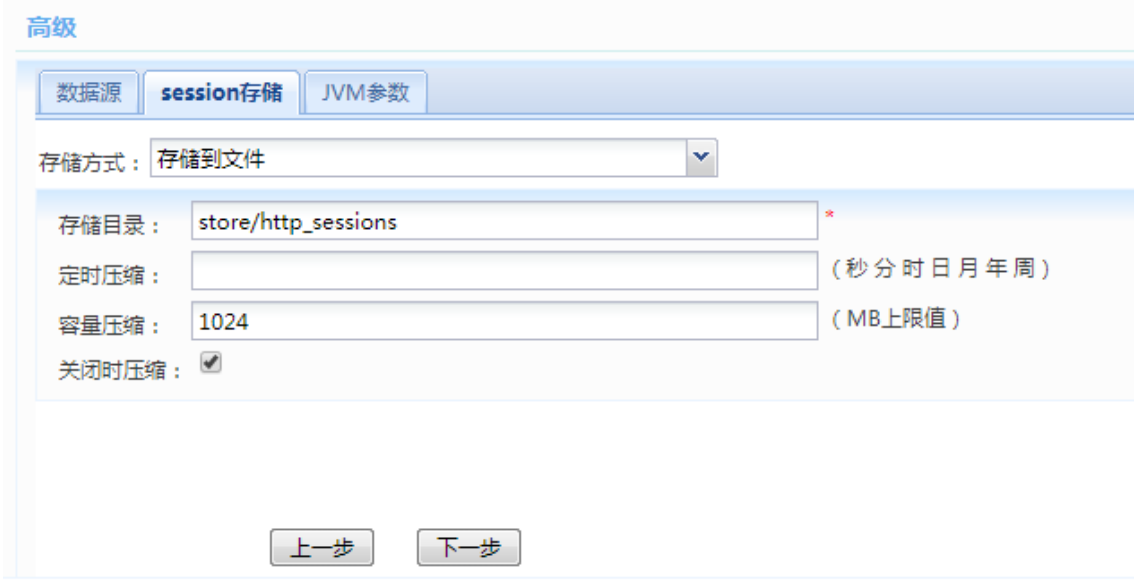
负载均衡器策略	指集群负载所使用的策略，包括随机、按网络流量、按请求数量、按业务耗时。
开启会话粘滞	是指在负载均衡模式下，用户的请求每次都被发送到第一次处理该请求的服务器上；这样可以避免集群中会话的频繁复制，提升集群的性能。
开启失效转移	是指在集群中，当某个节点失效时，服务可以转移到其他节点完成处理，确保会话的可用性，这种转移可以通过硬件或软件实现。

同时，还可以设置集群的高级属性，包括数据源、session存储和JVM参数：

数据源：勾选数据源后将数据源部署到集群中；



session存储：用户可以选择集群会话存储方式，包括文件存储、SQL存储和cache存储；



JVM参数：可以设置参数，包括内存参数、GC参数、其他参数。例如在测试应用程序test10.zip时，添加“其他参数”testapusic，格式为–Dtestapusic=1234；访问应用程序test10/testjvm.jsp时则会显示参数数据!

	注意
	注：如果不配置“高级属性”，则除数据源不做处理外，其他高级属性会自动选择系统默认值

点击“下一步”，进入一个配置界面

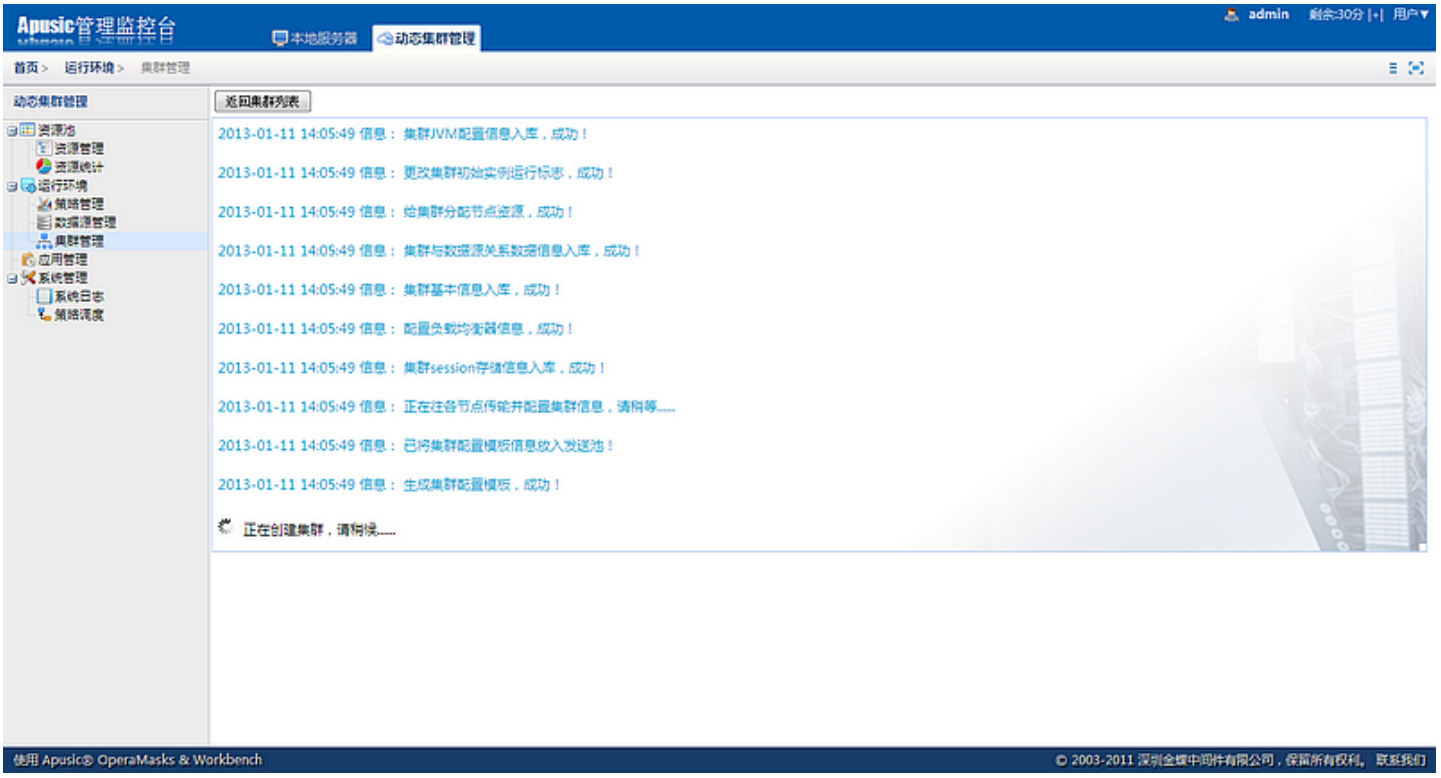
第四步：确认信息



列出前面配置信息，供用户确认信息是否有误。如果存在修改的配置，可以点击“上一步”回到对应的配置界面进行修改。

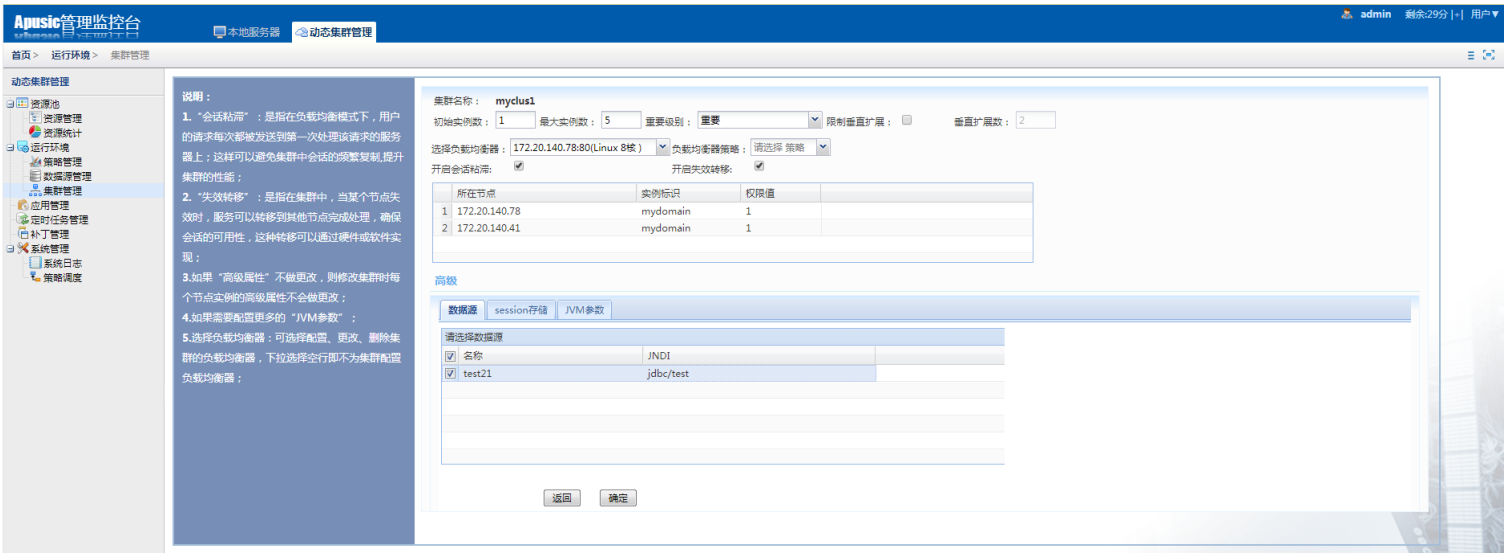
点击“完成”即开始创建集群，并在界面列出详细的部署日志信息直至部署成功或者失败，完成集群部署。

部署过程日志界面如下，通过该界面，可以了解整个应用部署的进展情况，部署提示完成后，可以点击“返回应用列表”按钮，进入应用列表界面。



三. 集群修改

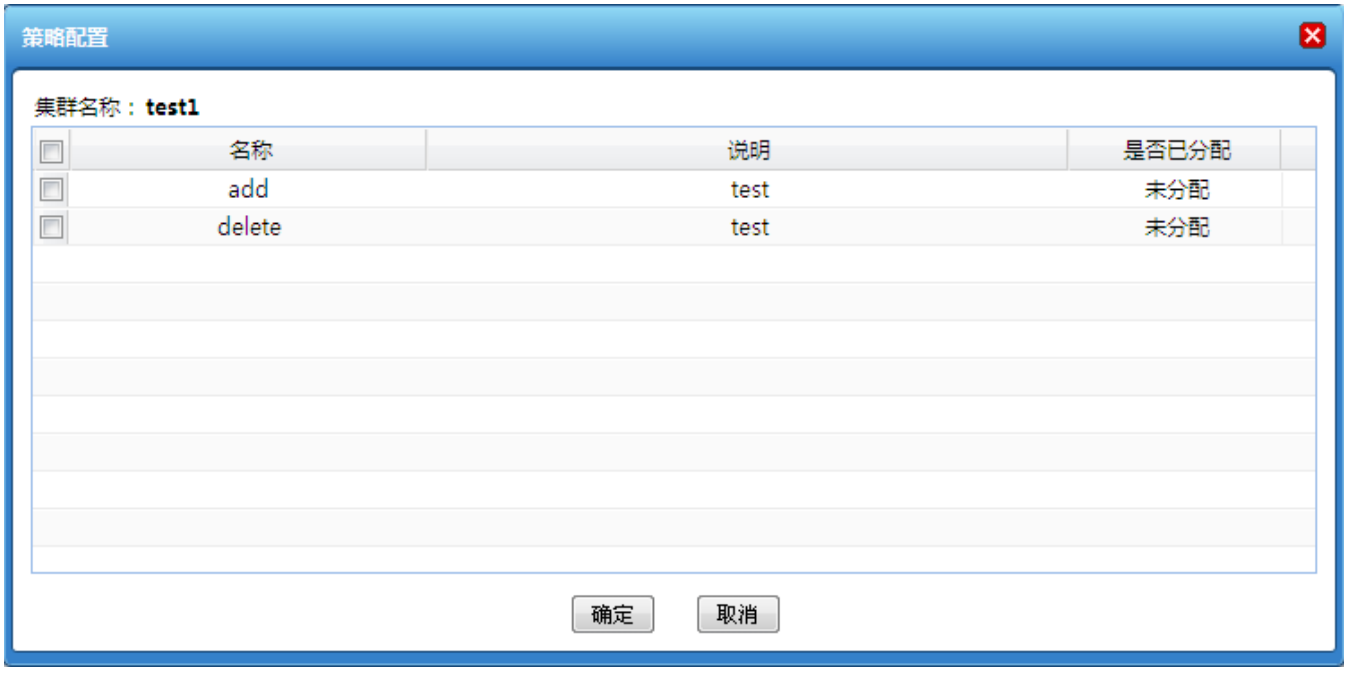
打开动态集群管理功能导航树，进入到“集群管理”界面。选中一个集群，点击“修改”按钮进入到集群修改界面：



在此界面可以对集群配置信息进行修改，配置项请参开集群创建界面。

四. 配置策略

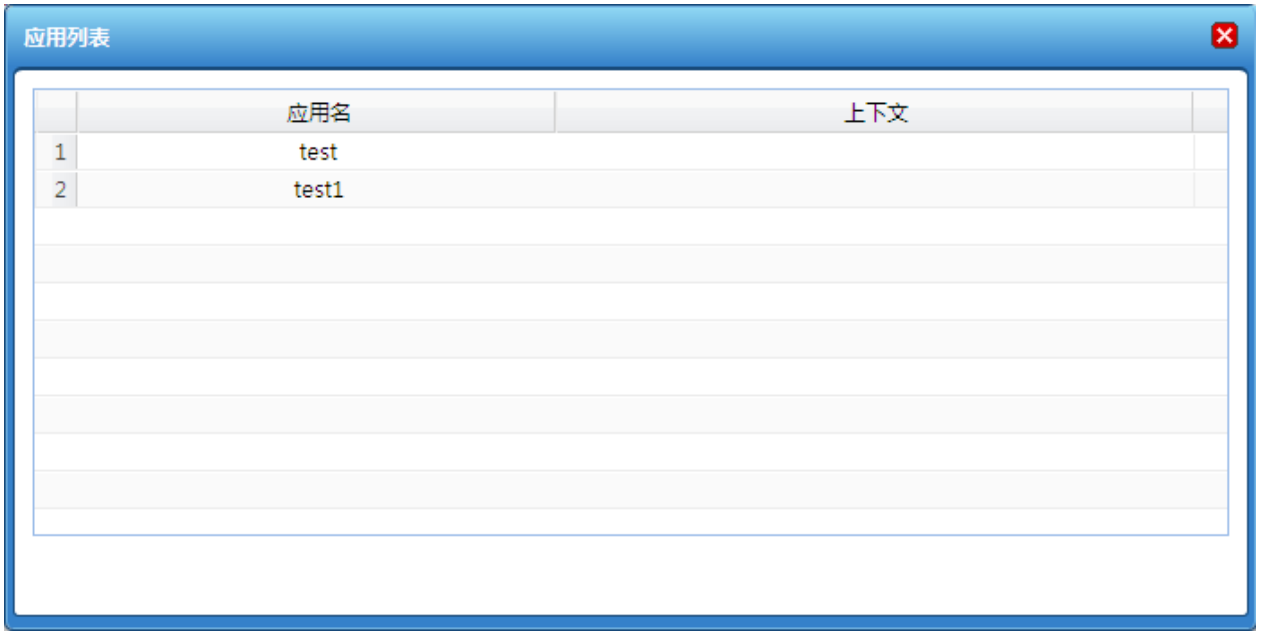
打开动态集群管理功能导航树，进入到“集群管理”界面。选中一个集群，点击“配置策略”按钮进入到策略分配界面：



在此界面可以对集群进行策略配置，配置完成后点击“确定”完成。

五. **查看应用**

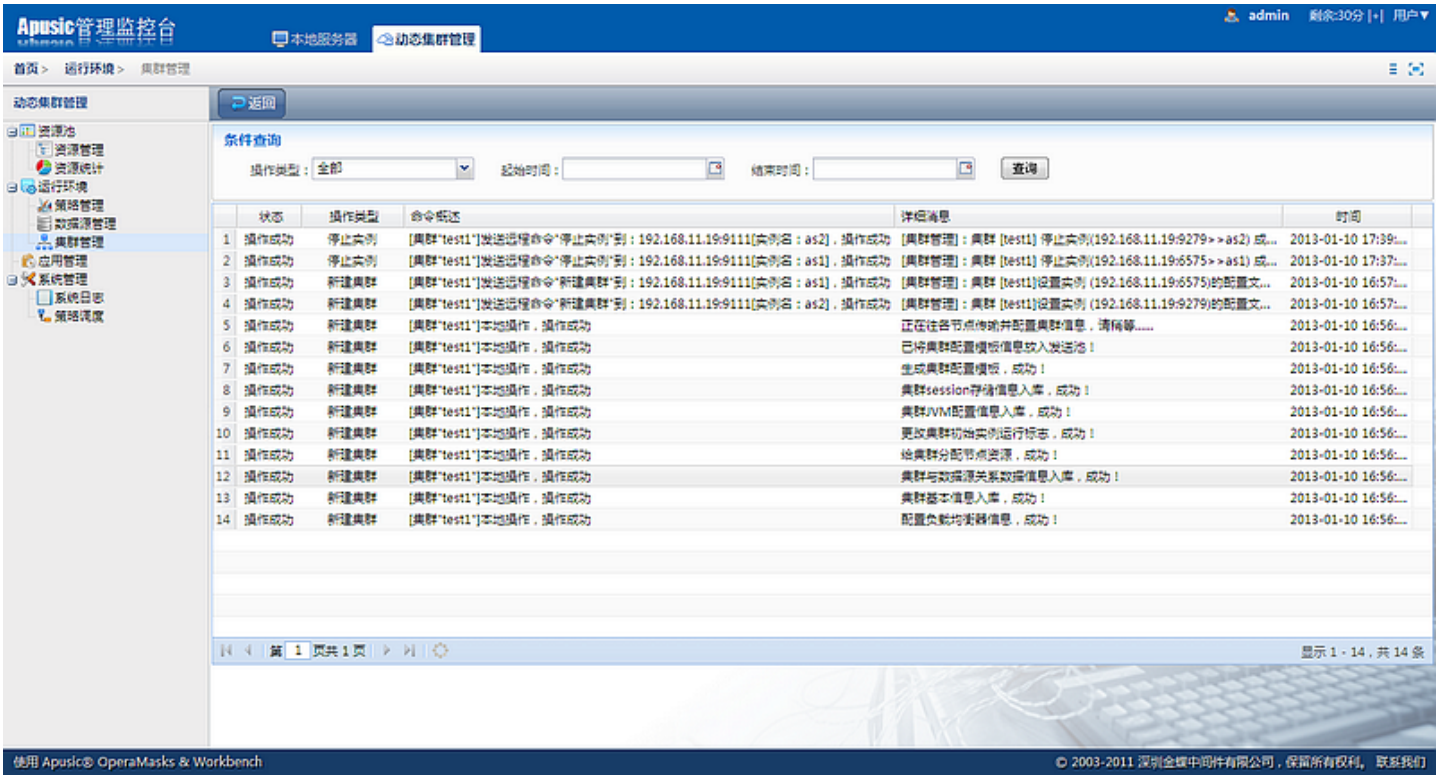
打开动态集群管理功能导航树，进入到“集群管理”界面。选中一个集群，点击“查看应用”按钮进入到应用列表界面：



在应用列表界面可以查看集群上部署的应用以及应用访问时的上下文。

六. **集群操作日志**

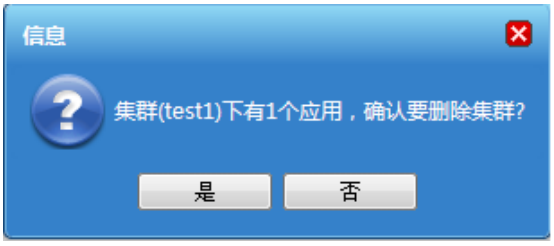
打开动态集群管理功能导航树，进入到“集群管理”界面。选中一个集群，点击“操作日志”按钮进入到集群操作日志查看界面：



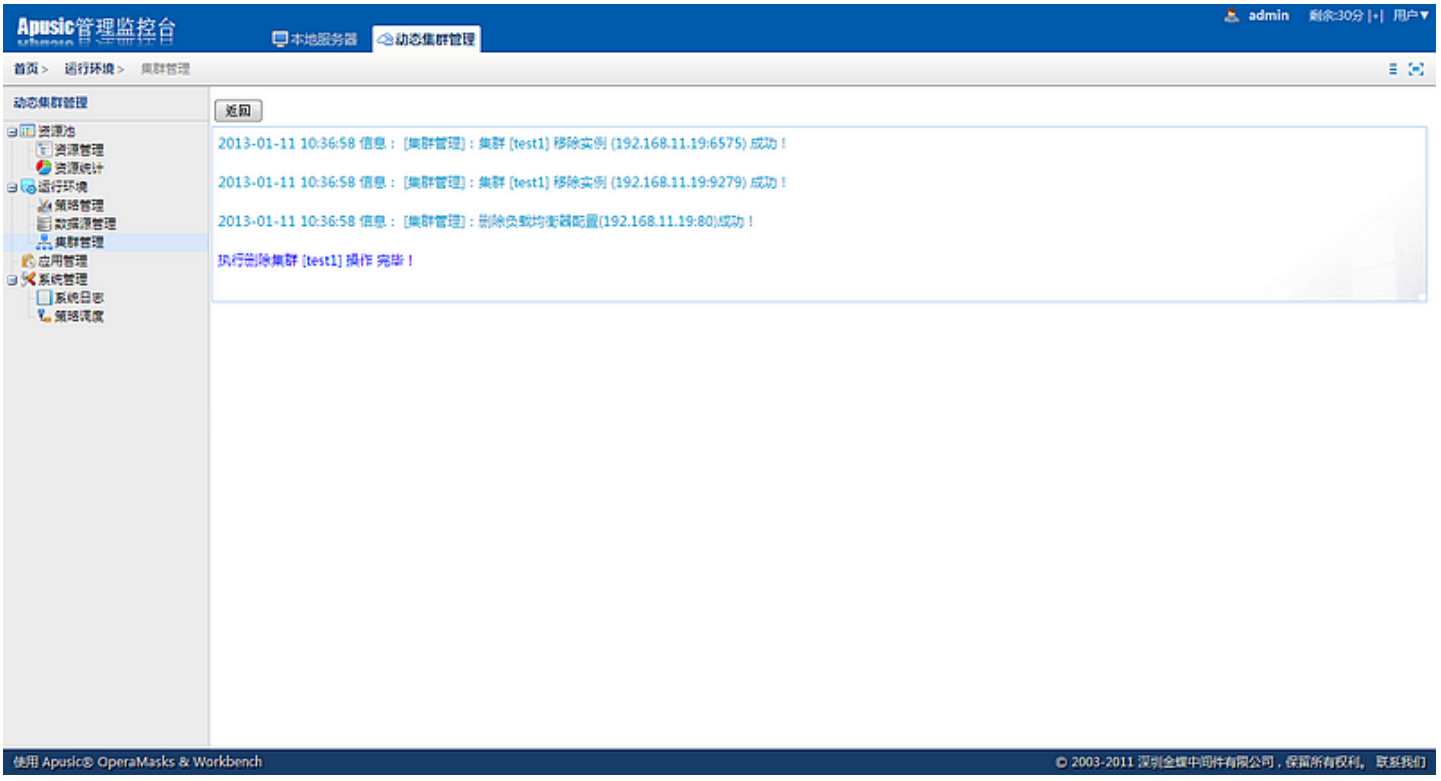
在操作日志界面可以查看集群操作的所有日志，并可以根据操作分类和时间进行查询。

七. **删除集群**

打开动态集群管理功能导航树，进入到"集群管理"界面。选中一个集群，点击"删除"按钮进行集群删除操作，在弹出的提示窗口中点击确定：

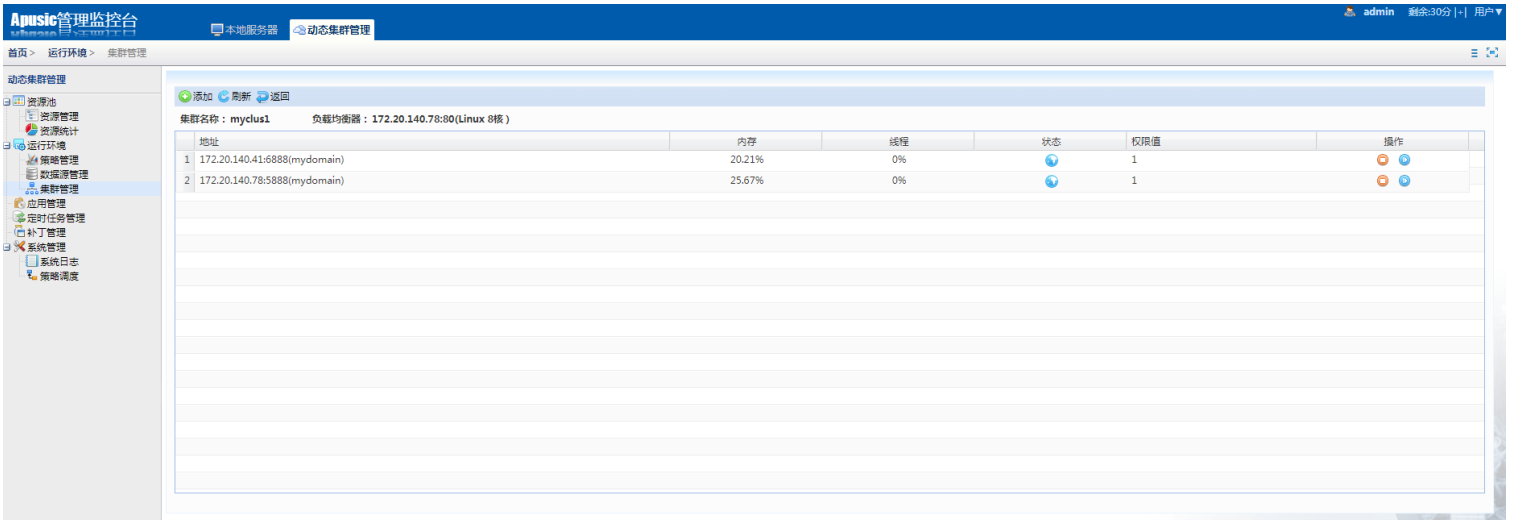


集群删除完毕后，查看日志确定是否成功删除。点击完成，返回集群列表界面：



八. 集群监控

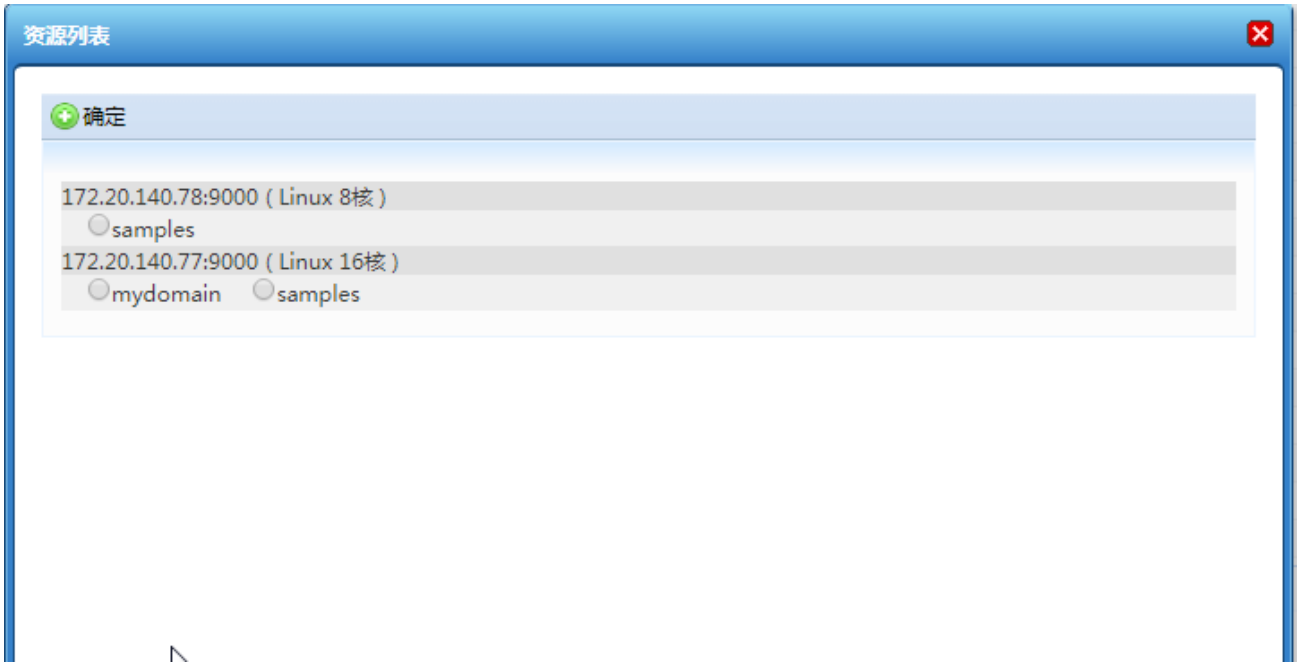
打开动态集群管理功能导航树，进入到“集群管理”界面。选中一个集群，点击“监控”按钮进入到集群监控界面下：



在集群监控界面下可以查看集群下每一个节点的状态，并可以查看线程和内存的使用比率。

(1). 添加节点

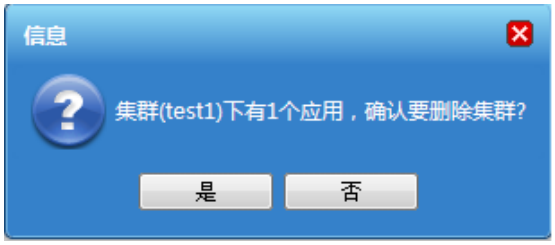
点击“添加”按钮，进入到选择实例列表：



选择一个空闲实例，点击确定，完成实例的添加。

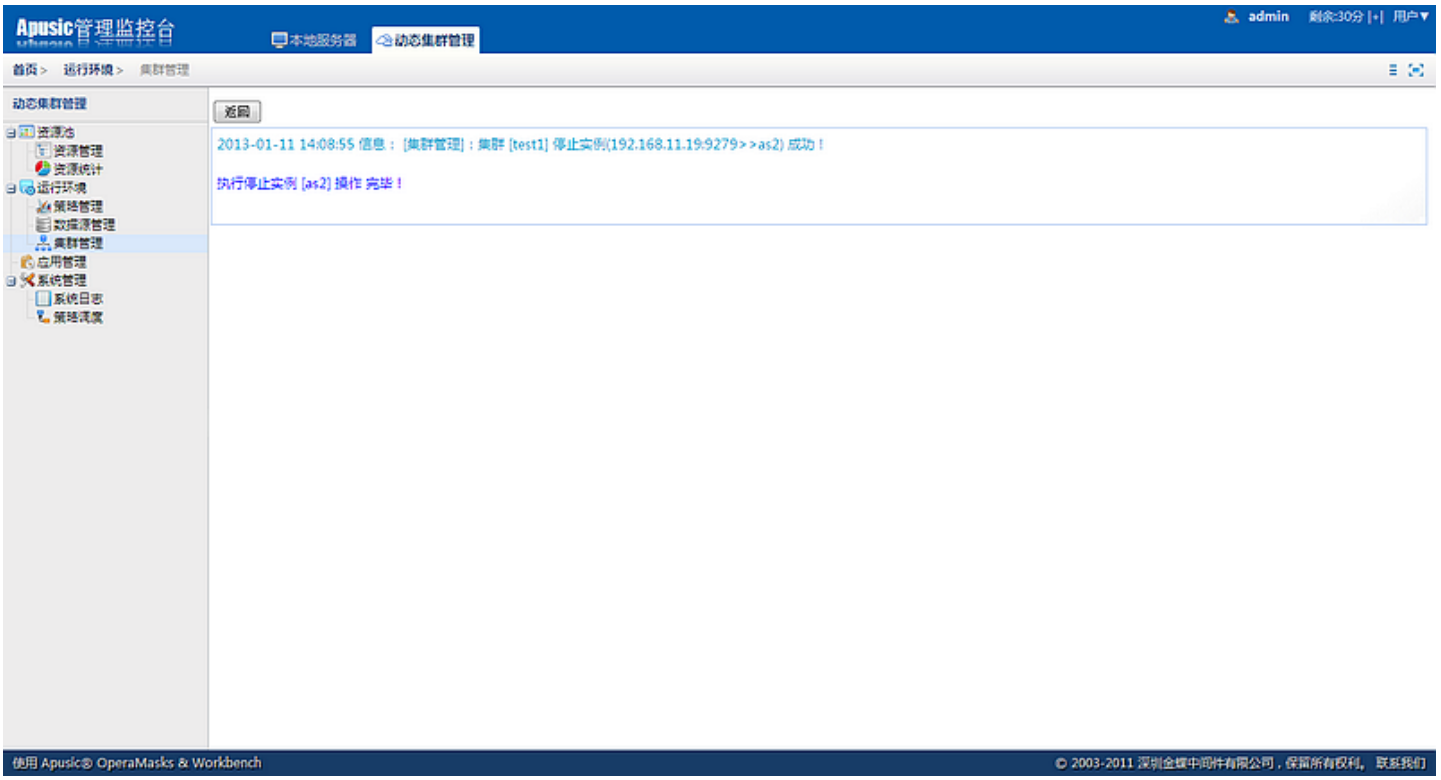
(2). **移除节点**

选择一个停止的apusic实例，点击删除，最后返回移除是否成功的信息。



(3). **实例操作**

选择一个apusic实例，在操作栏下可以通过启动，停止，重启按钮对实例进行相应的操作，操作成功后返回是否成功的信息



21.3.3.6 应用管理

21.3.3.6.1 部署应用

打开动态集群管理功能导航树，点击“应用管理”菜单下的“部署应用”菜单，进入到“应用列表”界面；点击“部署应用”按钮则进入整个配置流程分为：1、配置基本信息，2、选择集群，3、确认信息，4、进行部署。具体配置过程如下：

第一步：填写要部署的应用的基本信息，具体包括：

部署类型：分为三种，一种是部署服务器上的应用，一种是部署本地应用，还有一种是部署共享文件中的应用

文件上传路径：上传要部署的应用的war包或者ear包；

应用基础上下文：如应用的web模块配置了context-root，那么应用所有web模块的context-root会以应用基础上下文为前缀，访问路径变为http://hostname/base-context/context-root/。具体请参考帮助文档；

部署方式：立即部署并启动



选择立即部署并启动，表示部署的应用在部署完成后立即启动

部署方式：稍后部署并启动

启动时间：设置多少个小时或多少分钟后部署并启动应用



选择稍后部署并启动，表示在设置的时间后部署并启动应用。

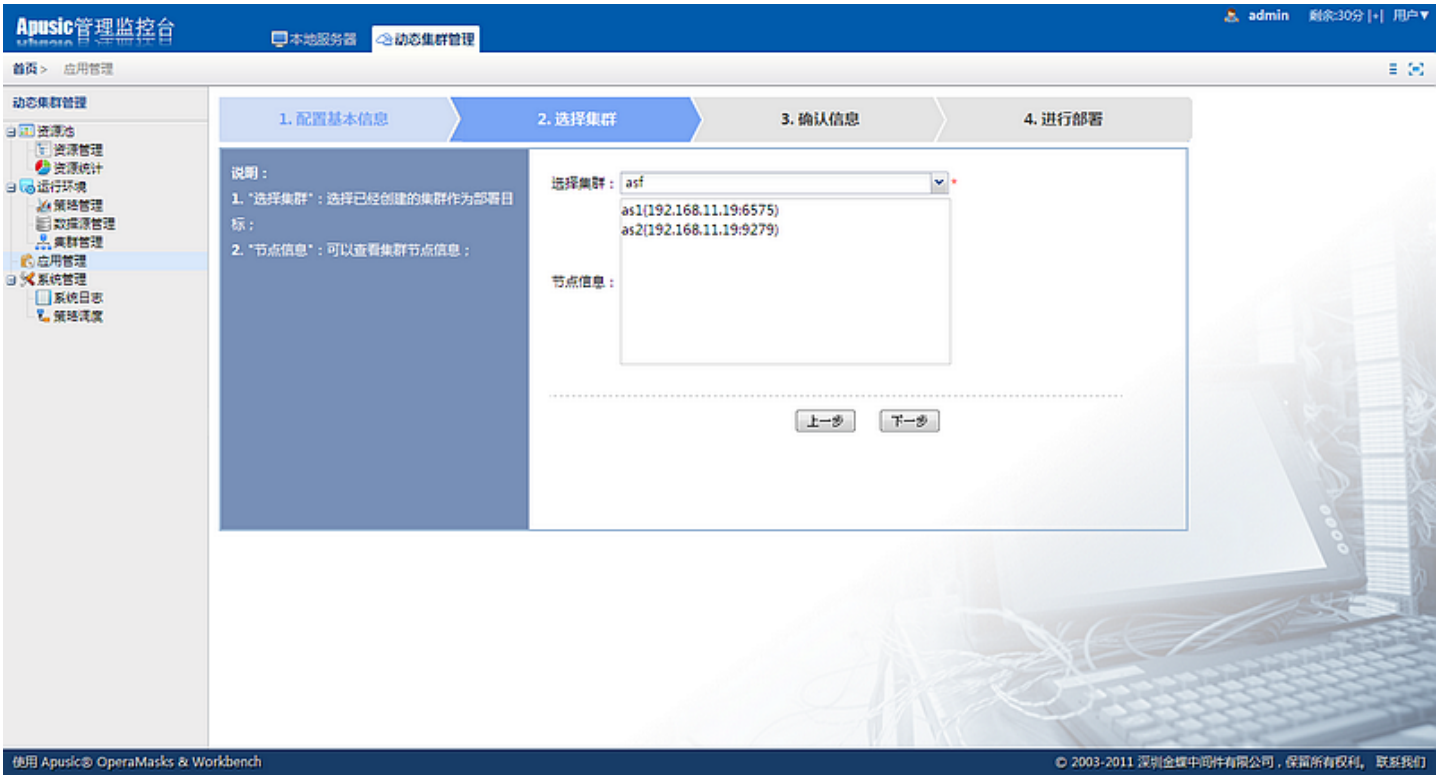
部署方式：仅部署



选择仅部署，表示只部署应用，应用部署后，需在应用列表中点击启动，才会启动应用。

点击"下一步"，进入下一个配置界面。

第二步：选择集群



用户可以在此界面为应用选择所要部署的集群，选择集群后可以在节点信息中查看集群的每一个节点的具体信息。

点击"下一步"，进入下一个配置界面

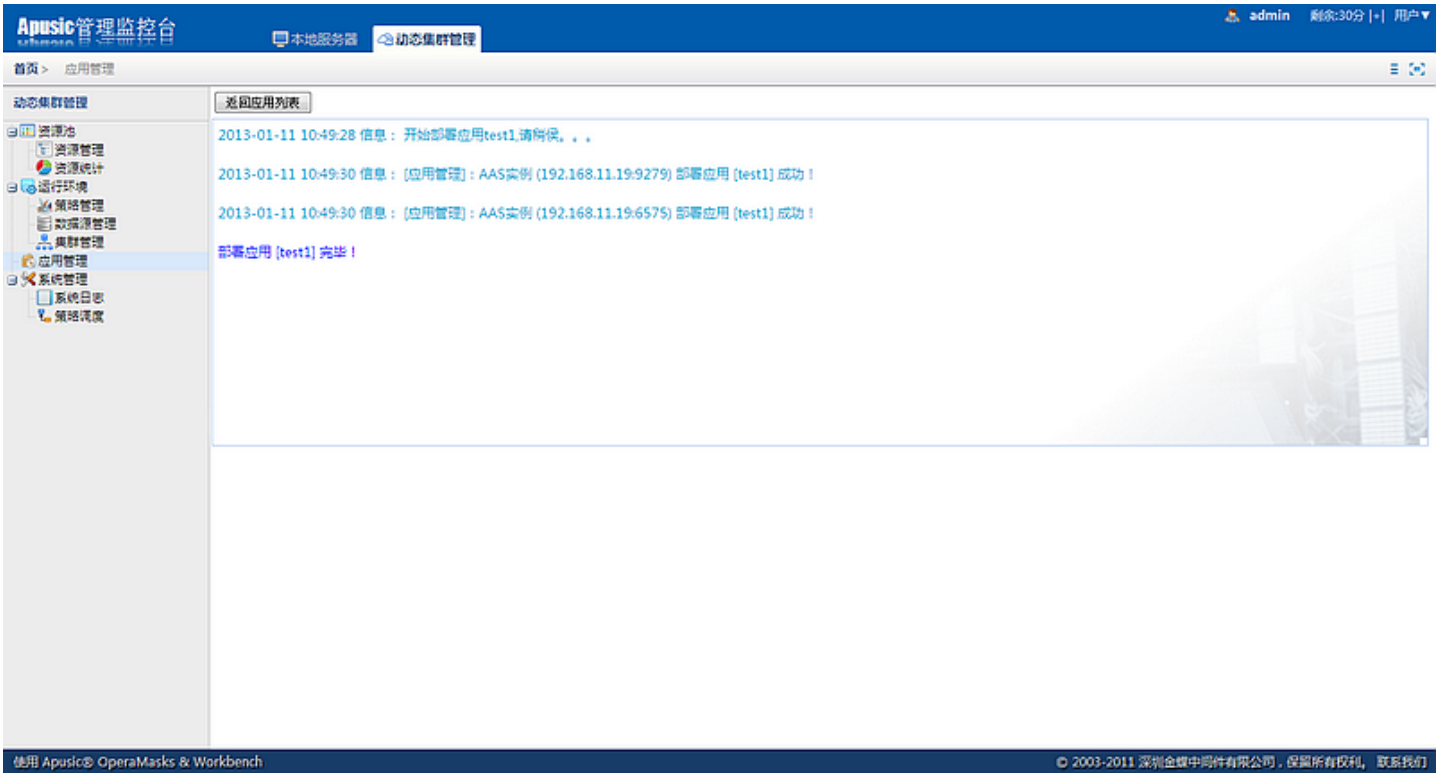
第三步:确认信息



列出前面配置信息，供用户确认信息是否有误。如果存在修改的配置，可以点击"上一步"回到对应的配置界面进行修改。

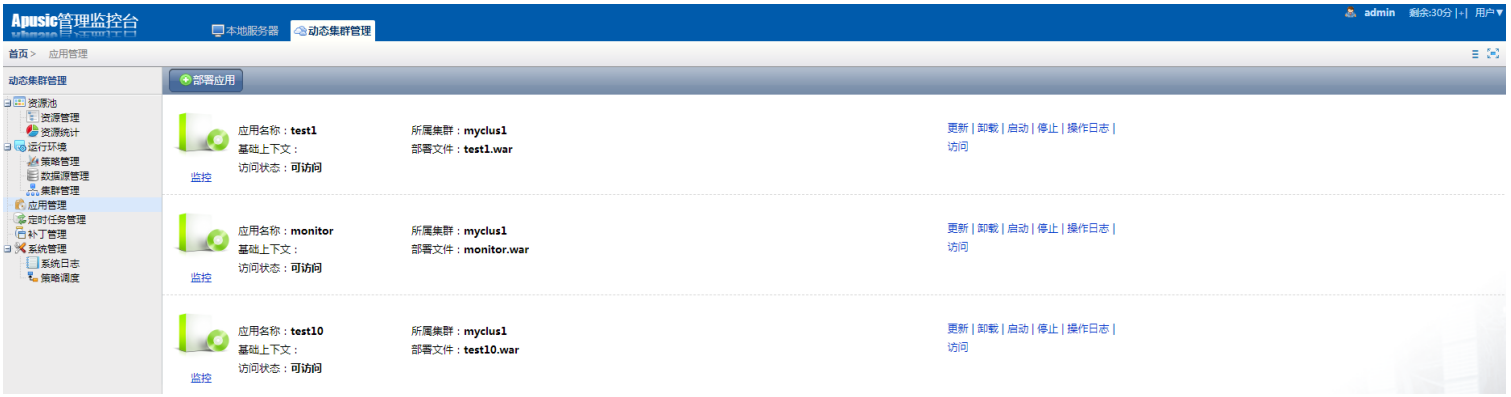
点击"确定"即可对上传的应用进行部署，并在界面列出详细的部署日志信息直至部署成功或者失败，完成应用集群部署。

部署过程日志界面如下，通过该界面，可以了解整个应用部署的进展情况，部署提示完成后，可以点击"返回应用列表"按钮，进入应用列表界面。



21.3.3.6.2 应用列表

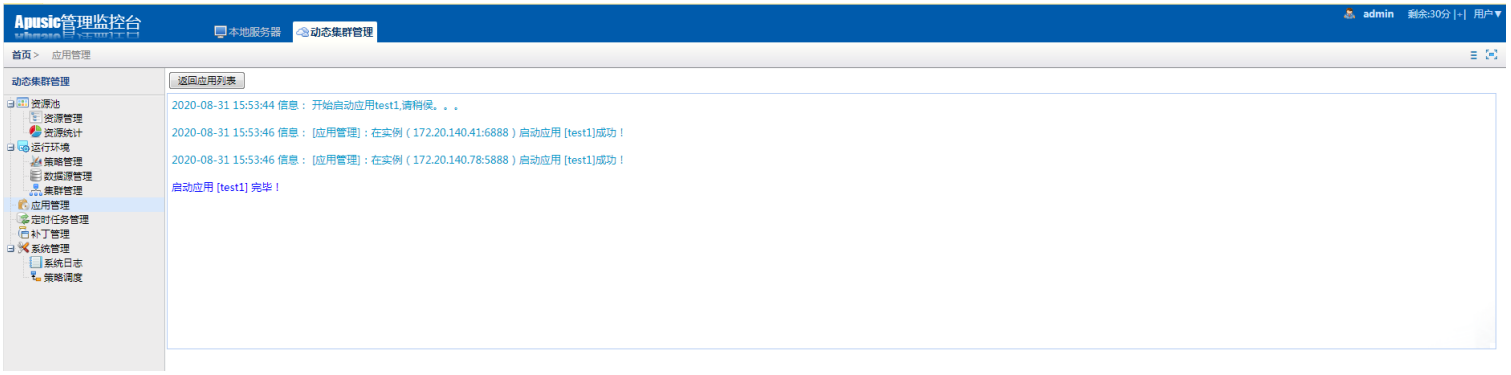
打开动态集群管理功能导航树，点击“应用管理”进入到应用列表操作界面。



21.3.3.6.3 应用操作

对成功部署的应用进行启动，停止和卸载的功能。

以应用启动操作为例，点击应用启动后，在弹出的提示信息中点击确定，进入到启动应用日志显示界面，如下：

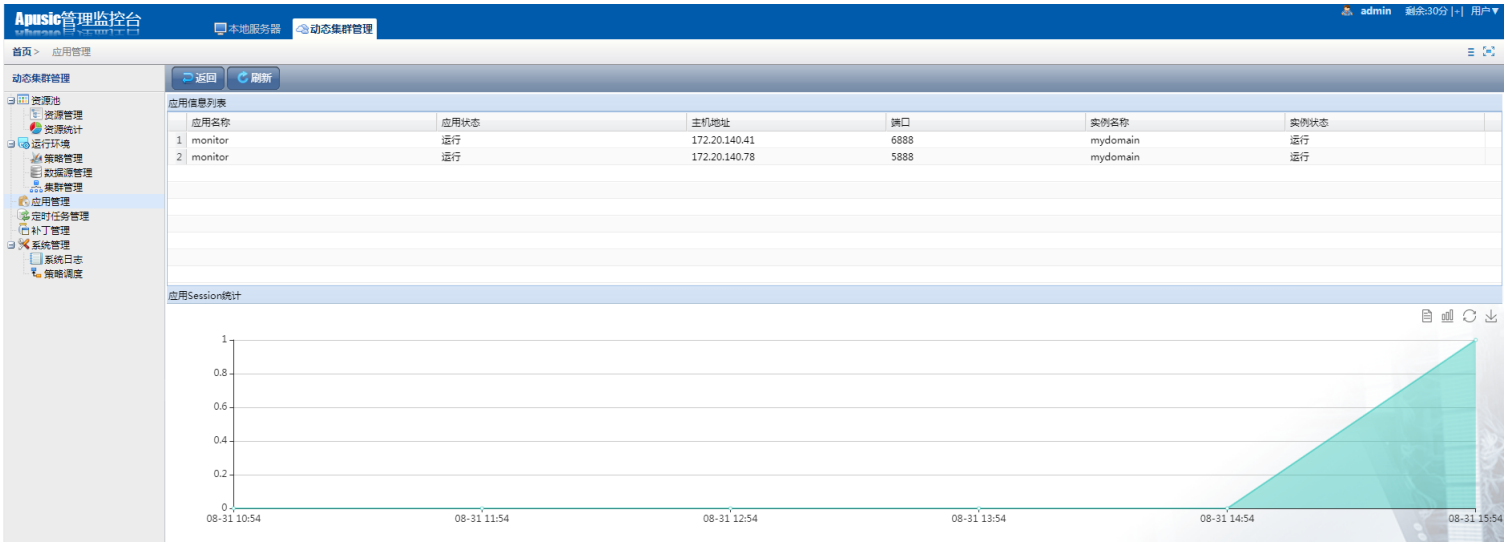


点击"返回应用列表"，返回到“应用列表”页面。

一. 应用监控

对部署该应用的所有资源节点进行监控的功能，系统定时刷新该窗口，查看各个节点实例以及实例上运行的应用的运行状态。

点击“应用列表”界面的“监控”按钮，打开应用监控窗口，如下：



二. 应用访问

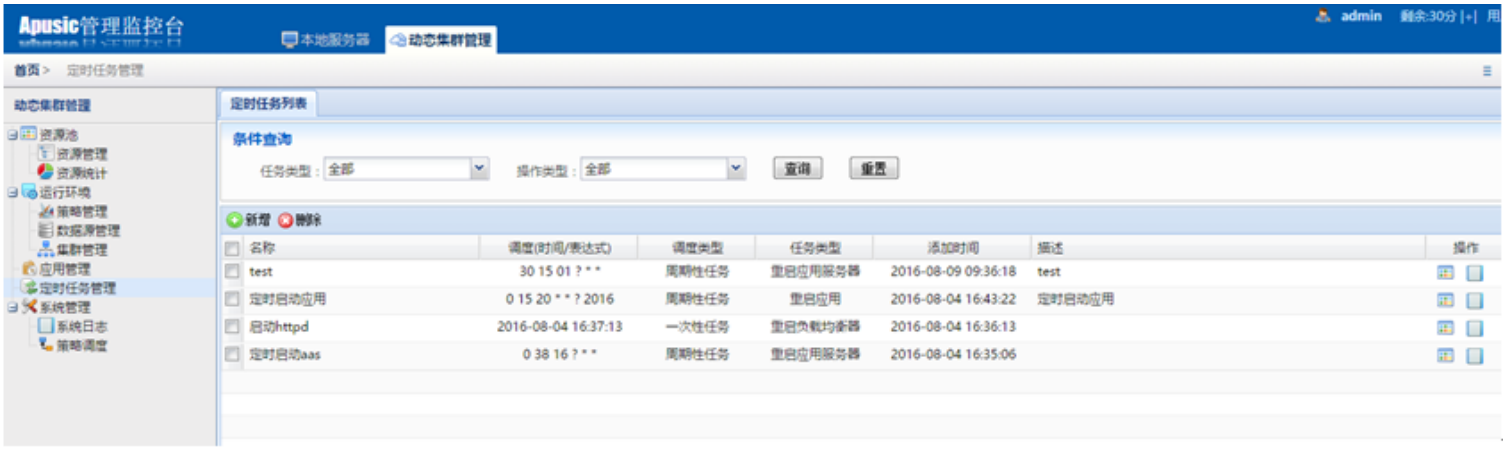
点击“应用列表”界面的“访问”按钮即可对应用进行访问。

注：访问应用程序需要先在集群配置负载均衡器

21.3.3.7 定时任务管理

定时任务管理是用于管理定时启动负载均衡器、应用服务器及应用的界面。定时任务分为：一次性任务和周期性任务，任务的类型可分为：重启负载均衡器、重启应用服务器、重启应用。

定时任务列表，定时任务列表显示创建的任务：



21.3.3.7.1 新增任务

进入动态集群管理系统，点击“定时任务管理”，再点击“新增”按钮，进入添加定时任务窗口。新增任务可以创建一次性任务和周期性任务。

一. 添加一次性任务

在添加定时任务时，如果选择的调度策略为“一次性任务”，则创建的就是一次性定时任务。



任务名称：填写任务的名称。

任务描述：填写对任务的描述。

调度策略：选择“一次性任务”或“周期性任务”,一次性任务请在此选择“一次性任务”。

调度时间：填写数字，表示几小时或几分钟后执行此任务。

任务类型：可以选择“重启负载均衡器”、“重启应用服务器”、“重启应用”。

新增或删除对应选择类型的服务：点击“新增”，选择相应类型的服务，点击“保存”确定。在列表中，选择相应服务，点击“删除”，可以删除服务。需要特别说明的是：这里新增时可选择的服器器、负载均衡器或应用，只会显示已经添加到集群中的服务器、负载均衡器及已经部署到集群中的应用。

二. 添加周期性任务

在添加定时任务时，如果选择的调度策略为“周期性任务”，则创建的就是周期性定时任务。



任务名称：填写任务的名称。

任务描述：填写对任务的描述。

调度策略：选择“一次性任务”或“周期性任务”,周期性任务请在此选择“周期性任务”。

cron格式：请参考“参考例子”标准填写。

任务类型：可以选择“重启负载均衡器”、“重启应用服务器”、“重启应用”。

新增或删除对应选择类型的服务：点击“新增”，选择相应类型的服务，点击“保存”确定。在列表中，选择相应服务，点击“删除”，可以删除服务。需要特别说明的是：这里新增时可选择的服器器、负载均衡器或应用，只会显示已经添加到集群中的服务器、负载均衡器及已经部署到集群中的应用。

21.3.3.7.2 删除任务

在定时启动任务列表中，选择某个要删除的作务，点击“删除”按钮，即可删除任务。



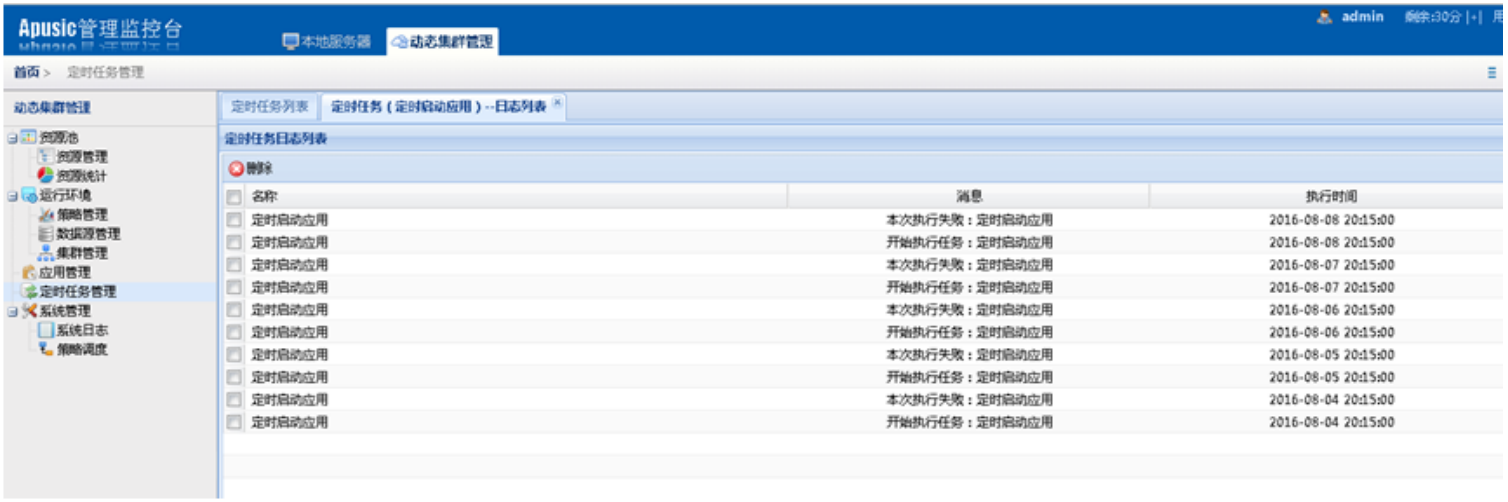
21.3.3.7.3 查看详细

在定时任务列表界面，选择任务后面的操作栏中的“查看详细”图标按键，可以看到任务的详细信息。



21.3.3.7.4 查看执行日志

在定时任务列表界面，选择任务后面的操作栏中的“查看执行日志”图标按键，可以看到任务的执行日志信息。



21.3.3.8 补丁管理

补丁管理用于管理应用服务器问题修复补丁包及应用服务器升级包。进入动态管理系统，点击“补丁管理”进入补丁管理界面。

21.3.3.8.1 新增

在补丁管理列表界面，点击“新增”按钮，进入“上传补丁”界面。



名称：请填写补丁的名称。

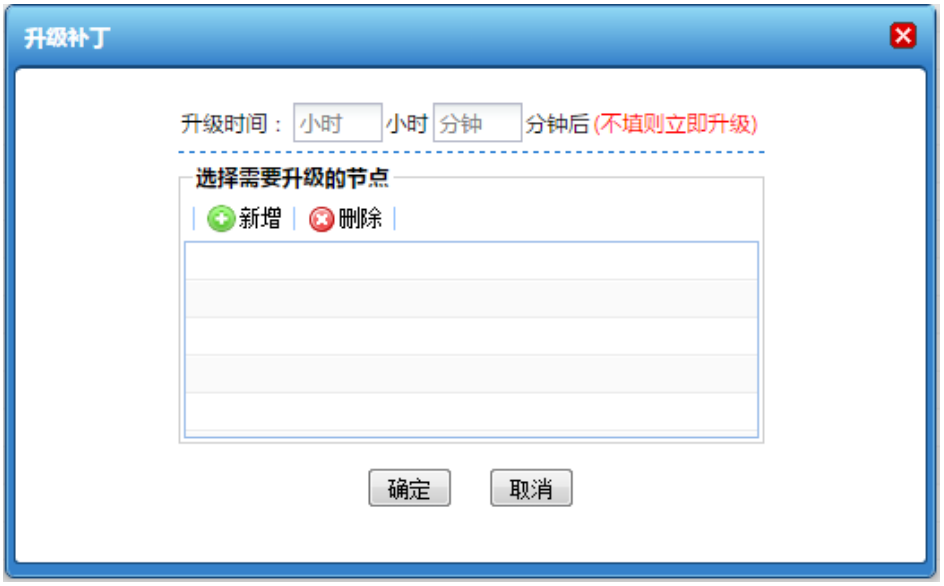
文件：请选择要更新jar包。

描述：请填写补丁的说明。

点击“确定”按钮确认，点击“取消”按钮取消本次操作。

21.3.3.8.2 升级

在补丁相对应的操作栏，点击“升级”按钮，选择要升级的节点。



升级时间：请填写预计升级的时间，填写后，将产生一个定时任务，到达设置时间后会进行升级。不填则立即升级。

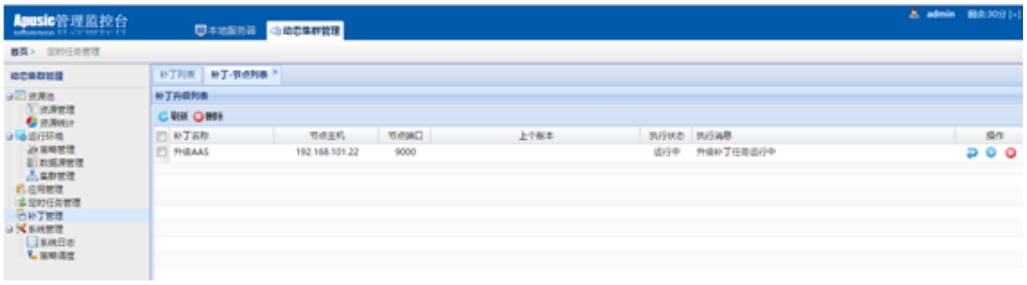
选择需要升级的节点：在“升级”补丁界面，点击“新增”按钮，进入节点选择界面，在节点选择界面，勾选需要升级的节点，则该节点下所有已加入到集群中的domains都将升级。



点击“确认”进行升级。

21.3.3.8.3 查看

在补丁相对应的操作栏，点击“查看”按钮，可以看到升级的节点列表。



恢复上一个版本：点击“恢复上一个版本”按钮，将取消本次升级操作，使系统恢复到上一个版本。

再次升级：点击“再次升级”按钮，将再次升级。

删除：点击“删除”按钮，将删除此补丁记录，但已经上传到各节点的应用服务器上的补丁包不会删除。

21.3.3.8.4 删除

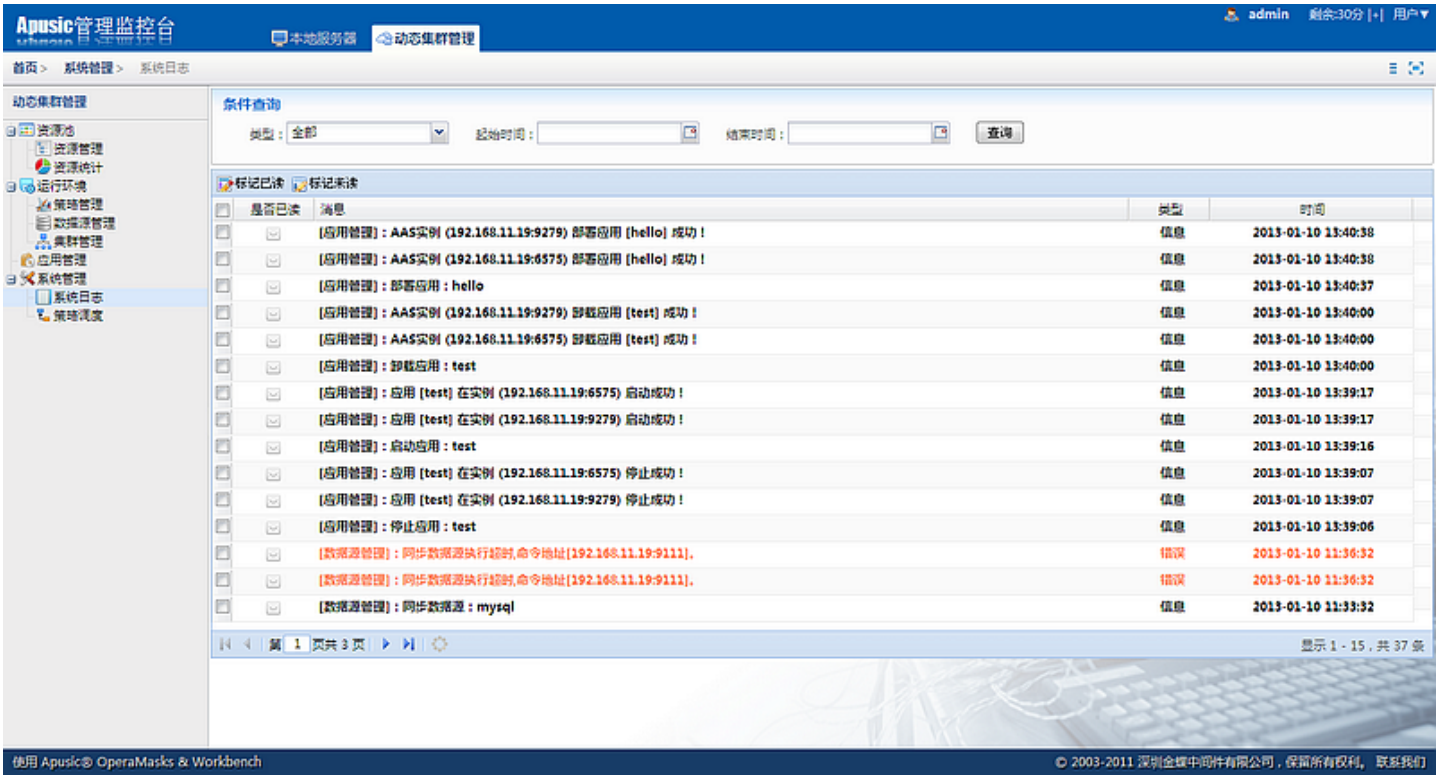
在补丁列表界面，选择一条或多条补丁记录，点击“删除”按钮，将删除这些补丁记录，但已经上传到各节点下的应用服务器上的补丁包不会被删除。

21.3.3.9 系统管理

21.3.3.9.1 系统日志

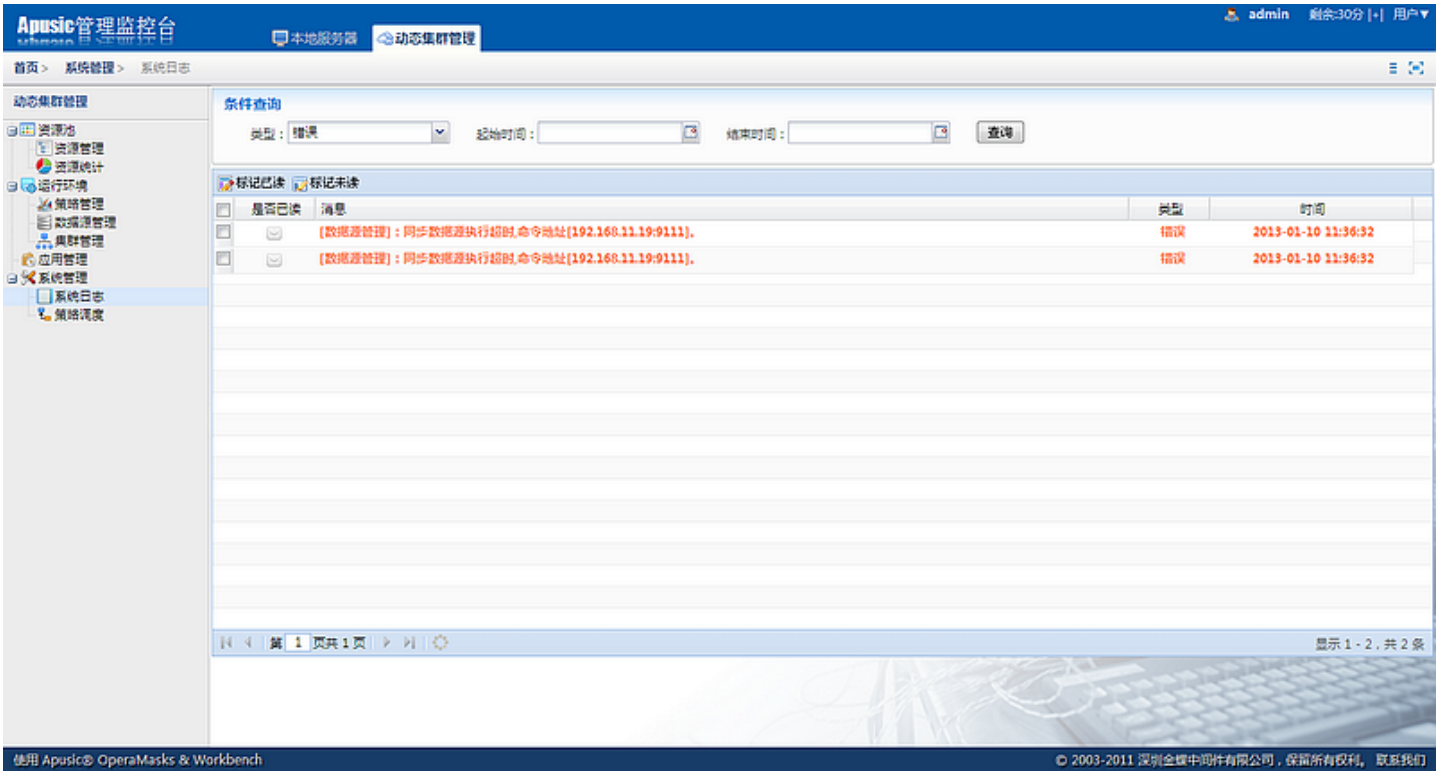
展示了系统的一些重要的日志信息。日志类型分为：警告，错误，信息三种，未读的日志用粗体显示，以区分已读日志。

打开动态集群管理功能导航树，点击“系统通知”菜单”，进入“系统通知”界面：



一. 系统日志查询

通过在条件查询面板输入查询条件，查询条件包括：类型，起始时间，结束时间。输入查询条件后，点击查询面板的"查询"按钮，可以对日志进行特定条件下的查询，下面是查询日志类型为：警告，所有时间段的日志：



二. 标记已读

在系统日志列表中选中未读的记录，点击"系统通知"界面的"标记已读"按钮，即可把未读的日志标记为已读。已读的日志，在首页中"告警日志"统计中将不会包含这条信息。

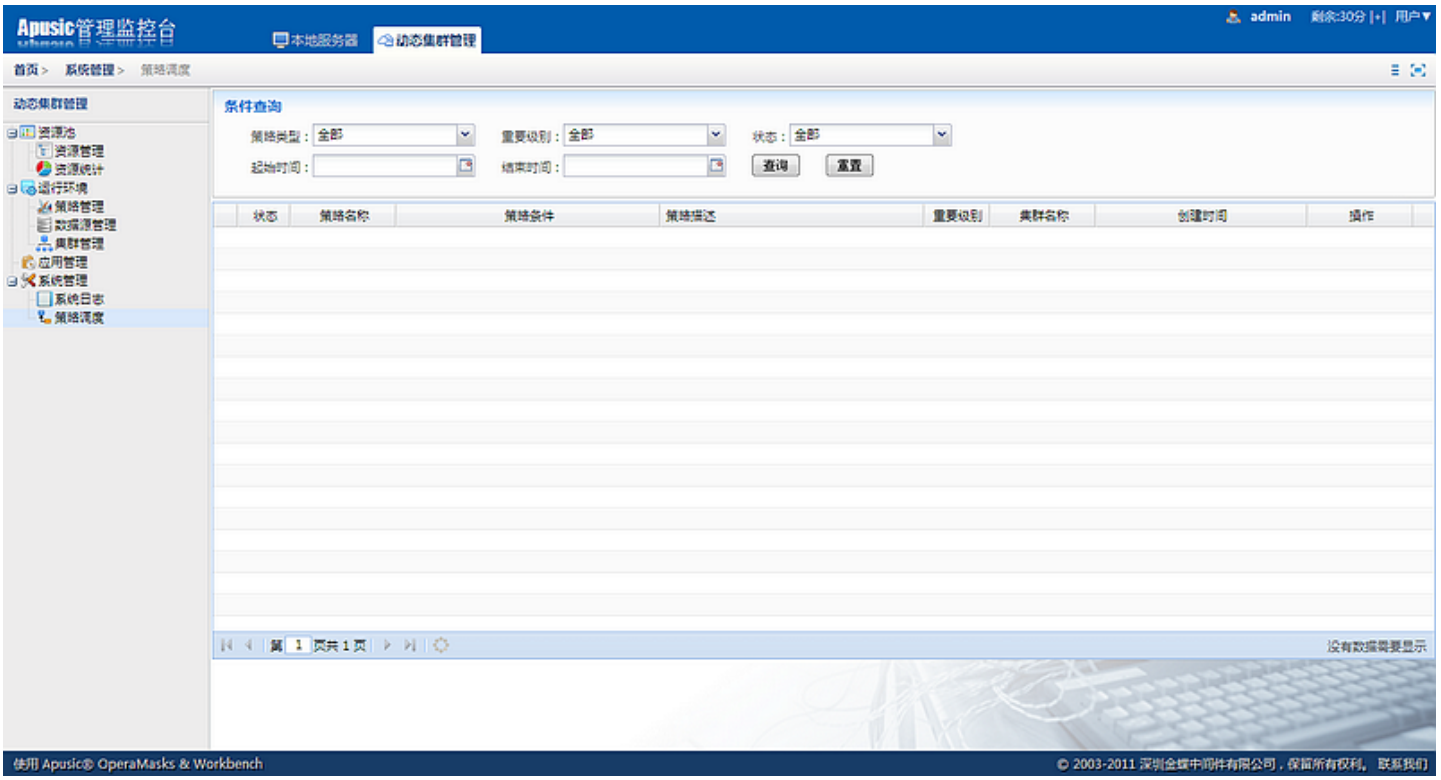
三. 标记未读

在系统日志列表中选中已读的记录，点击"系统通知"界面的"标记未读"按钮，即可把已读的日志标记为未读。未读的日志，在首页中"告警日志"统计中将会包含这条信息。

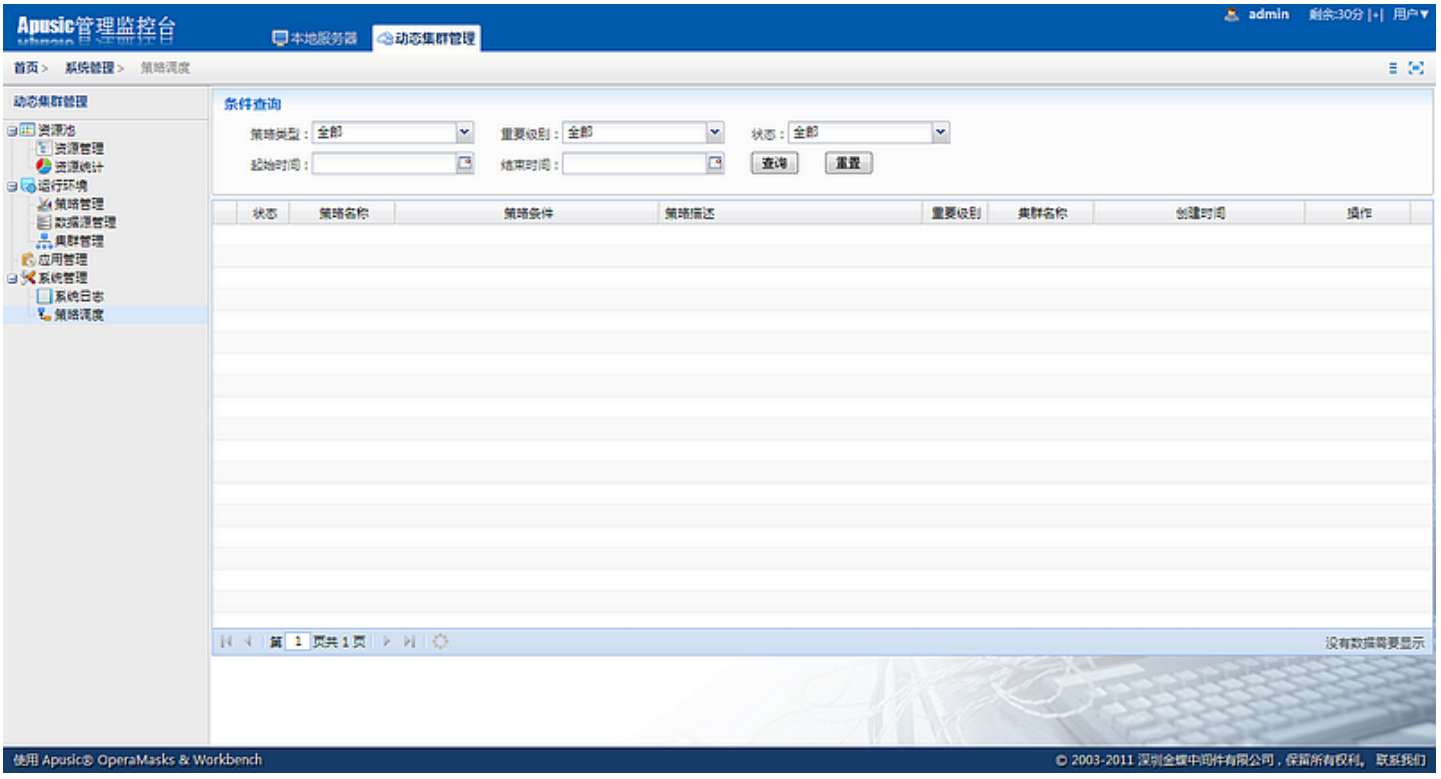
21.3.3.9.2 策略调度

展示了系统的策略调度信息与执行信息。在策略调度界面的列表下，可以对调度进行执行或者取消操作来完成策略的执行与取消。

如果给集群配置了手动策略，则策略运行时将会在策略调度列表下新增一个调度，执行这个调度将会触发手动策略，取消则不触发。



通过在条件查询面板输入查询条件，查询条件包括：策略类型，重要级别，状态，起始时间，结束时间。输入查询条件后，点击查询面板的“查询”按钮，可以对策略调度进行特定条件下的查询，下面是查询日志类型为：警告，所有时间段的日志：



21.3.3.10 证书管理

证书管理是用于管理证书的模块。进入动态集群管理系统，点击“资源管理”，选择一个节点，点击这个节点的实例管理，然后再选择一个实例，点击这个实例的“监控”按钮，就可进入到这个实例的管理控制台，然后在“系统配置”下可以看到“证书管理”。

21.3.3.10.1 基本概念

证书：公钥的载体，除了公钥外，里面还包括该证书的持有人的信息，例如姓名，地址...等。文件形式的证书后缀通常为crt或cert。

证书库：保存证书的存储库,通常的证书库格式有JKS（只在java程序中使用）和pfx(或p12)，证书库中除了包含证书外，还可以包含私钥。在ssl配置中，有时（双向认证时）除了配置证书库外，还需要配置信任证书库，信任证书库是用于保存信任的第三方证书用的。

证书请求文件：当向CA申请证书时，有时CA会要求提供证书请求文件（csr文件），该文件中包含申请者的个人信息，可以通过记事本等工具打开。

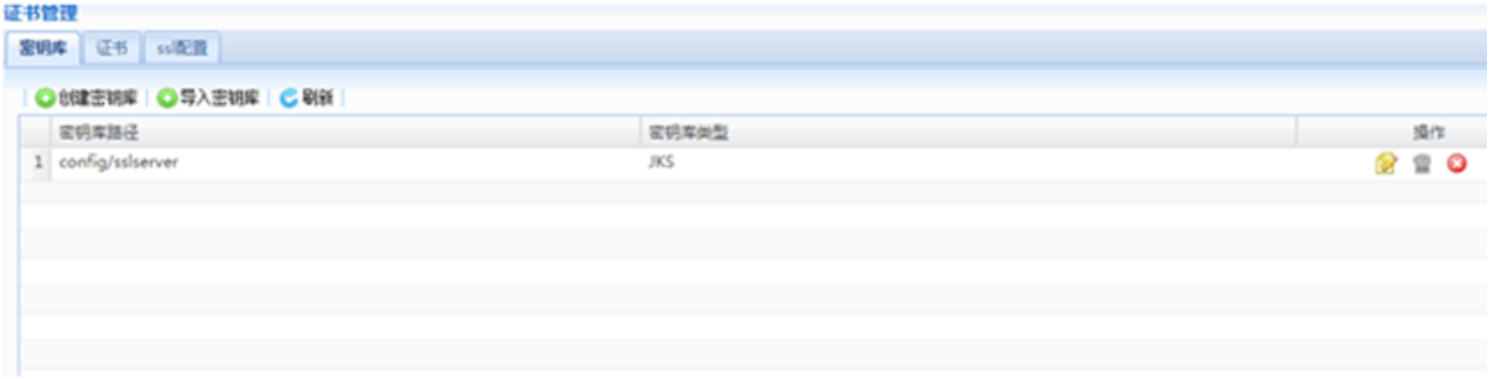
单向认证/双向认证：前者用于客户端认证服务器，即客户端认证服务器端传过来的证书；后者除了客户端需要认证服务器的证书外，服务器端也需要认证客户端的证书。直接信任该证书或信任该证书的颁发者证书都可以完成对该证书的认证。

21.3.3.10.2 使用自签名证书配置AAS SSL

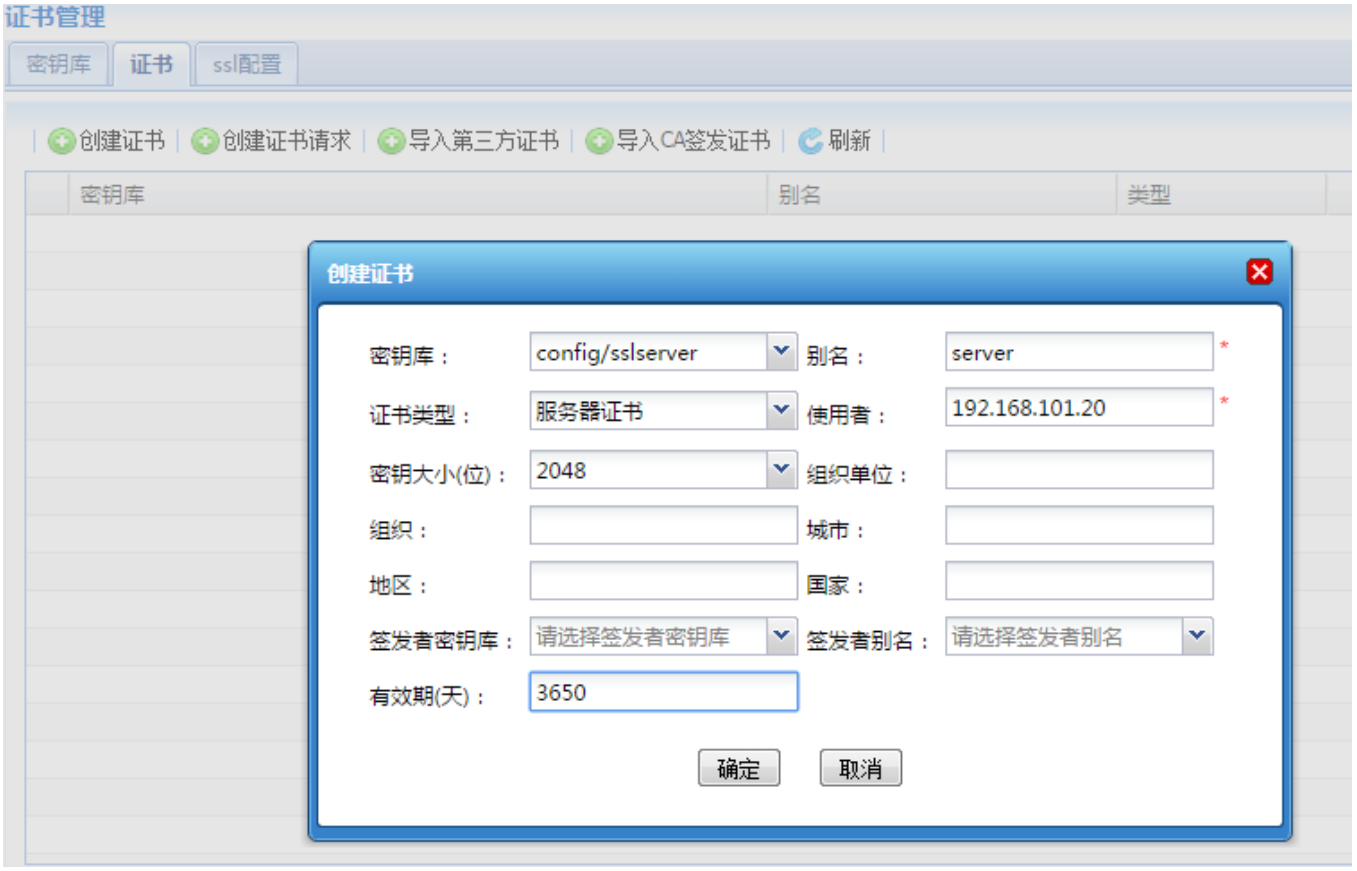
21.3.3.10.3 单向认证

(1). **直接认证服务器证书**

1. 添加证书库



2. 创建自签名证书

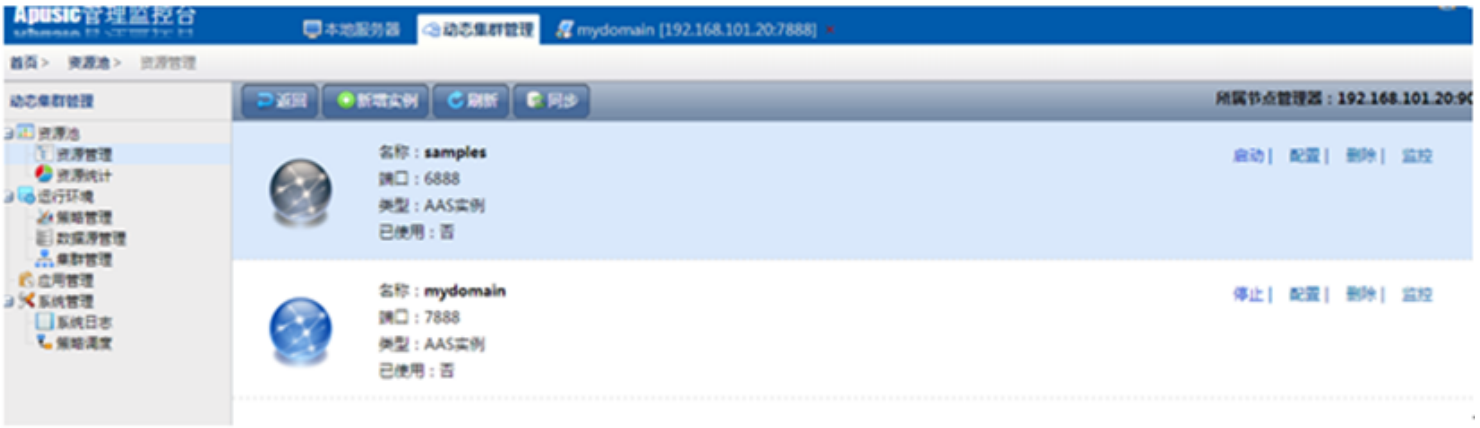


注意使用者必须填写服务器的域名，没有的话就填IP。

3. 配置SSL



SSL重新配置后要重启，在动态集群管理界面停止在启动即可。



4. 打开浏览器输入管理控制台地址



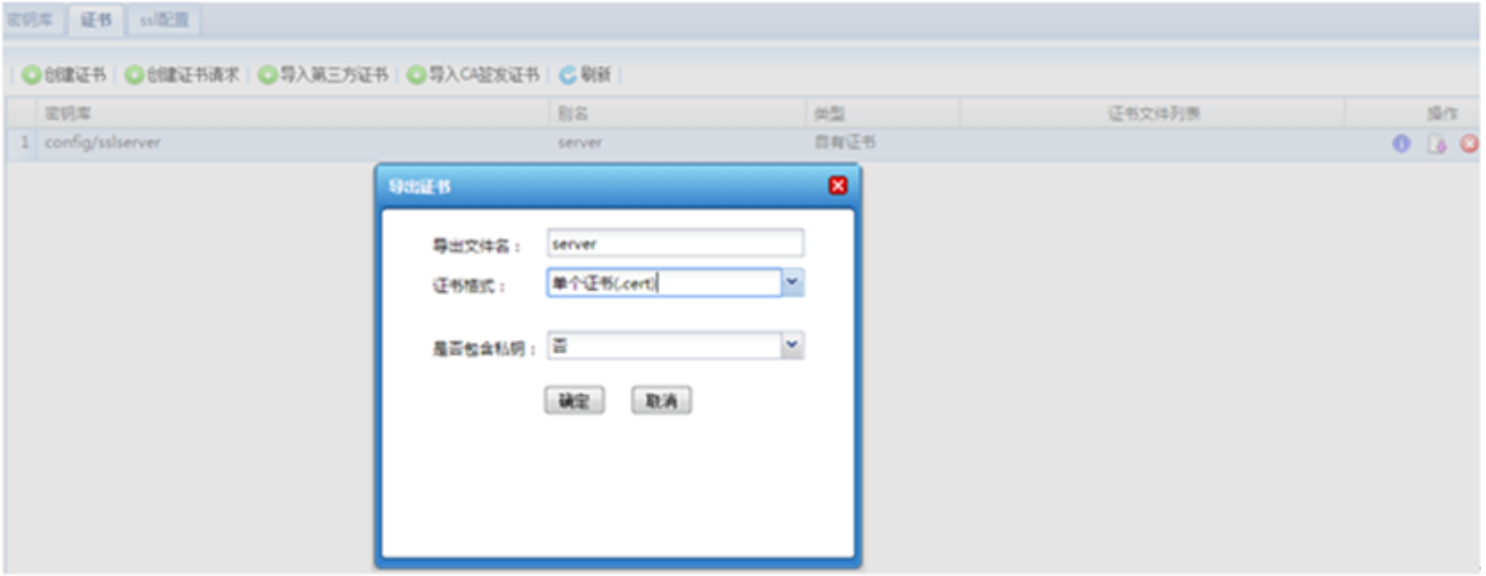
由于是自签名证书，浏览器无法确定是否该信任该证书，可以选择“继续浏览此网站”：



此时可以访问页面，但地址栏会显示证书错误。

5. 让浏览器信任服务器证书

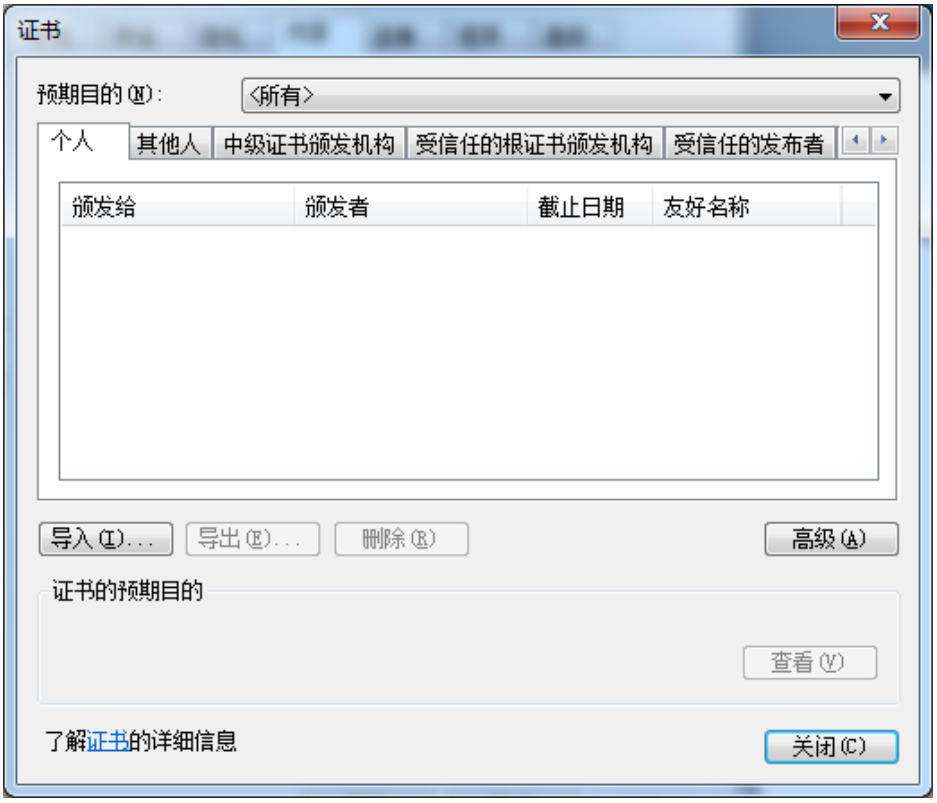
先导出服务器证书



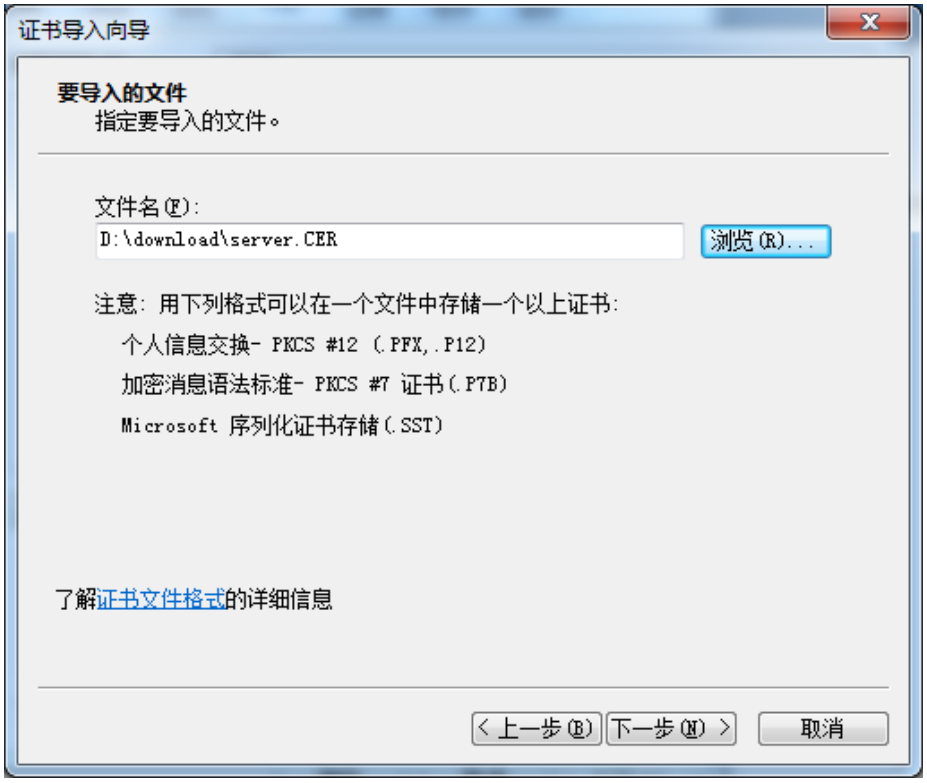
点击server.cer下载证书到本地：



在浏览器的"Internet选项□内容"页面点击"证书"按钮，打开证书配置界面：

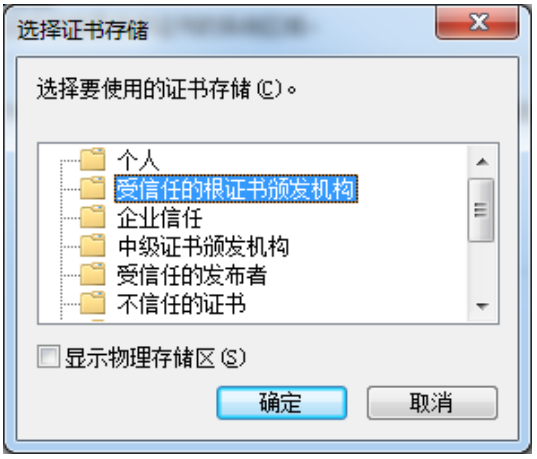


点击""导入"按钮，并选择刚刚下载的server.cer文件：

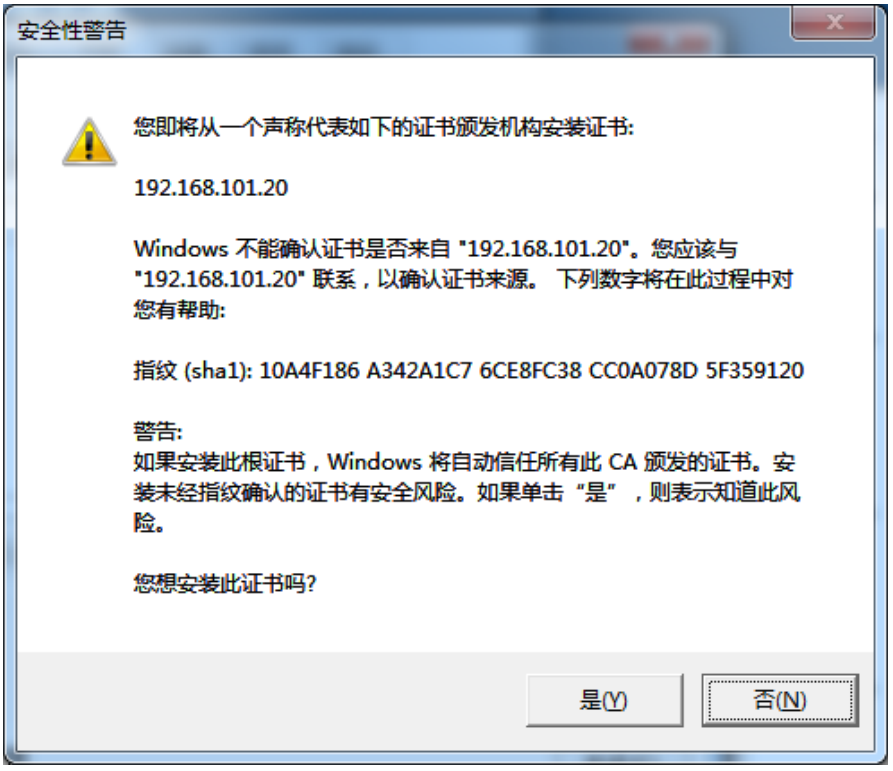


点击下一步，选择证书存储区域：

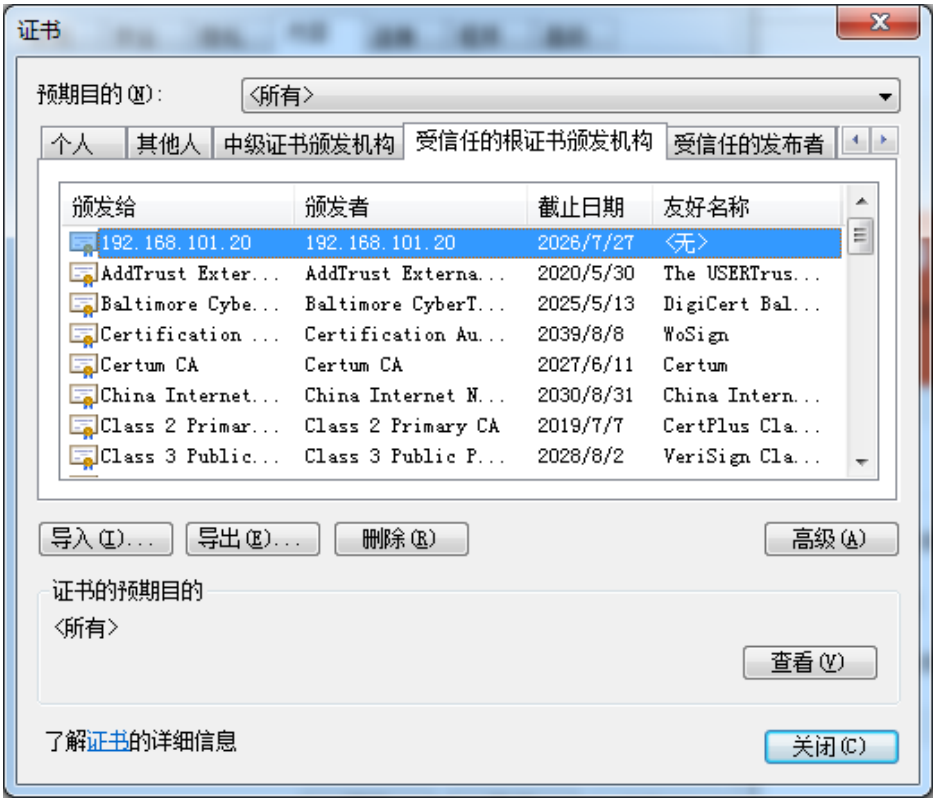




点击完成，系统会弹出警告对话框，点击"是"即可：



此时受信任的根证书页签中有导入的服务器证书：



重新访问管理控制台，此时地址栏上不再显示证书错误：



(2). 认证服务器证书的根证书

这个场景可用于使用一个自签名的根证书给多个应用服务器生成服务器证书，这样客户端只需要信任这个根证书即可，不需要再分别信任每个服务器证书。

1. 创建一个密钥库用于保存根证书



2. 创建自签名根证书



3. 创建服务器密钥库



4. 创建服务器证书并使用根证书签名



5. 配置ssl并重启应用服务器

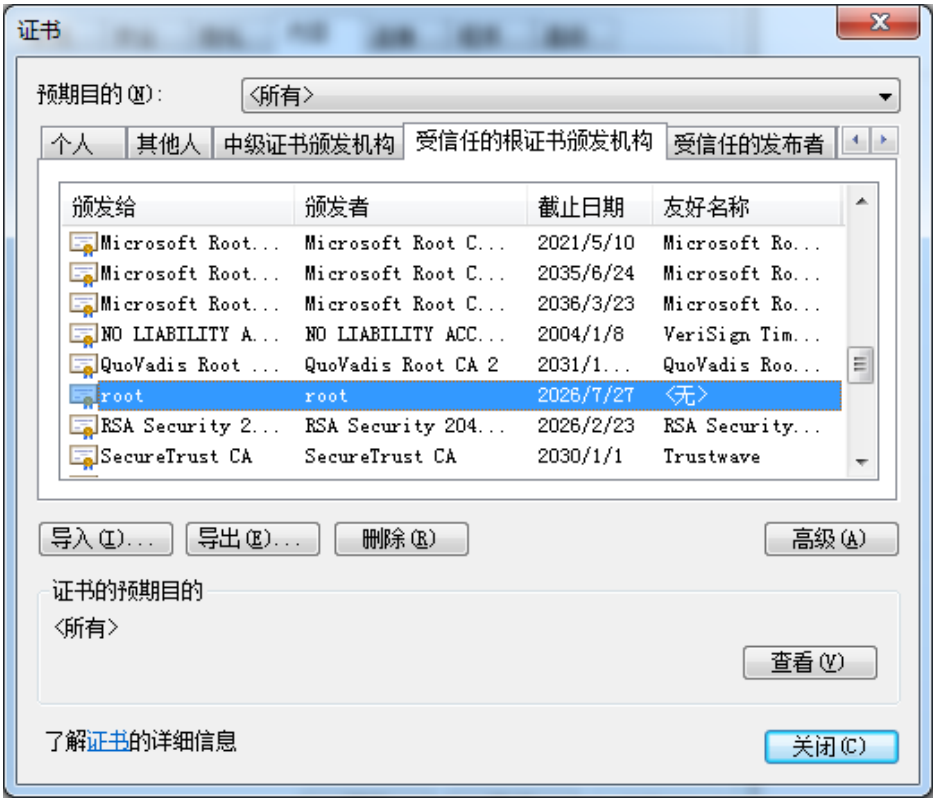


6. 打开浏览器访问管理控制台，此时还是显示证书错误



注意：每次测试的时候需要在IE的证书管理中删除之前测试导入的证书，并重启IE。

7. 导入根证书到IE浏览器



浏览器可以正常显示:



一. 双向认证

(1). 直接认证对方证书

服务器直接认证客户端证书，客户端直接认证服务器证书。

1. 创建两个证书库，分别用于保存服务器端证书和客户端证书



2. 分别创建服务器自签名证书和客户端自签名证书，并存入各自的密钥库



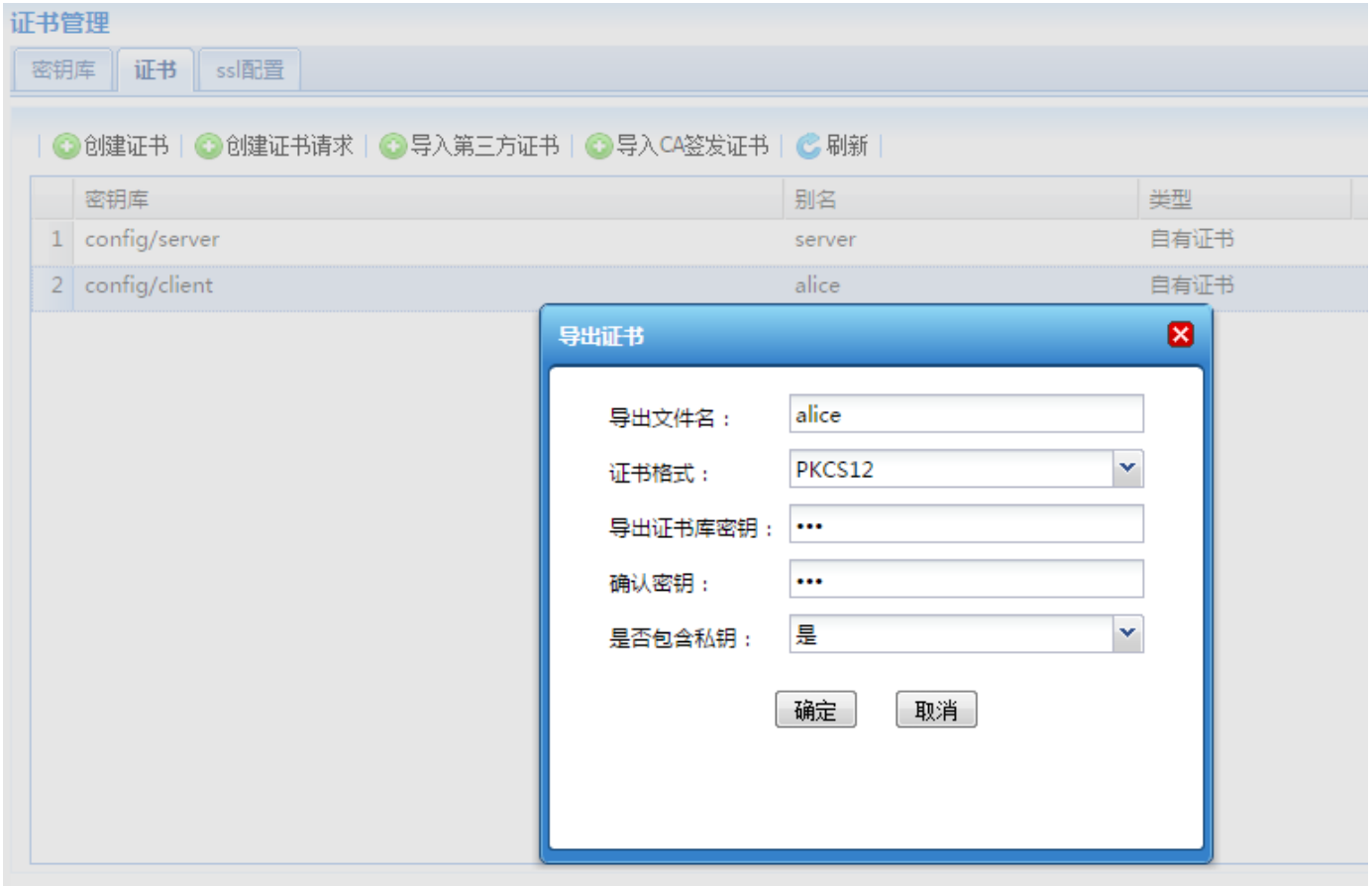
3. 配置SSL并重启

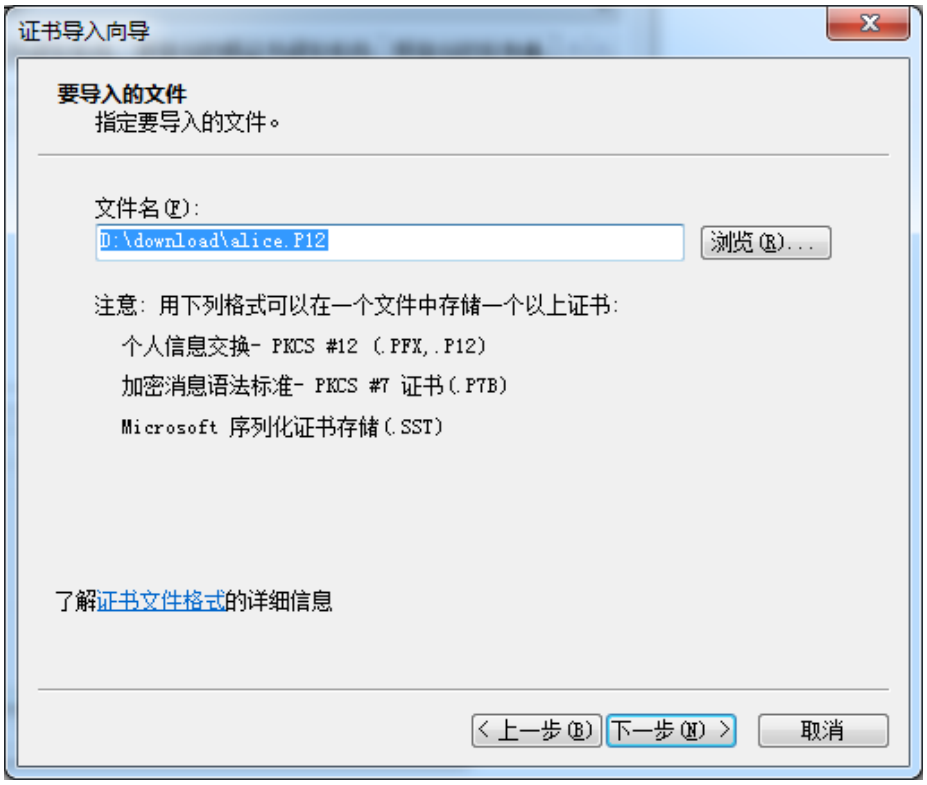


4. 打开浏览器访问管理控制台，此时网页无法正常显示

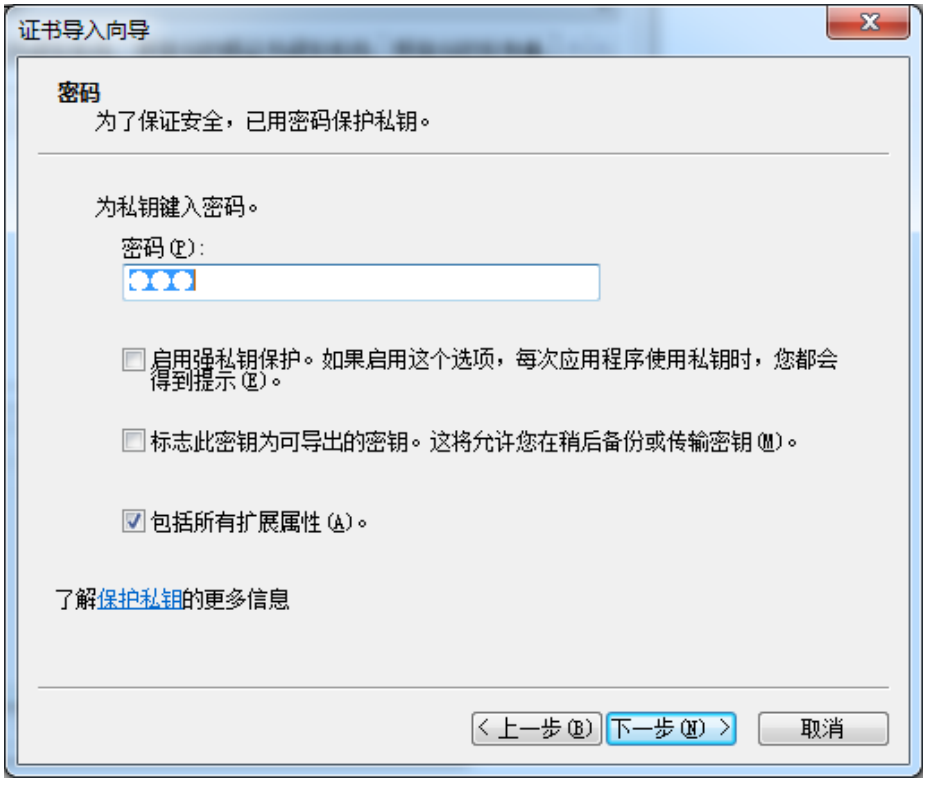


5. 导出客户端证书并安装到IE

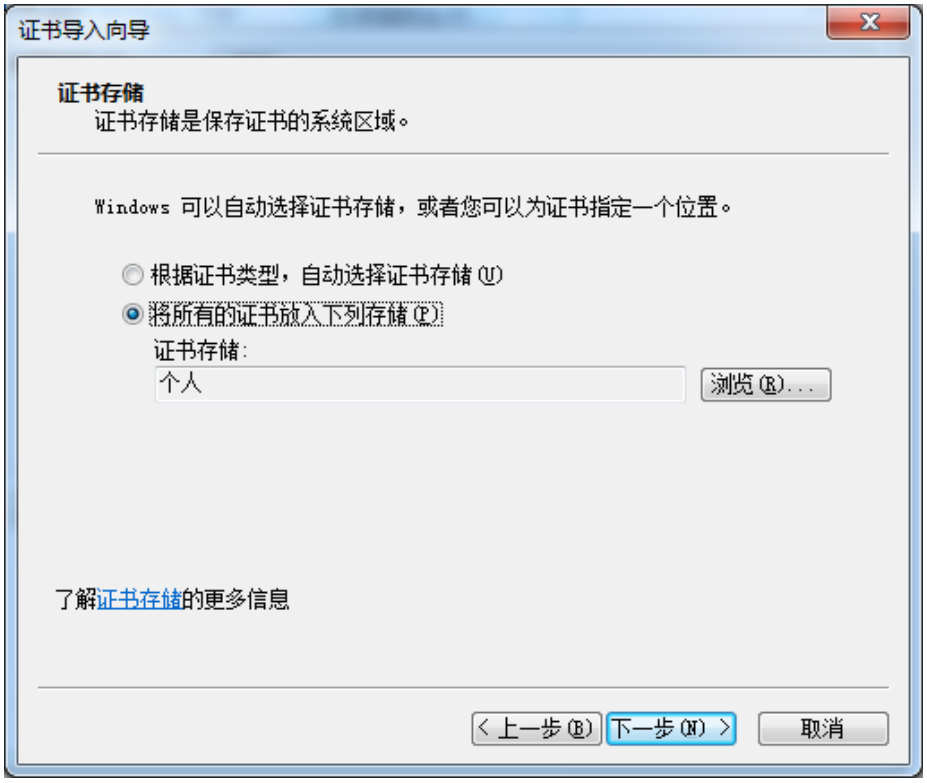




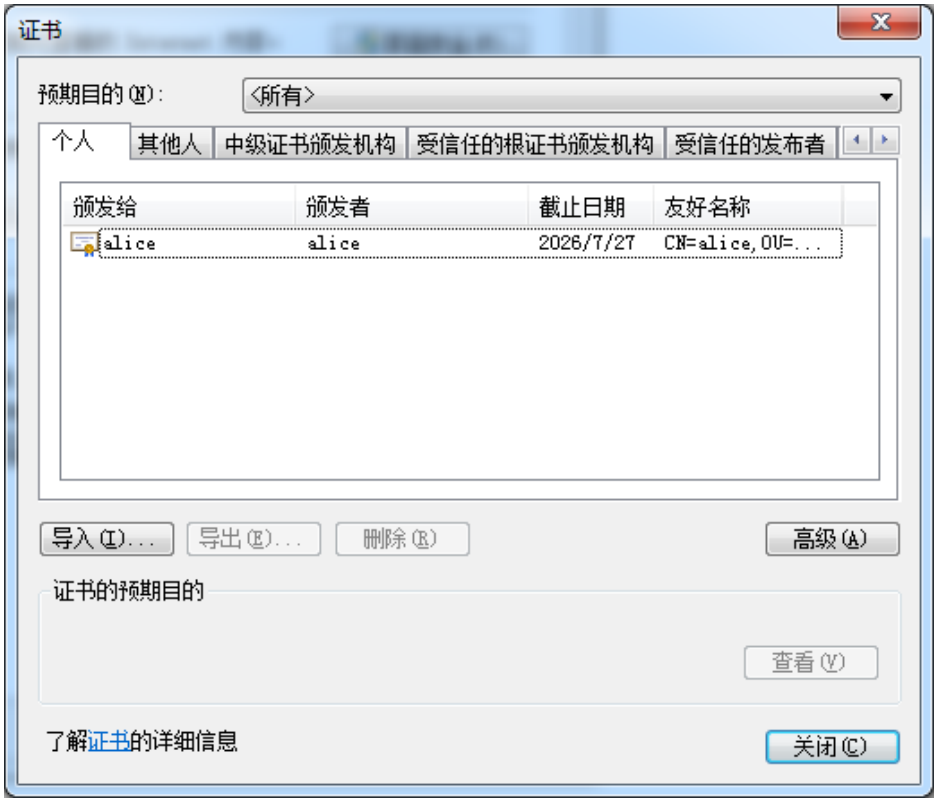
输入导出时指定的密码:



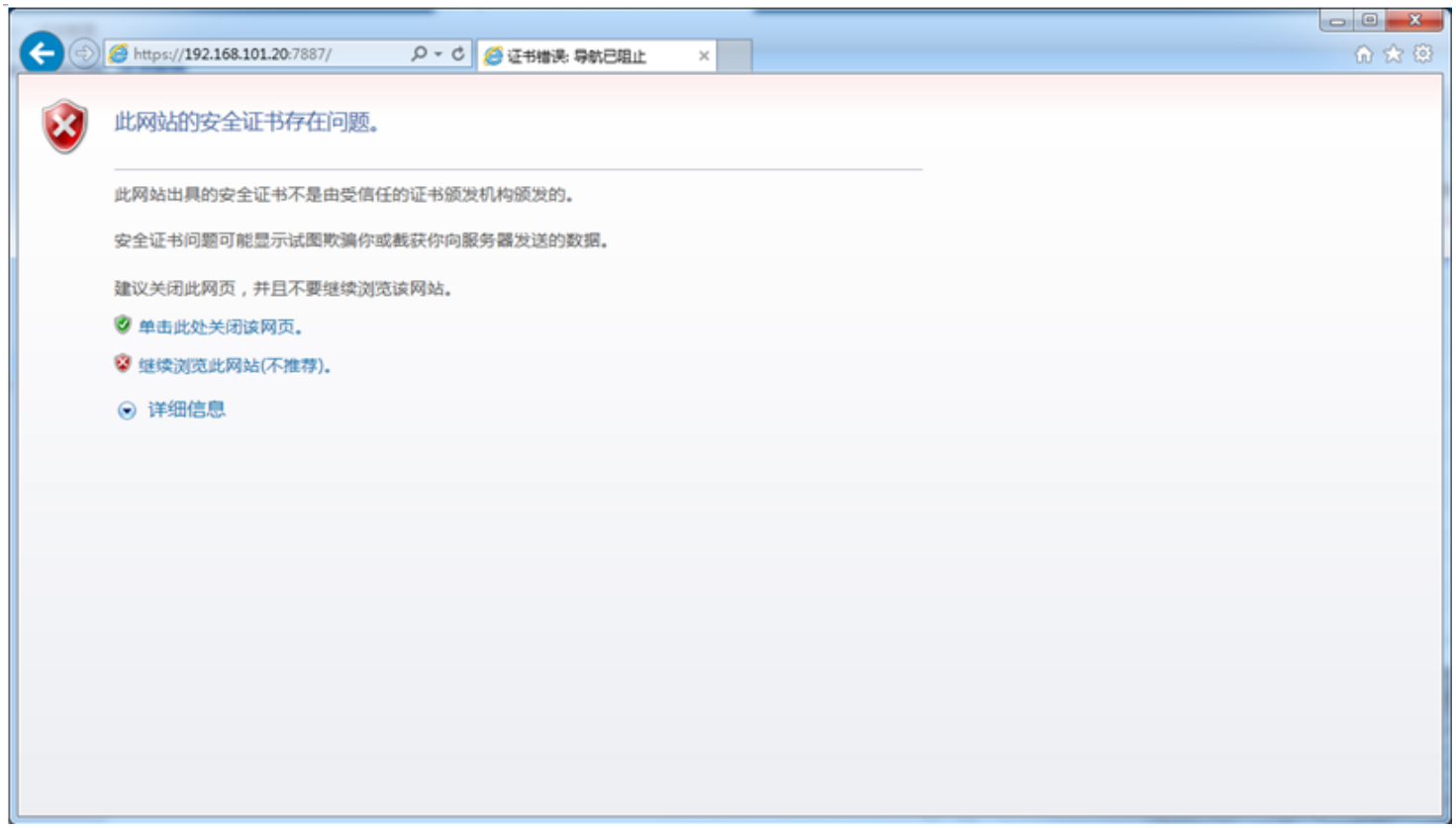
导入个人存储区即可:



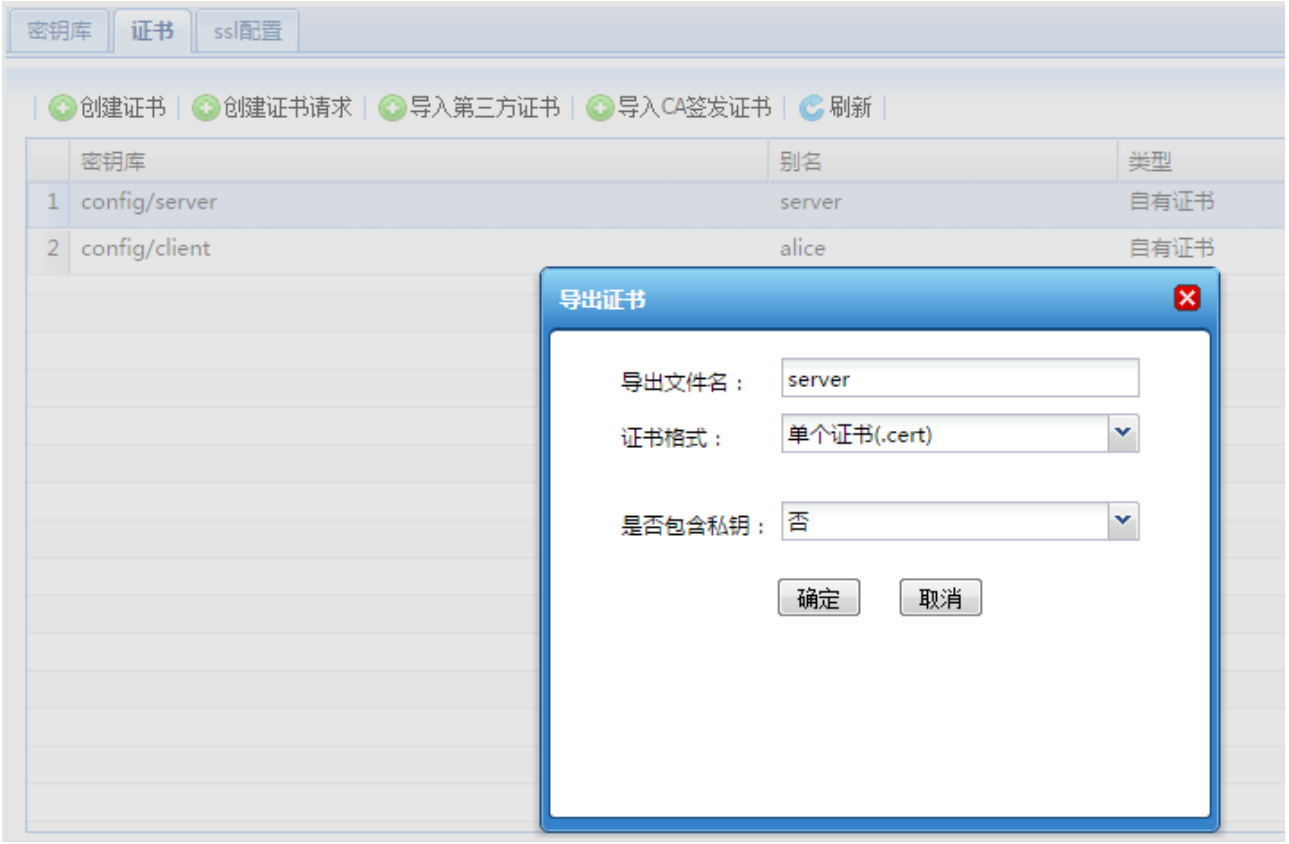
导入后的效果：

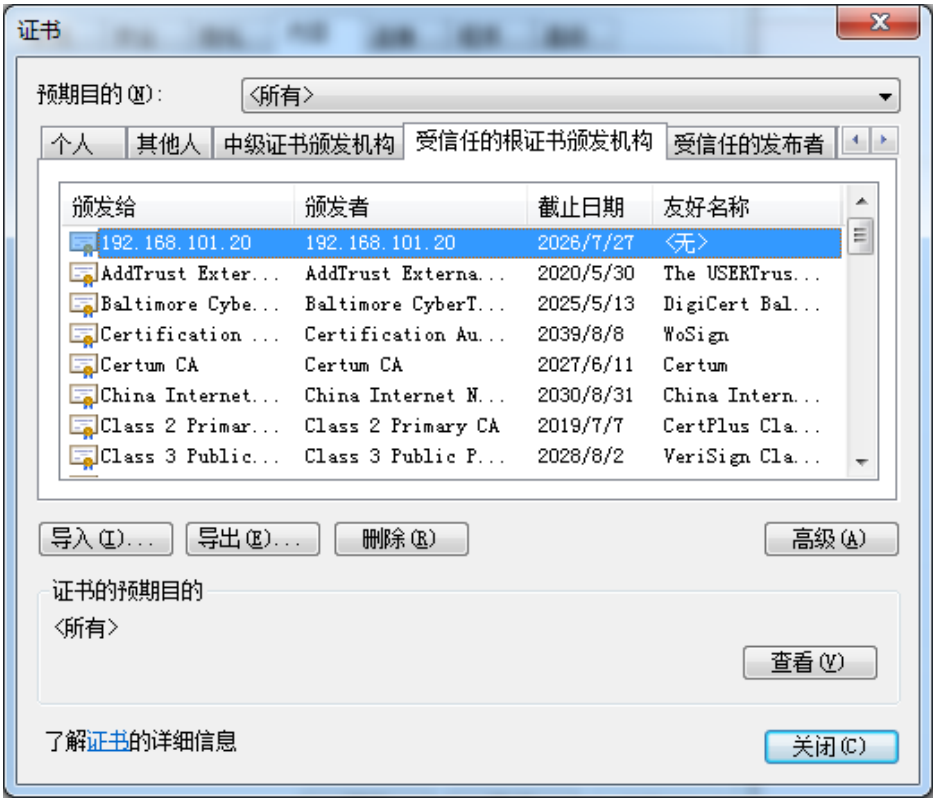


此时再访问管理控制台，显示服务器证书不受信任。



6. 导出服务器证书并导入IE





再次访问管理控制台，能正常显示：



(2). 认证对方证书的根证书

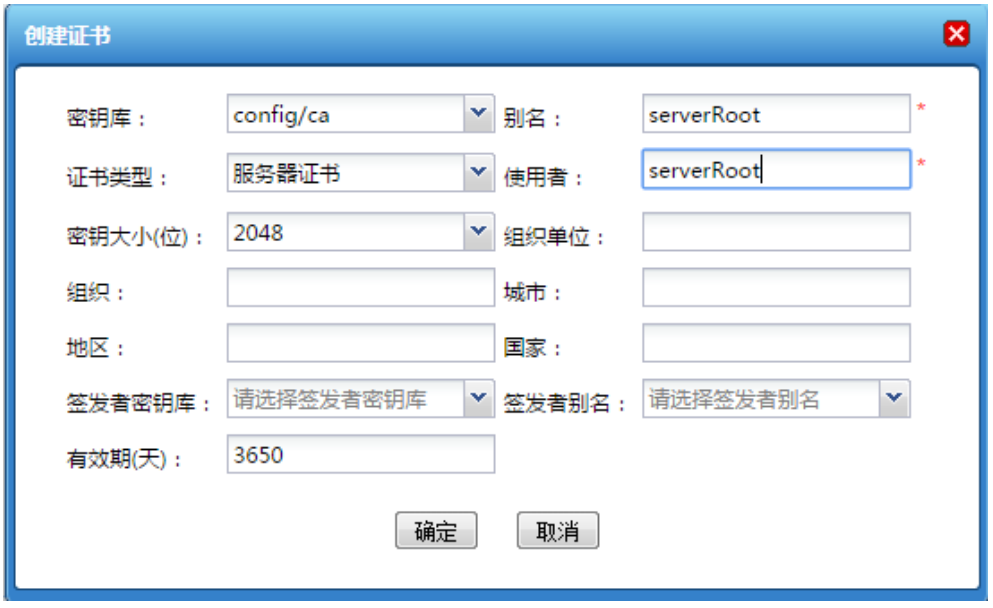
上面的场景虽然可以正常工作，但前提是客户端证书也是自己生成的。如果客户端证书是由其他机构生成的，那么服务器必须一个个将客户端证书导入信任库，使用起来会比较麻烦。此时可以通过将所有客户端证书的根证书导入到应用服务器的信任库，这样应用服务器只需要导入一次即可。

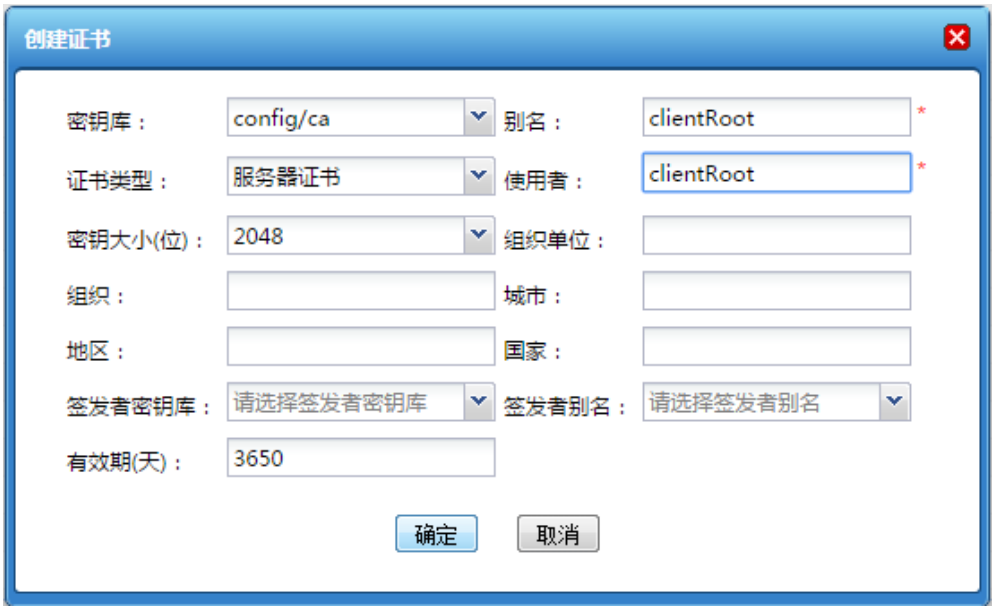
下面进行具体的配置描述，并结合上面章节，不直接信任服务器证书，而是信任服务器证书的根证书。

1. 创建三个密钥库，一个用于存放服务器证书，一个用于存放服务器信任的证书，一个用于存放根证书和客户端证书.

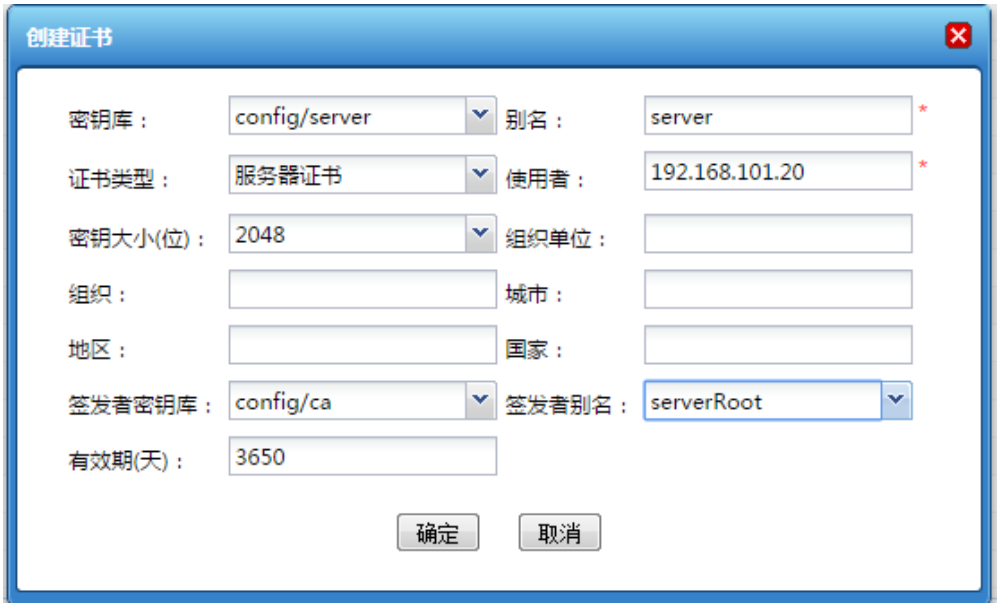


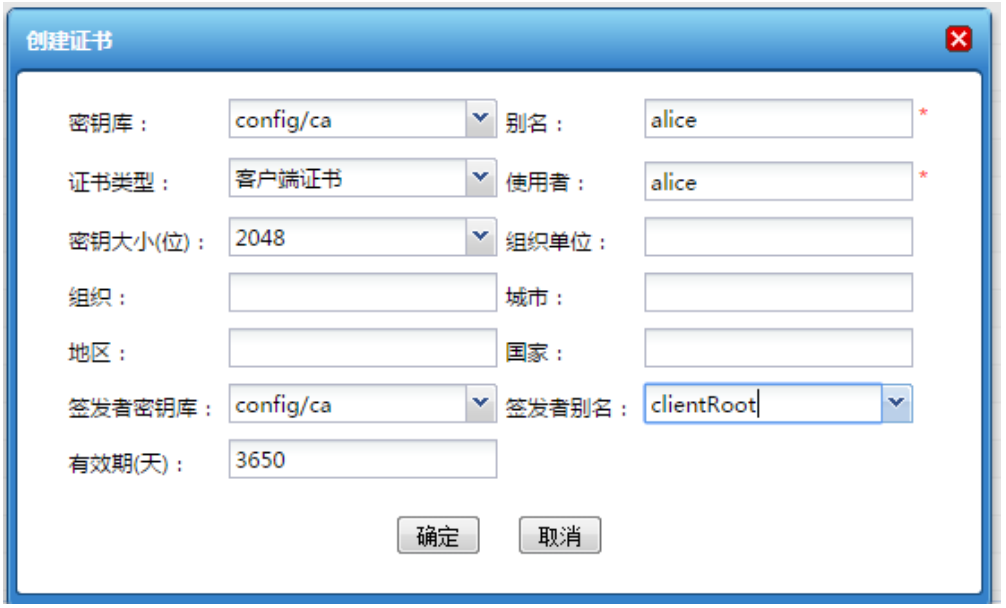
2. 创建服务器端根证书和客户端根证书，都是自签名证书



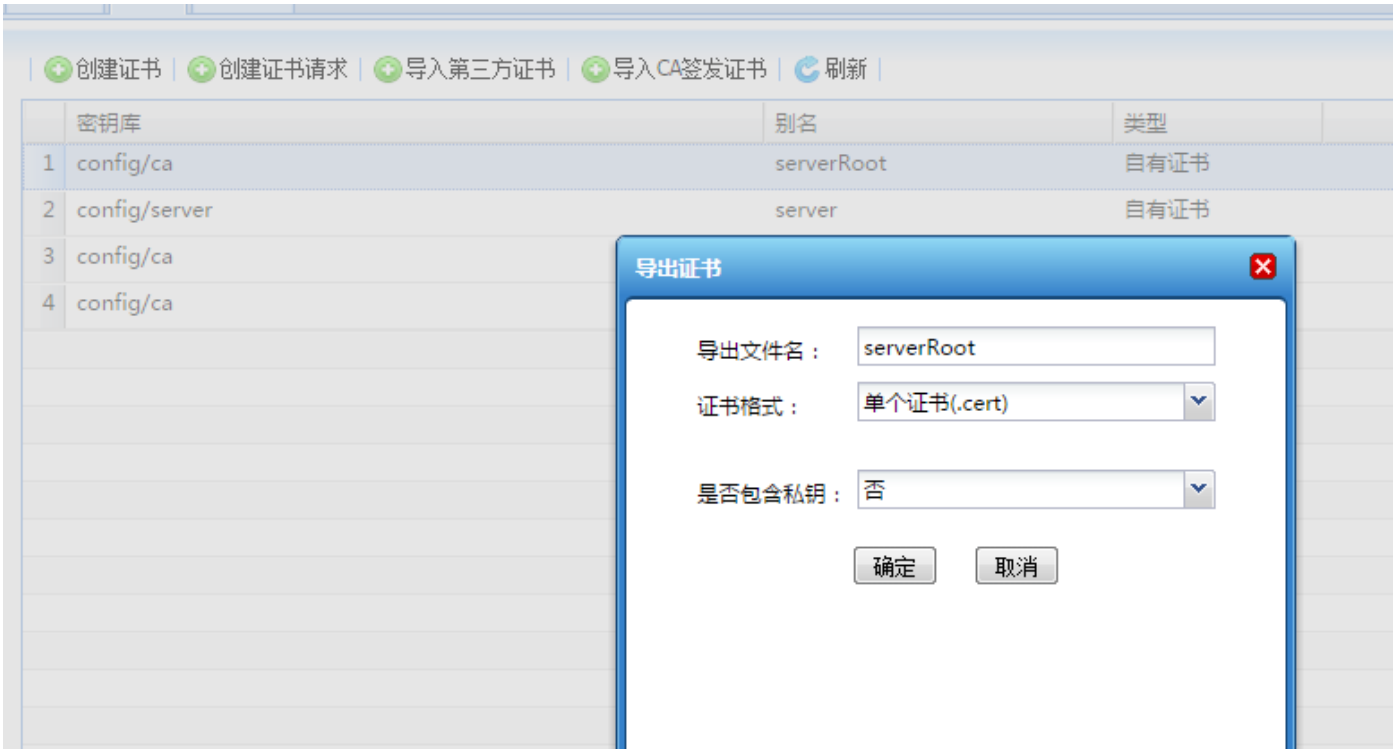


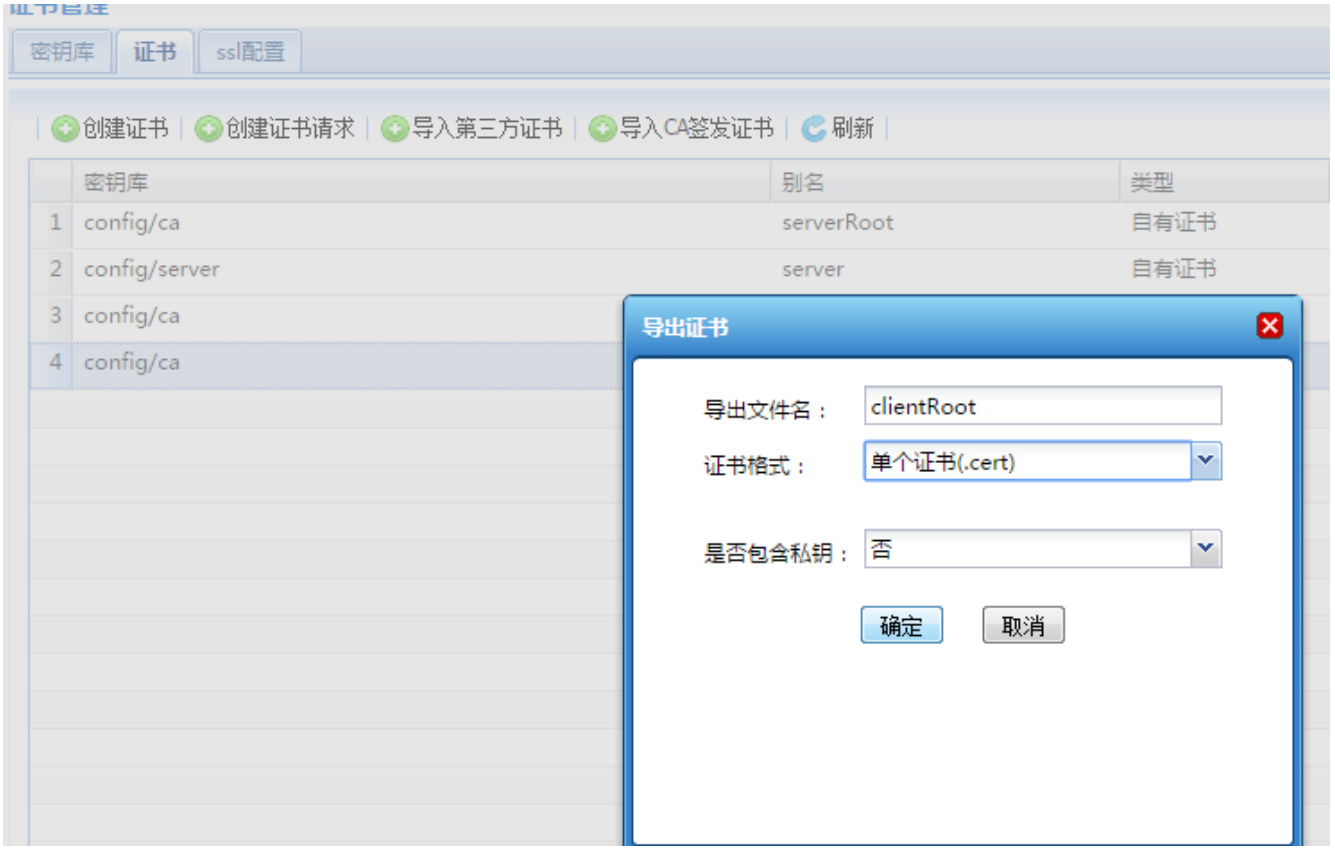
3. 创建服务器证书和客户端证书，并使用各自的根证书签名



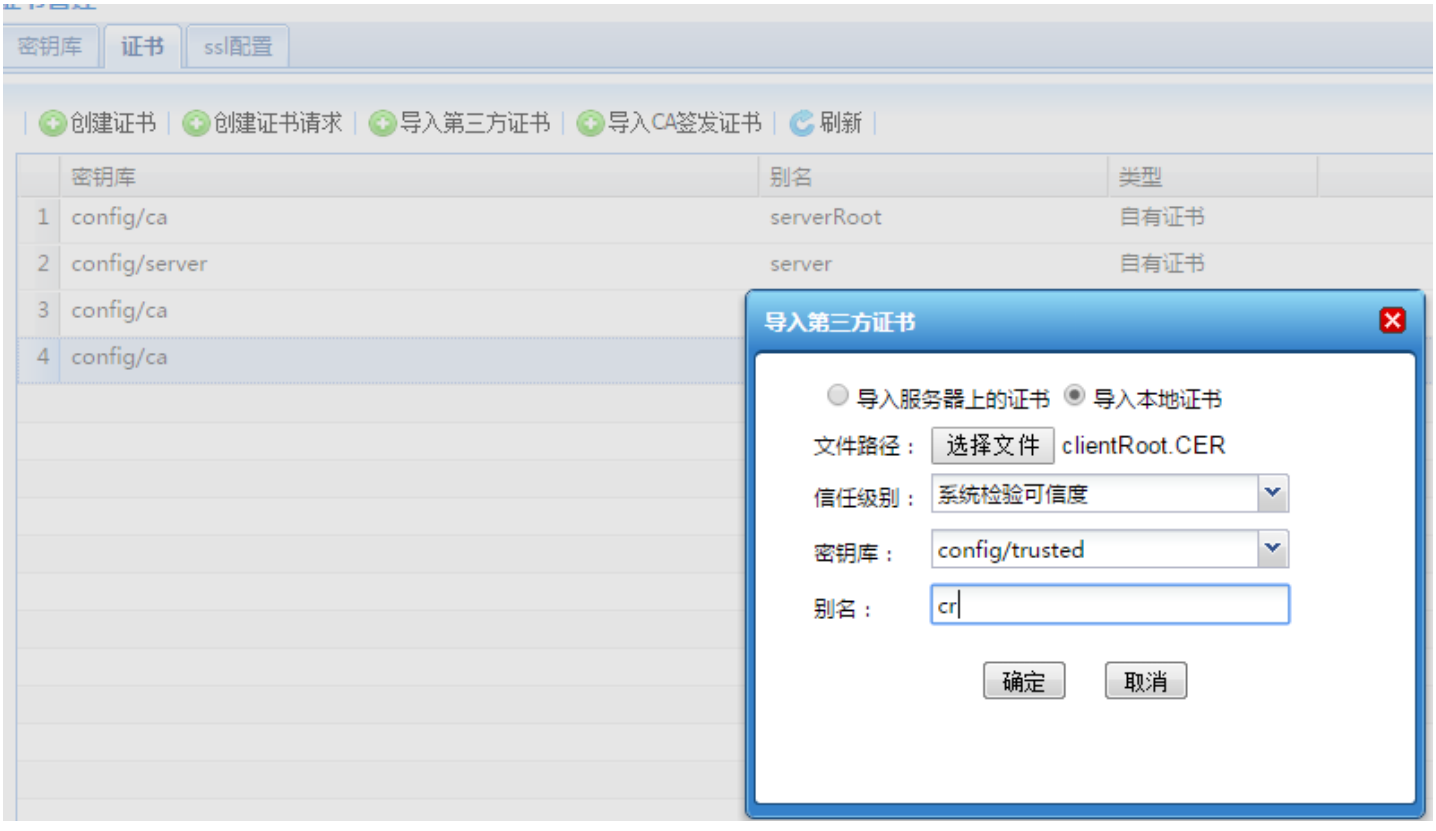


4. 导出客户端根证书和服务器端根证书

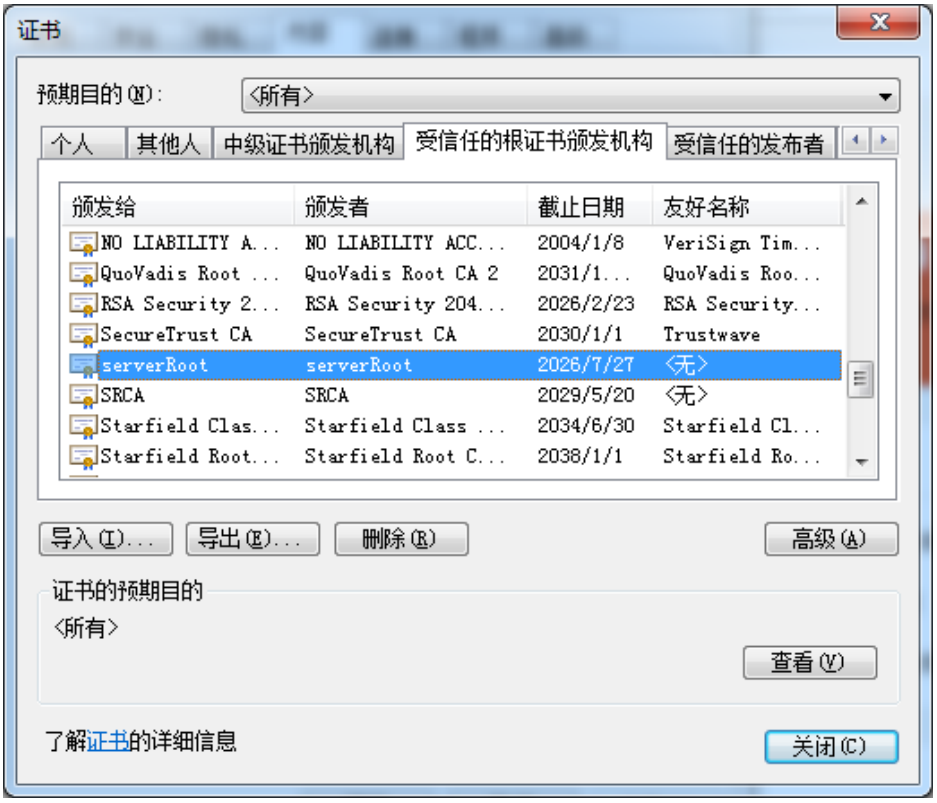




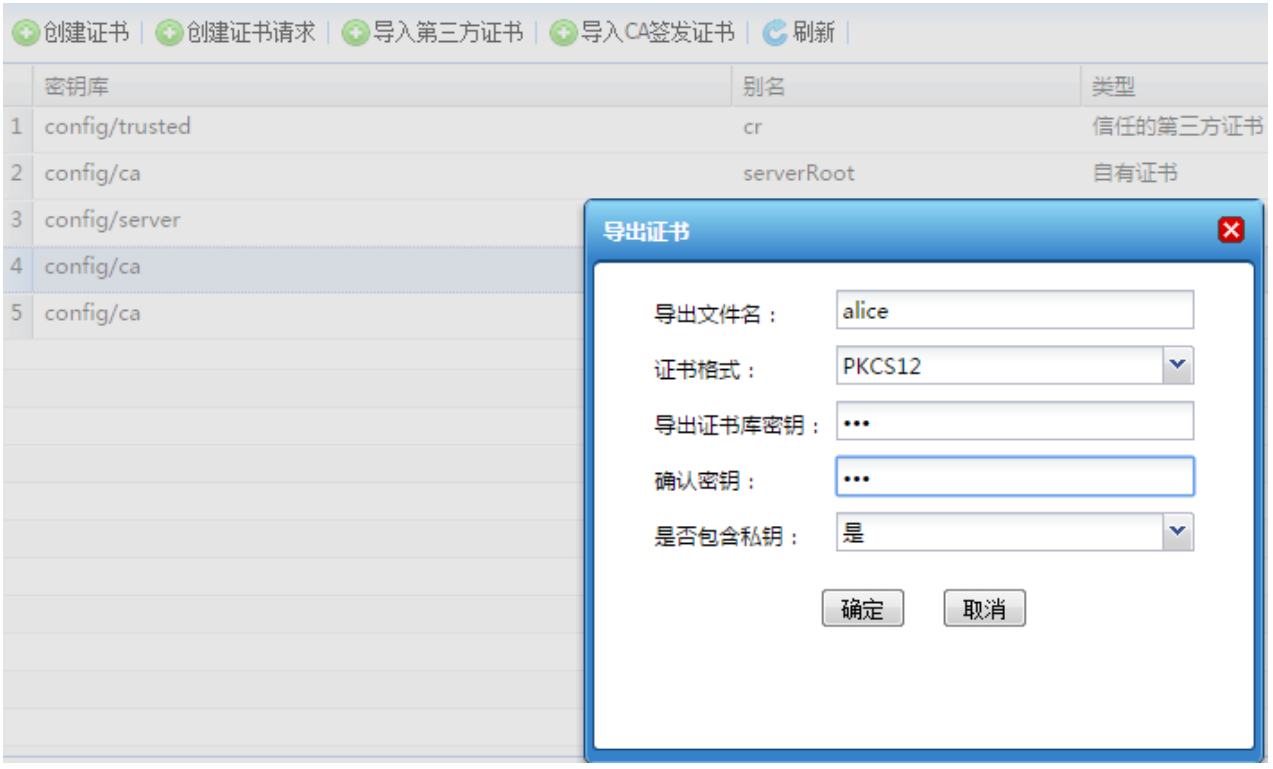
5. 下载导出的客户端根证书，并导入到服务器的信任库

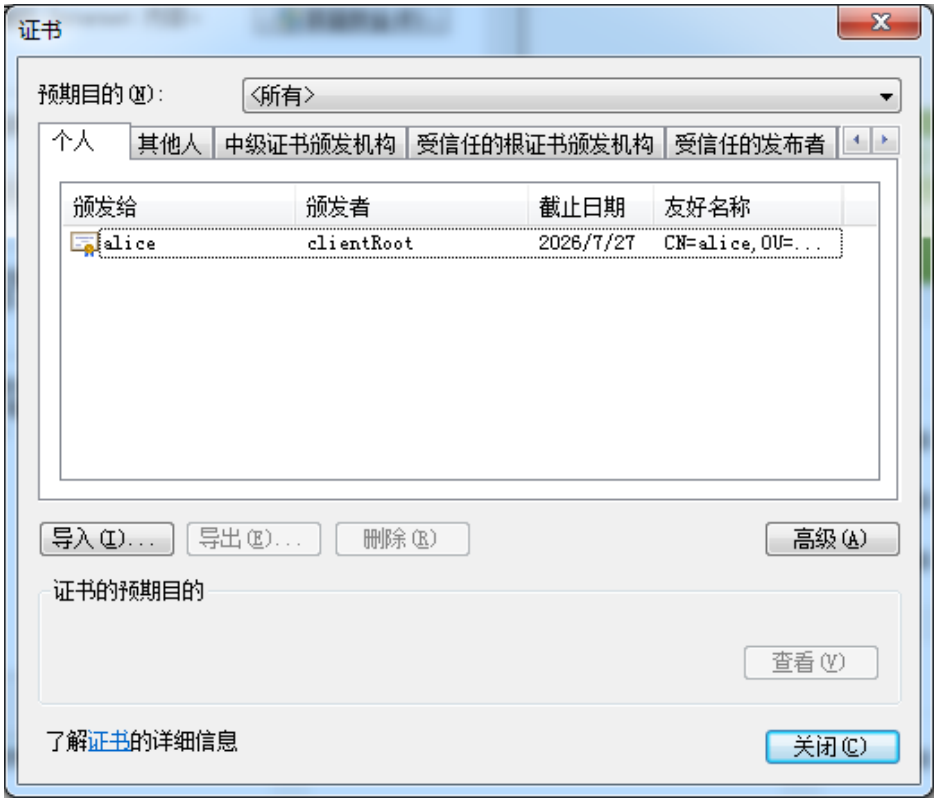


6. 浏览器安装导出的服务器根证书



7. 导出客户端证书，并安装到浏览器




证书
预期目的(N):
<所有>
个人
其他人
中级证书颁发机构
受信任的根证书颁发机构
受信任的发布者
颁发给
颁发者
截止日期
友好名称
alice
clientRoot
2026/7/27
CN=alice,OU=...
导入(I)...
导出(E)...
删除(R)
高级(A)
证书的预期目的
查看(V)
了解证书的详细信息
关闭(C)

8. 配置SSL并重启


证书管理
密钥库
证书
ssl配置
刷新
开启SSL:
单向认证端口:
7889
客户端认证方式:
双向认证
SSL协议:
TLSv1
双向认证端口:
7887
密钥库:
config/server
密钥库算法:
可信密钥库:
config/trust
可信密钥库算法:
CRL路径:
CRL检查间隔:
60
更新



9. 说明:

上面的配置服务器根证书和客户端证书是分开的，实际上可以是同一个，即客户端和服务器共用一个根证书，此时浏览器不需要再安装服务器端根证书，只需要安装客户端证书即可，因为导出的客户端证书库中包含根证书。

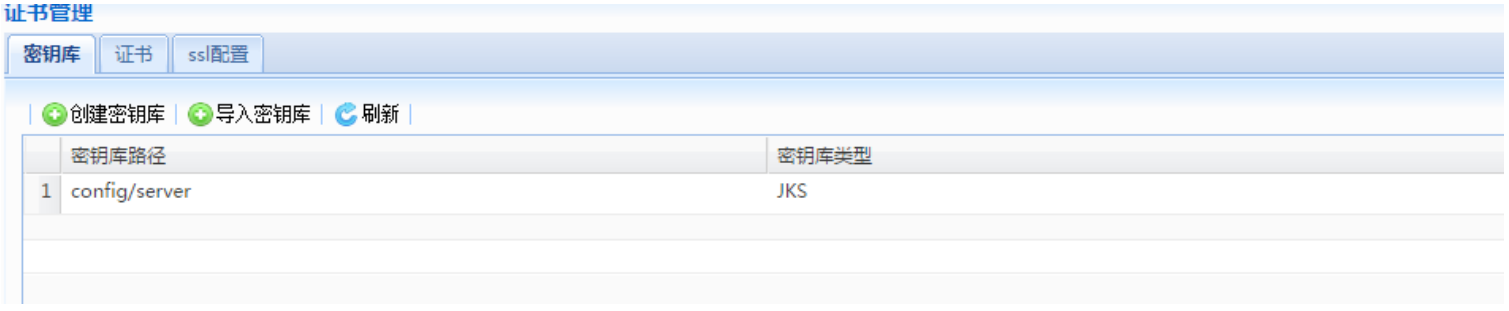
21.3.3.10.4 使用CA签发证书配置 AAS SSL

下面的配置介绍证书由第三方权威的CA机构生成，CA机构在生成证书时，有的会要求提供证书请求，有的不需要。

一. **单向认证**

(1). **需要证书请求**

1. 创建证书库



2. 生成证书请求



下载生成的server.CSR：



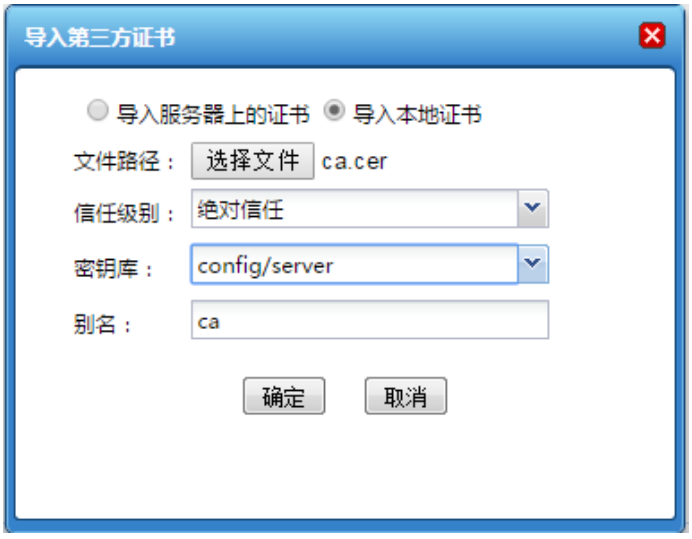
如果CA机构要求以文件形式提供，那么把下载的server.CSR文件传过去即可；如果要求提供文件内容，则需要把该文件用记事本打开，拷贝里面的内容进行提交：

server.CSR

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIICszCCAZsCAQAwbjELMAkGA1UEBhMCQ04xEDAOBgNVBAgTB1Vua25vd24xEDAOBgNVBAcTB1Vu
a25vd24xEDAOBgNVBAoTB1Vua25vd24xEDAOBgNVBAsTB1Vua25vd24xFzAVBgNVBAMTDjE5Mi4x
NjguMTAxLjIwMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAkso1Cs6KiIp9sq+6sGYZ
xGofp4qcs6/a5gWPulMSKgr4XLNCehHNGIPtENul93rJQjmo+CO33hH9vMIZKa+K7aEUU0AnYbvX
JTojUYkGehbZvHaoI5cDQuYyssDRJiaJgXp0GpFnLn5rYvWuaUpQPw9NQnXSg1nXgvvDxBHaV/Ap
vPIZyGYWkQSPebD0Nn0RUX/s0Ad/eQBpOztok7In2aNNbvMN9L3vxQaX4ADVD10ViAkTFMVC2k+t
Zo8QR/mhZOZ53jjCHAHQ7AFQk1ReoX5oQDmjmQKtABtBjtwTI99RzE8s9Z9hDmBtrtv8bJ/DOA84
BWawTt2WA/LbaCg6aQIDAQABoAAwDQYJKoZIhvcNAQELBQADggEBAHBByA895mWU0yjkV17NyjIX
bUZqSl2LQRzXgR6zBqzeKPhTuD8JX5pdM7kyA5il8Hd97UR6uuv6xQ+9whvbO+fJs9wF0A9zeKFH
/v8SmFuQXnWfsascE8Dq/nzAn9YfEzsZJN7SImnKybGX6x6S3fOiqd4j0Ge5yRjyaSe46ZvuMpMm
rL5wOEtZyCQem0ijFEP/rTIe6+2ht96ZxMeA1e7umZZ2Ty17EY43kbpH9tJoGPN8/7y1hFbf2qzm
MfWE0TbQX3Of6GJOaO0vM0wDUoJcmSjl0vb9yDCY62itcVXxEBzYL60dxcIRH0oTn2takYIdtCcK
Gm+d5Ep4wCsCNZ8=
-----END NEW CERTIFICATE REQUEST-----
```

3. CA接受证书申请后需要一定的时间进行证书制作，制作完成后会将证书文件传给申请者，申请者需要将该文件导入应用服务器；另外，如果CA同时也提供了自己的根证书，也需要将该根证书导入应用服务器。

导入CA根证书,选择导入第三方证书：



导入CA生成的证书，选择导入CA签发证书：



导入时别名一定要选择当时生成证书请求的那个证书条目。导入后可以看到别名为server条目已形成证书链：



4. 配置ssl并重启

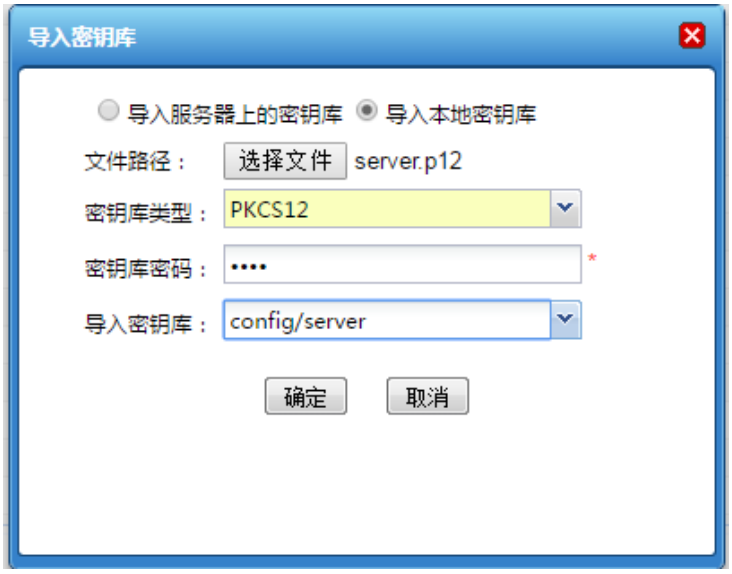




(2). **不需要证书请求**

这种情况所有工作都在CA完成，最后生成密钥库；应用服务器导入密钥库即可。

1. 将CA生成的密钥库导入应用服务器


导入密钥库
导入服务器上的密钥库
导入本地密钥库
文件路径：
选择文件
server.p12
密钥库类型：
PKCS12
密钥库密码：
导入密钥库：
config/server
确定
取消

2. 配置ssl并重启


证书管理
密钥库
证书
ssl配置
刷新
开启SSL:
单向认证端口:
7889
客户端认证方式:
单向认证
SSL协议:
TLSv1
双向认证端口:
7887
密钥库:
config/server
密钥库算法:
可信密钥库:
请选择可信密钥库
可信密钥库算法:
CRL路径:
CRL检查间隔:
60
更新



二. 双向认证

(1). 需要证书请求

1. 服务器证书同上，另外还需创建一个密钥库，并将CA根证书导入该密钥库。

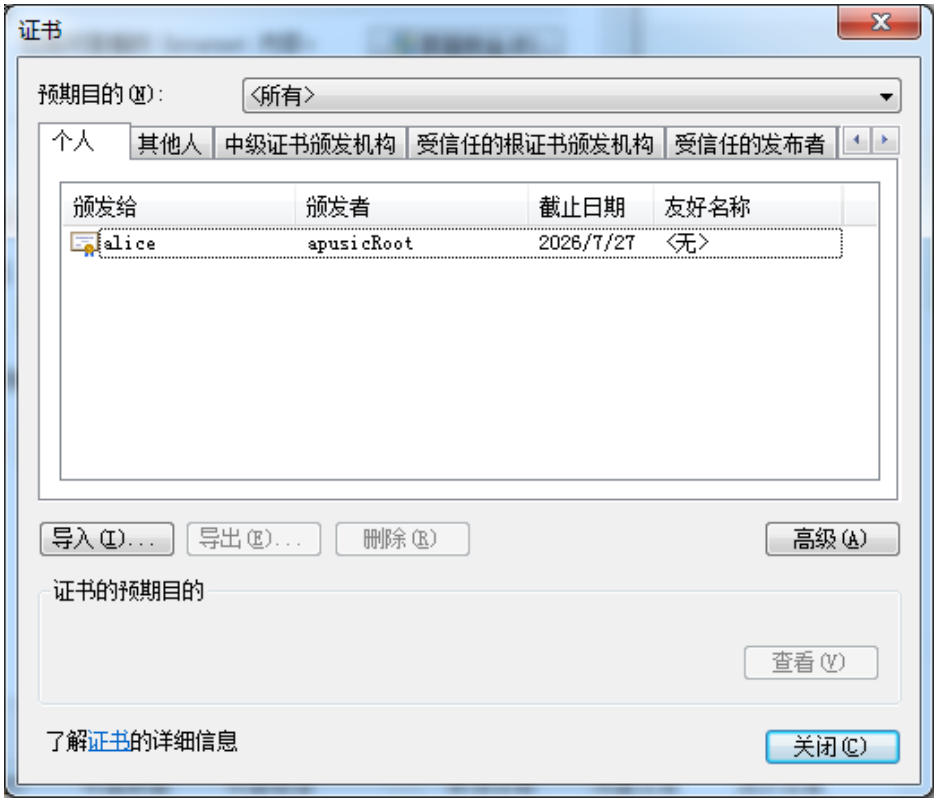


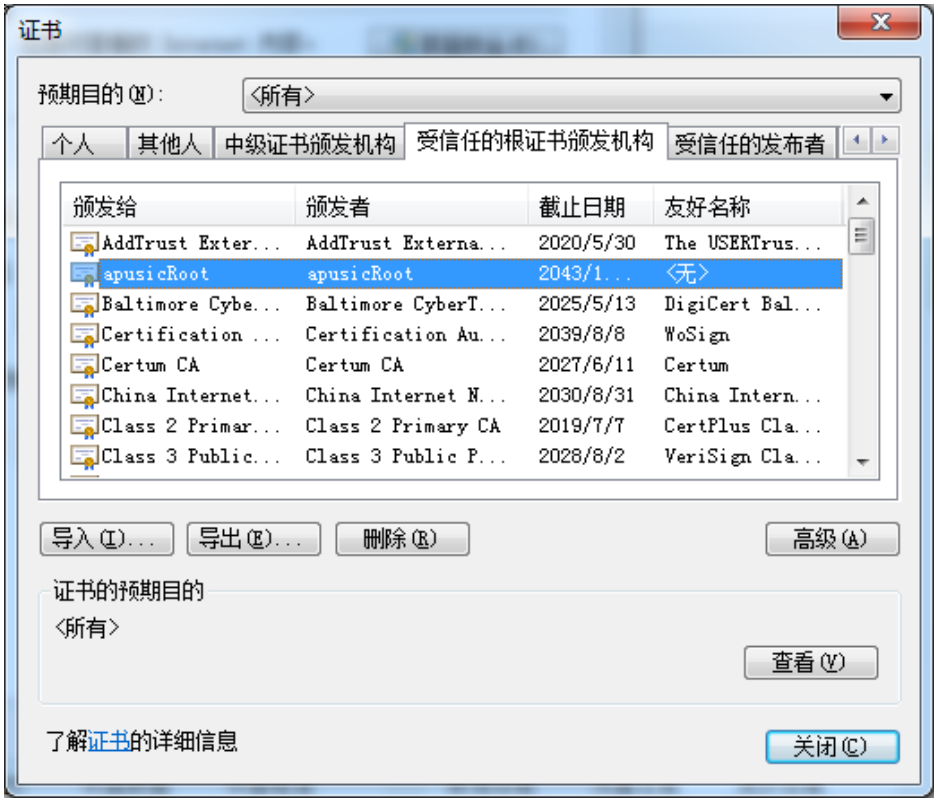
2. 生成客户端证书，客户端证书一般都不需要证书请求

```
openssl genrsa -aes256 -out private/client.key.pem 2048
openssl req -new -key private/client.key.pem -out private/client.csr -subj
"/C=CN/ST=GD/L=SZ/O=apusic/OU=kingdee/CN=alice"
```

```
openssl x509 -req -days 3650 -sha256 -CA certs/ca.cer -CAkey
private/ca.key.pem -CAserial ca.srl -in private/client.csr -out
certs/client.cer
openssl pkcs12 -export -inkey private/client.key.pem -in certs/client.cer -
out certs/client.p12
```

3. 将ca.cer和生成的certs/client.p12拷贝出来，并导入浏览器




证书
预期目的(N): <所有>
个人
其他人
中级证书颁发机构
受信任的根证书颁发机构
受信任的发布者
颁发给
颁发者
截止日期
友好名称
AddTrust Exter...
AddTrust Externa...
2020/5/30
The USERTrus...
apusicRoot
apusicRoot
2043/1...
<无>
Baltimore Cybe...
Baltimore CyberT...
2025/5/13
DigiCert Bal...
Certification ...
Certification Au...
2039/8/8
WoSign
Certum CA
Certum CA
2027/6/11
Certum
China Internet...
China Internet N...
2030/8/31
China Intern...
Class 2 Primar...
Class 2 Primary CA
2019/7/7
CertPlus Cla...
Class 3 Public...
Class 3 Public P...
2028/8/2
VeriSign Cla...
导入(I)...
导出(E)...
删除(R)
高级(A)
证书的预期目的
<所有>
查看(V)
了解证书的详细信息
关闭(C)

4. 配置SSL并重启


证书管理
密钥库
证书
ssl配置
刷新
开启SSL:
单向认证端口:
7889
客户端认证方式:
双向认证
SSL协议:
TLSv1
双向认证端口:
7887
密钥库:
config/server
密钥库算法:
可信密钥库:
config/trust
可信密钥库算法:
CRL路径:
CRL检查间隔:
60
更新



(2). **不需要证书请求**

1. 同上，另外还需创建一个信任证书库，并将ca.cer导入该证书



2. 其他步骤同上。

22 部分 XII. 第三方监控工具

22.1 Zabbix

Zabbix 是一个企业级的高度集成的开源监控软件，提供了分布式监控解决方案，可以用来监控设备，服务等的可用性和性能。Zabbix 使用灵活的通知机制，允许用户为几乎任何事件配置基于电子邮件的告警，以实现对服务器问题做出快速反应。Zabbix 基于存储的数据提供出色的报告和数据可视化功能。这使得 Zabbix 成为容量规划的理想选择。

22.1.1 安装

Zabbix 安装与部署请请查看其官方文档，这里不做过多赘述。

Zabbix 监控 Apusic应用服务器是基于 JMX，所以还需要安装 Zabbix Java gateway。

22.1.2 配置

1. Zabbix server 配置。

当 Java 网关启动并运行后，你需要告诉 Zabbix server 去哪里找 Zabbix Java 网关。通过在 server 配置文件中指定 JavaGateway 和 JavaGatewayPort 来完成这个操作。如果运行 JMX 应用程序的主机是由 Zabbix 代理监控的，则可以在 proxy 配置文件中指定连接参数。

修改 /etc/zabbix/zabbix_server.conf 配置文件，添加 JavaGateway 相关信息：

```
# Zabbix Java gateway 的 IP 地址（或主机名）。
JavaGateway=192.168.3.14

# Zabbix Java gateway 监听端口。
JavaGatewayPort=10052
```

默认情况下，Zabbix server不会启动任何与JMX监控相关的进程。如果你希望用到它，则必须指定Java pollers的数量。此操作与配置常规 pollers 和 trappers数量一样。

```
# Java 轮询器的 pre-forked 实例数量。
StartJavaPollers=5
```

2. 启动 AAS V9

3. 启动 Zabbix

22.1.3 Zabbix 监控

1. 导入 AAS V9 监控模板

下载Apusic应用服务器监控模板，zabbix6.2以上版本下载AAS V9 监控模板，zabbix6.2以下版本下载AAS V9 监控模板保存到本地。

然后导入到 Zabbix 模板库中。导入模板库的步骤为：依次点击配置 -> 模板 -> 导入按钮，如图所示：

点击选择文件，选择下载的 AAS V9 监控模版，如图所示：

点击导入，导入成功后，如图所示：

2. 创建 Zabbix 主机

如果已有主机，可以直接编辑主机添加JMX端口。创建主机的步骤为：依次点击配置 -> 主机 -> 创建主机，如图所示：

在创建主机界面中，填写主机名；选择主机组；添加 JMX 类型的接口；填写接口 IP 地址或 DNS 名称， 端口；模板输入框中选择步骤1导入的模板如下图所示：。

3. 查看监控数据

添加完主机后则可以在监控中看到监控数据, 如图所示：

点击最新数据，即可查看细节：

23 部分XIII 许可证授权

AAS需要有对应的许可证才能正常使用，通常情况下，金蝶天燕会根据用户购买的产品版本配套对应的许可证。

许可证位置为${APUSIC_HOME}/license.xml。

23.1 特征码获取

想要申请许可证，需要提供产品的特征码。特征码获取方式如下：

1. 可以运行mydomain/bin目录下命令startapusic获取，类似格式:

```
startapusic -ac [ethname or ip]
```

其中[ethname or ip]表示可以输入ip地址或者mac地址的名称，如可以输入常用ip地址172.24.1.116作为参数获取软件特征码(适用windows、linux)：

```
startapusic -ac 172.24.1.116
```

输出格式如下，Auth Code=右边的为软件特征码：

```
Auth Code=SZTY263537626
```

```
[root@l16host bin]# ./startapusic -ac 172.24.1.116
Using APUSIC_HOME:     /home/SP8/v9qiye
Using DOMAIN_HOME:     /home/SP8/v9qiye/domains/mydomain
Using JAVA_HOME:       /opt/java/jdk1.8.0_241
Using APUSIC_ENDORSED_DIRS: /home/SP8/v9qiye/lib/endorsed

Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
Auth Code=SZTY1640356587
```

2. 也可以输入网卡名称获取特征码(适用linux，可以通过ifconfig获取网卡名称)

```
startapusic -ac ens192
```

输出格式如下，Auth Code=右边的为软件特征码：

```
Auth Code=SZTY1640356587
```

```
[root@l16host bin]# ./startapusic -ac eth0
Using APUSIC_HOME:     /home/SP8/v9qiye
Using DOMAIN_HOME:     /home/SP8/v9qiye/domains/mydomain
Using JAVA_HOME:       /opt/java/jdk1.8.0_241
Using APUSIC_ENDORSED_DIRS: /home/SP8/v9qiye/lib/endorsed

Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
Auth Code=SZTY1640356587
```

23.2 提示解读

1. 如果使用了过期的授权文件，会出现如下的提示，提示时间过期以及获取特征码的命令。

```
get auth code:startapusic -ac [ip or ethname]
The server license is out of date
```

2. 如果使用了错误特征码进行启动，会出现如下的提示：

```
get auth code:startapusic -ac [ip or ethname]
not granted for SZTY1640356587 [Apusic Application Server9.0]4
```

其中后面的4为授权文件的version，和原有非kbc授权license的lic_ver标识相同。如果标识和产品名称、产品版本一致，就可以确定是特征码错误。也可以通过 startapusic –ac XX命令来打印特征码对比，XX为申请特征码时使用的ip或网卡名称信息。

3. 如果使用了其他产品的授权文件进行启动，如使用v10企业版的授权文件启动企业版v9.0企会出现如下的提示：

```
get auth code:startapusic -ac [ip or ethname]
not granted for SZTY1640356587 [Apusic Application Server9.0]46
```

提示信息的46表示这个授权是v10企业版的授权文件，[]里面是v9.0版本，所以不能启动

也可以通过startapusic –ac XX打印特征码进行对比特征码，XX为申请特征码时使用的ip或网卡名称信息。

24 部分XIV 常用参数说明及常见问题解决

24.1 Apusic应用服务器常用参数说明

24.1.1 VM.options配置

参数名称	用途说明
apusic.log.blocking.queueSize	设置日志保存队列的长度，如果到达了长度，则会根据属性apusic.log.blocking.queue设置的策略进行处理，默认值为2048。
com.apusic.errorPage.hideServerInfo	设置为true时，在现实错误页面信息时，不包含应用服务的版本等信息，默认值为false。
apusic.cookie.disabledRepeatedSessionID	该参数为true时，则用户的后续响应不再重复写出包含sessionID的cookie信息，默认为false。在与用友财务系统整合时，需要设置为true，否则会出现登陆失败的问题。
apusic.prefer.war.classes.include	表示以列表中的包前缀开头的类依然执行preferWar 规则，即Web应用会优先使用本应用中原有的类，即使其父包前缀已经在apusic.prefer.war.classes.exclude中出现。也就是说该参数用于从apusic.prefer.war.classes.exclude再排除一些类，相当于双重否定，用于精细控制。
apusic.prefer.war.classes.exclude	表示以列表中的包前缀开头的类不执行preferWar 规则，即Web应用会优先使用其父ClassLoader提供的类
apusic.webservice.enabled	设置为true，则表示开启webservice引擎，默认置为false
apusic.strict	是否严格遵循相关规范.可设置的值为true或false,默认为false。
apusic.log.clientIP	是否对每条日志记录记录客户端IP,可设置的值为true或false,默认为false。
apusic.redirect.proxy	请求重定向时，是否将服务器定制修正到代理服务器,可设置的值为true或false,默认为false.
apusic.fixscheme	请求重定向时，是否将http schema修正为https.可设置的值为true或false,默认为false。
com.apusic.jvm.route	用于配置负载均衡参数。
apusic.jsp.param.null.as.string	对标签jsp:param中传入参数值值为空的解决办法,可设置的值为true或false,如果不设置,默认为false.如果为true,则把null值转为字符串"null",否则转换为""。
apusic.prefer.war.classes	可设置的值为true或false,如果不设置,默认为false.如果为true则在应用和Apusic服务器有相同的类是,优先加载应用的类
apusic.enableLookups	可以提升getRemoteHost()方法访问速度,可设置的值为true或false,如果不设置,默认为false.

apusic.usingJbossEjb3JndiRule	为了适应从JBoss移植的应用中,JBoss默认生成JNDI的命名规则的命名规则,可设置的值为true或false,如果不设置,默认为false.
apusic.moduleDiscovery.charset	指定文件.moduleDiscovery的字符集,默认为ISO-8859-1,如果存在中文则需要修改为中文字符集.
apusic.http.request.maxHeaderLine	指定uri的允许长度,默认为8192.
apusic.request.charset.dependonheader	字符编码是否根据header决定,可设置的值为true或false,如果不设置,默认为true.
cookie.useContextRoot	cookie是否包含应用上下文路径.可设置的值为true或false,如果不设置,默认为true.
apusic.connect.anytime	设置服务启动后,是否随时可以进行链接.可设置的值为true或false,如果不设置,默认为false.
apusic.log.blocking.queue	决定采集日志时，当日志队列已满后，是丢弃日志还是在此堵塞等待队列.可设置的值为true或false,如果不设置,默认为false.true表示等待入列，false表示丢弃。
com.apusic.persistence.load	设置JPA是否自动加载,可设置的值为true或false,如果不设置,默认为true.
com.apusic.timerpool.size	值为有效数值,默认值为5
apusic.web.io.type	使用何种IO方式进行传输,可设置的值为 bio, nio。默认是nio。
apusic.jdt.compilerSourceVM	表示如果使用JDT编译对JSP进行编译时,原文件的版本,如果不设置,默认为1.5
apusic.jdt.compilerTargetVM	表示如果使用JDT编译对JSP进行编译时,Class类文件版本,如果不设置,默认为1.5
com.apusic.invocation.InheritContext	InvocationContext 类中有一个成员为threadlocal.使用true之后，初始化时使用InvocationLocal 实现，否则使用ThreadLocal.使用InvocationLocal 实现会继承父线程的信息，默认为false。主要的应用场景是事务边界的确定,如果为false，可能存在资源不受事务管理的情况，回滚失效。
com.apusic.web.moduleHandler	webservice注解发现类，会对class进行分析，是否存在webservice标记，如果没有使用AAS的webservice框架，则可以去掉
javax.persistence.spi.PersistenceProvider	JPA的实现，可以根据实际使用情况修改
apusic.classesloader.useDelegate	是否使用Apusic类加载器代理，默认为false。有些开源框架使用了类加载器作为键值，而apusic使用代理时，类加载器是不变的。
apusic.http.disabledMethods	需要限制的Http请求方法，默认为空。可以限制多个方法，不同的方法使用逗号分隔，如GET,PUT
apusic.http.status.forbiddenToNotFound	是否把状态吗403的提示信息转化为404提示信息，默认为false。主要是出于安全检测软件的需求
apusic.httpwriter.useDefaultEncoding	如果responseEncoding没有指定，是否使用默认编码ISO-8859-1,默认为false，表示使用操作系统编码

apusic.http.doChunkedItself	是否Servlet或者Fitler已经进行了chunked，默认值为false。如果为true而且在响应头包含Transfer-Encoding表示servlet或者fitler已经进行了chunked，aas不再进行chunked处理
com.apusic.web.ServletClassLoaderDelegate	指定ServletClassLoader的两种装载策略，默认为Composite，其他可选值还有Separated。
apusic.http.request.modifiedMap	通过request的getParameterMap能否返回一个可修改的Map，默认为false。
apusic.ajp.encoding	AJP协议编码，默认系统字符编码
apusic.cookie.disabled	是否禁用cookie,默认为false。禁用后则不能进行用户session跟踪。
apusic.cookie.forceEncodeUR	是否强制对URL编码，默认为false。如果为true，则在URL上加入jsessionid.
apusic.web.listener.reload	应用重新加载时，是否重新加载Listener
apusic.security.CheckPasswordComplexity	检查用户密码复杂度，默认为false。如果为true，则在建立新用户时，检查密码是否包含数字，字符大小写等
apusic.security.MinLengthOfPassword	用户密码最少长度
apusic.security.LockoutDuration	锁定时间(分钟)
apusic.security.LockoutEnable	开启用户锁定，如果用户登录失败次数超过设置值，则会锁定设置的时间。
apusic.security.LockoutResetDuration	失败次数重置时间(分钟)
apusic.security.LockoutThreshold	最大失败次数
apusic.transportGuarantee.useMutualAuthPort	如果应用强制使用安全认证登录，而客户使用了普通的端口登录，则重定向到双向认证端口，默认是重定向到单向认证端口
java.naming.factory.initial	JNDI工厂类，默认为：com.apusic.naming.jndi.CNContextFactory
japusic.http.session.serializer	session序列化配置,默认使用jdk方式,可配置的选项有：jdk,fst,kryo.
apusic.http.enable.pre.compression	启用预压缩,可配置的选项有：true,false.
jdbc.tx.auto.complete.decision	在数据源连接释放时，如果一些应用程序没有释放连接时，配置由服务器自动提交或回滚的处理。默认值不处理，可配置值为commit和rollback。
com.apusic.ciphersuite.exclude	可以设置过滤不安全的加密套件，值为加密套件的名称，多个值使用逗号分隔。
com.apusic.cookie.noSecurity	在使用https时cookie默认会设置secure属性，如果设置了该值为true，则不设置secure属性，可取值为true或false，默认值为false.
com.apusic.cookie.securityExclude	可设置在使用https时有哪些cookie不需要设置secure属性，值为cookie的名称，多个cookie名称使用逗号分隔。
com.apusic.cookie.httpOnly	该属性如果设置为true则在所有的cookie上设置httpOnly属性，如果为false，则根据cookie的配置进行设置，可取值为true或false，默认值为false。

24.1.2 web.xml全局配置参数

参数名称	用途说明
com.apusic.web.ServletClassLoaderDelegate	配置Class loader的策略,可以设置为Composite或Separated,默认的策略为Composite。
apusic.el.object.null.representation	可设置的值为true或false,如果不设置,默认为true.如果为false,则el表达式中不存在变量或返回值为null时,求值的结果为"";如果是true则返回null;在v6中,如果是字符串类型,如果值为null,则返回""
apusic.tld.notload.path	过滤不查找TLD文件的目录.如查找TLD文件不查找classes目录.
apusic.prefer.war.classes	可设置的值为true或false,如果不设置,默认为false.如果为true则在应用和apusic服务器有相同的类是,优先加载应用的类
directoryListing	是否列出文件夹下的文件.可设置的值为true或false,如果不设置,默认为false.
sendfileSize	参数sendfileSize,文件大小大于该值并且启用sendFile的情况下文件通过sendFile发送。默认值为49152。
special.symbol.suffix	资源访问路径的特殊后缀,不是实际文件名的一部分,只在URI中存在
request.charsetencoding	对请求的参数进行编码,可以解决中文为乱码的问题
sendErrorToClient	是否把错误堆栈的信息显示到客户端浏览器中.可设置的值为true或false,如果不设置,默认为true.
jsp.nocompile	是否对JSP不进行编译,对JSP预编译的可以设置为true.可设置的值为true或false,如果不设置,默认为false
jsp.keepGenerated	是否保留生成的Java源文件.可设置的值为true或false,如果不设置,默认为true.
jsp.keepOnError	只在发生异常的时候,才删除源文件.可设置的值为true或false,如果不设置,默认为true.
jsp.usePackages	是否Jsp编译后的class保存在不同的包中.可设置的值为true或false,如果不设置,默认为true.
jsp.useDataFile	是否把template text保存在独立的数据文件中.可设置的值为true或false,如果不设置,默认为true.
jsp.sourceLineNumbers	使用源Jsp文件的行号替换类文件的行号列表.可设置的值为true或false,如果不设置,默认为true.
jsp.useCharArray	是否用字符数组保存template text,提供性能.可设置的值为true或false,如果不设置,默认为true.
jsp.checkDependencies	检查所有依赖的文件的修改日期,决定是否进行重新编译.可设置的值为true或false,如果不设置,默认为true.
jsp.encoding	指定Jsp默认的字符编码

jsp.checkModified	检查Tag class, EL expected type, jsp Import class是否在jsp编译前进行了修改.可设置的值为true或false,如果不设置,默认为true.如果为false,则在应用运行时,不检查jsp的更新,否则检查更新
jsp.lookAHeadSize	往下分析多少字节来决定Jsp的编码
jsp.compiler	指定特定的Java编译器来编译生成的Java源文件
jsp.compiler.error.regexp	指定一个正则表单式来分析编译的错误信息

24.1.3 JVM参数

参数名称	用途说明
java.util.logging.manager	指定日志管理器,一般情况下不需要指定,但是在使用如-javaagent:D:aspectjweaver.jar时,需要在其前面指定,否则 apusic启动会出现异常。例如 -Djava.util.logging.manager="com.apusic.logging.manager.ServerLogManager"
com.apusic.web.ServletClassLoaderDelegate	设置Servlet Classloader的装载行为策略,对所有的应用都生效。例如：-Dcom.apusic.web.ServletClassLoaderDelegate=Separated
apusic.interval	设置apusic自动重启间隔时间。例如：-Dapusic.interval=1110 ，表示隔11天的10点自动重启。

在%domian_home%/bin文件夹下的startapusic脚本中配置JVM参数

set JVM_OPTS=-server -Xms128m -Xmx512m -XX:MaxPermSize=128m 若要参数起作用，需用startapusic –p 启动。

24.2 Apusic应用服务器常见问题及解决

24.2.1 解决安全软件检测后出现的问题

1. 解决http方法问题,禁用http方法配置：

在vm.options设置属性apusic.http.disabledMethods，值为要禁用的方法，多个方法用逗号分隔，如：

apusic.http.disabledMethods=DELETE,PUT,TRACE

2. 解决跨站点脚本问题 web.xml中添加下述配置:

```
<!--#################### Attack Filter ###################-->
<filter>
    <display-name>AttackFilter</display-name>
    <filter-name>AttackFilter</filter-name>
    <filter-class>com.apusic.util.AttackFilter</filter-class>
    <init-param>
        <param-name>enableAttackFilter</param-name>
        <param-value>true</param-value>
```

```
    </init-param>
</filter>
<filter-mapping>
    <filter-name>AttackFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

3. 解决在客户端显示错误堆栈问题

修改域下的config/web.xml下的参数sendErrorToClient 修改为false。

4. 解决管理控制台存在安全漏洞问题

建议1：去掉admin的部署。

建议2：修改管理控制台的上下文，不要使用admin，使用如dafadsa之类的，让软件预测不到。

5. 解决显示目录问题

修改域下的config/web.xml下的参数directoryListing 为false

屏蔽403提示,转化为404

在vm.options设置属性apusic.http.status.forbiddenToNotFound为true,例如：

```
apusic.http.status.forbiddenToNotFound=true
```

24.2.2 如何使用AAS进行应用预编译

AAS提供了对应用进行预编译的功能,步骤及说明如下:

1. 在%Apusic_Home%/bin目录下的jspc.cmd及jspc.xml，只是一个例子，是对默认的default应用中模块public_html进行编译
2. 如果需要修改为其他应用，可以修改jspc.xml，里面的格式和ant中的构建脚本的builde.xml写法一致，可以参考 http://ant.apache.org/manual/index.html。
    1. 属性<property name="src" ...> 指定需要编译的应用目录
    2. 属性<property name="dest"...> 指定编译后类路径的保存目录
    3. <classpath></classpath>设置需要的类路径
3. 构建完成后，需要设置应用的web.xml,增加参数

```
<context-param>
    <param-name>jsp.nocompile</param-name>
```

```
    <param-value>true</param-value>
</context-param>
```

设置后,访问应用的jsp时,就不会再进行编译动作,提供响应速度

24.2.3 编码问题

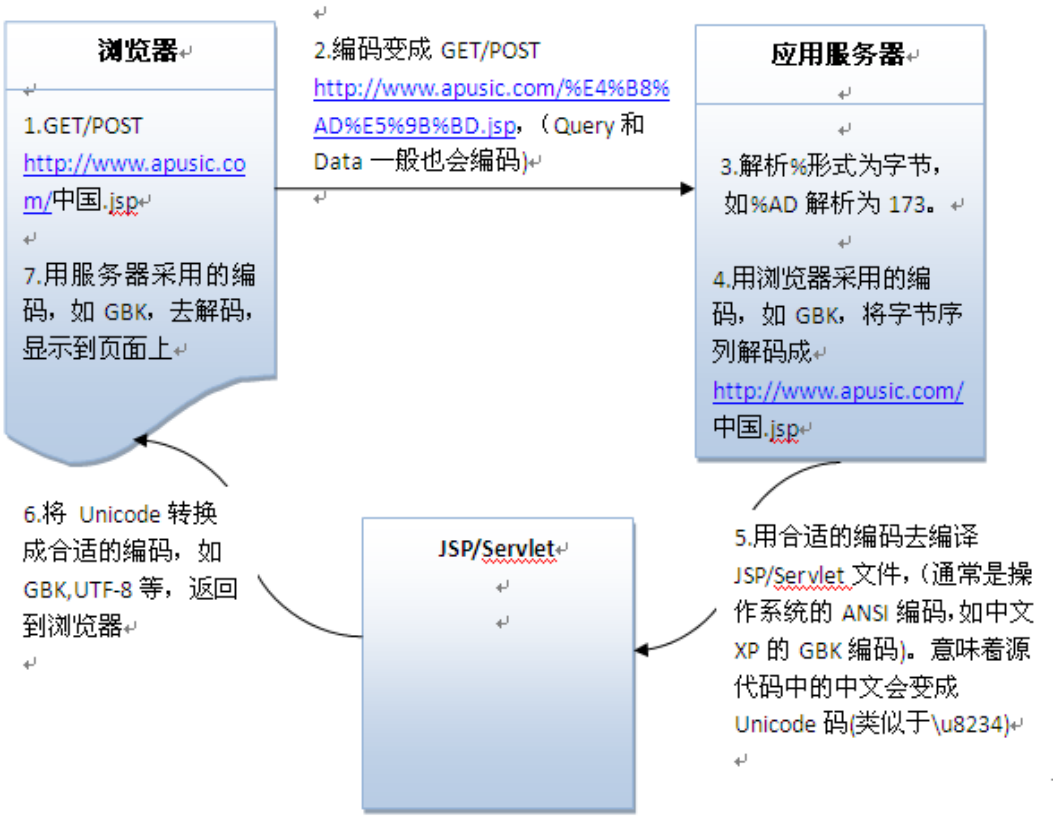
24.2.3.1 基础知识

- HTTP请求中，可能会出现中文的地方

1. Path：如www.apusic.com/中国.jsp
2. Query：如www.apusic.com/index.jsp?key1=汉字，"key1=汉字"就是Query字符串
3. Data：如表单中<input name="test2" type="text" value="中文"/>

	注意
	同一次请求，不同地方的编码格式很有可能是不一样的。

- HTTP请求中的编码转换



- 一般情况下的编码

Path:根据RFC 2732:Format for Literal IPv6 Addresses in URLs，URI（不仅Path，还包括Query）未指定特定的字符集，但实际应用中，Path部分浏览器通常是使用UTF-8编码，然后转成application/x-www-form-urlencoded格式。并且JAVA的URI类的实现也是使用了UTF-8。

Query:一般有两种情况:

一是用户直接在浏览器输入URL或使用超链接方式，那么浏览器会根据它设置的语言编码，如中国一般使用的是GBK。

另一种是用户在页面提交表单的方式，浏览器先会根据请求页面时得到的Context-Type的响应消息头中charset值（比如"text/html;charset=GBK"）去编码。如果没有，则采用设置的语言编码。

	**注意**
	强调一下，是指http响应消息头的Context-Type，而不是在网页中meta中的Context-Type。后者通常情况是不起任何作用的。

Data:一般采用于Query相同的编码

- 非一般情况下的编码

这种特殊情况的出现一般在以下场景：

1. 用户自己为汉字编码。那么可能出现Path使用GBK编码，Query与Data编码不一致等各种情况。
2. 用户自己编写客户端，并且不遵循规范或惯例
3. 浏览器不遵循规范或惯例

24.2.3.2 常见编码问题解析

- JSP/servlet中解析form提交的汉字得到的是乱码

原因：

这种情况出现的可能性很多，如用户自己加了编码过滤器等，但根本原因是服务器解析Query和Data的参数时，没有指定对编码。

解决方法：

1. 使用request.getParameter()前，先用request.setCharacterEncoding()指定编码，Apusic会根据设定的编码格式去编码Query和Data，（Tomcat默认只是将这个值作为Query的编码）。最佳实践为写一个SerlvetFilter用这个方法统一设置编码格式。
2. new String("中文".getBytes("ISO-8859-1"),YOUR_ENCODING);

Tips:这里使用ISO-8859-1，并不是因为HTTP协议采用ISO-8859-1，而是因为ISO-8859-1是单字节编码，用它去getBtyes()，可以将不进行编码地将字节取出。

3. Apusic特有设置：见Apusic的解决方案一节

- 浏览器中看到的 Jsp/Servlet 页面出现乱码

原因：

可能有几种情况：

1. request.getParameter()获得的值本身就没正确编码。
2. 使用了不合适的编码，如用ISO-8859-1去编码汉字。
3. 响应头中没有Context-Type或Context-Type没有指定编码，浏览器不知道用何种编码显示页面。

解决方法：

1. response.setContentType("text/html;GBK"); response.setCharacterEncoding ("UTF-8"); response.setLocale(Locale.CHINESE);这三个方法可以设置Context-Type中的编码。
2. Apusic特有设置：见Apusic的解决方案一节

- 中文路径无法访问

原因：

应用服务器没有判断对Path的编码。如果Path采用UTF-8一般不会出现这种情况，而如果是其他编码格式，比如用户自己编码了，那么就可能出现找不到路径的情况。

解决方法：

1. 采用英文路径。
2. Apusic特有设置：见Apusic的解决方案一节

- JSP源代码中的中文，编译后显示成乱码

原因：

JSP源文件采用的GBK编码,而编码时却指定UTF-8编码去编译。

解决方法：

1. 指定编译使用的编码。指定的方式有两种，一种是在JSP页面的首行，加上<%@ page contentType="text/html;charset=GBK" pageEncoding="GBK" %>，其中pageEncoding指定的就是编码时采用的编码。另一种是在web.xml设置，类似于

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>/*</url-pattern>
        <page-encoding>GBK</page-encoding>
    </jsp-property-group>
</jsp-config>
```

24.2.3.3 Apusic的解决方案

- Path和Query出现乱码

可以设置apusic.conf 下的webservice服务，增加属性URIEncoding，取值根据实际情况选择。如下选择了GBK的配置:

```
<ATTRIBUTE NAME="URIEncoding" VALUE="GBK" />
```

- 响应的设置

如果表单中通过Post方式提交的中文数据存在乱码，则可以在应用的web.xml里设置参数request.charsetencoding，取值根据实际情况选择。例子如下:

```
<context-param>
    <param-name>request.charsetencoding</param-name>
    <param-value>GBK</param-value>
</context-param>
```

如果表单数据通过get方式提交，则其内容的编码和Path及Query内容编码处理一致。

24.2.4 多个应用共享session

在实际应用中，可能需要在多个应用中共享Session数据，例如实现单点登录等。同域中的应用若需实现共享Session，则需要在Server.xml中配置如下属性：

base-context：应用的基础上下文，需要共享session的应用设置相同的base-context。

global-session：是否为全局session，此处应设为true。在此举例说明：

配置两个应用test1，test2共享session

1. 配置server.xml使用相同的base-context，设置global-session为true,具体配置如下：

```
<application name="test1" base="applications/test1" base-context="/test"
global-session="true" start="auto"/>
<application name="test2" base="applications/test2" base-context="/test"
global-session="true" start="auto"/>
```

2. 设置每一个应用里面的apusic-application.xml，增加context-root 的属性配置，如果没有配置，则使用base-context，会导致contextroot冲突。如上面的例子。

test1的配置：

```
<apusic-application>
    <module uri="">
        <web>
            <context-root>/</context-root>
```

```
        </web>
    </module>
</apusic-application>
```

test2的配置:

```
<apusic-application>
    <module uri="">
        <web>
            <context-root>/test2</context-root>
        </web>
    </module>
</apusic-application>
```

3. 测试，在test1应用有一个index.jsp页面，设置一个sesion值，然后定向到/test2应用的getsession.jsp页面，并获取test1设置的值，获取成功。

index.jsp页面大致如下:

```
<%

session.setAttribute("test1","你好吗");

System.out.println("access index.jsp");

%>
    <a href="<%=request.getContextPath()%>/test2/getSession.jsp">test2同一个session</a>
</td>
```

getsession.jsp页面大致如下:

```
<%

System.out.println("access get Session");

out.print("session value:"+session.getAttribute("test1"));

%>
```

24.2.5 Spring的nativeJdbcExtractor问题

描述：没有提供spring的nativeJdbcExtractor实现导致在oracle数据库中插入blob类型数据时出现异常

```
com.ufgov.gk.common.system.exception.OtherException: SqlMapClient operation;
uncategorized SQLException for SQL []; SQL state [null];
error code [0];

- The error occurred in sqlmap/AsFile.xml.

- The error occurred while applying a parameter map.

- Check the AsFile.insertAsFile-InlineParameterMap.

- Check the parameter mapping for the 'fileContent' property.

-Cause: org.springframework.dao.InvalidDataAccessApiUsageException:

OracleLobCreator needs to work on [oracle.jdbc.OracleConnection], not on
[com.apusic.jdbc.adapter.ConnectionHandle]:specify a corresponding
NativeJdbcExtractor;

nested exception is java.lang.ClassCastException:
com.apusic.jdbc.adapter.ConnectionHandle cannot be cast to
oracle.jdbc.OracleConnection;

nested exception is com.ibatis.common.jdbc.exception.NestedSQLException:
```

检查数据库发现As_file表中file_content字段是一个blob字段，报错信息提示:

```
OracleLobCreator needs to work on [oracle.jdbc.OracleConnection], not on
[com.apusic.jdbc.adapter.ConnectionHandle]
com.apusic.jdbc.adapter.ConnectionHandle cannot be cast to
oracle.jdbc.OracleConnection
```

问题原因：这个问题在spring框架中比较常见。对于blob和clob的字段存储，spring的NativeJdbcExtractor会根据数据源和AS类型的不同，提供不通的API接口，从oracle的本地jdbc的jar报包中抽取对象类，spring针对常用的应用服务器都有对应的NativeJdbcExtractor接口（如webshpere、weblogic jboss等），但是在apusic中并未提供处理接口。

解决方法：nativeJdbcExtractor是需要根据不同的应用服务器实现,因此需要apusic应用服务器提供自己的实现类，具体如下：

可以编译ApusicNativeJdbcExtractor.java，此类是apusic对nativeJdbcExtractor的实现，具体参开代码如下：

```
package org.springframework.jdbc.support.nativejdbc;

import java.lang.reflect.Method;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import org.springframework.util.ReflectionUtils;

/**
 * Implementation of the {@link NativeJdbcExtractor} interface for Apusic,
 *
 * <p>
 * supporting Apusic Application Server 6.0+.
 *
 * <p>
 * Returns the underlying native Connection, Statement, etc to application
code instead of Apusic' wrapper
 * implementations.
 *
 * <p>
 * The returned JDBC classes can then safely be cast, e.g. to
<code>oracle.jdbc.OracleConnection
 * </code>.
 *
 * <p>
 * This NativeJdbcExtractor can be set just to <i>allow</i> working with a
Apusic connection pool: If a given object is
 * not a Apusic wrapper,
 *
 * <p>
 * it will be returned as-is.
 *
 * @author weiyongsen
 *
 * @since 07.08.2011
```

```
 *
 * @see com.apusic.jdbc.adapter.ConnectionHandle#getActualConnection
 * @see com.apusic.jdbc.adapter.StatementHandle#getActualStatement
 * @see com.apusic.jdbc.adapter.ResultSetHandle#getActualResultSet
 */
public class ApusicNativeJdbcExtractor extends NativeJdbcExtractorAdapter {
  private static final String WRAPPED_CONNECTION_NAME =
"com.apusic.jdbc.adapter.ConnectionHandle";
  private static final String WRAPPED_STATEMENT_NAME =
"com.apusic.jdbc.adapter.StatementHandle";
  private static final String WRAPPED_RESULT_SET_NAME =
"com.apusic.jdbc.adapter.ResultSetHandle";
  private Class wrappedConnectionClass;
  private Class wrappedStatementClass;
  private Class wrappedResultSetClass;
  private Method getUnderlyingConnectionMethod;
  private Method getUnderlyingStatementMethod;
  private Method getUnderlyingResultSetMethod;

  public ApusicNativeJdbcExtractor() {
    try {
      this.wrappedConnectionClass =
getClass().getClassLoader().loadClass(WRAPPED_CONNECTION_NAME);
      this.wrappedStatementClass =
getClass().getClassLoader().loadClass(WRAPPED_STATEMENT_NAME);
      this.wrappedResultSetClass =
getClass().getClassLoader().loadClass(WRAPPED_RESULT_SET_NAME);
      this.getUnderlyingConnectionMethod =
this.wrappedConnectionClass.getMethod("getActualConnection", (Class[]) null);
      this.getUnderlyingStatementMethod =
this.wrappedStatementClass.getMethod("getActualStatement", (Class[]) null);
      this.getUnderlyingResultSetMethod =
this.wrappedResultSetClass.getMethod("getActualResultSet", (Class[]) null);
    } catch (Exception ex) {
      throw new IllegalStateException("Could not initialize
ApusicNativeJdbcExtractor because Apusic API classes are not available: " +
ex);
    }
  }
```

```
  protected Connection doGetNativeConnection(Connection con) throws
SQLException {
    if (this.wrappedConnectionClass.isAssignableFrom(con.getClass())) {
      return (Connection)
ReflectionUtils.invokeMethod(this.getUnderlyingConnectionMethod, con);
    }
    return con;
  }

  public Statement getNativeStatement(Statement stmt) throws SQLException {
    if (this.wrappedStatementClass.isAssignableFrom(stmt.getClass())) {
      return (Statement)
ReflectionUtils.invokeMethod(this.getUnderlyingStatementMethod, stmt);
    }
    return stmt;
  }

  public PreparedStatement getNativePreparedStatement(PreparedStatement ps)
throws SQLException {
    return (PreparedStatement) getNativeStatement(ps);
  }

  public CallableStatement getNativeCallableStatement(CallableStatement cs)
throws SQLException {
    return (CallableStatement) getNativeStatement(cs);
  }

  public ResultSet getNativeResultSet(ResultSet rs) throws SQLException {
    if (this.wrappedResultSetClass.isAssignableFrom(rs.getClass())) {
      return (ResultSet)
ReflectionUtils.invokeMethod(this.getUnderlyingResultSetMethod, rs);
    }
    return rs;
  }
}
```

放到应用目录的类目录，修改应用对应的nativeJdbcExtractor配置即可。

24.2.6 管理应用运行在独立端口配置的问题

部分客户需要把管理应用和业务应用分开部署在不同的端口上，现在介绍把管理应用独立部署在6886端口上需要修改的配置。

1. 在apusic.conf配置一个tcp服务，端口属性如下：

```
<SERVICE CLASS="com.apusic.web.http.tcp.TCPEndpoint"
NAME="Endpoint:type=tcp,service=httpd">
    <ATTRIBUTE NAME="Port" VALUE="8080" />
    <ATTRIBUTE NAME="Backlog" VALUE="50" />
    <ATTRIBUTE NAME="Timeout" VALUE="300" />
</SERVICE>
```

2. 把管理应用webtool.ear从%APUSIC_HOME%/lib目录中剪切到另外一个路径中，如%APUSIC_HOME%/admin。由于webtool.ear在%APUSIC_HOME%/lib下，启动时会自动部署，所以需要它先删除掉，以免冲突。

3. 在server.xml配置管理应用，如下：

```
<application name="admin" base="%APUSIC_HOME%/admin/webtool.ear " virtual-
host="192.168.12.114:8080,localhost:8080" start="auto"/>
```

4. 重新启动后，只能通过192.168.12.114:8080或者localhost:8080才能访问管理应用。

24.2.7 AIX系统下ORBSingleton问题

在AIX系统下，启动Apusic应用服务器时报如下异常：

com.ibm.rmi.corba.ORBSingleton incompatible with com.sun.corba.se.spi.orb.ORB

解决办法：

方案一：在启动脚本startapusic中设置

```
-Dorg.omg.CORBA.ORBSingletonClass=com.sun.corba.se.impl.orb.ORBSingleton
```

方案二：修改ORBfactory为Apusic的实现，在启动脚本startapusic增加如下设置

```
-Dcom.apusic.corba.ORBFactory=com.apusic.corba.plugin.ee.ORBFactoryImpl
```

24.2.8 应用服务器启动慢的问题

如果在应用服务器的时候发现很慢，则可能存在如下原因：

1. 在AASV8.0中，使用了新的管理控制台，在应用服务器启动时会自动部署，由于管理控制台应用文件比较大，所以在解压部署时需要的时间比较长。

2. 建立的数据源所需要的数据库没有启动或由于网络原来连接不上，应用服务器启动时会尝试去连接数据库，尝试超时后，初始化数据源失败，才会继续启动其它服务。

3. 其它自动部署的应用(放在%DOMAIN_HOME%/applications目录下的应用)比较大，导致解压部署时需要比较长的时间。

24.2.9 配置使用openJPA

当前AAS版本支持JPA2.0规范，默认使用eclipselink实现，按下面配置即可在AAS中使用openJPA实现：

1. 把openJPA相关的jar包拷贝到%apusic_home%/lib/ext目录下
2. 在%domain_home%/config/vm.options文件中修改属性javax.persistence.spi.PersistenceProvider，如下：

```
javax.persistence.spi.PersistenceProvider=org.apache.openjpa.persistence.Persis
```

24.2.10 GemFile与Apusic集成

GemFire 是一个在分布式云架构中提供实时，一致性接入数据密集应用的数据管理平台。GemFire池化内存，CPU，网络资源和本地硬盘来管理应用对象和行为。它使用动态复制和数据分块技术来实现高可用，改善性能，高扩展性和容错能力。GemFire不仅仅是一个分布式数据中心，还是提供可靠的异步事件和保证信息传输的内存数据管理系统。

GemFire为应用提供3种部署架构：peer to peer, client/server, 和 WAN gateway。

Apusic Application Server 暂时只提供client/server的架构方式存储Session。

24.2.10.1 安装配置GemFile

1. 从GemFile官网下载安装包后解压缩
2. 设置系统环境变量，除了Java环境变量外，还要设置gemfile的运行变量，以windows为例，如下所示：

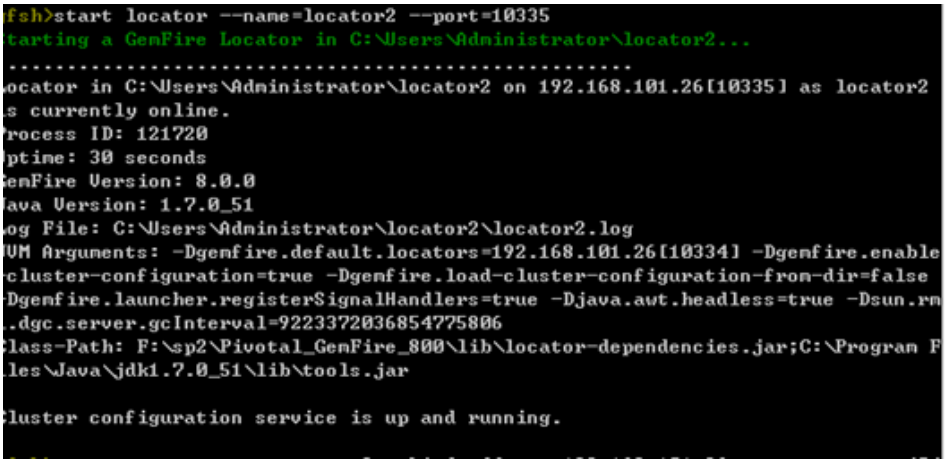
```
GEMFILE= F:\\Pivotal_GemFire_800
GF_JAVA= %JAVA_HOME%\\bin\\java.exe
Path= ;%GEMFILE%\\bin;%JAVA_HOME%\\bin;%JAVA_HOME%\\jre\\bin;
```

3. 启动GemFile

a. 进入到gemfile安装目录的bin目录下,运行gfsh命令进入到gfsh界面下。

b. 启动locator。

在gfsh下执行：`start locator --name=locator2 --port=10335`，其中name和port可以自定义，端口号在apusic配置中要使用。成功创建后如下所示：

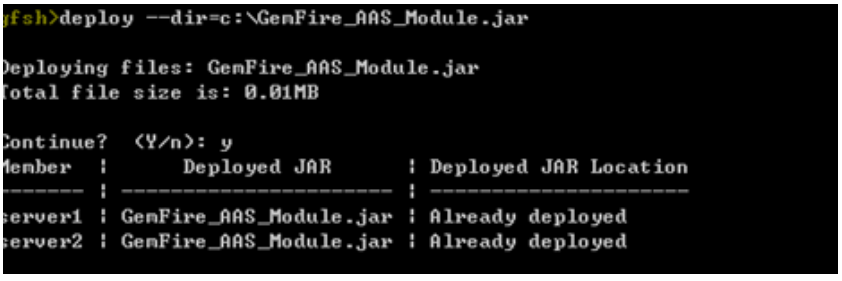


c. 启动server。

在启动locator后，进一步启动server,示例如下： `start server --name=server2 --bind-address=192.168.101.26 --server-port=40405 --locators=192.168.101.26[10335]`，其中，name为server名称，bind-address为server端所在服务器地址，server-port为server监听端口，locators为第二步中创建的locator，端口为10335。



d. 部署GemFile_AAS_Module.jar 在gfsh命令行下执行部署命令，例如：deploy --dir=c:\GemFire_AAS_Module.jar,部署完毕后如下所示，将在每一个server下均部署GemFile_AAS_Module.jar,如下图所示：



24.2.10.2 配置apusic应用服务器

1. 配置session存储

进入到apusic目录下，在AAS9\domains\mydomain\config下修改apusic.conf文件。修改session存储服务如下：

```
<SERVICE CLASS="com.apusic.web.session.GemFireSessionStoreService">
    <ATTRIBUTE NAME="CacheXmlFile" VALUE="config/cache-client.xml" />
    <ATTRIBUTE NAME="RegionName" VALUE="AAS_SESSIONS" />
    <ATTRIBUTE NAME="Refid" VALUE="PARTITION_REDUNDANT" />
    <ATTRIBUTE NAME="EnableLocalCache" VALUE="True" />
    <ATTRIBUTE NAME="LogFile" VALUE="logs/gemfire.log " />
</SERVICE>
```

属性说明:

- CacheXmlFile：GemFire的配置文件路径。
- RegionName: GemFire中使用的Region的名字。AAS将使用此名字查找服务器端是否创建此Region，若无，则将创建此名字的Region。默认为：AAS_SESSIONS。
- Refid：创建Region时的参数。参考GemFire中的Region Shortcuts。默认为：REPLICATE_PERSISTENT_OVERFLOW。
- EnableLocalCache：是否允许本地缓存, 默认为: True。注: 只有在允许本地缓存的情况下,才能回调Session的监听的sessionDestroyed事件。
- LogFile: GemFire的日志输出文件。默认为：logs/gemfire.log。

2. 新增配置GemFire的配置文件

在AAS9\domains\mydomain\config下新增配置GemFire的配置文件cache-client.xml，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE client-cache PUBLIC
gemStone Systems, Inc.//GemFire Declarative Caching 8.0//EN"

"http://www.gemstone.com/dtd/cache8_0.dtd">
<client-cache>
    <!-- To configure the client to use a locator instead of a server,modify
the locator host and port as necessary. -->
    <pool name="sessions" subscription-enabled="true">
        <locator host="localhost" port="10335" />
    </pool>
</client-cache>
```

3. 将GemFire_AAS.jar拷贝至apusic_home的lib目录下，例如：AAS9\domains\lib。

24.2.10.3 启动apusic应用服务器

配置完毕后，进入到APUSIC_HOME\domains\mydomain\bin目录下，运行startapusic启动apusic应用服务器即可。

24.2.11 应用程序设置停止时销毁应用创建的线程池、定时器

在context.xml中添加

```
<Context clearReferencesStopThreads="true"
clearReferencesStopTimerThreads="true"
clearReferencesHttpClientKeepAliveThread="true"></Context>
```

24.2.12 开启IPV6

在启动脚本，如startapusic中去掉`-Djava.net.preferIPv4Stack=true`，保存后启动。再使用IPv6地址访问即可。

24.2.13 同一域下创建多个监听器，使应用可以部署到不同的端口上

1、在${DOMAIN_HOME}/application/config/apusic.config 中添加监听器。

注意得加在配置文件较后位置，否则服务器启动时会出现空指针异常。

内容如下：

```
<SERVICE        CLASS="com.apusic.web.http.tcp.TCPEndpoint"
NAME="Endpoint:type=tcp,service=httpd">
    <ATTRIBUTE NAME="Port" VALUE="8080" />
    <ATTRIBUTE NAME="Backlog" VALUE="50" />
<ATTRIBUTE NAME="Timeout" VALUE="300" />
</SERVICE>
```

添加多个监听器需要在`NAME`属性后面添加任意参数名区分，例如 a=1，如下：

```
<SERVICE        CLASS="com.apusic.web.http.tcp.TCPEndpoint"
NAME="Endpoint:type=tcp,service=httpd,a=1">
    <ATTRIBUTE NAME="Port" VALUE="8080" />
    <ATTRIBUTE NAME="Backlog" VALUE="50" />
<ATTRIBUTE NAME="Timeout" VALUE="300" />
</SERVICE>

<SERVICE        CLASS="com.apusic.web.http.tcp.TCPEndpoint"
NAME="Endpoint:type=tcp,service=httpd,a=2">
    <ATTRIBUTE NAME="Port" VALUE="8081" />
    <ATTRIBUTE NAME="Backlog" VALUE="50" />
<ATTRIBUTE NAME="Timeout" VALUE="300" />
</SERVICE>
```

2、配置完成后，启动AAS；此时每个端口可以访问。

3、部署应用时，设置"**虚拟主机**"为 IP+端口 或 虚拟主机名+端口；没有配置虚拟主机参数默认所有的监听端口都能访问该应用。

24.2.14 配置CORS（跨域资源共享）

在应用的 `web.xml` 或者 `default-web.xml` 设置如下内容：

```
 <filter>
        <filter-name>CORS</filter-name>
        <filter-class>com.apusic.util.filters.CorsFilter</filter-class>
        <init-param>
            <param-name>cors.allowed.origins</param-name>
            <param-value>*</param-value>
        </init-param>
        <init-param>
            <param-name>cors.allowed.methods</param-name>
            <param-value>GET,POST,HEAD,PUT,DELETE,OPTIONS</param-value>
        </init-param>
        <init-param>
            <param-name>cors.allowed.headers</param-name>
            <param-value>Accept,Authorization,Cache-Control,Content-
Type,DNT,If-Modified-Since,Keep-Alive,Origin,User-Agent,X-Mx-ReqToken,X-
Requested-With</param-value>
        </init-param>
        <init-param>
            <param-name>cors.exposed.headers</param-name>
            <param-value>Set-Cookie</param-value>
        </init-param>
        <init-param>
            <param-name>cors.support.credentials</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>CORS</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

使命

推动民族软件创新与发展

愿景

数字政府服务领航者

价值观

致良知、走正道、行王道

Apusic 金蝶天燕

深圳金蝶天燕云计算股份有限公司

电话：400-555-800

网站：www.apusic.com

北京总部：北京市顺义区金蝶软件园　深圳总部：深圳市南山区金蝶软件园　上海总部：上海市浦东区金蝶软件园

全国统一服务热线

4008-555-800



金蝶天燕云计算股份有限公司(简称“金蝶天燕云”)成立于2000年，前身为“金蝶中间件公司”，是金蝶集团旗下新一代软件基础云平台服务商，云计算国家标准制定企业，国家信创产业核心软件企业。金蝶天燕是国家863重点研发计划与核高基重大专项承接企业，也是“两网一站四库十二金”国家重点工程的基础平台提供商，产品广泛应用于政府、军工、金融、能源等关键行业，累计服务客户总数超过10万家。

Apusic 金蝶天燕

云计算国家标准制定企业

金蝶集团旗下基础软件企业

信息技术应用创新核心企业

官网：www.apusic.com