PKUAS 2010 技术手册

北京大学

PKUAS 2010 技术手册

北京大学

目录

插图清单

表格清单

范例清单

前言

1. 关于JEE应用服务器

应用服务器是一种为方便应用软件开发与维护而逐步形成的基础软件，它是网络环境中应用系统的高层运行平台，其主要目的是使应用系统的代码更为简单，使开发人员的精力可以更加集中于系统的逻辑部分。

JEE应用服务器是指符合J2EE or Java Enterprise Edition 5规范的应用服务器。

2. 关于PKUAS

构件运行支撑平台PKUAS是由北京大学信息科学技术学院软件研究所自行设计开发，兼容Java EE 5(Java Platform, Enterprise Edition 5)规范、面向Internet的 开放式构件运行支撑平台。PKUAS在具有全面功能的同时，易于配置、管理，对资源要求低而运行效率高，具有出色的性价比特征，非常适合中小企业、政府部门等。PKUAS具有开放的构件化体系结构，具有强大的互操作能力和出色的定制与扩展能力，可以为企业量体裁衣，方便灵活地构造面向特定领域的应用中间件。同时，PKUAS最大限度为用户着想，能够在不中断系统运行的情况下对应用系统进行在线演化，保护关键业务的进行，具备下一代中间件的领先特征。

PKUAS兼容Java EE 5规范和Enterprise JavaBean3.0规范中的核心内容， 全面支持基于J2EE/EJB体系结构的应用， 已经通过了J2EE蓝图程序Java PetStore（JPS） （1.3版本） 、J2EE性能基准测试程序ECperf(1.0版本)和ObjectWeb发布的开源Jonas兼容性测试集的测试。

PKUAS占用资源少，对系统硬件配置要求低，可在奔腾166MHz、48兆内存、200M空闲磁盘空间的机器上正常工作。PKUAS支持多种操作系统：Windows 2000, Windows XP，Windows Server 2003, Windows Vista, Win 7和Unix/Linux系统。PKUAS要求Java SE 1.6或以上版本支持。

3. 关于本书

本书面向PKUAS的用户，也可以供PKUAS应用服务器开发人员参考。

本书介绍了PKUAS的设计和实现，目的是便于用户了解和使用PKUAS应用服务器。书中对所涉及到的JEE相关内容不作具体介绍，这方面内容请参看SUN公司有关文档。

本书一共分为十六章。第一章引言，第二章介绍PKUAS的安装与配置，第三至十四章分别从体系结构、容器与服务等方面介绍PKUAS的设计和实现，第十五章是基于PKUAS的应用系统开发指南，第十六章列出了本书的参考资料。

第 1 章 安装与配置

1.1. 目标与需求

本节描述了如何安装和配置PKUAS的步骤。如果安装配置有问题，请访问在线文档 http://dev.sei.pku.edu.cn/trac/pkuas 。

1.2. 获取系统

目前PKUAS的稳定版本是0.2，可以从 http://dev.sei.pku.edu.cn/trac/pkuas/wiki/downloads 获得可运行版本和源码。

用户也可以从SVN地址 http://dev.sei.pku.edu.cn/svn/pkuas/trunk 获取PKUAS的最新源码。

1.3. 系统要求

PKUAS的运行要求JDK版本在1.6或以上，并正确设置 JAVA_HOME 环境变量。

1.4. 编译

编译步骤针对使用PKUAS源码的用户，如果用户获取的是可运行版本，可直接进入安装步骤。

PKUAS的源码使用Maven管理。可以从 http://maven.apache.org/download.html 下载最近发布的Maven二进制版本（本文使用的版本为2.2.1），解压到适当位置，设置环境变量MAVEN_HOME为解压路径、Path=%MAVEN_HOME%/bin;%Path%。

获得PKUAS的源码后，进入PKUAS项目根目录pkuas/trunk目录，可以找到一个pom.xml文件，该文件是基于Maven的PKUAS项目的顶级项目描述文件。

在PKUAS项目根目录pkuas/trunk中，键入命令mvn install，开始编译PKUAS项目（若要清除之前编译生成的结果，可用命令mvn clean install）。由于PKUAS依赖的包较多，首次编译PKUAS时Maven需要从网络下载PKUAS依赖的包，编译过程可能需要较长时间。

完成上面的过程后，检查pkuas/trunk/assemblies/pkuas-dist/target/pkuas-dist-2010-bin.dir目录，此时PKUAS运行版本的全部目录和文件都已经在该目录下生成。

1.5. 安装

如果用户使用可运行版本，直接解压即可。如果用户手动编译PKUAS，可以将Maven生成的PKUAS可运行版本，即pkuas/trunk/assemblies/pkuas-dist/target/pkuas-dist-2010-bin.dir目录下的所有内容拷贝到文件系统上一个合适的位置。下文解压路径或可运行版本所在路径称为PKUAS_HOME。

1.6. 配置

用户可以在在etc/pkuas.xml文件中配置pkuas的各种信息。

1.6.1. 配置内核属性

在Kernel->Properties中，设置系统属性，比较重要的属性是pkuas.localhost，该属性决定了pkuas绑定的IP地址。缺省为127.0.0.1。Kernel-Properties中的所有属性都通过System.setProperty被设置到系统环境变量中。

1.6.2. 配置数据源

PKUAS启动时将相应地启动在etc/pkuas.xml文件中配置的数据源服务。在etc/pkuas.xml文件中配置的每个数据源都对应一个Service标签。配置数据源的Service标签遵从下面的格式。

```
<Service Class="pku.as.datasvc.PoolDataSource" Name="your datasource JNDI name">
  <Attribute Name="expirationTime" Value="a positive integer value which specifies
  <Attribute Name="minCapacity" Value="a positive integer value which specifies th
  <Attribute Name="maxCapacity" Value="a positive integer value which specifies th
  <Attribute Name="user" Value="user name used to connect DB" />
  <Attribute Name="password" Value="password used to connect DB" />
  <Attribute Name="URL" Value="URL string used by the JDBC client to connect DB" /
  <Attribute Name="driverClassName" Value="JDBC driver class name" />
</Service>
```

Service标签的Class属性指定数据源实现类的类名，Name属性指定所配置数据源的JNDI名称；expirationTime指定一个数据库连接不使用时保持打开的最长时间（以秒为单位）；minCapacity指定打开的最少连接数；maxCapacity指定打开的最多连接数；user指定连接底层数据库时使用的用户名；password指定连接底层数据库时使用的密码；URL指定连接底层数据库时使用的URL；driverClassName指定连接底层数据库时使用的JDBC驱动程序类名。

PKUAS附带了一个示例应用程序：petstore。要正常运行petstore应用程序，需要在数据库中为之创建必需的数据库对象。petstore被配置成使用数据源jdbc/pkuas，这在PKUAS中对应连接到内置数据库Derby的数据源。附件提供了在内置数据库Derby中为petstore应用程序创建数据库对象和插入初始记录的SQL脚本，在启动PKUAS前将这两个脚本文件放到var/db/sql目录下，PKUAS启动时会自动在内置数据库中执行这两个脚本。

1.7. 启动

在命令行中进入目录%PKUAS_HOME%/bin，在Windows系统中运行run.bat（Linux系统中运行run.sh）即可启动PKUAS。

启动完毕后，在浏览器中输入 http://localhost:8080/petstore，如果PKUAS安装正确，在浏览器中就能看到示例程序petstore的首页了。

1.8. 部署应用程序

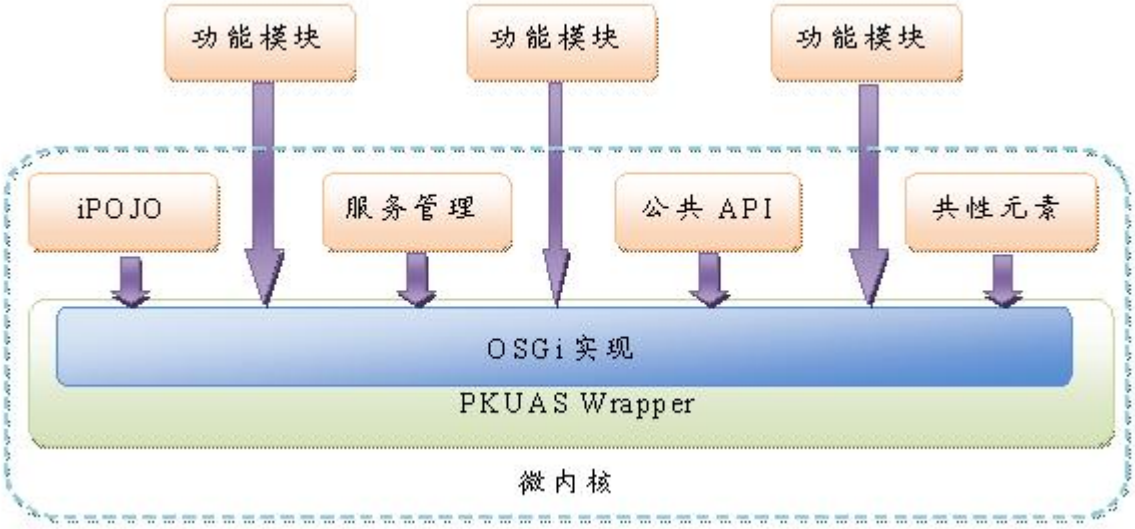
PKUAS同时支持启动前部署和运行中部署。

部置应用程序时，只需把需要部署的WAR、JAR或EAR文件拷贝到%PKUAS_HOME%/application目录下，即可完成应用程序的部署工作。

第 2 章 体系架构

2.1. 目标与需求

2.2. PKUAS体系结构概述



PKUAS的体系结构如上图所示，采用“微内核+功能模块”的结构。下面分别介绍：

- 微内核：PKUAS的微内核采用符合OSGi Release 4规范的内核实现Felix。运行在微内核上的每一个构件模块称为一个Bundle。微内核负责管理Bundle的生命周期，包括运行时刻的动态安装、卸载、启动和停止等。Felix实现之上还包含以下几个属于微内核的模块（以下模块同时也作为Bundle部署在OSGi平台上）：

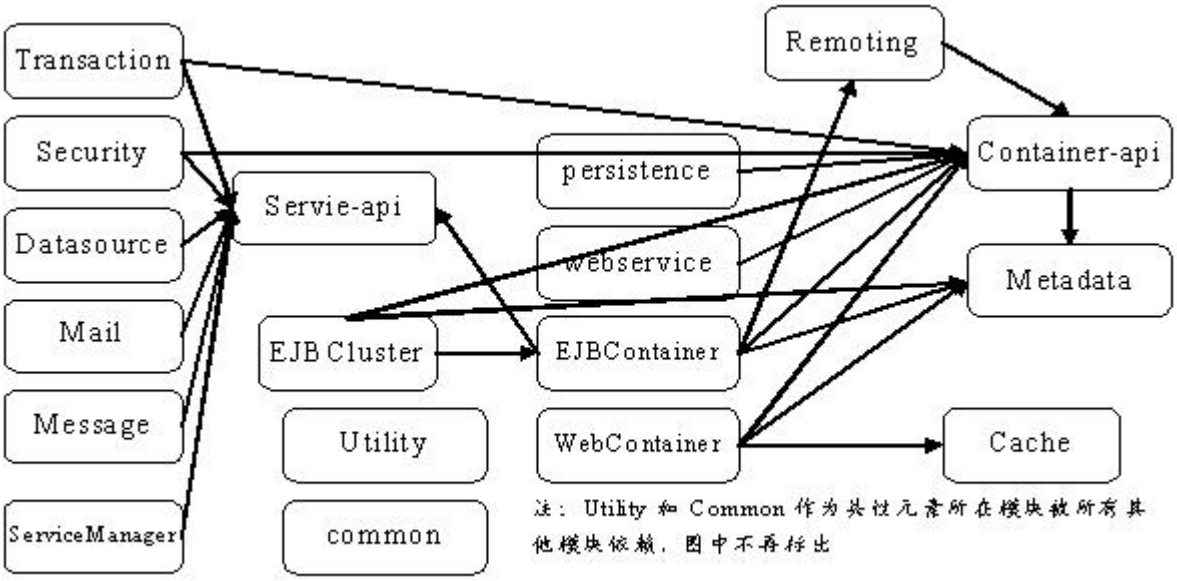
  1. iPOJO：iPOJO是一个基于OSGi的服务构件模型，其主要作用是简化OSGi的Bundle开发。它仅仅通过一个metadata.xml描述文件，就可以实现Bundle的生命周期和动态依赖管理。

  2. 服务管理：该模块完成被管理对象类的管理和配置功能。其中，被管理对象类是指每个功能模块中实现了被管理接口、可以被微内核管理生命周期的类。服务管理与iPOJO交互实现管理和配置功能。

  3. 公共API：这个模块实际上是PKUAS的SPI(Service Provider Interface)。它抽取了所有模块接口类的interface，并集合在一起形成了公共API模块。这样，模块之间通过依赖注入实现了松散耦合。

  4. 共性元素：主要包括日志、对象序列化等所有模块的AOP(Aspect Oriented Programming)元素，PKUAS公共类及被多个模块使用的第三方包。

- 功能模块：每一个PKUAS功能模块作为一个OSGi Bundle，实现不同的功能。PKUAS划分的模块及模块之间的相互关系如下图所示：



其中的功能模块主要分为三类：

1.容器系统：容器是构建运行时所处的空间，负责构件的生命周期管理以及构件运行需要的上下文管理，同时为系统的非功能性约束（例如安全、事务等）提供服务接入点。其中抽取的接口形成Container-api模块。

2.服务：实现系统中与业务无关的功能和约束，如通信、安全、事务等。PKUAS的服务可以通过微内核动态的增加、替换和删除，具有很强的可定制性和灵活性。具体说来，PKUAS采用了一种松耦合的服务集成框架，通过引入适配器模型和反射机制，容器系统的实现仅仅依赖于服务抽象出来的Service-api，而用ServiceInfo文件描述具体装载哪个服务实现，从而实现容器系统和服务实现的演化相互独立。

3.共性元素：提供被多个模块依赖的功能。

2.3. 应用服务器启动过程

1.pku.as.launcher.felix.PKUAS从配置文件default-config.properties和pkuas-autodeploy-bundles.properties获取将要启动的Bundle列表（其中deafault-config.properties声明Felix所需要的系统Bundle，pkuas-autodeploy-bundles.properties声明PKUAS默认启动的Bundle），然后启动Felix内核。

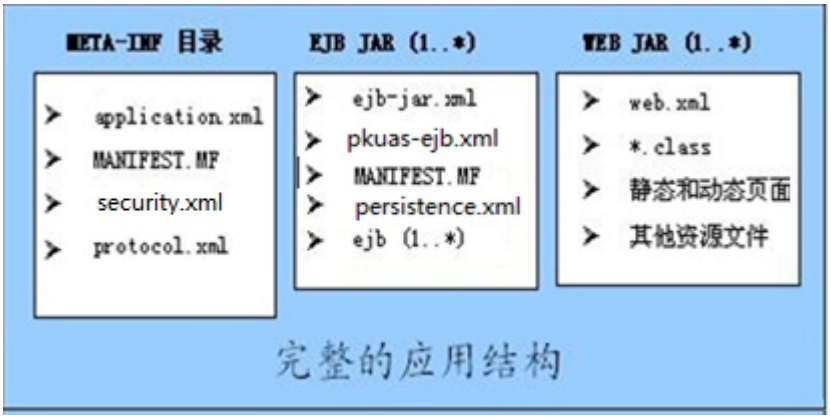
2.Felix内核根据pkuas-autodeploy.properties中声明的顺序加载各个Bundle。

3.Felix内核创建Service Manager的一个实例，由Service Manager完成Bundle Service的启动。

4.Bundle启动完毕之后控制流转到Application Manager，由它加载所有的应用。

2.4. 应用部署和启动过程

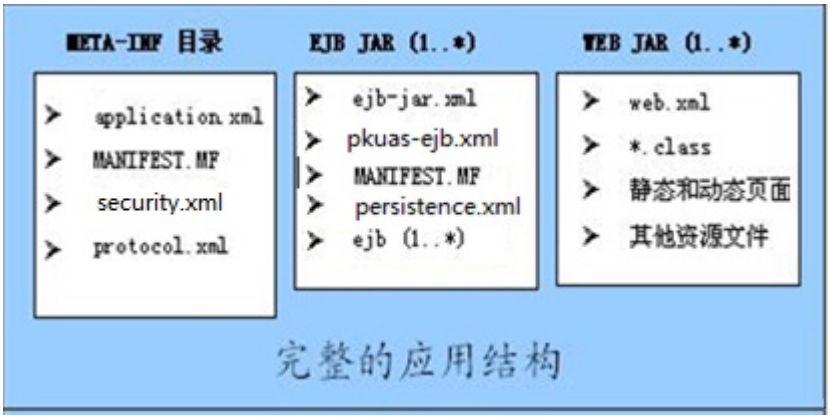
首先了解PKUAS的应用（Application）结构如下图所示：



PKUAS的部署单位为EJB应用或者Web应用。也就是说，不能直接部署整个应用（包括EJB和Web构件的EAR文件），必须分别部署EJB构件和Web构件。

ApplicationManager（pku.as.application）类负责对应用进行管理、部署、卸载、重新部署、启动和停止，该类使用了Singleton设计模式，以保证在一个PKUAS中只有一个ApplicationManager实例。

ApplicationManager与其它几个重要类的关系如下图：



2.5. 应用部署的主要步骤

在Application中实现，有两大步骤:

1.生成application的信息。包括应用本身的信息以及应用部署的信息。

2.对application进行部署。包括生成application对应的容器，生成通讯器，生成客户端容器，装配intercept链，绑定JNDI等。

2.6. 应用部署的主要流程

在ApplicationManager.loadApplication中实现，主要包括两个步骤：

1.把所有jar包和ear包拷到deployed目录下，然后对每一个调用deployApp方法进行部署。

2.ApplicationManager.deployApp部署一个应用:

(1).调用ApplicationManager.construcApp方法来构建一个application，得到这个应用的实例（具体过程在下面介绍）。

(2).为application设置包括命名、事务、安全在内的方法引用。

(3).调用application.deploy方法对应用进行部署。

(4).生成一个线程AutoDeployThread来监控部署的应用的变化。

2.7. 如何构建一个应用

ApplicationManager.construcApp方法

1.目的：主要是为了生成application的信息ApplicationInfo，包括它本身内部EJB和Web应用的信息以及部署信息。

2.构建的流程

(1).判断该应用是否更新过（通过AppInfoStore类的最近更新时间）。

(2).如果更新过：调用deployCMP方法新进行解压缩。先进行CMP转换处理，对于CMP，要调用CMP20Engine.work方法将CMP转换为BMP。然后拷贝并解开jar包或ear包。

(3).生成AppClassLoader，地址设为部署目录和服务器端的codebase。

(4).生成ApplicationDeployer，调用它的buildApplication方法，目的是解析各种配置文件和EJB、Web应用信息，最终得到应用的信息ApplicationInfo（具体过程在下面介绍）。

(5).通过ApplicationFactory得到Application实例。

(6).生成application的信息ApplicationInfo。

3.由ApplicationDeployer实现application的ApplicationInfo的生成。ApplicationDeployer的buildApplication方法得到application的各种配置文件中已经的部署信息。

(1).Xml文件用castor解析:通过三个Parse过程（parserApplicationXML，parserSecurityXML，parserProtocolXML）解析application.xml,security.xml,protocol.xml，得到application内部的各个module。

(2).解析EJB的jar包（processEJB）：为内部每个bean装配EJBClassLoader，将每个类存到classList中。

(3).解析Web的war包（processWeb）。

(4).构建EJB（buildEJBcomponents）。

• 解析pkuas-ejb.xml。

• 解析ejb-jar.xml（ejb部署信息，解析注解时优先级比代码的高）：解析每个Bean，得到元数据信息bmi（支持EJB2.0和3.0）

• 解析式通过调用ParserManager的processAnnotaion方法，返回一个class对应的bmi。

• ParserManager中提供MDB，SLB，SFB三种parser，分别调用它们的processAnnotation方法进行解析。

• 过程遵循EJB规范。

• 根据上面解析得到bmi的类型生成具体的BeanDeploymentInfo（EJB2.0中是SessionMetaData，EntityMetaData和 MessageDrivenMetaData，3.0中只有SessionMetaData ， MessageDrivenMetaData），作为部署的具体信息。

• 构建Web构件（buildWebComponents。

(5)初始化每个构件的bdi，做本地JNDI名绑定（只在EJB3.0中实现）。

4.构建EJB的buildEJBcomponents方法中，提供了加入一组InjectionDeployer的步骤，表示用户可以加入自己的deployer，实现对特定EJB定制自己的部署方式，这个deployer和ApplicationDeployer在地位上是平行的，都会被执行，在ServerConfig的getDeployerList方法中来获得deployer，所以用户要部署什么样的deployer，可以在ServerConfig的这一方法中获得，通常应该是读取一个关于部署器的配置文件，此处没有实现这一功能。

2.8. 对application进行部署

1.在ApplicationManager中生成application后，调用application的deploy方法进行部署。

2.采用动态代理调用deploy。

(1).目的：为了能将applicaton注册到MBeanServer中进行管理。

(2).自动生成动态代理ApplicationProxy，将deploy和对MBeanServer的注册（在ApplicationManagermentInterceptor类的intercept方法中进行）都放在它的deploy方法中执行。

3.在deploy方法中调用start方法启动应用，包括启动JRMP通讯器服务，生成通讯器的stub，生成客户端容器代理类ClientContainer。

4.生成容器实例：用哈希表contains存储应用中用到的所有容器，messageDrivenContainers存储用到的消息Bean的容器，取出应用中所有的构件，分别生成对应的容器，将容器注册到MBeanServer中，并进行初始化（主要是

生成interceptor链）消息Bean容器和其他容器的处理方法不一样，消息Bean容器时直接new出实例，然后进行注册，其他容器是用EJBContainerFactory生成，同时为容器设置事务、安全、命名等服务；启动Web应用。

5.完成部署后，启动一个AutoDeployThread线程，用于监听部署的应用在运行时的变化。

第 3 章 EJB 容器

3.1. 目标与需求

3.2. 概述

应用服务器中容器的作用主要是：

1.容器是异构的EJB构件的运行支撑环境。实现容器与构件的合约；维护构件运行的上下文（如安全与事务上下文）；管理构件的生命周期（如类装载、实例化、缓存、释放等），主要由类装载器、实例池与相应的对象持久化机制实现。

2.容器是为分布的EJB构件提供远程访问能力的载体。EJB调用者使用命名查找的方式得到EJB的远端代理，用来和服务器端的EJB进行交互。容器需要为分布式环境下的EJB生成相应的远端代理。

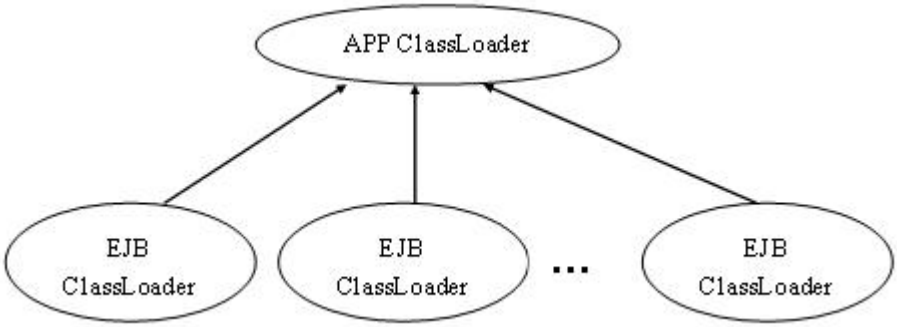
3.构件容器为受控的EJB构件提供企业计算特性。容器需要为系统的非功能约束（安全、事务等）提供检查点以实现关注点分离，供公共服务接入。

在PKUAS的容器系统中，每一类EJB对应一种容器，包括无态会话构件容器、有态会话构件容器、实体构件容器、以及消息驱动的构件容器。应用（Application）将相关联的容器组织到一起。

PKUAS中的应用（Application）是容器系统管理构件的基本组织单位，与实际部署的应用程序包一一对应的。一个构件种类对应一个容器类型，而应用程序包里的一个构件对应一个容器实例。应用实际上是容器管理框架的主要实现。首先，它是对容器的一种分组，将逻辑上相关的容器实例组织在一起，防止不同应用之间构件的相互干扰。同时，每一个应用都有自己独立的类加载器，可以防止非本应用的代码随意访问应用中的构件，从而提高了安全性。其次应用将容器、通讯器和服务集成在一起，构成了构件平台的核心。应用在启动时，根据应用程序包中的部署文件，装载相应的通讯器、容器以及容器中的服务截取器。在PKUAS中，每个应用对应一个通讯器的，由应用负责接收客户的请求，然后转发给相应的容器处理。

3.3. 类加载机制

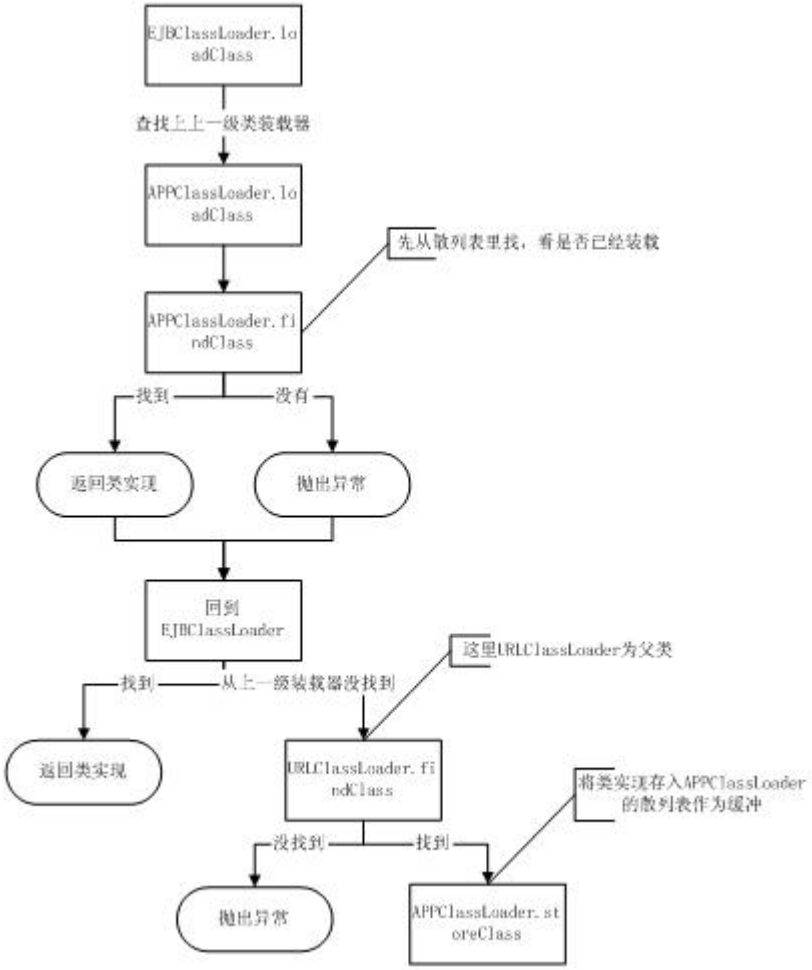
PKUAS中每一个应用采用两层ClassLoader结构，第一层是应用的类加载器AppClassLoader（pku.as.container.app，见modules/apis/container-api/src/main/java），第二层是各种类型的EJB ClassLoader。如下图所示：



每一个PKUAS的应用都对应一个自己的AppClassLoader实例。AppClassLoader继承URLClassLoader（java.net.URLClassLoader，用于从指向 JAR 文件和目录的 URL 的搜索路径加载类和资源），用一个散列表来存储已经加载的类的类名和实现之间对应关系，并提供方法装载某个构件中所有的类。

EJBClassLoader（pku.as.container.ejb.EJBClassLoader）类同样继承URLClassLoader，并包含一个指向父AppClassLoader的引用，它进行具体的类装载。

类装载的流程如下图所示：



EJBClassLoader在装载一个类时，如果找不到该类，首先委托上一层的AppClassLoader装载。AppClassLoader则在已装载类的散列表中寻找该类，如果找到，直接返回类实现，否则，再委托URLClassLoader装载类，并将类实现加入在散列表中。

3.4. 客户调用工作流程

对于一个调用，PKUAS的工作流程如下：

1.客户调用由应用服务器提供的命名服务查找构件的类型接口。

2.客户调用类型接口的创建方法，构件stub（运行时刻由Interceptor动态生成）将客户请求发送给服务器。

3.服务器传输层接收到客户请求后，根据请求中的标识，分发给相应的容器处理，即调用相应容器的invoke（pku.as.ejb.Container）方法。

4.Invoke方法调用容器初始化时注册在容器中的Interceptor的intercept（pku.as.DefaultInterceptor）方法，该方法负责依次调用Interceptor的beforeInvoke方法进行调用前的处理。

5.容器调用实例管理器获得一个实例，并生成相应的构件标识（Instance ID）。

6.容器依次调用各个截取器的afterInvoke()接口（步骤4和6中的beforeInvoke和afterInvoke方法是进行客户端的处理），进行调用后的处理。

7.容器返回构件标识（Instance ID）。

8.通讯机制将构件标识包装成构件的实例引用。（这个应该由通讯框架完成）

9.客户得到实例引用后，调用其中的实例方法，构件stub将请求发送给服务器。

10.服务器接收到客户请求（InvocationRequest）后，根据请求中的标识（方法名），分发给相应的容器处理。

11.容器依次调用各个截取器的beforeInvoke()接口，进行调用前的处理。

12.容器调用实例管理器，根据请求中包含的构件标识，调用命名服务的lookup方法得到指定的构件实例。

13.容器调用构件实例处理用户请求，通过反射机制invoke用户指定的方法。

14.容器依次调用各个截取器的afterInvoke()接口（步骤11和步骤14中的beforeInvoke和afterInvoke方法是进行服务器端的处理），进行调用后的处理。

15.容器返回调用结果。

16.客户得到调用结果。

3.5. 实例管理

EJB容器用实例池化（instance pooling）技术来管理大量Bean的实例。因为EJB的客户端从不直接访问Bean的实例，而是通过容器来访问，因此，容器只需要维护少量的Bean实例，并重复使用这些Bean实例为不同的请求提供服务就可以了。这种技术可以大大减少处理全部请求所需要耗费的资源总量。实例管理的基类是pku.as.ejb.im.BaseInstanceManager。不同类型的EJB实例管理类继承该类。

1.无态会话Bean的实例管理（pku.as.ejb.stateless.StatelessInstanceManager）：无态会话Bean不需要维护任何内部状态，每次独立的方法调用也不依赖于任何实例变量，因此它的实现相对简单。一个无态会话Bean的生命周期存在三种状态：

- 无状态：Bean实例尚未被初始化的状态，通常表示一个Bean实例生命周期的开始和结束。

- 池状态：Bean实例已经被容器初始化，但是尚未与一个EJB请求相关联。
- 就绪状态（ReadyKey）：Bean实例已经与一个EJB请求相关联，并且已经做好了响应业务方法调用的准备。

2.消息驱动Bean的实例管理（pku.as.ejb.message.MessageDrivenInstanceManager）：同无状态会话Bean一样，消息驱动Bean也不需要维护任何内部状态。因此当消息到来时，实例管理器只要简单的调用fetchFromPool方法从实例池中取出一个实例（若实例池中没有可用的实例，则重新new一个实例），使用完毕后调用toPool方法把实例放回实例池中即可。

3.有态会话Bean的实例管理（pku.as.ejb.stateful.StatefulInstanceManager）：有态会话Bean的实例管理最大的区别是需要在不同的方法调用之间维护会话状态，实例管理器要负责对实例进行钝化（passivation，解除有态会话bean实例与相关EJB对象之间的关联，并保存其状态的操作）和激活（activate，根据所关联的EJB对象将Bean实例的状态重新回复的操作）操作。因此实例管理器需要保存实例钝化器（pku.as.ejb.stateful.StatefulPassivator）的引用。

4.实体Bean的实例管理（pku.as.ejb.entity.EntityInstanceManager）：实例管理器负责实现实例的钝化和激活操作。

在PKUAS中，实例池的实现采用两种数据结构，一种是集合，一种是队列。

1.队列：处于队列中的构件实例没有标识，处于池状态的Bean。

2.散列表：处于散列表中的每一个构件实例都有惟一标识，用于处于就绪状态的Bean（包括处于会话中的有态会话Bean）。

此外，实例池在实现中把处于事务中的和没有处于事务中的实例分开管理。

第 4 章 通信框架

4.1. 目标与需求

4.1.1. 第1.1小节标题

第1.1小节内容

4.1.2. 第1.2小节标题

第1.2小节内容

4.2. 第2小节标题

4.2.1. 第2.1小节标题

第2.1小节内容

4.2.2. 第2.2小节标题

第2.2小节内容

第 5 章 管理框架

5.1. 目标与需求

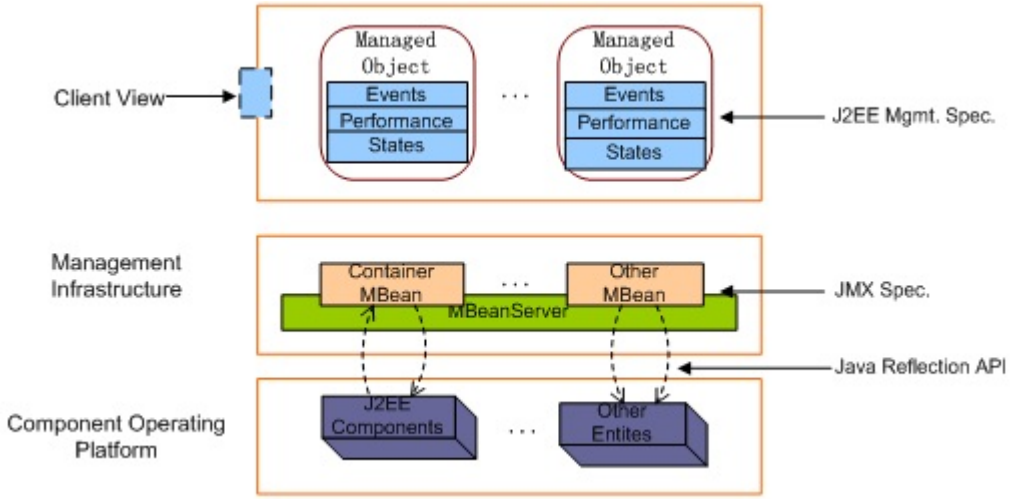
J2EE构件运行支撑平台是符合J2EE规范的构件运行平台，为了满足用户需求和运行环境的要求，J2EE构件运行支撑平台提供的功能和质量服务日益增多。部署在构件运行支撑平台上的应用和平台的公共服务必须能够被有效地观测和管理，以便给用户提供符合要求，能适应变化的服务。平台的管理模块用于提供对平台和应用的行为和状态的管理设施。PKUAS的管理框架负责提供平台的基础管理设施、管理互操作接口和管理工具。其中正确地观测平台的状态和行为、有效地调整和操纵平台的状态和行为是平台基础管理设施的关注的两部分内容；管理互操作接口对外提供管理接入的途径；管理工具提供面向用户的友好的管理界面。

Sun公司提出的JSR77被J2EE1.4规范纳入，作为J2EE的标准管理模型。这个管理模型包括元数据模型、性能监控模型、事件处理模型和状态模型，以及管理互操作接口MEJB。J2EE规范要求J2EE应用服务器必须支持其中的元数据模型和管理互操作接口。而对于性能监控模型、事件处理模型和状态模型的实现是可选的。J2EE管理模型标准化了J2EE应用服务器的管理机制，使得管理成为独立于平台实现的部分。PKUAS的管理框架实现了J2EE的标准管理模型作为平台的基础管理设施，并且提供了MEJB来访问管理框架的内容。此外，PKUAS也提供了基于Web的管理界面来管理平台和应用。对Web管理工具设计的介绍将在另外的文档中进行。

对于管理框架而言，图1红色圆圈中的模块都是平台管理的内容。在PKUAS管理框架体系中，管理框架通过操纵这些模块的元层实体来操纵这些模块的行为和表现。此外，管理框架与PKUAS的系统启动模块、JMS消息服务、安全服务存在直接交互关系。

5.2. 方案与设计

PKUAS的管理机制主要由Java反射机制、Jmx Server和JSR77管理规范构成，它们的关系如下图所示。

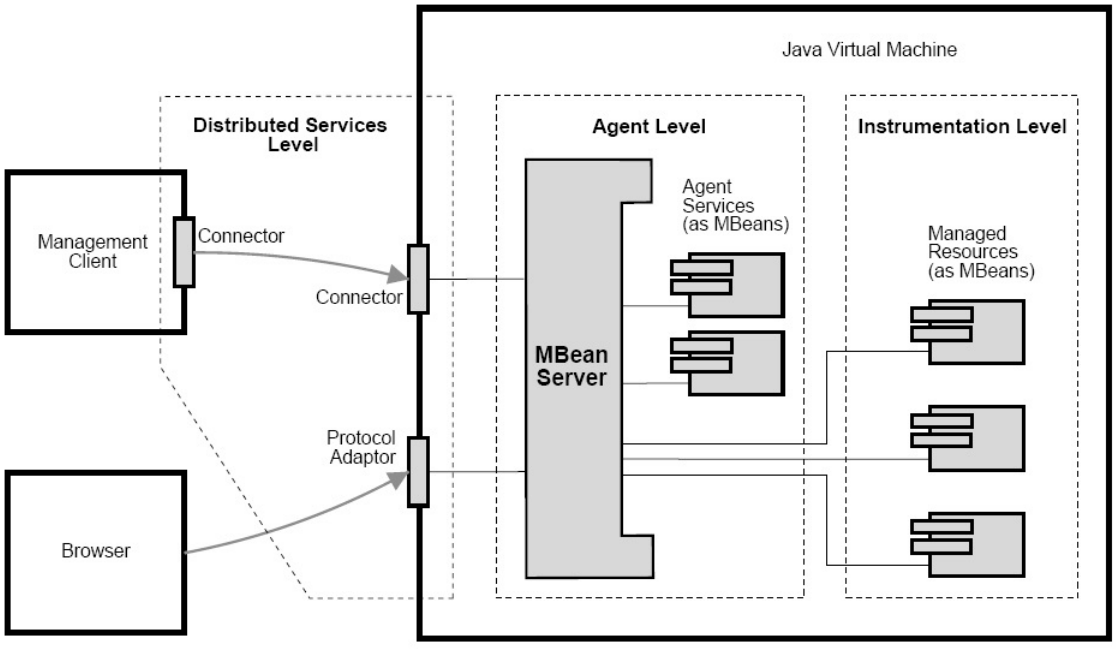


5.2.1. JAVA反射机制

Java Reflection API让编程人员可以在运行时刻才确定需要对哪个类、那个方法进行操作。这个API在语言级上支持了在运行时刻对类、接口的方法和属性域进行操纵和管理，而在整个管理过程中，管理者不需要知道被管理对象是哪个类或者接口，它们具体包含什么属性域和方法。比如，确定一个对象的类名；获取一个类的修饰符、属性域、方法和超类；获取一个接口的常量声明和方法声明等。

5.2.2. JMX管理框架

JMX是一个为Java应用程序植入管理功能的框架，它的体系结构可分为以下三个层次。



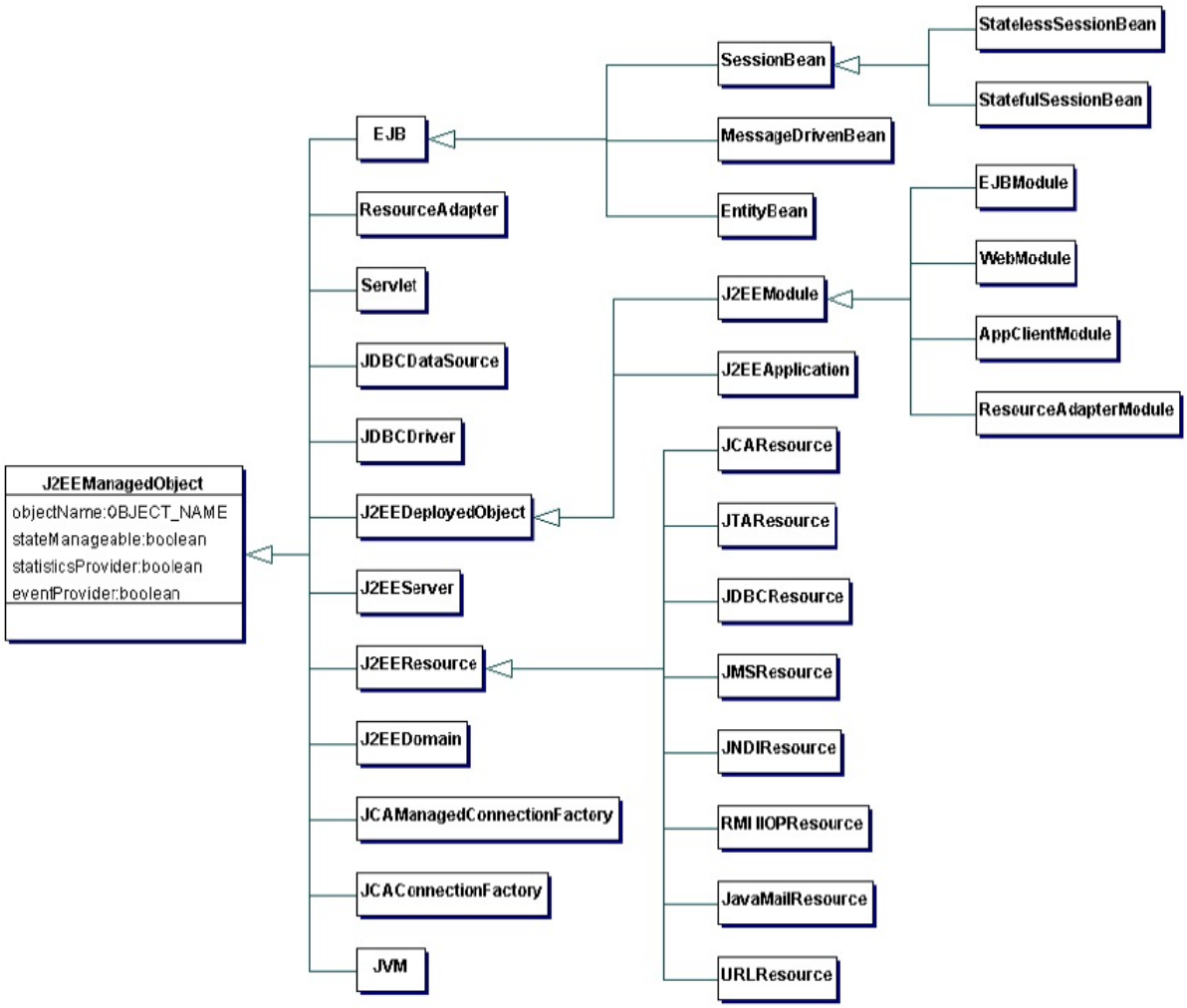
- 工具层：工具层定义的是JMX可管理资源所在的层。这些可管理资源可以编写或封装成MBean 。一共有四种类型的MBean：标准、动态、开放、模型。

- 代理层：代理层定义了用来管理同一个Java虚拟机上的工具层实体的代理，包括MBeanServer和动态类装载服务、监控服务、计时服务和关系服务等标准代理服务，标准代理服务同样被实现成MBean，受MBeanServer（有时候也称为jmx server）的管理。

- 分布式服务层：分布式服务层包含协议适配器和连接器两种类型的组件，它们实现为MBean的形式。通过它们可以连接到外部的管理应用。例如，外部应用可以通过RMI进行管理或者基于浏览器进行管理。

5.2.3. J2EE管理规范---JSR77

Sun Microsystems 和其他构件运行支撑平台厂商提交给JCP(Java Community Process)[JCP]的JSR77(Java Specification Request 77)[SUN 02a]制定了J2EE构件运行支撑平台的标准管理模型，J2EE1.4规范要求每一个符合规范要求的构件运行支撑平台必须支持这个管理模型。它标准化了构件运行支撑平台的管理机制，使得管理本身成为独立于平台的内容，任何一个支持J2EE管理规范的平台可以被任何一个符合管理规范要求设计的管理工具进行管理。建立在JMX对Java资源对象的管理能力之上，这个管理模型提供了元数据模型、事件模型、状态模型和性能数据模型，并且规定了客户如何来访问这些模型数据。



5.2.4. PKUAS中管理功能的框架

- 建立基于反射的平台管理框架，平台管理的不同反射层是框架的主体。其中，PKUAS的平台服务和运行在平台上的应用构件构成反射塔结构中的基层；PKUAS对平台服务和容器系统的MBean封装，构成反射塔结构中的第一元层；第二元层由构件运行支撑平台具体的管理元数据组成，包括J2EE管理视图的元数据和RSA视图的元数据。
  - 支持J2EE管理视图让平台管理符合了标准。
  - 在管理视图层中引入运行时软件体系结构给管理者提供了易于理解的应用结构信息，符合了面向应用的管理要求。
  - 基于JMX的微内核以及反射塔结构使得平台管理具有良好的可扩展能力。
- 在此基础上，提供性能监控、事件处理和访问控制这三个核心支撑机制。性能监控机制给管理者提供平台和应用运行时的状态和行为信息；事件处理机制用于记录、追踪平台管理的敏感操作以及状态、行为的变化。访问控制机制则保证反射式管理操作以受限的方式进行。
  - 通过可配置的方式，在不同的应用语境中可以选择性地启用三个支撑机制，改善平台管理的定制能力。
  - 特别地，给管理者提供在线启动/停止性能监控的能力，保证平台管理符合在线要求。
  - 此外，访问控制机制给平台管理提供了安全性保障。

5.2.5. 如何实现性能监控

首先，management模块为大多数J2EE Object实现了对应的数据统计类，比如最简单的StatisticImpl除了名字等不变属性外，记录了2个性能的数据。

```
public abstract class StatisticImpl
      implements Statistic, Serializable
{
  private static final long serialVersionUID = -3427364348020739916L;
  protected String name;
  protected String units;
  protected String description;
  protected long startTime;
  protected long lastSampleTime;
  public StatisticImpl(String name, String units, String description)
  {
     this.name = name;
     this.units = units;
     this.description = description;
```

```
        this.startTime = System.currentTimeMillis();
    }
```

这些数据统计类在J2EE MBean初始化的时候也被注册进了MBServer，只要查找这些MBean就可能得到相应的J2EE object的性能数据。

```
 public class StatelessSessionBean extends SessionBean
implements StatelessSessionBeanMBean
{
 /**
  * construct method
  * @param name type String
  */
 protected StatelessSessionBean(String name,ObjectName parent,ObjectName containe
  super(J2EETypeConstants.StatelessSessionBean,name,parent,container);
  stats=new StatelessSessionBeanStatsImpl();
 }
```

5.3. 启动和使用

5.3.1. 管理模块的启动过程

- 在配置文件pkuas-autodeploy-bundles.properties中配置默认启动pkuas-management-2008-ipojo.jar，由Felix将其装载到OSGi平台上。
- ServiceManager根据配置文件pkuas.xml启动JSR77Manager服务。
- JSR77Manager初始化
  - 在MBServer（即JMX Server)中注册一个J2EEDomain，一个J2EEServer和一个JVM。
  - 找到ServiceManager和EarManager。
  - 为ServiceManager中的每一个服务创建一个管理对象，为MBServer中的每一个Application中的每个EJB创建一个J2EE管理对象。
- 监听ServiceManager和ApplicationManager，如果创建了新的应用，则在MBServer中创建相应的MBean管理对象，同步更新。

5.3.2. 如何使用管理框架

管理模块作为一个bundle运行在服务器的OSGi平台上，用户可以通过一个EJB应用来访问管理模块从而获得系统此时的运行信息。将管理模块的内核和客户端分开的好处有三条：

- EJB支持远程访问，解决了JMX规范只允许处于同一个虚拟机的客户访问MBean的缺点，扩展了符合JMX规范要求的元数据的访问方式。
- PKUAS提供了开放式的互操作框架。对于MEJB来说，管理员可以通过多种互操作协议，如IIOP、JRMP、SOAP、EJBLocal等来访问它，继而通过它访问管理框架，扩展了管理操作的协议支持能力。
- 可以利用J2EE安全模型给EJB提供的访问控制机制和安全传输机制，如JAAS和IIOP-SSL来提高管理操作的安全性。

管理模块实现了javax.management.j2ee.Management所定义的标准接口：

```
public Object getAttribute(ObjectName name, String attribute)

public AttributeList getAttributes(ObjectName name, String[] attributes)

public String getDefaultDomain()

public ListenerRegistration getListenerRegistry()

public Integer getMBeanCount()

public MBeanInfo getMBeanInfo(ObjectName name)

public Object invoke(ObjectName name, String operationName,
  Object[] params, String[] signature)

public boolean isRegistered(ObjectName name)

public Set queryNames(ObjectName name, QueryExp query)

public void setAttribute(ObjectName name, Attribute attribute)

public AttributeList setAttributes(ObjectName name, AttributeList attributes)
```

下面的例子展示了开发人员如何通过编写一个EJB应用来实现“查看平台所有服务的应用”

```
  /* 创建MEJB */
Context ic = new InitialContext ();
ManagementHome home = (ManagementHome) PortableRemoteObject.narrow
(ic.lookup(“ejb/mgmt/MEJB”),ManagementHome.class);
Management mejb = home.create ();
/*得到构件运行支撑平台的J2EE管理视图数据*/
ObjectName searchPatten = new ObjectName(“SingleDomain:j2eeType=J2EEServer,
name=PKUAS”);
Set mbeans = mejb.queryNames(searchPatten,null);
ObjectName mbean = (ObjectName)(mbeans.iterator().next());

/*得到平台上的所有公共服务*/
Iterator it = ((ArrayList)mejb.getAttribute(mbean,"mResources")).iterator();
while(it.hasNext()){
ObjectName t_obj = (ObjectName)it.next();
/*输出每一个公共服务的名字*/
System.out.println("[Service] " + t_obj.toString());
```

5.4. 现提供的管理工具

JSR77Manager以Struts结构建立了一个网站，它作为一个应用放在了服务器端，用户可以访问服务器8080端口下的manage路径来访问。

JSR77管理工具管理了服务器所提供的大部分服务，包括EJB和WEB应用，命名、数据源、数据库管理、安全（用户管理）、消息、事务、以及EJB/WEB容器的管理。

第 6 章 WEB容器

6.1. 目标与需求

容器是一种服务调用规范框架，J2EE大量运用了容器和组件技术来构建分层的企业级应用。在J2EE规范中，主要包括四种容器（Applet容器、Application Client容器、EJB容器以及Web容器）和各类J2EE服务（事务服务，安全服务，资源管理服务等）。

一般地，Web容器在接收到客户的Web请求时，会由相应的Web构件进行简单的业务处理，之后再将请求转发给EJB容器、由相应的EJB构件处理复杂业务逻辑并跟数据层进行交互。

Web容器负责管理两类构件——Servlet构件和JSP构件，它们分别对应Servlet和JSP两个规范。由于JSP规范实际上是对Servlet规范的扩展与延伸，所以通常这两类构件由同一个容器（即Web容器）管理。Web容器负责提供支撑部署在Web容器中的Servlet/JSP构件运行所需的机制，使JSP、SERVLET直接跟容器中的服务接口交互，不必关注其它系统问题；当作为应用服务器中的一个模块时，Web容器需要提供被应用服务器其他模块访问的接口，例如为管理工具提供启动、停止和配置（包括虚拟主机配置）等一系列管理操作的管理接口，为部署工具提供部署Web构件的部署接口等。

6.2. 方案与设计

PKUAS设计了一种新的Web容器的集成框架。该框架包括两个层面，面向应用服务器其他模块（包括部署工具模块，管理工具模块等）的外层，以及面向Web容器实现的内层。 体系框架如图1-1所示：

图 1-1 Web 容器的集成框架图

6.2.1. 框架外层（外层接口）：

由一组接口构成，它定义了应用服务器其他模块（包括管理工具、部署工具和其他模块等）与Web容器进行交互的接口。应用服务器中的其他模块与Web 容器的交互全部通过这组接口进行，而不对Web容器的具体实现有任何依赖。

通常，一个完整的Web服务器包含一个或多个的“虚拟主机”（所谓虚拟主机，就是在一个物理的服务器上配置多个域名，每个域名对应相同或不同的应用。这样，客户端看起来好像是有多个主机；Web 容器在收到请求时，根据客户端给出的域名确定处理该请求的虚拟主机），在每个虚拟主机中又可以部署一个或多个Web应用。

根据这样一种层次结构，我们抽象出如下三个接口作为Web容器集成框架的外层接口：

6.2.1.1. WebController

该接口代表了整个Web 容器，任何需要使用或操作Web 容器的模块都需要通过该接口进行。该接口提供的方法包括启动与停止Web 容器，或者添加删除虚拟主机以及获取已定义的虚拟主机实例等。

6.2.1.2. VirtualHost

该接口代表一个虚拟主机。虚拟主机同时是部署Web应用的容器，一个应用在同一时间只能被部署在一个虚拟主机上。该接口提供的方法包括启动与停止该虚拟主机和部署与卸载Web 应用等。

6.2.1.3. VirtualApp

该接口代表了一个部署在服务器上的Web 应用。通过该接口可以得到具体应用的信息，包括应用根目录的本地路径和映射的上下文路径等。另外该接口还提供部署、启动、停止和卸载应用的方法

这三个接口反映了Web容器的逻辑层次结构，能够满足应用服务器其他模块对Web容器的访问，并且完全独立于所集成的Web容器，很好地将Web 容器的内部实现细节隐藏了起来。

在PKUAS的Web容器中，实现了两个重要的类Server.java和HostWrapper.java。

（1）Server.java在启动PKUAS服务器时设置tomcat的相关参数，包括maxThreads、connectionTimeout等。然后启动tomcat。Server.java的实现，从机制上为之后tomcat的升级或替换提供的可能性。

（2）HostWrapper.java在网服务器部署应用时，对web应用进行管理。对于每一个web应用，都包裹在一个container中，服务器管理该应用或客户端调用该应用时，都通过HostWrapper.java来进行管理和访问

6.2.2. 框架内层（内层接口）：

Web容器框架内层的主要功能是为外层接口到特定Web容器提供的方法进行适配，即对特定Web容器进行包装、扩展或改良，用于匹配外层接口。为了完成该功能，内层需要实现的内容包括：

（1）接入应用服务器管理框架所需的接口。Web容器集成到应用服务器中，需要遵从一定的管理规则，即实现接入应用服务器内核的管理接口。例如一般应用服务器都会采用Java Management Beans接口（Java Management Extensions规范的组成部分）来实施统一管理，因此Web容器为纳入应用服务器整理管理框架也必须实现MBeans接口。

（2）实现框架外层中定义的接口，将框架外层所定义的功能映射到Web容器所提供的功能。若集成的Web没有相应的功能，则通过类似功能进行模拟或者直接实现该功能。一般来说特定的Web容器实现都会有相关模块进行匹配。例如，我们使用的是当下比较流行Apache Tomcat5.0作为集成的Web容器，它的结构就和外层接口有比较好的对应关系：!WebController的功能映射到Tomcat Engine模块；!VirtualHost的功能映射到Tomcat Host模块；!VirtualApp功能映射到Tomcat Context模块。

下面将着重介绍一下我们所使用的Web容器tomcat的内部结构。

6.2.3. tomcat内部结构

tomcat是一种Servlet/jsp容器（更实质性的说是Servlet容器，因为jsp最终还是被编译成servlet来执行的），为来自web客户的请求提供服务。

tomcat服务器由一系列可配置的组件构成，其中的核心组件是Catalina Servlet容器，它是所有其他tomcat组件的顶层容器。tomcat的组件可以在conf/server.xml文件当中进行配置，每个tomcat组件在server.xml文件中进行配置，每个tomcat组件在server.xml文件中对应一种配置元素。以下代码以XML的形式展示了各种Tomcat组件之间的关系:

```
<Server>
 <Service>
  <Connector/>
   <Engine>
    <Host>
     <Context>
     </Context>
    </Host>
   </Engine>
 </Service>
</Server>
```

这些元素分四类:

1.顶层类元素:包括<Server>元素和<Service>元素,他们位于整个配置文件的顶层.

2.连接器类元素　:代表了介于客户与服务之间的通信接口,负责将客户的请求发送给服务器,并将服务器的响应结果传递给客户.

3.容器类元素:代表处理客户请求并生成响应结果的组件,有3种容器类元素,它们是Engine、Host和Context.　Engine组件为特定的Service组件处理所有客户请求,Host组件为特定的虚拟主机处理所有客户请求,Context组件为特定的Web应用处理所有客户请求.

4.嵌套类元素:代表了可以加入到容器中的组件,如<Logger>元素、<Valve>元素和<Realm>元素。

图1-2描述了tomcat配置的结构。

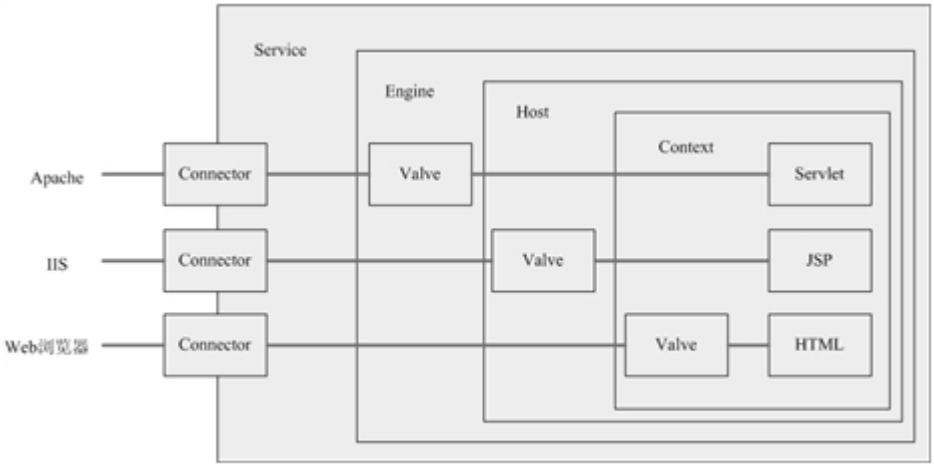


图 1-2 Tomcat 配置体系结构图

下面，再对一些基本的Tomcat元素进行介绍。

<Server>元素

代表了整个Catalina Servlet容器,它是Tomcat实例的顶层元素.可包含一个或多个<Service>元素.

<Service>元素

包含一个<Enginev元素,以及一个或多个<Connector>元素,这些<Connector>元素共享同一个<Engine>元素.

<Connector>元素

代表和客户程序实际交互的组件,他负责接收客户请求,以及向客户返回响应结果. 每个Connector 实例实际上实现的是一种网络传输协议，它将通过这种协议传入的客户端请求进行分析，构造相应的Request 和Response实例，找出适合相应该请求的Container 实例，调用该Container的invoke方法并将Request和Response实例作为参数传入，最终将处理结果或者错误信息反馈给客户端。

<Engine>元素

每个<Service>元素只能包含一个<Engine>元素. <Engine>元素处理在同一个<Service>中所有<Connector>元素接收到的客户请求. 一个Engine 的子Container 通常是一个或多个的Host 实例。

<Host>元素

一个<Engine>元素中可以包含多个<Host>元素.每个<Host>元素定义了一个虚拟主机, 与一个特定的主机名以及任意个“别名”相绑定。它可以包含一个或多个Web应用. 每个客户端的请求都会根据主机名映射到相应的Host 进行处理。Host 的下级Container 通常是一些Context 实例。

<Context>元素

每个<Context>元素代表了运行虚拟主机上的单个Web应用. 它包括了Web 应用的所有Servlet类、JSP文件、静态页面和图片、Jar包、环境变量、各种配置参数以及其他各种资源。Context的下级Container 通常是Wrapper 的实例。一个<Host>元素中可以包含多个<Context>元素.

<Host>元素

一个<Engine>元素中可以包含多个<Host>元素.每个<Host>元素定义了一个虚拟主机, 与一个特定的主机名以及任意个“别名”相绑定。它可以包含一个或多个Web应用. 每个客户端的请求都会根据主机名映射到相应的Host 进行处理。Host 的下级Container 通常是一些Context 实例。

<Context>元素

每个<Context>元素代表了运行虚拟主机上的单个Web应用. 它包括了Web 应用的所有Servlet类、JSP文件、静态页面和图片、Jar包、环境变量、各种配置参数以及其他各种资源。Context的下级Container 通常是Wrapper 的实例。一个<Host>元素中可以包含多个<Context>元素.

6.3. Web应用调用工作流程

Web容器在启动应用服务器时就要启动，具体流程是：（1）启动模块通过外层接口，分别在服务器上创建Virtual Host: localhost，端口号为8080；在该端口上创建Connector；（2）启动Web容器tomcat；（3）创建一个tomcat的engine实体。

当用户部署了一个Web应用，即发布一个带.war包的.ear包到application上时，web容器会为每一个Web应用分配并维护一个Container(包括其相应的URL）。Web容器在应用部署阶段的另一个主要功能，是将应用中相应EJB的JNDI写入tomcat的JNDI中，因为tomcat在处理web请求时不会查找相应的EJB的JNDI.

当用户要进行Web访问时，在浏览器中键入相应的URL，

(1) 之前创建的Connnector会一直监听是否有web请求，Connector监听来自客户端的请求后会创建一个Request实例，并进行相应初始化；

(2) 然后创建相应的Response实例并初始化；

(3) 解析客户端的URL地址，将其映射到相关的Web应用（与具体的JSP网页绑定）；若没有找到相应的应用，则报错；

(4) 解析网页，这里以MyHelloWorld.ear为例加以说明。如下是index.jsp的代码:

```
<%@ page contentType="text/html; charset=GBK"%>
<%@ page import="com.foshanshop.ejb3.MyHelloWorld, javax.naming.*"%>
<%@ page import="java.util.Properties"%>
<%
try {
Properties props = new Properties();
InitialContext ctx = new InitialContext(props);
MyHelloWorld helloworld = (MyHelloWorld) ctx.lookup("com.foshanshop.
out.println(helloworld.SayHello("ZhaoJingyu"));
} catch (NamingException e) {
out.println(e.getMessage());
}
%>
```

(ⅰ)为本Web应用创建Context实例ctx;

(ii)创建一个EJB中定义的类MyHelloWorld的实例helloworld，并初始化；这里需要调用ctx中的lookup()方法，同时需要EJB容器中的classloader加载相应的EJB类;

(iii)调用类MyHelloWorld的方法SayHello（）；

(iv)将所需输出的内容以String的形式返回给response实例;

(5) 服务器返回相应response给浏览器，用户可以看到相应的输出内容。

由此，完成一个Web应用的访问过程。

第 7 章 命名服务

7.1. 目标与需求

命名服务(Naming Service)提供了一种为对象命名的机制，可以定位任何通过网络可以访问的机器上的对象，使得用户可以在无需知道对象位置的情况下获取和使用对象。

使用命名服务，首先要将对象在命名服务器上注册，然后用户就可以通过命名服务器的地址和该对象在命名服务器上注册的JNDI名找到该对象，获得其引用了。

7.2. 方案与设计

命名服务作为PKUAS提供的主要公共服务之一，在其构件化体系结构中的位置如下图所示：

命名服务与事务服务、消息服务、数据连接服务等协作，帮助这些服务在执行中获得接口、连接、构件等。

7.2.1. 命名服务的执行

在PKUAS中，用户使用命名服务，首先通过SmartCtx对象来查找需要的对象，PKUAS会把SmartCtx的请求转发给相应互操作协议的命名服务的代理，代理再把请求转发到具体的命名服务。

下面以查询请求解释这个过程：

7.2.1.1. 生成初始上下文

上下文包含了命名服务发生的环境和内容。它是一个对象，它的属性实际上就是前述命名服务所保存的一组绑定。一般的，一个命名服务包含多个上下文对象，这些上下文对象以“树”的形式组织逻辑，即存在一个根上下文，它有若干个子上下文，这些子上下文通过再产生子上下文的形式向下生长。引入子上下文的目的是为了 支持跨越命名空间多层次的复合名称的查找，便于按照原子名称的顺序在对应的子上下文中查找具体的绑定。

初始上下文是上下文树的根。如果要获得初始上下文，需要通过初始上下文工厂实现。

在PKUAS中，就是通过SmartCtxFactory构造具有一定环境属性的SmartCtx初始上下文，并使用传递给构造方法的环境参数中以及所有应用程序资源文件中定义的属性初始化其环境。

特殊的，如果是来自Web应用（如Servlet）的请求，则要通过ServerCtxFactory构造SmartCtx初始上下文。

7.2.1.2. 查询名称

一般的，查询名称的过程就是一个解析名称的过程，即一个不断地将名称转化为命名上下文的过程。一个命名上下文或者将名称直接映射到具体的属性上，或者进一步映射到一个子上下文上，并将名称传递给那个子上下文。

在解析一个名称时，首先将该名称传递给初始上下文，然后不断地对名称在上下文中迭代，直到找到对应的属性，或者发现该名称在该上下文中无法解析。

而如果是查找一个混合名称，则需要在多个不同的命名空间中解析。

在PKUAS中，就是在初始上下文中调用命名服务器（Namingserver）中的find方法，间接调用NameParserImpl解析该名称。

7.2.1.3. 返回结果

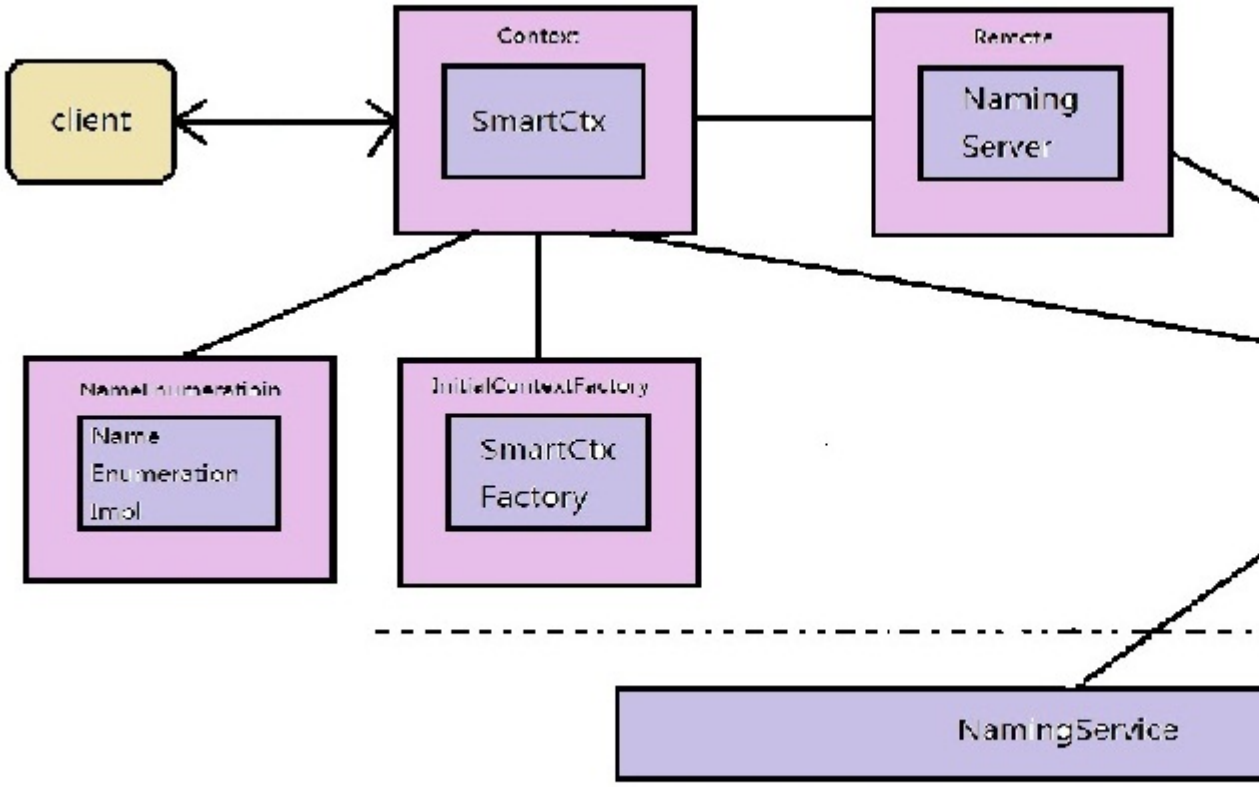
在PKUAS中，查询返回的结果可能是单个IOR，也可能是多个，因此要调用NameEnumerationImpl的方法来枚举结果。

7.2.1.4. 命名服务管理

在PKUAS中，由NamingService 负责管理整个命名服务的执行过程。它包含的了命名服务的所有属性，包括命名服务器名、端口号等，它执行的操作包括构造命名服务器、开始/结束命名服务等。

7.2.1.5. 名称解析

在SmartCtxFactory、NamingServer和NamingService中处理名称时使用NameParserImpl进行名称解析



7.2.2. 命名服务的实现

在PKUAS中，命名服务维护了一组JNDI名称与IOR的映射。在执行查找请求时根据一定的负载平衡算法选择一个特定于某个节点的对象引用返回给客户。

PKUAS将命名服务的实现划分为本地和全局两个层次。

7.2.2.1. 本地命名服务

本地命名服务存储单个构件/应用/服务器数据的名称和引用的绑定，比如构件的可定制属性、构件的别名、本地EJB的互操作地址、服务器的配置信息等。

这些信息存储在本地服务器的容器中，以供本构件查找。

7.2.2.2. 全局命名服务

另外一些需要存储在网络中的信息，如容器的地址等，则需要使用全局命名服务。

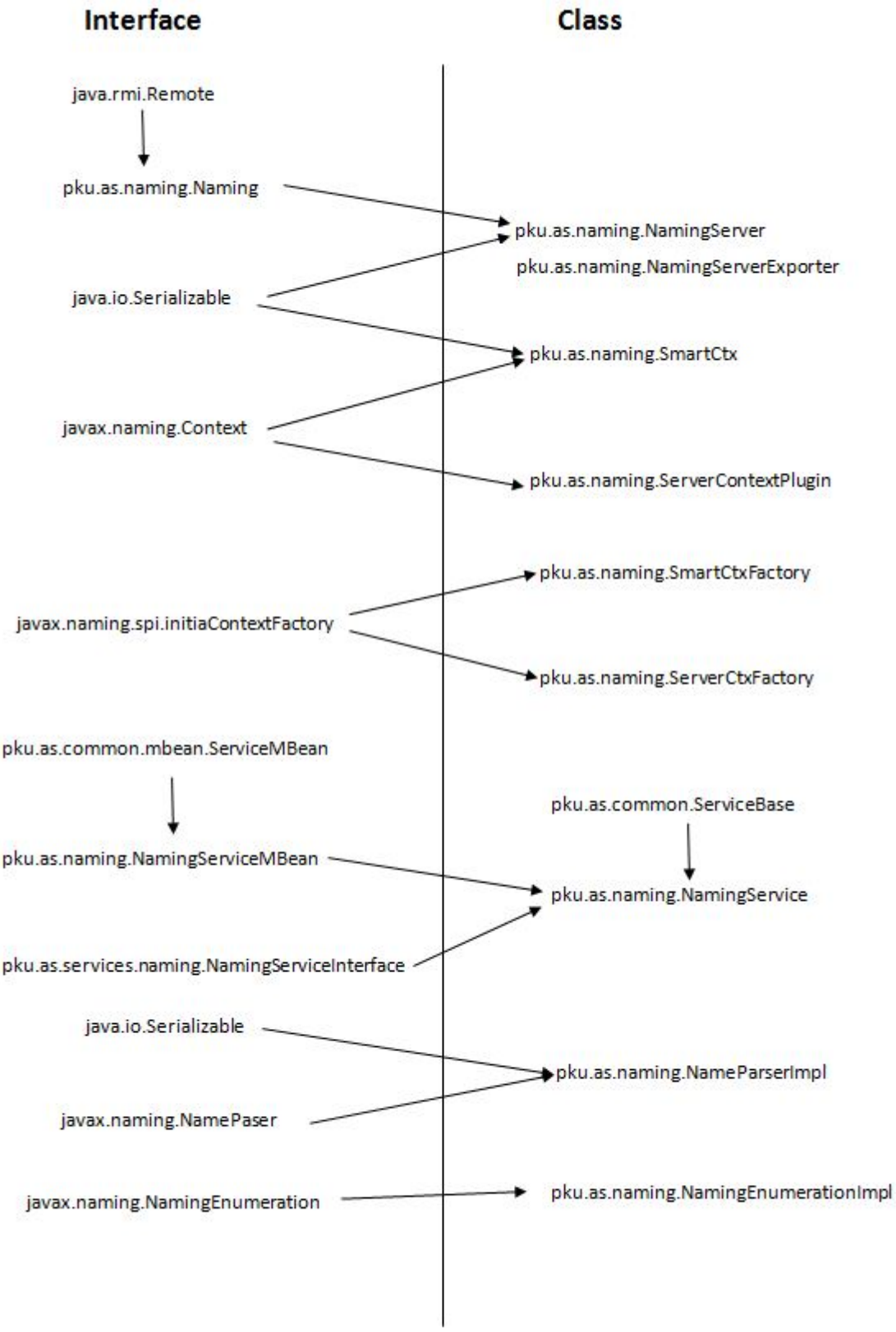
PKUAS集成位于不同服务器上的命名服务实例：用户将待集成的命名服务器注册到全局唯一的命名服务中，全局命名服务通过Proxy插件等将当前命名服务器中的名称绑定到其中。这样，当用户或其它服务使用全局命名服务查找对象时，查询请求将被送往全局命名服务，由全局命名服务查询其中的所有绑定，并找到该对象实际所在的地址，返回给用户或服务该对象的引用。

以EJB为例，PKUAS在应用部署时刻为其生成home接口的IOR，并将IOR绑定到本地命名服务上，对于需要负载平衡的EJB则还会将其绑定到全局命名服务上。全局命名服务通过集群服务将名称组播到组内所有命名服务。在接收到查询请求后，全局命名服务首先在自身维护的绑定信息中查找相应的对象引用，如果找不到，就将请求转发给本地命名服务。如果仍然找不到，则将请求通过集群服务转发给组内其他节点的命名服务。当得知某个节点失效时，全局命名服务将属于该节点的名称信息删除。

7.2.3. 使用命名服务

```
//设置初始化上下文的参数，主要是设置命名驱动的类名，
// java.naming.factory.initia指定初始上下文工厂，pkuas对应的是pku.as.naming.SmartC
//java.naming.provider.url的值包括提供命名服务的主机地址和端口号，pkuas对应的是rmi

Properties p=new Properties();
p.setProperty("java.naming.factory.initial","pku.as.naming.SmartCtxFactory");
p.setProperty("java.naming.provider.url","rmi://localhost:2001");

// 初始化上下文环境，生成一个上下文实例
Context ctx = new InitialContext(p);

//查找一个对象
Object ojb = ctx.lookup("someJndiName");
```

7.3.1. 接口与服务类概览



7.3.2. 主要接口

7.3.2.1. pku.as.naming.Naming

pkuas命名服务对外提供的接口。

该接口扩展了javax.rmi.Remote接口，声明了一组上下文操作（建立/重新/消除绑定、查找、两种列表、创建/销毁子上下文），是远程虚拟机可调用的本地虚拟机方法。

实现：

pku.as.naming.NamingServer

7.3.2.2. pku.as.naming.NamingServiceMBean

pkuas中，命名服务作为消息Bean被管理。这个接口就是对外提供了将命名服务作为消息Bean的管理方法。它扩展了pku.as.common.mbean.ServiceMBean接口，提供获取和设置数据端口和命名端口的抽象方法。

实现：

pku.as.naming.NamingService

7.3.3. 主要类

7.3.3.1. pku.as.naming.NamingServer

命名服务器类是pkuas命名服务的核心。

此类实现了pku.as.naming.Naming和 java.io.Serializable两个接口，可序列化地实现了pku.as.naming.Naming中声明的命名上下文操作（建立/重新/消除绑定、查找、三种列表、创建/销毁子上下文），供SmartCtx中的方法调用。

NamingServer是所有命名服务的实际执行者。

在NamingServer中，首先以map为基础，实现一组基本操作，再调用这组基本操作来完成Naming中声明的上下文操作。

如果名称是一个CompoundName，在绑定第k个component时，要求前k-1个component都已经绑定到了对象，并且要求其第一个component绑定在一个命名服务器上。

对于返回子上下文的列表方法，NamingServer返回给SmartCtx一个Collection对象，再由SmartCtx将Collection对象转化为NameEnumeration对象返回给客户端。

7.3.3.2. pku.as.naming.SmartCtx

SmartCtx是pkuas命名服务器和客户端进行交互的上下文类，即客户端发出SmartCtx的请求，pkuas返回SmartCtx的响应。

此类实现了javax.naming.Context和java.io.Serializable两个接口。

提供resolveObject()方法，对对象进行预处理，将非MarshalledObject变成MarshalledObject，便于将对象传入rmi调用；

提供resolveName()方法，对名称进行预处理，对于以"java:"开头的名称，解析它之后的部分；对于以空开头的名称，解析它之后的名称，对于其它非空名称，给它增加前缀。

调用命名接口，实现命名上下文的所有操作（建立/重新/消除绑定、查找、两种列表、创建/销毁子上下文），而不同于命名服务器，它对每种名称为Name类型参数的方法，都名称为String类型参数的相应方法：对于名称为Name类型的操作，首先调用resolveName()方法和resolveObject方法对名称和对象进行预处理；对于名称为String类型的操作，直接用命名解析器解析这个名称。

7.3.3.3. pku.as.naming.ServerContextPlugin

服务器上下文插件，实现了javax.naming.Context接口。

7.3.3.4. pku.as.naming.SmartCtxFactory

SmartCtx工厂，实现了InitialContextFactory接口，用于创建具有特定环境属性值的SmartCtx初始上下文。

7.3.3.5. pku.as.naming.ServerCtxFactory

ServerCtx工厂，实现了InitialContextFactory接口，用于创建具有特定环境属性值的ServerCtxPlugin初始上下文。

7.3.3.6. pku.as.naming.NamingEnumerationImpl

命名枚举实现，实现了javax.naming.NamingEnumeration接口。主要是将NamingServer返回的非序列化的对象集合转化为序列化的对象枚举，返回给客户端。

7.3.3.7. pku.as.naming.NamingService

该类实现了NamingServiceMBean和pku.as.service.naming.NamingServiceInterface接口，执行对pkuas命名服务的管理。

7.3.3.8. pku.as.naming.NamePaserImpl

命名解析器。

实现了javax.naming.NamePaser和java.io.Serializable接口。

以jndi语法为属性（从左到右分析、不忽略、消除空白、以'/'为分割）。

主要提供两个方法：

- getInstance() 返回一个命名解析器实例
- parse(String) 返回String的CompoundName名称

7.3.3.9. pku.as.naming.NamingServerExporter

这个类将pkuas命名服务器需要被外界调用的方法暴露给外界，提供一个pkuas监听的端口，从而实现RMI远程方法调用。

7.3.4. 其它相关类和接口说明

7.3.4.1. java.rmi.Remote

Remote 接口用于标识其方法可以从非本地虚拟机上调用的接口。任何远程对象都必须直接或间接实现此接口。只有在“远程接口”（扩展 java.rmi.Remote 的接口）中指定的这些方法才可远程使用。

扩展：

pku.as.naming.Naming

7.3.4.2. java.io.Serializable

类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。序列化接口没有方法或字段，仅用于标识可序列化的语义。

实现：

- pku.as.naming.NamingServer
- pku.as.naming.SmartCtx
- pku.as.naming.NamePaserImpl

7.3.4.3. javax.naming.Context

此接口表示一个命名上下文，它由一组名称到对象的绑定组成，声明了一组上下文操作（查找、建立/重新/消除绑定、重命名、列表、创建/销毁子上下文）。

pkuas中的上下文是一颗树，树的节点是含有名称的对象。A/B/C这样的混合名称在命名树中的存储就是:A->B->C.命名树的逻辑存储方法是哈希表。

实现：

- pku.as.naming.SmartCtx
- pku.as.naming.ServerContextPlugin

7.3.4.4. javax.naming.spi.IntialContextFactory

此接口表示创建初始上下文的工厂。

JNDI 框架允许在运行时指定不同的初始上下文实现。初始上下文是使用初始上下文工厂创建的。初始上下文工厂必须实现InitialContextFactory 接口，该接口提供了一个方法，用于创建实现 Context 接口的初始上下文的实例。此外，工厂类必须是公共的，必须有一个不接受任何参数的公共构造方法。

实现：

- pku.as.naming.SmartCtxFactory
- pku.as.naming.ServerCtxFactory

7.3.4.5. javax.naming.NamePaser

此接口用于解析取自分层的名称空间的名称。NameParser 包含解析这些名称所需的语法信息知识（比如从左到右的方向、名称分隔符，等等）。 在使用 equals() 方法比较两个 NameParser 时，当且仅当它们服务于相同的名称空间时才返回 true。

实现：

- pku.as.naming.NamePaserImpl

7.3.4.6. javax.naming.NamingEnumeration

此接口用于枚举 javax.naming 和 javax.naming.directory 包中的方法所返回的非序列化的对象集合。

当某一方法（比如 list()、listBindings() 或 search()）返回 NamingEnumeration 时，在返回所有结果之前，将保留所遇到的所有异常。在枚举结束时抛出异常（由 hasMore() 完成）。

实现：

- pku.as.naming.NamingEnumerationImpl

7.3.4.7. pku.as.common.mbean.ServiceMBean

该接口提供对消息Bean的管理服务。

扩展：

- pku.as.naming.NamingServiceMBean

7.3.4.8. pku.as.service.naming.NamingServiceInterface

此接口提供创建企业级命名上下文的方法，也属于命名服务管理接口。

实现：

- pku.as.naming.NamingService

7.3.4.9. pku.as.common.ServiceBase

该抽象类扩展了javax.management.NotificationBroadcasterSupport类。

而javax.management.NotificationBroadcasterSupport提供 NotificationEmitter 接口的实现。该类可以用作发送通知的 MBean 的超类。

默认情况下，通知调度模型是同步的。也就是说，当某一线程调用 sendNotification 时，将在该线程中调用每个侦听器的 NotificationListener.handleNotification 方法。可以通过重写子类中的 handleNotification 或者通过将 Executor 传递给构造方法来重写此默认值。

如果过滤器或侦听器的方法调用抛出 Exception，则该异常不会阻止调用其他侦听器。不过，如果过滤器、Executor.execute 或 handleNotification 的方法调用（未指定任何 Excecutor 时）抛出 Error，则将该 Error 传播到 sendNotification 的调用者。

通常不会同步调用使用 JMX Remote API 添加的远程侦听器。也就是说，当 sendNotification 返回时，不保证任何远程侦听器都已经收到通知。

实现：

- pku.as.naming.NamingEnumerationImpl

7.3.5. 对EJB3实例的处理

由于EJB3不必通过EJBHome创建Bean, 通过JNDI lookup即可获得EJB对象，因此需要在命名服务中作处理。

PKUAS设计了LookupAware接口，对此提供支持。 容器启动时，将对应JNDI名称的一个LookupAware匿名对象注册到JNDI中。当NamingServer接受到查询请求时，若发现是LookupAware,则回调该接口的lookupObject方法，对于EJB3，会创建相应的EJB3实例。

7.3.6. 应用示例

下面的代码表示调用 192.168.1.22 主机上的PKUAS，Naming端口为2001

```
System.setProperty(Context.PROVIDER_URL, "rmi://192.168.1.22:2001");
System.setProperty(Context.OBJECT_FACTORIES","pku.as.naming.SmartCtxFactory");
InitialContext ctx = new InitialContext();
```

下面的代码表示调用本机上(rmi://127.0.0.1:2001)的PKUAS，Naming端口为 2001

```
System.setProperty(Context.OBJECT_FACTORIES","pku.as.naming.SmartCtxFactory");
InitialContext ctx = new InitialContext();
```

第 8 章 安全服务

8.1. 目标与需求

本模块为J2EE应用中EJB和WEB提供统一的认证和授权服务。整个应用（客户端构件与服务器端EJB构件和Web构件）使用统一的认证机制、安全域和角色映射。在独立接入方式中，安全检查在对服务器端的EJB构件进行调用时进行；在Web接入方式中，安全检查既可以在对Web页面访问时进行，也可以将检查延迟到对EJB构件访问时进行。

8.1.1. 基于容器的安全

在J2EE中，构件的安全通过它们的容器实现。容器通过两种途径提供安全：声明和程序接口。

声明型安全：

通过部署描述符描述。部署者将逻辑安全需求和物理的运行环境安全机制进行映射。

程序接口型安全：

如前所述，通过两个对象的4个API进行。

对于EJB，通过EJBContext的isCallerInRole 和getCallerPrincipal方法。

对于servlet和JSP，通过HttpServletRequest的isUserInRole和 getUserPrincipal 方法。

8.1.2. 分布的安全

某些J2EE产品，可能不同的容器分布于网络上不同位置，这样构件间通信就引入安全问题。因此，要在构件间建立安全关联(共享的安全状态信息)，以保证通信的安全。建立过程如下：

- 为客户端认证目标，或反之。
- 协商安全质量。
- 建立构件间的安全上下文。

建立过程一般是由容器完成的。

8.1.3. 鉴权模型

J2EE鉴权模型基于安全角色的概念。 在部署描述符中的method-permission元素中，描述对方法的安全权限要求。

8.1.4. 角色映射

安全服务依赖于将请求的Principal和资源的安全角色的对应。

8.1.5. HTTP登录网关

J2EE 1.3版本没有定义不同安全策略域的互通。当访问其他J2EE产品时，可能不能互通。此时，构件可能选择先通过HTTP登录到外面的服务器。HTTP over SSL可以提供不同J2EE产品间的安全通信。我们称这种使用HTTP和SSL访问远端服务为HTTP登录网关(Login Gateway)。

8.1.6. 用户认证

根据终端用户的不同，有两种认证：基于Web客户的，和基于应用客户的。

8.1.6.1. web客户

Web 客户必须能够向Web Server认证用户。

WEB server可以使用如下认证机制：

HTTP Basic Authentication：要求客户输入名称和密码。没有加密，使用普通的Base64编码传输。可以通过某些方式加强，比如使用HTTP over SSL。

HTTPS Client Authentication：强认证机制，HTTP over SSL。用户必须拥有一个公钥证书。J2EE平台必须支持该机制。 Form Based Authentication：提供灵活的界面。

Web Single Signon

因为HTTP是无状态协议，但是很多WEB应用需要支持会话。因此，必须保证用户只需要认证一次，就可以访问所有的资源，只是在跨安全域时需要重新认证。容器使用凭证为会话建立安全上下文。

Login Session

在J2EE 平台中, 登录会话由Servlet容器支持。当用户成功认证后，容器为用户建立了登录会话，该会话包括用户的凭证信息。当客户是无状态时，服务器必须为他建立安全上下文，并通过某引用提供给客户使用。Cookie或者URL重写技术可以用于传递该引用。如果使用SSL,则不必依赖应引用。

8.1.6.2. 应用客户

指可以不通过Web Server就可以直接访问EJB的程序实体。常用于对客户的认证。

通常，只有当要访问被保护的资源时，才进行认证，称为延迟认证(Lazy Authentication)。

8.1.7. 用户认证需求

8.1.7.1. web客户

J2EE 产品提供商必须为Web 用户认证支持下述机制：

- Web Single Signon

  所有兼容J2EE的WEB server 必须支持Single Signon。必须支持一次登录会话跨越几个应用。

- Login Sessions

  所有J2EE产品必须支持Servlet规范中定义的登录会话。延迟认证也必须支持。

- Required Login Mechanisms

  三种机制必须被支持。

  HTTP Basic Authentication：包括HTTP over SSL。

  SSL Mutual Authentication：基于证书的认证。下述加密机制必须被支持：

  SSL_RSA_EXPORT_WITH_RC4_40_MD5 和 SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

- Unauthenticated Users

  WEB容器要为客户提供匿名访问身份。如果用户没有被认证，则HttpServletRequest的方法getUserPrincipal返回null。

  EJB规范要求调用EJBContext的方法getCallerPrincipal时，必须返回一个合法的Principal，永不能为null。 而WEB容 器内的构件必须能够访问EJB构件，即使其还没有任何用户通过认证。此时，J2EE产品必须提供一个Principal使用。

8.1.7.2. 应用客户

为了满足EJB容器和WEB容器的鉴权和认证要求，application client容器必须对应用的用户进行鉴权。进行鉴权的技术没有具体规定，可以集成J2EE产品的认证系统以提供single signon； 或者当应用启动时，容器就进行认证；或者延迟认证。

容器要提供一个适当的用户接口，以接受用户认证数据。

应用的回调句柄必须支持javax.security.auth.callback包中标识的所有Callback对象。

8.1.7.3. 对资源的认证需求

企业内的资源可能分布在不同的安全域中，尤其是可能不和构件处于一个安全域。J2EE产品必须能在资源所在的安全域中进行认证。下列机制被支持：

- Configured Identity.

  在部署时声明对资源进行访问的principal和相关的凭证数据。如果认证数据存储在容器中，则J2EE产品提供商负责保密性。必须支持部署时的指定，不能依靠程序代码。

- Programmatic Authentication.

  由应用构件在运行时提供principal和认证数据。应用构件可以通过多种方式获得该数据，包括通过参数，通过上下文等。

8.1.7.4. 鉴权需求

J2EE产品必须满足下列需求：

- Code Authorization

  J2EE产品可能限制使用某些J2SE类和方法，以保证系统安全。满足最低安全要求的应用必须能被部署。

- 对调用者鉴权

  必须支持调用者标识的传播。

对于一个J2EE产品的一个特定应用，其对所有EJB的调用，通过EJBContext的方法 getCallerPrincipal返回的Principal和调用链中的第一个EJB返回的Principal相同。如果调用链的第一个EJB是被servlet或者JSP调用的，则返回的Principal必须和HttpServletRequest的方法getUserPrincipal返回的Principal名称相同。该要求仅仅当EJB使用user-caller-identity时有效。

J2EE 产品也必须支持Run-AS特性。此时，初始调用者的身份不会被传递。

8.1.7.5. 部署需求

所有的J2EE产品必须实现EJB，JSP和Servlet规范中定义的安全控制语义，并且能将部署描述符中的逻辑定义和运行系统对应。J2EE产品必须提供部署工具，以将安全角色和运行时的负责安全的实体对应。

一个应用可能定义一个角色，代表所有认证的用户，或者没有认证的用户。

8.2. PKUAS的设计实现

安全服务作为一个系统服务，由pkuas加载启动。

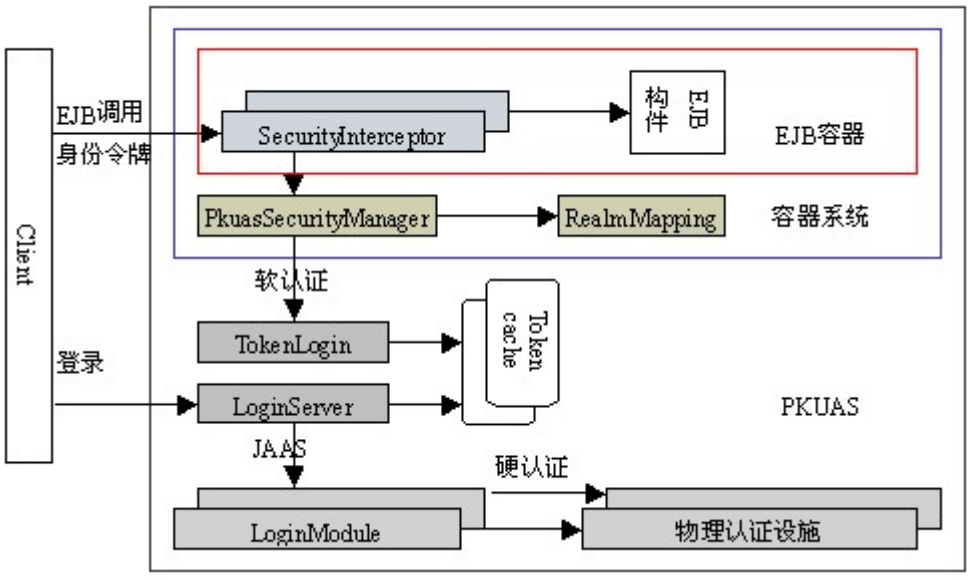
安全服务定义了SecurityServiceMBean接口，由SecurityService类实现该接口。

SecurityService类主要完成以下任务：

- 启动认证服务器LoginServer。
- 启动EJB安全管理器代理PkuasSecurityManagerProxy，该代理实现了远程访问PkuasSecurityManager的接口EJBSecurityManagerRemote，完成对EJB调用的安全检查。
- 启动安全设施管理器代理SecurityAdminProxy。该代理实现了远程管理底层安全认证机制的接口，可以远程增加、删除、修改用户。
- 集成符合JACC规范的授权策略模块，该模块完成应用服务器中的授权检查。应用服务器开发者在security.conf文件中声明授权策略提供商的相关参数，服务启动时，动态加载授权策略模块。

8.2.1. EJB容器的安全

通常，客户端对应用进行一次完整的业务操作流程，包含若干次WEB调用和EJB调用。这样一次完整的业务流程，就是一次“会话”的概念。由于EJB构件在处理收到的请求时，从请求本身得不到会话信息，这就迫使EJB构件对每一次EJB调用都执行安全检查。因此，EJB请求消息中，需要携带安全上下文，包括用户身份和凭证信息（通常是用户名和密码）。然而，在每次请求中都携带敏感的身份和凭证信息是不明智的。因此，在PKUAS的安全体系中，客户执行EJB业务操作前，要到认证服务器显式登录一次，获得一个令牌。该令牌是认证服务器分配给用户的临时身份凭证，代表一次会话。当用户注销登录，或者操作超时，该令牌就会失效。用户以后每次操作都携带该令牌，服务器检查操作请求中令牌的合法性。这样就降低了用户密码等安全信息的被窃取的概率。EJB容器的安全机制如图所示。



上图左边的Client可以是WEB客户端，也可以是Standalone客户端。PKUAS目前利用CORBA IIOP（Internet Interoperable ORB Protocol）消息传递EJB请求，IIOP消息头中特别为传输安全等上下文定义了消息域，安全信息保存在该消息域中。用户执行一次登录获得了令牌后，一个关键问题是如何透明地将安全上下文绑定到IIOP消息中。为了解决该问题，PKUAS为客户端提供了LoginClient模块，客户端只需要简单调用LoginClient提供的登录和注销方法即可。下图给出了

LoginClient执行登录请求的一段代码。在成功登录获得令牌后，将安全上下文设置到SecurityContext对象的线程相关变量中。当EJB客户桩Stub构造IIOP消息时，自动从SecurityContext中取出安全上下文。由于客户桩Stub是由工具自动生成的，因此安全上下文的绑定对用户而言是透明的。WEB客户端的调用略微复杂一些，在调用LoginClient的login后，还要建立令牌和HTTP会话的关联关系。在PKUAS中，令牌和HTTP会话之间关联关系的管理通过扩展Tomcat来实现。在WEB客户端发出EJB调用时， WEB容器会自动在IIOP消息中加入对应会话的安全上下文，对用户而言也是透明的。

```
public static Object login(String domain, Principal sp, Object passwd)
    throws LoginException, RemoteException
{   ……
    try { //variable ls is the remote instance of LoginServer
        token = ls.login(domain, sp, passwd) ;
    } catch (LoginException le) {
        throw le;
    } catch (RemoteException re) {
        throw re;
    }
    SecurityAssociation.setPrincipal(sp);
    SecurityAssociation.setCredential(token);
    return token ;
}
```

8.2.1.1. 安全截取器

在PKUAS中，EJB容器收到请求后，并不是直接将该请求转发给相应的EJB，而是要经过一系列的截取器（Interceptor），进行预处理。截取器机制体现了良好的“关注点分离”（Separation Of Concern）的思想，各种截取器可以完成不同的功能和服务，彼此却互不干扰；同时，截取器提供了对容器的行为进行扩展的能力——要增加新的行为，只需增加新的截取器。目前PKUAS中有三个主要的截取器：安全（ContainerSecurityInterceptor）、事务截取器（ContainerBMTInterceptor、ContainerCMTInterceptor）。

在PKUAS中，一个容器系统逻辑上对应一个客户应用，一个容器逻辑上对应一个应用构件。安全截取器是针对容器的，每个容器一个。安全截取器从EJB请求中取出安全上下文，即用户身份和令牌，交给SubjectSecurityManager进行认证和访问控制检查。

与安全截取器紧密相关的对象有安全管理器（PkuasSecurityManager）、域映射管理器（RealmMapping）、授权管理器（AuthorizationManager）。安全管理器和域映射管理器在一个容器系统范围内唯一；安全上下文中的用户身份是物理身份，在进行访问控制时，必须得到当前应用域为该物理身份定义的逻辑角色，根据逻辑角色判断访问权限。RealmMapping维护了应用安全域中物理用户和逻辑安全角色的映射关系，并提供访问接口。授权管理器负责管理J2EE应用中每个模块（ejb-jar、web-war）的访问控制。

当接受请求的EJB需要调用其它EJB时，调用者EJB传递的用户身份可以是该EJB持有的特定代理身份，即“run-as-specified-identify”方式；也可以将初始消息中的用户身份传递下去，即“use-caller-identity”方式，具体方式依赖于应用组装者在部署描述文件中的设置。安全截取器要负责处理运行时刻的相关工作。

8.2.1.2. 认证服务器

在PKUAS中，存在两种认证服务器：软认证服务器TokenLogin和物理认证服务器LoginServer。物理认证服务器LoginServer提供远程登录接口，负责处理客户的登录请求，根据JAAS的配置，调用相关的登录模块LoginModule，由相应的LoginModule负责实际物理认证。成功登录的用户会得到临时身份凭证—令牌Token。LoginServer维护了令牌缓存列表，记录令牌分配情况。软认证服务器TokenLogin只提供验证EJB调用中令牌合法性的本地接口，它只被PkuasSecurityManager调用。在实现上，TokenLogin和LoginServer是一个实现体。下图给出了TokenLogin和LoginServer的接口。

```
public interface TokenLogin
{
    public boolean verifyToken(String domain, Principal p, Object token)
        throws LoginException, RemoteException;
}
```

```
public interface LoginServer extends Remote
{
    public Object login(String domain, Principal p, Object credential)
        throws LoginException, RemoteException;
    public void logout(String domain, Principal p, Object token)
        throws RemoteException;
}
```

通过上述安全机制，PKUAS实现了J2EE规范的基本安全需求。PKUAS数据传输的安全是通过安全套接层协议SSL实现的，利用了JSSE提供的API和参考实现，在此不再详述。

在JAAS配置文件中说明了应用的安全域jwdomain，配置用于进行认证的认证对象：LoginModule。EJB容器有一个安全截取器，首先截取了Comm模块发送给EJB容器的调用请求，授权给和该容器关联的安全管理器进行认证。

对于缺省的安全管理器JaasSecurityManager，它的认证基于从消息中提取的Principal对象和Credential对象。通过执行配置的LoginModule，得到认证结果。

8.2.1.3. 安全管理器SubjectSecurityManager

SubjectSecurityManager继承自EJBSecurityManager，提供了本地调用接口，SecurityManagerRemote提供了远程访问接口。

本地访问接口SubjectSecurityManager，提供给ContainerSecurityInterceptor使用。远程接口SecurityManagerRemote，提供给WEB服务器使用。因为WEB服务器使用和EJB容器相同的认证机制，而它有可能分布在别处，不和EJB容器在一个JVM中，因此，需要一个远程访问接口。

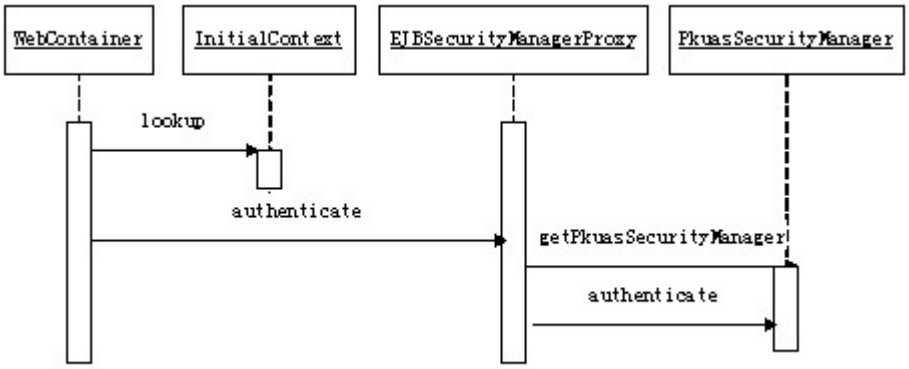
实际中，PkuasSecurityManager类实现了SubjectSecurityManager接口，负责安全检查。PkuasSecurityManager调用软认证服务器TokenLogin进行软认证（所谓软认证，是指没有在实际的安全基础设施中进行物理认证，只是对令牌进行认证调用）；调用EJBAuthorizationManager进行访问控制。PkuasSecurityManager同时

还为EJB编程型安全模型提供支持，EJBContext的接口最后都是通过PkuasSecurityMananger获得相关信息。

远程访问通过代理SecurityManagerProxy实现。SecurityManagerProxy实现了SecurityManagerRemote接口，把对SecurityManagerRemote的调用，转化为对相应PkuasSecurityManager实例的调用。SecurityManagerProxy由安全服务启动，它控制所有应用的远程安全检查，根据调用参数，调用相应的PkuasSecurityManager实例。

SecurityManagerRemote的类定义如下，其中各个方法调用中的domain参数，标识了应用安全域，应用安全域在应用描述文件application.xml中由security-domain项配置。

下面是调用序列图



8.2.1.4. 授权管理器AuthorizationManager

PKUAS中的授权遵循JACC规范，为了更好地进行访问权限的控制，每个jar包和war包分别拥有一个授权管理器EJBAuthorizationManager，WebAuthorizationManager，这两个类集成自抽象类AuthorizationManager。它们的主要职责为：

- 将部署描述符中的有关访问权限的安全声明转换成JACC规定Permission，存放在JACC的PolicyConfiguration中，每个jar包或war包对应一个PolicyConfiguration。
- 管理PolicyConfiguration状态的改变（commit、remove）。
- 进行访问控制检查，事实上授权管理器将访问检查工作交给JACC授权策略模块中的Policy类来完成。　依据JACC的要求，PKUAS自己实现了一个JACC授权策略模块，完成授权的基本功能。改模块包括类PkuasPolicy、PkuasPolicyConfiguration、PkuasPolicyConfigurationFactory。

8.2.2. web容器的安全

一般来说，Web容器的安全重点在控制资源的访问上，但也有一些程序将资源访问的认证与授权和业务逻辑相统一。从PKU-AS的整体考虑，希望能够让用户在整个访问过程中只进行一次登录，这就需要将Web容器的安全和EJB容器的安全统一起来。

Web容器中的安全基于Tomcat的实现，只考虑Servlet引擎部分的安全。这主要是因为从现实情况看，将Tomcat直接作为Web Server并不合适，这就需要PKU-AS能够保持与Web Server之间的独立性，从而保证使用PKU-AS的用户可以根据自己需要选择合适的Web Server。

PKU-AS中Web容器的安全通过修改集成的Tomcat实现。保证：

- 修改后的Tomcat能够符合Servlet规范的规定，实现2.3中提到的标准的身份认证与授权，完全符合规范的应用程序可以不加修改的部署、运行于PKU-AS而不用修改其安全设定。
- 修改后的Tomcat作为为PKU-AS统一的安全认证与授权的一部分，保持与后端EJB Server的一致性。
- 具体实现方式为自定义安全截取器（Security Interceptor）来对用户进行认证和授权。
- Tomcat中不做独立的LoginModule，只是作为一个认证授权的点存在，实际的工作通过调用SecurityManager接口而实现。Tomcat只完成用户信息向SecurityManager的传递、返回认证授权信息（如Pricipal、Cridential）与session的绑定，以及之后每次请求与对应安全上下文的映射。

当客户输入一个网络资源地址、或通过超级链接连向一个网络资源，客户将向WebSever（web容器）发出一个http的客户请求（HttpRequest）。这里，我们只讨论客户将访问一个受保护的资源的情况。

在Tomcat中，有两个RequestInterceptor — AccessInterceptor、SimpleRealmInterceptor — 处理身份认证与授权。为保证PKU-AS对Servlet标准规定的身份认证与授权的支持，暂不改动这两个Interceptor及其在Tomcat中的位置。 从总的过程来看，web容器将向客户发出要求身份认证的请求，要求用户输入用户名和密码，再由web容器根据用户信息进行身份认证和授权。当用户通过认证并得到授权后，web容器正常服务；否则返回出错页面。

第 9 章 事务服务

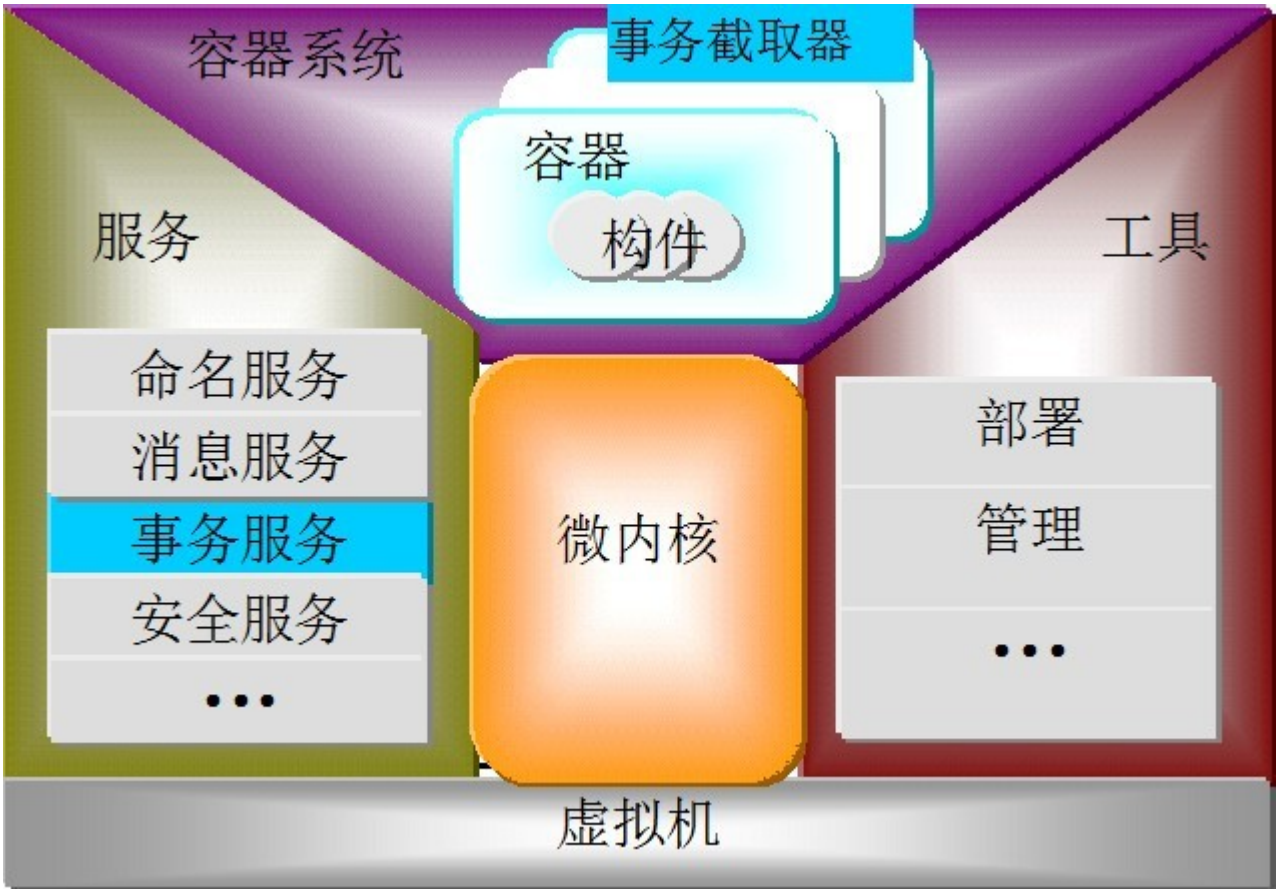
9.1. 目标与需求

开发者在开发EJB构件的时候，可以利用应用服务器提供的事务服务来提高构件运行的可靠性。在构件开发过程中访问事务服务使得构件内部的操作具有ACID性质。PKUAS提供的事务服务给EJB构件、Web构件、Standalone应用、JMS会话等提供了事务语义支持。

事务服务模块的设计和实现提供完全支持JEE规范的事务模型，使得应用服务器可以为应用提供两种访问事务的途径：一种是编程式访问；另一种是声明式访问。PKUAS完全支持J2EE规范的事务模型，可以为应用程序提供强大的事务支持，保证业务操作的进行。

9.2. 方案与设计

PKUAS提供了对JTA事务服务的实现。事务服务是PKUAS公共服务的一部分。在下图中，蓝色部分标明了PKUAS事务服务涉及的内容，其中左侧是事务管理器的实现，也是事务服务的主体；容器系统中的事务截取器（包括EJB构件管理类型和容器管理类型，此外PKUAS对消息驱动容器提供了单独的截取器操作）；事实上，事务服务与公共服务中的消息服务、数据服务均存在交互关系。



pku.as.transaction包中是PKUAS事务管理器的实现类，pku.as.persistence.transaction包提供了持久化上下文中事务管理器的实现，pku.as.application.interceptor则提供了几个事务相关的截取器的实现。

9.2.1. 底层实现

关于事务服务JTA接口的底层实现，PKUAS2010采用了jotm，JOTM (Java Open Transaction Manager)是一个独立的开源事务管理器，它实现了XA协议并且与JTA API兼容。在jotm的事务管理实现中，使用的事务管理类为org.objectweb.jotm.Current，事务类为org.objectweb.jotm.TransactionImpl。

9.2.2. 事务截取

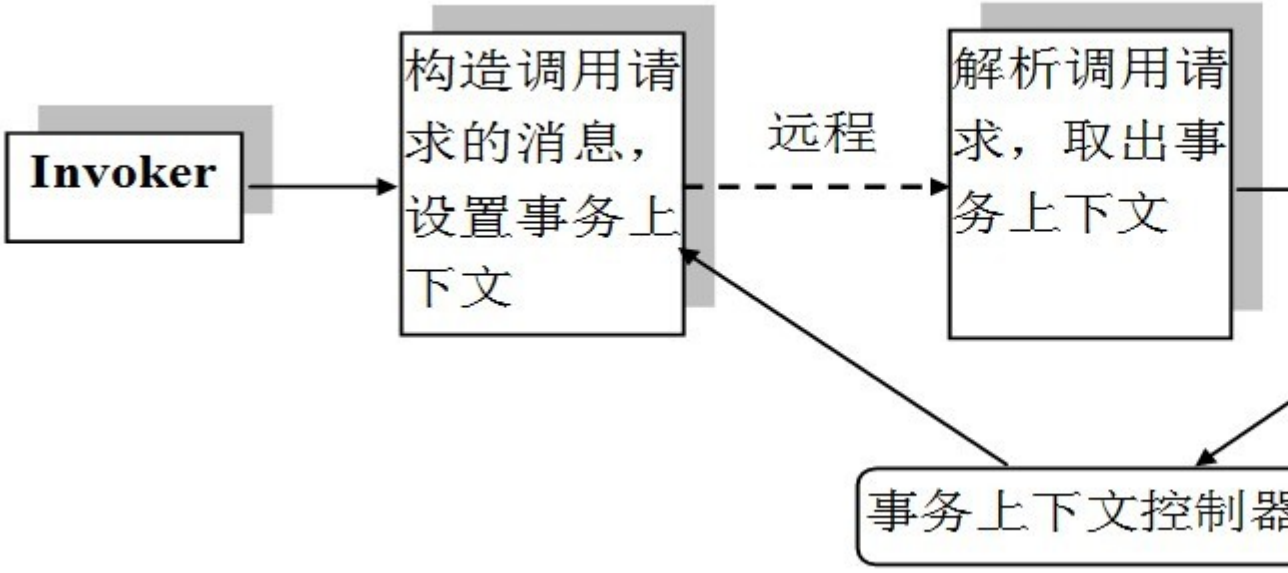
在PKUAS中，提供非功能性服务的公共服务均被封装为一个个截取器的实现，组成一个截取器链。对于事务服务，同样提供了相应的截取器，用于对构件管理的事务和容器管理的事务提供事务处理支持。

在PKUAS的事务截取器的实现中，TransactionInterceptor类继承了DefaultInterceptor类，为和事务服务相关的截取器提供了父类，而CMTInterceptor与BMTInterceptor继承了TransactionInterceptor类，给会话、实体EJB提供了声明式与编程式的事务服务支持。每一个截取器的总体结构是类似的。对于事务服务而言，当调用请求达到容器系统后，容器系统将请求转给截取器链来处理，其中包括事务截取器。在调用Bean的业务逻辑前执行事务截取器的beforeInvoke方法，提供合适的事务支持；在调用Bean的业务逻辑后执行事务截取器的afterInvoke方法，进行事务决策。

9.2.3. 上下文传播

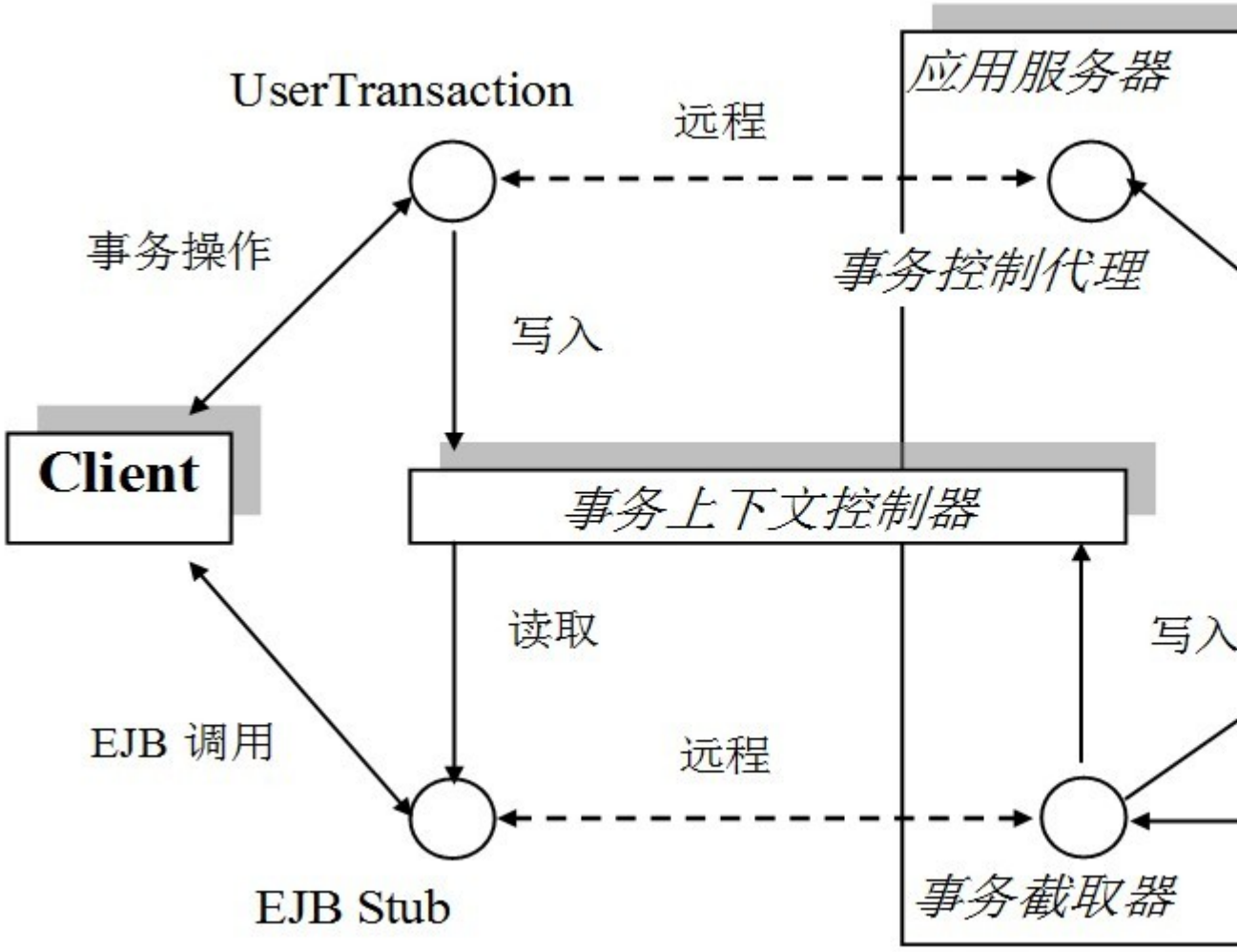
上下文传播是事务服务设计和实现中的关键概念。"传播"的思想使得跨容器（客户应用容器，EJB容器，Web容器，Applet容器）的事务控制成为可能，提供了对大粒度事务的支持。

PKUAS将事务的接收与传播工作都放在事务截取器中进行。事务截取器依据从调用请求中接收到的事务上下文信息和构件部署描述符中的事务声明信息进行事务决策。事务上下文(或者安全上下文等)利用EJB调用进行传播的策略可以给开发者提供最大程度的透明性，开发者甚至不需要知道事务上下文传播这件事情，完成业务逻辑开发后，在运行时刻由应用服务器负责底层的传播细节。PKUAS通过定制自己的RMI-IIOP消息来实现这个技术。如下图所示：



事务上下文控制器负责存储当前线程关联的事务上下文，在容器启动一个事务或者用户编程式启动一个事务时，PKUAS访问事务上下文控制器来设置当前线程的事务上下文。调用者发出调用请求时，首先会构造RMI-IIOP请求。在构造过程中，访问事务上下文控制器获得事务上下文，并在消息体中设置；在事务决策后，依据决策结果重新调整事务上下文控制器的状态。图4中的部署信息是指被调用方法的事务属性信息，J2EE规范提出了六种事务属性信息：
NotSupported、Required、Supports、RequiresNew、Mandatory、Never。这六种事务属性针对有事务上下文传播和没有事务上下文传播的情况，约定了事务决策的内容。当图4中的EJB构件再调用其他构件时，它就充当了Invoker的角色，接下来的过程按照图示进行。

对于客户应用构件调用EJB构件的情形，PKUAS通过引入一个事务操作代理来到达远程的事务发起和传播。在下图中，客户通过访问UserTransaction接口来发起事务。PKUAS在实现这个接口的过程中，让它可以获取位于服务器端的事务控制代理的远程引用，事务控制代理实现：



UserTransaction接口中的所有方法，客户对UserTransaction接口的方法调用通过远程引用转给服务器端的事务控制代理来处理，并向客户返回结果。UserTransaction接口从事务控制代理得到的结果可能是事务的上下文，也可能是事务的状态或其他信息。在发起一个远程事务并得到它的上下文时，同样访问事务上下文控制器设置上下文关联。此后，客户进行EJB调用时的流程前一个图示是一样的。通过这种方式，远程客户可以进行远程事务操作。

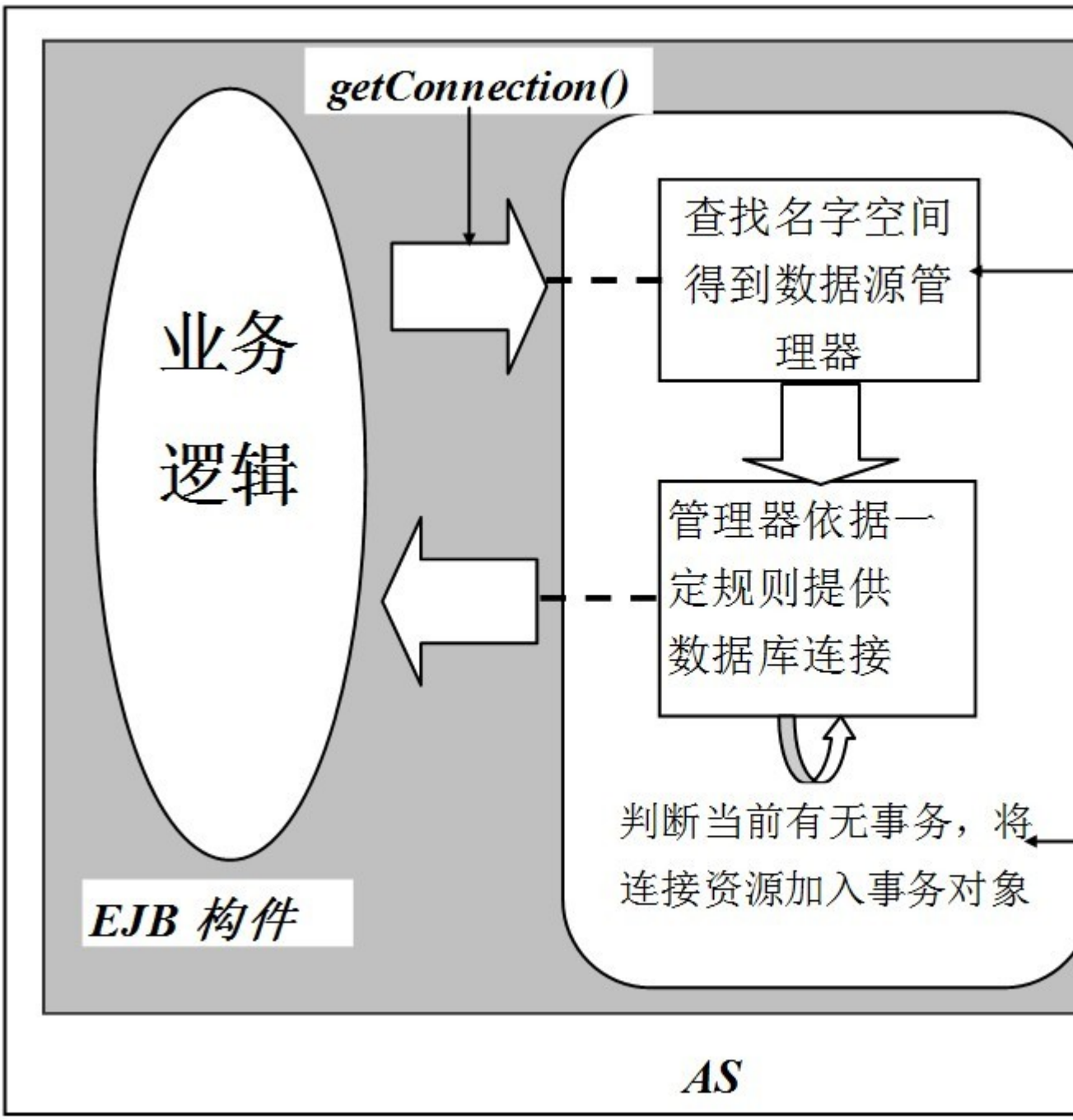
9.2.4. 事务资源管理

JDBC Connection和JMS Session是两类常见的事务性资源。事务性资源在使用过程中可以表现事务性特征。在用户访问事务性资源时，并不需要自己关心这些资源是否真正被包括到一个启动事务中，并体现出事务性特征。应用服务器对事务性资源的访问的设计使得整个过程对用户都是透明的。这种透明性减轻了用户的编程考虑，也提高了应用的可靠性。在JTA事务范畴下，对事务性资源的访问设计是针对EJB构件内部的业务方法。

9.2.4.1. JDBC连接资源

PKUAS提供了对Connection接口的服务器相关实现，EJB构件的业务方法遵循一定的模式来得到这个实现，实现对事务性资源的访问。在下图中，数据源管理器对连接资源的维护和供应、得到和关闭连接时对事务对象的操作是在JTA事务环境下

设计数据库资源使用的关键点。数据源管理器维护了一个连接池，池中的每个连接项保留数据源管理器执行选择算法时需要的信息。连接池中的连接项数目、更新策略、供应和回收方式均由数据源管理器负责。数据源管理器在从连接池中挑选到合适的连接项后，访问事务管理器判断当前事务，如果存在则把连接项中的连接资源加入当前的事务对象。最后将服务器相关的Connection接口实现返回给EJB构件的业务方法。



9.2.4.2. JMS会话资源

在PKUAS中，对于JMS Session的访问流程相对简单。在返回用户Session对象时，判断当前是否处于一个已启动的事务上下文中，如果是则将Session对象加入到这个事务对象中，PKUAS中是把Session接口实现的包装类加入事务对象。最后返

回Session对象给用户。在此过程中，区别两种情况下返回的Session接口实现，以满足对事务性特征的不同需求。

9.3. 使用事务管理

9.3.1. 声明式

在声明式使用方式中，用户不必关心具体的事务操作。哪个构件运行需要事务服务，只要在构件的部署描述文件声明即可，应用服务器负责具体的事务控制。使用这种方式，大大减轻了开发者的关注内容，提高了开发效率。

在J2EE/EJB体系中，容器给构件提供了部署和运行的环境，对构件的调用请求首先要经过容器，容器依据解析构件的部署描述符或者annotation得到的信息，在截取调用请求以及完成构件的业务逻辑后提供事务服务的支持。这种使用方式是容器管理的事务(CMT)，为了使用这种方式，EJB构件的开发者必须依据部署描述文件的DTD或者annotation的要求，正确的声明自己的事务需求。

9.3.1.1. JTA事务中的事务属性

事务属性的设置准确地决定容器管理事务的行为

- NotSupported：如果客户的调用请求中带着事务上下文，容器将切断服务线程和这个事务上下文的关联。在调用结束后，容器负责恢复这个关联。
- Required：如果客户的调用请求中带着事务上下文，容器将在客户的事务上下文中执行业务操作，如果客户没有带事务上下文，容器将发起一个事务，并将这个事务的上下文和当前服务线程关联。
- Supports：客户调用请求带有事务上下文的处理和Required相同，客户调用请求不带有事务上下文的处理和NotSupported相同。
- RequiresNew：客户调用请求带有事务上下文时，容器将切断服务线程和这个事务上下文的关联，并且新发起一个事务，在调用结束后，容器负责恢复这个关联。客户调用请求不带有事务时，容器新发起一个事务。
- Mandatory：客户调用请求带有事务上下文时，处理逻辑与Required相同，客户调用请求不带有事务上下文时，容器将抛出 javax.transaction.TransactionRequriedException。
- Never：客户调用不带有事务上下文时，处理逻辑与NotSupported相同，客户调用带有事务上下文时，容器将抛出例外，如果是local client抛出 javax.ejb.EJBException，如果是remote client抛出 javax.rmi.RemoteException。

可以通过在会话Bean或者消息驱动Bean类上，或者在作为业务接口一部分的那些方法中的一个方法上，标注@TransactionAttributeType注释，来指出一个方法的事务属性，默认属性为REQUIRED。

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public void foo() {
 //...
}
```

9.3.2. 编程式

PKUAS事务服务也支持编程式的事务控制。编程式的事务控制可以为用户提供更灵活的控制方式和更小的控制粒度，对于高级用户和特殊场合可以起到比声明式控制更好的效果。

为了能够手动地开始和提交容器事务，应用程序必须有一个接口支持它，UserTransaction接口就是JTA中指定的对象，应用程序可以维持和调用它来管理事务边界。一个UserTransaction的实例实际上不是当前事务的实例，但是它是一种提供一事务API的代理，并且代表当前事务。可以使用@Resource注释把UserTransaction实例注入到BMT组件中。当使用依赖查找时，可以使用保留的名字"java:comp/UserTransaction"从环境命名上下文中找到它。

```
// UserTransaction接口
public interface javax.transaction.UserTransaction {
 public abstract void begin();
 public abstract void commit();
 public abstract int getStatus();
 public abstract void rollback();
 public abstract void setRollbackOnly();
 public abstract void setTransactionTimeout(int seconds);
}
```

```
// 使用UserTransaction接口
@Resource UserTransaction tx;

void foo() {
 //...
 try {
  tx.begin();
 }
 finally {
  tx.commit();
 } catch (Exception e) {
  // handle exceptions from UserTransaction methods
 }
}
```

第 10 章 消息服务

10.1. 目标与需求

10.1.1. JMS

JMS(Java Message Service)是Sun公司提出的一种基于Java语言的消息中间件的实现规范，它给Java应用提供了进行创建、发送、接收消息的能力。 J2EE1.3规范将JMS纳入，作为J2EE应用服务器必须提供的标准服务之一，同时J2EE规范提出消息驱动EJB作为JMS消息的企业级消费者。一方面JMS消息给J2EE应用提供了进行异步的消息通信的能力；一方面J2EE应用也可以通过消息驱动EJB来有效地处理JMS消息，这样的通信模式给 J2EE应用带来了更强的灵活性，更能适应分布式环境的要求。

10.1.2. JMS的目标

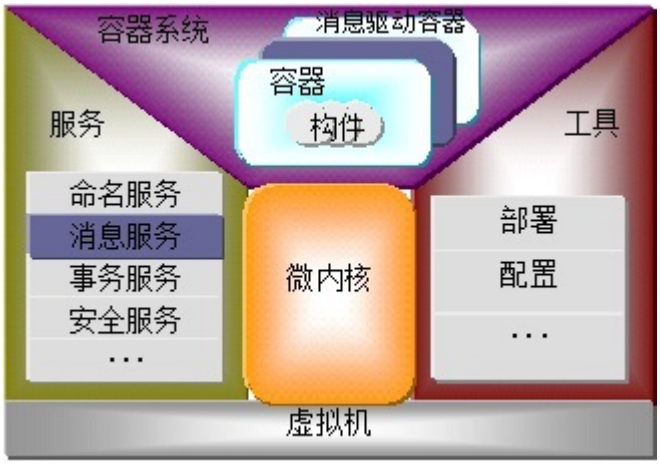
JMS定义了一系列通用的企业消息概念和工具，它试图最小化Java语言程序员使用企业消息产品而必须了解的概念集。 它致力于最大化消息应用的可移植性。

10.2. 方案与设计

10.2.1. PKUAS中的JMS设计方案

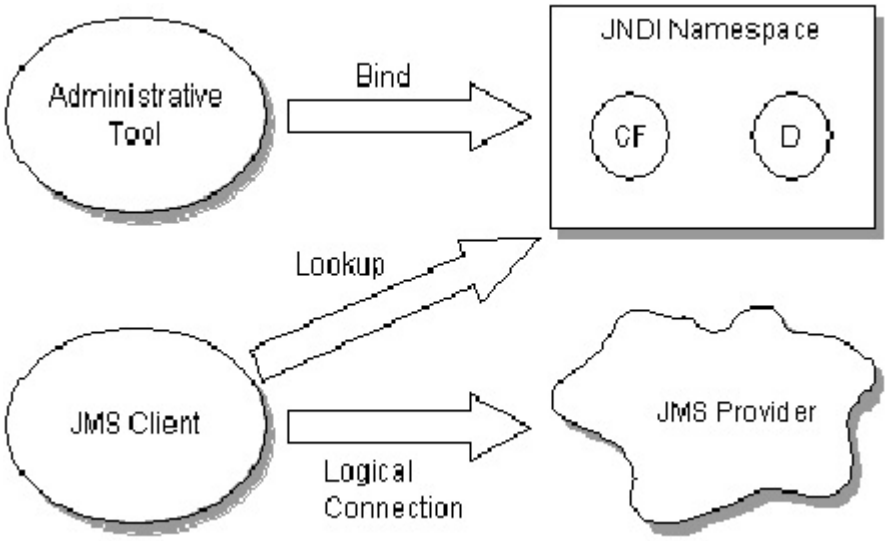
PKUAS通过集成第三方JMS中间件(默认使用的JORAM)作为消息服务的底层实现体，设计和实现了消息驱动容器来为消息驱动的EJB提供运行环境、公共服务支持和实例池管理，此外提供了JMS核心类的封装实现。

JMS消息服务和消息驱动EJB的引入使得运行在PKUAS之上的J2EE应用具有了异步通信的能力。为了提供给不同的应用和平台不同的消息服务，PKUAS提供了多消息的服务框架，应用服务器运行时，可以有多个消息服务同时存在，每一个消息服务可以由不同的消息引擎或者同一个消息引擎的不同实例来实现。



10.2.2. JMS的异步通信模型

JMS的应用模型包括JMS客户（JMS client）、本地客户(Native client)、消息(Message)、JMS供应者(JMS provider)和管理对象(Administered object)五部分。



上面的这个图展示了这几个部分之间的交互方式。管理工具(Administrative Tool)将连接工厂(ConnectionFactory)和消息目的地(Destination)这两类管理对象绑定到名字空间 (Namespace)中，JMS客户从名字空间中获取管理对象后，和JMS供应者发生交互，最终建立JMS客户和消息目的地的关联。在通信时刻，JMS 供应者通过维护JMS客户和消息目的地的关联来缓存和转发JMS消息。 JMS客户之间通过JMS供应者这个媒介进行通信，互相之间可能一无所知。参与通信的JMS客户并不需要同时处于活动状态。发送者发送一个消息后，可以接着做其它工作，而接收者也可以在若干时间以后再激活自己开始接收。JMS供应者负责维护消息，直至消息被接收者处理。JMS提供消息应答 (acknowledgement)、持久化消息(persistence message)和本地事务(Local Transaction)等可靠性机制来确保达到once-and-only once消息传递语义的要求。 针对企业级消息系统对不同消息通信模式的支持，JMS同时支持点对点(Point-to-Point)和发布/订阅(Publish/Subscribe?)两种消息模式，每一种消息模式有一个独立的消息域，在不同的域中使用域相关的API，遵循不同的语义规定。在点对点的消息模式中，消息的接收者只能有一个，而发布/订阅的消息模式支持多个消息接收者。

J2EE1.3规范开始将JMS列为自己的标准服务，使得JMS可以在企业级应用中发挥自己的优势和作用。同时，为支持企业级的消息处理，JMS规范提出了一套支持消息并发处理的机制和相应的API，来协助实现和应用服务器的衔接与交互。依据JMS规范约定的交互模式，实现J2EE应用服务器和JMS供应者的正确衔接，应用服务器才能支持JMS消息服务和消息的并发处理。

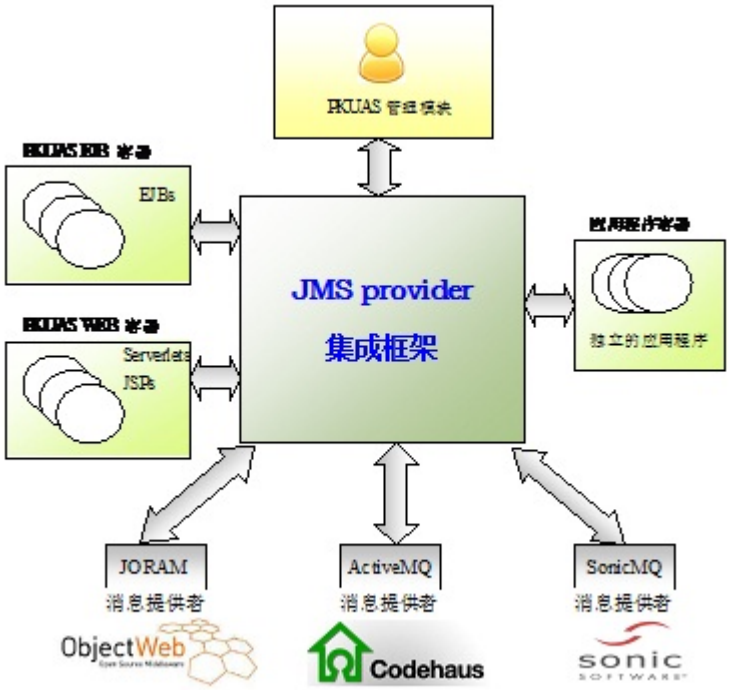
10.2.3. 多消息服务集成框架

PKUAS在实现JMS消息服务时，实现标准规范的基础上，着重考虑体现自己的研究特色。为了提供消息处理能力，并且针对不同的应用和平台提供不同的消息服务，集成了第三方的消息中间件JORAM，ActiveMQ，SonicMQ；为了体现自己的研究特色，通过提供JMS的封装实现来一方面实现和底层消息中间件的连接，另一方面

在封装实现中加入特定于平台的相关设计。此外，PKUAS在系统的配置文件中提供了预设的消息目的地的配置能力。 PKUAS多消息的服务框架可以解决以下问题：

- 由于不同的消息引擎具有不同的特性(性能，安全，可靠，稳定，扩展功能，非标准的编程接口)，所以不同的应用可能需要使用不同的消息服务。
- 遗产系统可能要求使用某个特定的消息引擎（消息引擎特定的编程方式）；
- 消息服务不只是被应用访问，还可以用来构建平台系统如SOA/ESB。不同的平台可能依赖于具体的消息引擎，比如!ServiceMix依赖于ActiveMQ。

PKUAS多消息的服务框架使得应用服务器运行时，可以有多个消息服务同时存在，每一个消息服务可以由不同的消息引擎或者同一个消息引擎的不同实例来实现，使用者可以对每一个消息服务配置JMS管理对象(连接工厂和消息目的地)，应用部署人员可以根据需要选择不同的消息服务。



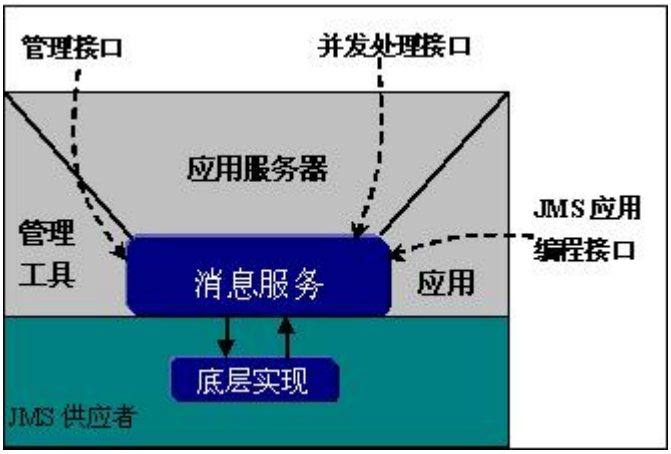
上图为多消息服务集成框架的示意图。它为应用程序提供编程接口(API)，为集成消息提供者提供接口（SPI），为消息驱动EJB的部署提供注册支持和为 J2EE提供管理接口。 API具有通用性、简单性和规范性的特点，它不依赖于消息服务的具体实现，它的使用不应该增加用户程序的复杂性，并且应该符合消息服务规范。在这个集成框架下，只要实现简单的接口(SPI)就可以非常方便地集成不同的消息提供者。消息驱动EJB作为一种特殊消息消费者，它必须注册到J2EE平台的消息系统中，框架提供一种方法，可以让消息驱动容器方便的完成消息驱动EJB的注册。消息服务作为J2EE平台的组成部分，它必须接受平台管理。

10.2.4. PKUAS中的JMS服务实现概况

PKUAS的消息服务主要通过JMS实现。JMS提供持久化消息缓存机制，实现了隐式的异步通信模式，有效地提高了J2EE应用的灵活性和可扩展性。

为了支持消息驱动的EJB，PKUAS提供了消息驱动的容器。对JMS消息服务和消息驱动EJB的支持，使得运行在PKUAS上的J2EE应用具有了异步通信的能力，应用的灵活性得到了扩展，同时参与通信构件的松耦合关联提高了应用的可维护性和可扩展性。

在JMS消息服务的支撑下，J2EE体系的四种容器：EJB容器、Web容器、应用客户(Application Client)容器和Applet容器都可以发送和接受JMS消息，进行异步通信。下图是PKUAS中消息服务和其他相关部分的交互关系图



JMS 供应者是消息中间件的实现体，提供了消息服务的底层实现。应用服务器通过封装JMS供应者提供的API实现类和管理类形成Wrapper层，实现和JMS 供应者的衔接，同时应用服务器参与实现JMS消息的并发处理接口，和消息服务发生交互。此外，消息服务还向PKUAS管理工具提供管理接口。

首先，JMS作为一项标准服务的纳入，使得PKUAS中EJB容器、Web容器、应用客户容器和Applet容器构件之间可以进行消息互操作。在同步通信模式中，四种容器中的应用需要利用名字服务定位远程接口进行互操作，操作的方向性和参与方都有限制，JMS消息扩展了上述基于接口定位的互操作，引入了异步的概念，方便了各类型构件之间的通信。通信的参与者按照为普通JMS客户(使用JMS API、单线程地进行消息发送和接收操作的客户应用, 独立的Standalone应用、Web构件和EJB构件都可以是普通的JMS客户)约定的API可以进行消息的发送和接收。

其次，消息驱动EJB作为一种新的EJB类型，充当了一类特殊的JMS消息消费者。JMS为这类企业级消息消费者约定了不同于普通JMS客户的消息处理流程，多个消息监听者可以并发接收JMS消息，继而通过消息驱动容器指派给多个消息驱动EJB实例处理。这种消息处理机制提高了消息处理能力，体现了应用服务器对JMS消息的企业级处理的支持。消息驱动EJB作为EJB容器中的一种构件，除具有企业级的消息接收能力外也可以像普通JMS客户一样发送和接收 JMS消息

如果在发布/订阅这个消息模型域中进行消息操作，JMS消息的消费者既可以是企业级的消息驱动EJB，也可以是普通的JMS客户，或者两者同时都是消息的消费者；如果是点对点的消息模型域，消息消费者只能有一个，要么是消息驱动的EJB，要么是普通的JMS客户。无论利用JMS消息进行通信的是消息驱动的 EJB还是普通的JMS客户，通信的各个参与者都隐藏在JMS供应者背后，不必获取对方的引用。消息发送和接收的动作在时间上是分离的。约定了消息的格式后，通信参与者只需要知道消息目的地就可以完成通信过程，整个过程满足隐式异步通信的特

10.3. PKUAS中JMS的详细设计

10.3.1. 消息服务的启动

PKUAS支持运行时的多消息服务，可以根据需要选择和不同的消息服务，然后有针对性的进行配置，配置文件在PKUAS-DIR/etc/pkuas.xml。每一个如下的片段代表对一个消息服务的配置，以JORAM为例：

```
<Service Class="pku.as.message.jms.JMSService" Name="JORAM">
 <!--Attribute Name="JMSServer" Value="localhost:16010"/-->
 <Attribute Name="providerAgentClass"
  Value="pku.as.message.jmsadapter.joram.JORAMAdapter" />
 <Properties Name="administeredObjects">
  <!--connection factories used at server side,
   all code running within PKUAS' JVM,include all types of EJBs,
   should uses these connection factories to access JMS-->
  <Set Name="connectionFactory_ServerSideUsedList"
   Value="JMSAgent_CF,joram/JMSAgent_CF" />
  <Set
   Name="queueConnectionFactory_ServerSideUsedList"
   Value="JMSAgent_QCF,joram/JMSAgent_QCF" />
  <Set
   Name="topicConnectionFactory_ServerSideUsedList"
   Value="JMSAgent_TCF,joram/JMSAgent_TCF" />
  <!--connection factories used used by pure JMS client,
   standalone JMS client should uses these connection factories to access JMS-->
  <Set Name="connectionFactoryList"
   Value="jms/ConnectionFactory,jms/joram/ConnectionFactory" />
  <Set Name="queueConnectionFactoryList"
   Value="JQCF,jms/QueueConnectionFactory,jms/joram/QueueConnectionFactory" />
  <Set Name="topicConnectionFactoryList"
   Value="JTCF,jms/TopicConnectionFactory,jms/joram/TopicConnectionFactory" />
  <!-- Destinations defined here!-->
  <Set Name="queueList"
   Value="dummyQueue,JunitQueue1,JunitQueue2,JunitQueue3,joram/JunitQueue1,joram
  <Set Name="topicList"
   Value="dummyTopic,JunitTopic1,JunitTopic2,joram/JunitTopic1,joram/JunitTopic2
 </Properties>
</Service>
```

Class="pku.as.message.jms.JMSService" 指定消息服务的实现类；Name="JORAM"指定消息服务的名字，多消息服务系统中，每一个消息服务应该具有

唯一的名字，这里是指消息服务的名字是 JORAM。 在这个片段中，您可以对消息引擎（JMS Provider）和JMS管理对象进行配置。下面将详细解释各个配置项

- "JMSServer"属性指定JMS Provider的地址，如果要使用PKUAS自带的JMS Provider 则将VALUE设置为空（VALUE=""）或者没有JMSServer这一属性。如果使用自带的JMS Provider，PKUAS会自动启动和关闭JMS Provider；若使用了独立的JMS Provider，则需要手动启动和关闭它

- providerAgentClass指定JMS Provider的一个代理类，每一个消息服务会通过这个类来对JMS Provider进行管理，一般不需要修改此项设置

- "administeredObjects"属性集指定消息服务需要提供的管理对象，分为两类：连接工厂和消息目的地。

  - 第一部分是配置运行在服务器内部的实体(各类EJB，JSP/Servlet，以及其他运行在服务器中的代码)需要的连接工厂：connectionFactory_ServerSideUsedList，queueConnectionFactory_ServerSideUsedLis同一类型的连接工厂有多个时用逗号分隔；

  - 第二部分配置独立运行的程序需要使用的连接工厂：connectionFactoryList, queueConnectionFactoryList, topicConnectionFactoryList同一类型的连接工厂有多个时用逗号分隔

  - 第三部分配置消息目的地，它们可以被服务器中的实体和独立运行的程序使用，queueList和topicList分别用于配置队列和主题，用逗号分隔同一类型的多个消息目的地。

10.3.2. 多消息框架的SPI

在多消息框架中，可以增加消息底层的不同实现机制，但是也要编写相应的适配器，其实现JMSProviderAdapter接口，实现start(), stop(), createQuene(), createTopic(), deleteDestination()等接口。通过相应的适配器使用消息中间件的接口来完成真正服务的实现。

```
public interface JMSProviderAdapter extends Serializable {
    String AS_CONF_HOME = "etcHome";

    /**
     * The MOM will be started if isMOMEmbeded ==true. then create an
     * XAConnectionFactory,an XATopicConnectionFactory, an
     * XAQueueConnectionFactory,a ConnectionFactory,a TopicConnectionFactory and
     * a QueueConnectionFactory.
     *
     * @param isMOMEmbeded
     * @param url
     *            the MOM URL,used when isMOMEmbeded==false
     * @throws Exception
     */
    public void start(boolean isMOMEmbeded, String url) throws Exception;

    /**
     * Stop the JMS Agent,this will stop JMS provider this method should be
     * called only if isMOMEmbeded in start method is true.
     */
    public void stop();

    /**
     * Create a Queue to be named 'name',if a queue with this name already
     * exists in pkuas' name space, the queue will be return immediately.
     *
     * @param name
     */
    public Queue createQueue(String name) throws Exception;

    /**
     * Create a Topic to be named 'name',if a topic with this name already
     * exists in pkuas' name space, the topic will be return immediately.
     *
     * @param name
     */
    public Topic createTopic(String name) throws Exception;

    /**
     * Delete a destination .
     *
     * @param name
     */
    public void deleteDestination(String name) throws Exception;
```

10.3.3. 消息驱动构件的解析和部署

PKUAS在部署消息驱动的EJB构件，创建消息驱动的容器实例时，根据构件部署描述文件提供的信息创建服务器会话池对象 (ServerSessionPool)，然后创建连接对象(connection)。连接对象依据构件描述文件提供的订阅信息和服务器会话池对象，创建连接消费者对象(ConnectionConsumer)。连接消费者对象包含了订阅者的

所有信息，持有服务器会话池对象的引用。启动连接对象后，应用服务器便为消息驱动的EJB参与消息的并发处理做好了准备。连接消费者对象是消息订阅者的底层代理，由JMS供应者发来的消息通过JMS连接首先到达这个对象，它通过服务器会话池对象得到一个或多个服务器会话对象（ServerSession），然后将消息分派给每一个!ServerSession所关联的会话对象(Session)进行处理处理，即可以同时将消息交给一个或多个会话对象，从而实现对并发处理的支持。其中在创建连接消费者对象时提供max- messages参数来控制每个会话对象(Session)一次最多处理的消息数量，Connection.createConnectionConsumer(Destination destination, String messageSelector, ServerSessionPool sessionPool, int maxMessages)，从而控制JMS消息的并发处理程度。

10.3.4. JMS API类的封装

在PKUAS选择集成的消息中间件产品实现了点对点和发布/订阅模式所属的域内容。为了在整合第三方实现的过程中保持一定的灵活性，并且能够利用到 PKUAS其他部分提供的支持，对JMS中每一个域的API，PKUAS进行了选择性包装。目前的包装类包括连接工厂、连接和session。根据JMS 规范，JMS客户和EJB构件在使用JMS API时有不同的语义要求，正确的实现这些语义要求也是PKUAS进行包装实现的一个动机。

连接工厂（ConnectionFactory）

PKUAS中的公共服务包括消息服务。消息中间件产品的启动是作为消息服务的启动的一部分。启动消息中间件产品后，利用它对外提供的管理接口来创建连接工厂。连接工厂一旦创建完毕马上被包装到PKUAS相关的工厂实现类中。连接工厂和目的地(destination)是JMS中的管理对象，由消息中间件建立后，应用服务器负责把它们发布到名字空间中。JMS中利用连接工厂得到连接是需要认证的，认证过的连接可以访问授权给连接的认证用户的目的地。有包装类作为缓冲，应用服务器可以提供默认的认证，或者和服务器的安全机制结合起来，在包装类中进行身份解析。这样对于目的地的访问控制，建立连接控制和EJB构件的安全机制可以有机的结合起

连接(Connection

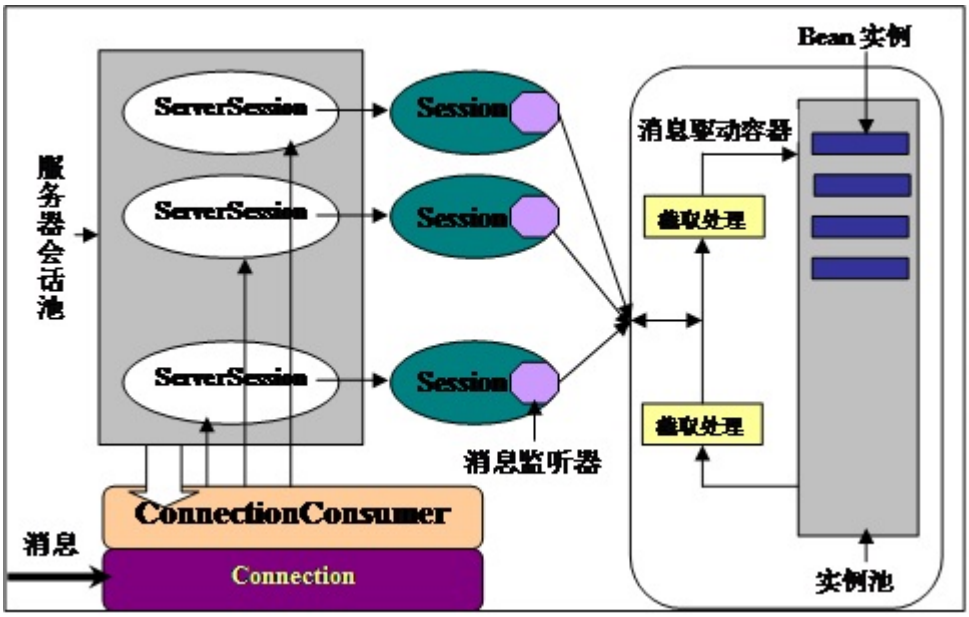
消息中间件产品集成到应用服务器环境中后，比单纯作为一个独立的产品会有额外的限制。为了和应用服务器的服务支持有效的合作，符合该产品实现规范的某些操作可能是不被允许的。对于JMS来说，session对象可以包含一组具有事务特征的操作，在本地事务(LocalTransaction)的支持下，这个对象的所有操作可以被看成是一个原子操作，要么全部执行，要么一个都不执行。这个性质有效的提高了JMS消息操作的可靠性和一致性。在应用服务器环境中，如果客户在EJB构件内部使用这个特性就可能会和应用服务器提供的分布式事务(distributed transaction)冲突。所以在EJB内部不能使用JMS的本地事务操作。Session对象是连接创建的，根据创建时的不同参数设置指名 session对象是否要使用本地事务。限制EJB开发者在开发EJB构件时使用JMS提供的本地事务是必须的，为了做到这一点，PKUAS在连接的包装类中忽略创建session对象的参数设置。EJB开发者在开发时不用考虑这个限制。

Session

Session对象可以使用本地事务，在应用服务器环境中使用分布式事务服务。J2EE体系中JMS Session和JDBC连接一样是事务性资源，在事务上下文中创建一个session时，必须把这个session资源包括到当前事务中去。为了做到这一点，PKUAS在session的包装实现类中加入分布式事务支持，让session对象在创建时可以访问事务管理器，得到当前事务把自己加入当前事务。JQueueSession是在protected javax.jms.QueueSession getMOMQueueSession()进行处理的（TransactionManager tm = (TransactionManager)(TxManager.getInstance())；tx = tm.getTransaction()；）；JTopicSession在protected TopicSession getMOMTopicSession()中进行了类似的处理。 实际上在连接和session中都涉及对是否能使用本地事务的判断。连接中判断是否创建具有本地事务性质的session对象；Session对象的本地事务操作要判断当前操作的合法性。对一个消息中间件产品的集成不能破坏它原本的能力，PKUAS在集成Joram，ActiveMQ，SonicMQ等消息中间件后仍然支持单纯的JMS应用。这样的应用独立于容器环境执行，不受J2EE环境的限制。 PKUAS利用扩展Thread类来执行服务器端的EJB构件操作，通过判断当前线程的实现类来区分是否处于服务器端执行。对于单纯的JMS客户端，就不必限制某些特性的使用，这一切都可以在上面所说的三个包装实现类中实现。 对JMS编程接口的包装使得可以替换底层集成的消息中间件，应用服务器的改动却很少。在消息中间件升级维护时可以更好的体现出它的优势。

10.3.5. 消息驱动对JMS消息的接收过程

连接消费者类是消息订阅者的底层代理，JMS供应者将消息通过底层连接发送到连接消费者类后，它通过持有的PKUAS提供的!ServerSessionPool类，得到一系列的ServerSession，每个!ServerSession都包含一个对Session的引用。连接消费者类将JMS消息指派给各个!ServerSession类后，!ServerSession类再将消息的处理权交给Session类。 PKUAS提供了统一的!MessageListener接口实现，并在Session类中提供这个实现，这样如图5所示，JMS消息经由Session 类，转给!MessageListener接口实现处理。消息驱动容器是应用服务器端JMS消息并发处理路径上的最后一个环节。容器中的处理流程和其他类型EJB的容器类似，首先进行截取处理，然后访问实例管理器得到消息驱动的EJB实例来执行消息消费动作



第 11 章 数据服务

11.1. 目标与需求

Java EE采用多层体系结构设计。表示层面向客户，企业级数据库系统面向最终的持久化存储介质，中间是业务逻辑层，在Java EE中，业务逻辑层通常还分为Web容器层和EJB容器层两个层次。数据服务模块的功能是与企业级数据库系统进行交互，向业务逻辑层提供可用的数据存取服务，把业务逻辑中计算获得的数据进行持久化的保存，并且提供高效而可靠的对这些数据的访问能力。从CMP(容器管理的持久化)引入Java EE规范开始，业务逻辑开发者通常意识不到数据服务模块的存在，CMP的实体会话构件封装了对数据服务的访问操作。所以，从某种意义上说，数据服务与Java EE平台的关系更为紧密，Java EE构件开发者只需要关心如何配置数据源，如何在构件开发过程中声明对数据服务的使用即可。而对于Java EE平台开发者来说，则更关心数据服务的设计和实现，期望能够满足业务逻辑层的需求，同时尽量提高数据操作的效率。

PKUAS中的数据服务提供的功能包括：（1）建立并且初始化与底层数据库的关联，创建满足需求的可用的数据源；(2)提供数据连接池服务，分离物理连接和逻辑连接，以降低频繁的创建和释放数据库连接带来的性能影响；(3)提供对JTA事务的支持。

11.2. 方案与设计

11.2.1. 连接池管理

PKUAS中的数据源实现类为PoolDataSource，PKUAS中的EJB构件和Web构件通过命名服务查找或注释声明（由容器注入）的方式获取数据源实现类的实例，然后调用其getConnection方法获得一个到底层数据库的连接。

PoolDataSource提供了连接池管理的功能。PoolDataSource内部使用Pool类来实现连接池管理的大部分功能。Pool类维护的主要数据结构：空闲连接列表，保存当前未用（即未与任何事务关联）的底层数据库连接；事务→连接映射，用于检索当前同指定事务关联的底层数据库连接，保证处于相同事务中的数据库连接请求总是获得同一个底层数据库连接。

从连接池提供数据库连接服务的生命周期来看，连接池管理分四个阶段进行：连接池初始化、接受数据库连接请求、释放数据库连接、停止连接池服务。

- 连接池初始化

  数据服务启动时，将在空闲连接列表中创建一定数量的空闲连接备用，创建备用连接的数量在外部配置文件中指定。

- 接受数据库连接请求

  PKUAS中的EJB构件或Web构件通过调用数据源接口的getConnection方法请求数据库连接。处理请求时，先检查请求是否处于事务当中。若请求处于事务当中，则先以请求所处的事务从事务→连接映射中检索连接，若不能检索到连接，

再从空闲连接列表中抓取新的连接，并在事务→连接映射中添加新的事务/连接映射项。若请求不是处在事务当中，则直接从空闲连接列表中抓取新连接。

- 释放数据库连接

  释放数据库连接有两种情况：EJB构件或Web构件通过Connection接口的close方法关闭没有处在全局事务中的连接；一个全局事务被提交或回滚后，释放该全局事务中的数据库连接。不管是哪种情况，实际的物理数据库连接并不被关闭，而是在清除与事务有关的一些标志后被加入到空闲连接列表中重复使用。

- 停止连接池服务

  数据服务停止时，连接池中所有打开的到底层数据库的连接被关闭。

11.2.2. 对JTA事务的支持

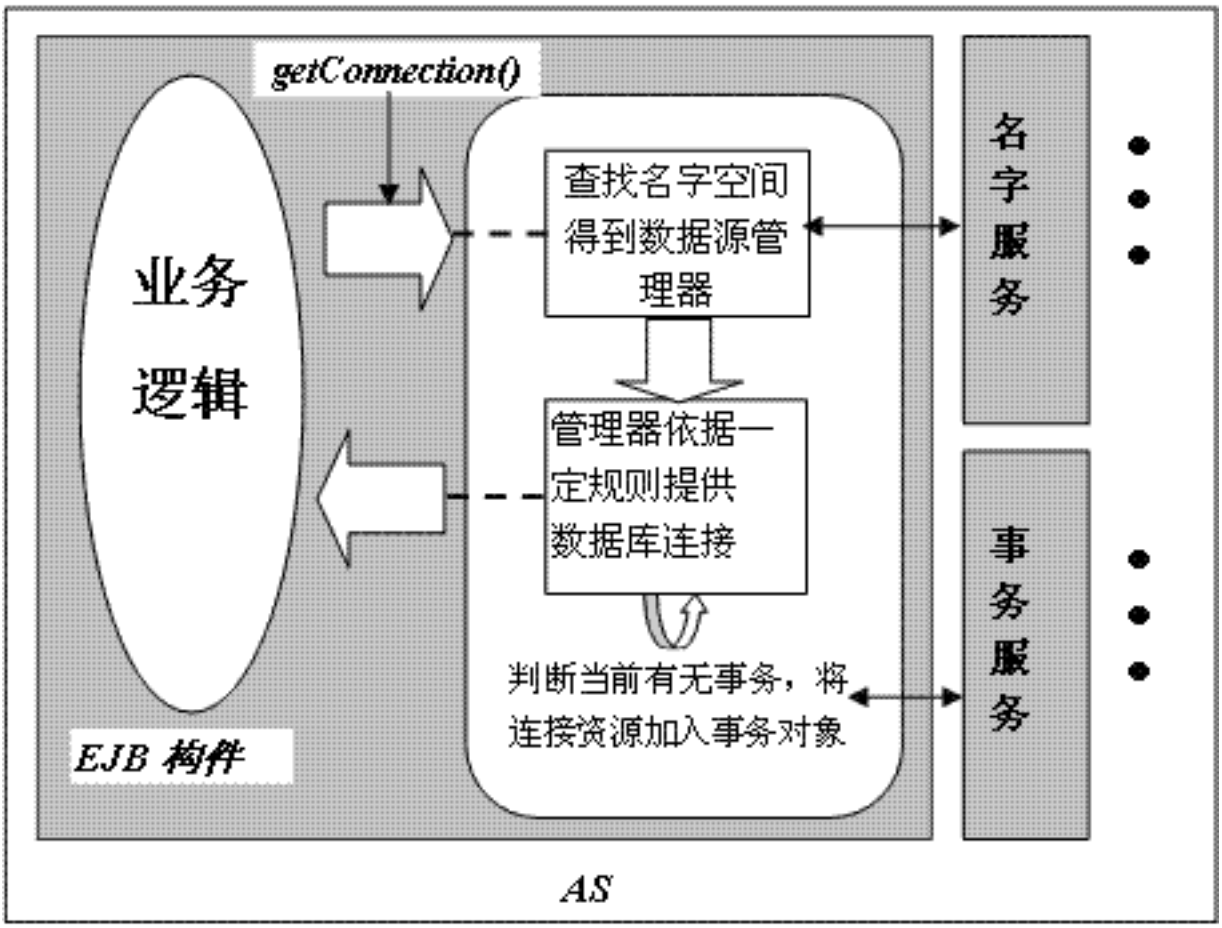
一个典型的非分布式事务可能按下面的步骤进行：获取到数据源的连接；开始事务；利用连接访问数据源中的数据；提交或回滚事务；释放连接。而一个典型的分布式事务可能按下面的步骤进行：开始事务；获取到数据源A的连接；利用连接访问数据源A中的数据；获取到数据源B的连接；利用连接访问数据源B中的数据；提交或回滚事务；释放事务中的连接。下面的示例代码描述了两种事务在程序中的用法。

```
/*非分布式事务*/
DataSource ds;
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
…//Some data access operations on connection conn
conn.commit(); // Or conn.rollback();
conn.close();

/*分布式事务*/
DataSource dsA;
DataSource dsB;
UserTransaction userTransaction;
userTransaction.begin();
Connection connA = dsA.getConnection();
…//Some data access operations on connection connA
Connection connB = dsB.getConnection();
…//Some data access operations on connection connB
userTranscation.commit(); // Or userTransaction.rollback();
```

在比较两种事务的用法的时候，考虑这一点：上面的分布式事务访问了两个数据源，在这两个数据源上的操作是如何关联到同一个事务中去的？在上面的示例代码中找不出哪一行是用来执行这种关联操作的。事实上，在JTA事务模型中这种关联操作是由支持分布式事务的数据源在后台进行的。

PKUAS中的数据服务封装了底层（物理数据库）数据源和数据连接，通过提供对XA接口（XADataSource、XAResource、XAConnection）的实现，提供了与平台事务服务交互的能力。数据库连接是JTA事务模型的基础。要保证数据库连接是事务性的（可以进行回滚和提交操作），必须在获取数据库连接时透明的和事务服务进行交互，如下图所示。

图 11.1. 数据库连接参与分布式事务



EJB构件或Web构件在通过!DataSource接口的getConnection方法获取连接时，!DataSource实现类会与平台的事务管理器交互，判断当前线程是否处于事务当中，如果当前线程处于事务当中，则调用Transaction接口的enlistResource方法将与所返回Connection对象关联的XAResource对象加入到当前线程所处事务的Transaction对象中。XAResource接口提供了在所关联资源（这里即是数据库连接）上的回滚、提交等方法。通过与各个事务性资源关联的XAResource对象，事务管理器就可以统一控制全局事务中各个数据源上的局部事务按两阶段提交方式提交或回滚事务了。

11.2.3. 对JMX管理的支持

数据服务作为PKUAS公共服务的一部分，与其他公共服务一样被实现成符合JMX规范要求的MBean。!PoolDataSource类实现了PoolDataSourceMBean接口，通过该接口PKUAS可以有效而灵活的管理数据服务，包括数据服务的启动、结束和参数调整等。PoolDataSourceMBean接口的定义如下：

```
public interface PoolDataSourceMBean extends ServiceMBean{
 String getServiceName();
 String getDataSourceName();
 String getDriverClassName();
 int getExpirationTime();
 int getMaxCapacity();
 int getMinCapacity();
 String getPassword();
 String getURL();
 String getUser();
 void setDriverClassName(String s);
 void setExpirationTime(int i);
 void setKeepAliveTime(int i);
 void setPollTime(int i);
 void setMaxCapacity(int i);
 void setMinCapacity(int i);
 void setPassword(String s);
 void setURL(String s);
 void setUser(String s);
 void addPoolListener(PoolListener pl);
 void removePoolListener(PoolListener pl);
}
```

11.3. 使用PKUAS中的数据服务

PKUAS中的数据服务注册为PKUAS中的JNDI资源，任何能够访问到PKUAS中的JNDI资源的构件都可以访问PKUAS中的数据服务。为了在PKUAS中正确地启用数据服务，需要在外部配置文件中添加基本的数据源配置项，这些配置项包括：数据源实现类的类名，数据源的JNDI注册名，连接超期时间，数据源中的最少连接数和最多连接数，连接底层数据库时使用的用户名、密码、访问URL、JDBC驱动程序的类名。

访问数据服务前，需要获取在JNDI中注册的数据源对象的引用。有两种方式可供PKUAS中的Web构件和EJB构件选择：使用JNDI命名服务查找、在程序中用注释声明（使用时由容器注入）。如果在外部配置文件中配置的数据源的JNDI名为jdbc/MyDB，则可以按下面的方式获取该数据源的引用。

```
/*使用JNDI命名服务查找*/
public class XxxClass{
  public xxxMethod(...){
    javax.naming.InitialContext ctx = new javax.naming.InitialContext();
    javax.sql.DataSource dataSource=(javax.sql.DataSource)ctx.lookup("jdbc/MyDB");
    java.sql.Connection connection=datasource.getConnection();
  }
```

```
}

/*注释声明*/
public class XxxClass{
  @Resource(name="jdbc/MyDB")
  javax.sql.DataSource dataSource;

  public xxxMethod(...){
    java.sql.Connection connection=datasource.getConnection();
  }
}
```

11.4. 附-PKUAS中的内置数据库

PKUAS内置了关系型数据库Apache Derby。内置数据库主要用来存放运行PKUAS需要的各种系统数据。当然，用户也可以为PKUAS配置基于内置数据库的数据源。在外部数据库不可用，而又需要快速运行一些较小的测试程序的情况下，使用内置数据库就是很好的选择。

PKUAS中的内置数据库将系统数据和应用程序使用的数据分开放置。系统数据（包括应用程序部署信息、安全信息等）被保存到system数据库中，而应用程序使用的数据被保存到pkuas数据库中。PKUAS每次启动时，会将所有应用程序使用的数据删除，所以用户不可以使用内置数据库保存需要长时间存在的数据。

PKUAS中的内置数据库先于PKUAS中的各种服务启动，因此，在PKUAS尚未完全启动的时候，PKUAS的各种服务就可以使用内置数据库。但是直到数据服务启动后，PKUAS中的服务和应用程序都只能通过JDBC驱动程序来访问内置数据库，而不能通过数据源访问内置数据库。下面分别说明这两种访问内置数据库的途径。

(1)通过JDBC驱动程序访问内置数据库

通过JDBC驱动程序访问数据库的关键在于获得连接数据库需要的4个参数：驱动程序类名、连接URL、连接用户名和密码。这几个参数是具体于内置数据库引擎和PKUAS的，参数值如下。

以network server模式（默认模式）启动内置数据库的情况：

- 驱动程序类名：org.apache.derby.jdbc.ClientDriver
- 系统数据库URL：jdbc:derby://主机IP:所配置的端口/system
- 应用数据库URL：jdbc:derby://主机IP:所配置的端口/pkuas

主机IP和端口的值在etc/pkuas.xml中设定。

以exclusive模式（通过设置系统属性pkuas.db.servermode的值为false启用该模式）启动内置数据库的情况：

- 驱动程序类名：org.apache.derby.jdbc.EmbeddedDriver

- 系统数据库URL：jdbc:derby:system
- 应用数据库URL：jdbc:derby:pkuas

为了向应用程序提供这几个参数，PKUAS设置了下面几个系统属性。

- pkuas.db.url.systemdb 连接系统数据库(system)的的URL
- pkuas.db.url.defaultdb 连接应用数据库(pkuas)的URL

Derber的启动参数可在etc/pkuas.xml中设置

```
<!-- Embedded DB -->
<Service BundleName ="pku.as.pkuas-db" Class="pku.as.db.PKUASDerbyDBController">
  <Attribute Name="serverMode" Value="false" />
  <Attribute Name="port" Value="1527" />
  <Attribute Name="addr" Value="127.0.0.1" />
</Service>
```

这三个参数是可选设置的，serverMode缺省为true, port缺省为1527, addr 缺省为 "0.0.0.0"。为了安全，可将addr设置为"127.0.0.1".

PKUAS开放了内置数据库的访问权限，PKUAS中的任何服务或应用程序都能以空用户名连接到内置数据库。下面给出了通过JDBC驱动程序访问内置数据库的代码示例。

```
/*访问系统数据库(system)*/
String url = System.getProperty("pkuas.db.url.systemdb");
java.sql.Connection conn = java.sql.DriverManager.getConnection(url);
//do some operations on the Connection
conn.close();

/*访问应用数据库(pkuas)*/
String url = System.getProperty("pkuas.db.url.defaultdb");
java.sql.Connection conn = java.sql.DriverManager.getConnection(url);
//do some operations on the Connection
conn.close();
```

(2)通过数据源访问内置数据库

也可以在PKUAS中配置基于内置数据库的数据源，然后通过数据源来访问应用数据库(pkuas)。配置方法与基于外部数据库配置数据源几乎没有差别，仅有的不同的是，为JDBC驱动程序配置的4个参数都为空。下面给出了配置示例。

```
<Service Class="pku.as.datasvc.PoolDataSource" Name="jdbc/pkuas">
    <Attribute Name="expirationTime" Value="300"/>
    <Attribute Name="minCapacity" Value="10"/>
    <Attribute Name="user" Value=""/>
    <Attribute Name="password" Value=""/>
    <Attribute Name="URL" Value=""/>
    <Attribute Name="driverClassName" Value=""/>
    <Attribute Name="maxCapacity" Value="100"/>
</Service>
```

配置了数据源，PKUAS中的服务或应用程序就可以通过命名服务来查找数据源，然后从数据源获得到到内置数据库的连接。

(3) 查看内置Derby数据库的状态

Derby提供了一个命令行工具 ij，可去 http://mirror.bjtu.edu.cn/apache/db/derby/ 下载。

运行 ij ij:> connect 'jdbc:derby://localhost:1527/pkuas;';

第 12 章 持久服务

12.1. 目标与需求

PKUAS中的持久模块作为一个相对独立的模块，既可以在脱离容器的独立应用程序中使用，也可以被引入EJB容器，为EJB提供持久服务。 目前，PKUAS集成toplink essentials作为其JPA实现。

12.2. 方案与设计

12.2.1. 持久化的概念

12.2.1.1. 持久化单元和实体管理器工厂

一个持久化单元（persistence unit）逻辑上包括以下内容：

- 一个实体管理器工厂、由实体管理器工厂创建的多个实体管理器、持久化单元的配置信息；
- 持久化单元中被管理实体类的集合；
- 用来将实体类映射到关系表的元数据。

通常，在程序中为每个持久化单元创建一个实体管理器工厂，并让该实体管理器工厂被各个线程共享，当某个线程向实体管理器工厂请求实体管理器时，实体管理器工厂使用持久化单元的配置信息（在一个名为persistence.xml的文件中定义）创建一个定制的实体管理器，返回给请求线程使用。

持久模块按下面的策略来确定持久化单元中包含的被管理实体类。

(1) 可以在persistence.xml文件中显式声明被管理实体类。

(2) 若是在容器中使用持久服务，容器根据所部署应用的位置确定一个实体类的搜索路径；若是在独立应用程序中使用持久服务，持久模块根据persistence.xml文件所在的META-INF目录确定一个实体类的搜索路径。

(3) 若在persistence.xml文件中指定只管理显式声明的被管理实体类，则在persistence.xml文件中显式声明的各个被管理实体类的集合就是持久化单元的被管理实体类集合；否则，除了要将在persistence.xml文件中显式声明的被管理实体类加到持久化单元的被管理实体类集合中，持久模块还要从前面确定的实体类搜索路径开始搜索class文件，将那些在class文件中定义的声明了javax.persistence.Entity注解的类也加入到持久化单元的被管理实体类集合中。

持久化单元中的映射元数据可以有两种提供方式：一种方式是以Java注解的形式在类文件中提供，另一种方式是以XML配置文件的形式在类文件外部提供。

12.2.1.2. 持久化上下文和实体管理器

持久化上下文（persistence context）是一个由被管理实体类的实例组成的集合（注意与持久化单元中被管理实体类集合的区别），所有这些实例都处于被管

理状态。每个实体管理器背后都要关联一个持久化上下文，实体管理器负责持久化上下文中实例的创建并控制这些实例的生命周期。应用程序通过实体管理器进行实体持久化操作，而事实上，实体管理器将具体访问数据库的操作交给持久化上下文以异步方式来完成。

有两种类型的持久化上下文：事务范围（transaction-scoped）的持久化上下文和扩展（extended）持久化上下文。持久化上下文的范围决定了被管理实体受管理的生命周期，若是事务范围的持久化上下文，则在事务结束时，持久化上下文中的所有实体在与后台数据库取得同步后即变为不受管理的实体，若是扩展持久化上下文，虽然在事务结束时也要将持久化上下文中的所有实体与后台数据库同步，但是这些实体仍然是在持久化上下文中受管理的实体，只有当实体管理器实例不再存在时，持久化上下文中的实体才会变为不受管理的实体。

究竟使用哪种范围的持久化上下文，这是由使用持久服务的容器或应用程序决定的，但是也有使用惯例。通常，容器中的无态会话bean通过为javax.persistence.PersistenceContext注解的PersistenceContextType元素指定TRANSACTION值（默认值）声明由容器注入使用事务范围的持久化上下文的实体管理器；有态会话bean则通过为javax.persistence.PersistenceContext注解的PersistenceContextType元素指定EXTENDED值声明由容器注入使用扩展持久化上下文的实体管理器；而由于独立应用程序总是自行创建实体管理器工厂、实体管理器，可以自由控制销毁实体管理器的时机，从而可以让实体管理器关联的持久化上下文跨越多个事务。

需要注意，实体管理器不是线程安全的，因此不要让多个线程并发访问同一个实体管理器。通常的做法是，为每个线程分配一个私有的实体管理器。

12.2.2. PKUAS的持久化实现

容器在部署JPA应用的过程中，需要主动去发现在JPA应用中定义的持久化单元（持久化单元的信息在persistence.xml文件中配置），并将持久化单元的配置信息交由JPA实现处理。

在PKUAS中，EJB容器使用pku.as.container.spi.InjectionDeployer的实现类pku.as.jpa.JPADeployer来发现并调用JPA实现部署在JPA应用中定义的持久化单元。JPA实现在成功部署一个持久化单元后，向EJB容器返回所部署持久化单元的实体管理器工厂（javax.persistence.EntityManagerFactory）实例。实体管理器工厂实例将被EJB容器用来创建供JPA应用操纵实体的实体管理器（javax.persistence.EntityManager）实例。

```
package pku.as.jpa;

public class JPADeployer implements InjectionDeployer {
 public void deploy(ApplicationInfo appInfo, String moduleName, EJBClassLoader ejb
     //read persistence unit information
        PersistenceUnitReader persistenceUnitReader = new PersistenceUnitReader((E
        List<PersistenceUnitInfo> unitList = persistenceUnitReader.readPersistence

        //get the JPA provider in PKUAS
        PersistenceProvider persistenceProvider = ApplicationManager.getPersistenc
```

```
            if(persistenceProvider == null)
                throw new Throwable("No persistence provider is available");

            //deploy the persistence units
            Map<String, EntityManagerFactory> persistenceUnitMap = new HashMap<String,
            for(PersistenceUnitInfo unit: unitList){
                EntityManagerFactory factory = persistenceProvider.createContainerEnti
                persistenceUnitMap.put(unit.getPersistenceUnitName(), factory);
            }
            appInfo.addPersistenceUnitMap(moduleName, persistenceUnitMap);
        }
    }
```

表 12.1.

配置项	标签	说明	在PKUAS中的处理方式
持久化单元名称	persistence-unit	每个持久化单元都应有一个唯一的名称	必须在persistence.xml文件中指定
事务类型	transaction-type	在JPA实现使用JTA事务还是资源本地事务	总是使用容器中的JTA事务
JPA实现（JPA Provider）	provider	JPA实现提供的javax.persistence.实现类的类名	不需要spi.PersistenceProvider
JTA数据源	jta-data-source	供JPA实现使用的JTA数据源的JNDI名称	必须在persistence.xml文件中指定
非JTA数据源	non-jta-data-source	供JPA实现使用的非JTA数据源的JNDI名称	不指定，总是使用JTA数据源
映射文件	mapping-file	定义对象关系映射信息的配置文件	若存在，则在persistence.xml文件中指定
jar文件	jar-file	指定JPA实现从中搜索实体bean的jar文件	若存在，则在persistence.xml文件中指定
class	class	显式指定的实体bean的类名	若存在，则在persistence.xml文件中指定
exclude-unlisted-classes	exclude-unlisted-classes	是否排除未显式指定的实体bean	默认值为false，可以在persistence.xml文件中指定新值

配置项	标签	说明	在PKUAS中的处理方式
属性	properties	一个或多个具体于JPA实现的属性	若存在，则在persistence.xml文件中指定

对于上面的配置项，需要特别指出的是，PKUAS容器中的持久化单元总是使用JTA事务和JTA数据源（容器管理的实体管理器使用的都是JTA事务，而应用程序管理的实体管理器可以使用两种事务中的任何一种）。默认搜索未显式指定的实体bean。

12.2.3. 对JPA实现的包装

尽管toplink essentials可以用在任何Java EE应用服务器中，我们仍然需要对其作必要的包装，以使其能够与PKUAS良好衔接。对toplink essentials的包装主要包括两项工作：（1）自定义事务控制器；（2）转发日志。

12.2.3.1. 自定义事务控制器

在JPA实现中使用Java EE应用服务器中的JTA事务，具体到PKUAS中就是要让toplink essentials获得在PKUAS的JNDI上下文中注册的的事务管理器。toplink essentials提供了一种机制，允许JPA应用通过配置项指定一个toplink事务控制器（oracle.toplink.essentials.sessions.ExternalTransactionController）的实现类，oracle.toplink.essentials.sessions.!ExternalTransactionController中定义了用来操纵事务的各种方法，toplink essentials通过这个事务控制器来操纵JTA事务，如开始事务、提交事务等。我们不需要创建一个直接实现oracle.toplink.essentials.sessions.!ExternalTransactionController的类，我们选择继承toplink essentials中的oracle.toplink.essentials.transaction.JTATransactionController，oracle.toplink.ess间接实现了oracle.toplink.essentials.sessions.ExternalTransactionController，我们只需覆盖oracle.toplink.essentials.transaction.JTATransactionController中的acquireTransactionManager方法，在这个方法中查询PKUAS中的事务管理器并返回给JPA实现即可。我们为PKUAS提供的自定义事务控制器为pku.as.jpa.toplink.PKUASTransactionController。

```
package pku.as.jpa.toplink;

public class PKUASTransactionController extends oracle.toplink.essentials.transact
 public static final String TRANSACTION_MANAGER_JNDI_NAME = "java:transaction/Tran

 @Override
 protected TransactionManager acquireTransactionManager() throws Exception {
  Context context = new InitialContext();
  TransactionManager txManager = (TransactionManager)context.lookup(TRANSACTION_MA
  return txManager;
 }
```

```
}
```

在实现了自定义的事务控制器后，还需要为JPA应用配置使用该事务控制器，也就是为持久化单元配置属性项toplink.target-server，属性值为自定义的事务控制器的类名。我们不希望将这种涉及应用服务器内部实现的配置暴露给应用程序，因此持久化单元的这一属性只能在PKUAS内部自动赋值。PKUAS并没有直接使用toplink essentials提供的javax.persistence.spi.PersistenceProvider实现类oracle.toplink.essentials.PersistenceProvider，而是使用我们自己提供的从该类继承的一个子类pku.as.jpa.toplink.PersistenceProvider，我们将为持久化单元的toplink.target-server属性赋值的工作放在这个子类中完成。这样一来，在JPA应用中就不需要专门为持久化单元指定自定义事务控制器的实现类了。

```
package pku.as.jpa.toplink;

public class PersistenceProvider extends oracle.toplink.essentials.PersistenceProv
 protected static final String TRANSACTION_CONTROLLER_CLASS_NAME = "pku.as.jpa.top

 @Override
 public EntityManagerFactory createContainerEntityManagerFactory(PersistenceUnitIn
  ...
  props.put("toplink.target-server", TRANSACTION_CONTROLLER_CLASS_NAME);
  ...
  return super.createContainerEntityManagerFactory(unitInfo, props);
 }
}
```

12.2.3.2. 转发日志

PKUAS提供了基于apache commons logging的日志服务，toplink essentials也有自己的日志实现，有必要将toplink essentials的日志转发给PKUAS中的日志实现，由PKUAS中的日志实现统一处理toplink essentials的日志和其他来源的日志。

toplink essentials允许在JPA应用中配置日志实现类，PKUAS通过为JPA应用配置自定义日志实现类来实现日志转发。日志实现类必须实现toplink essentials提供的oracle.toplink.essentials.logging.SessionLog接口。同样，我们不需要直接实现oracle.toplink.essentials.logging.SessionLog接口，可以继承间接实现了这一接口的oracle.toplink.essentials.logging.DefaultSessionLog类。PKUAS为JPA应用配置的自定义日志实现类为pku.as.jpa.toplink.PKUASLogger。

```
package pku.as.jpa.toplink;

public class PKUASLogger extends oracle.toplink.essentials.logging.DefaultSessionL
 private static Log logger = LogFactory.getLog(pku.as.jpa.toplink.PersistenceProvi
```

```
@Override
public synchronized void log(SessionLogEntry logEntry) {
 int level = logEntry.getLevel();
 String message = formatMessage(logEntry);
 if (level == 1)
  finest(message);
 else if (level == 2)
  finer(message);
 else if (level == 3)
  fine(message);
 else if (level == 4)
  config(message);
 else if (level == 5)
  info(message);
 else if (level == 6)
  warning(message);
 else if(level >= 7)
  severe(message);
}

@Override
public void info(String message) {
 logger.info(message);
}

@Override
public void warning(String message) {
 logger.warn(message);
}

@Override
public void config(String message) {
 logger.debug(message);
}

@Override
public void fine(String message) {
 logger.debug(message);
}

@Override
public void finer(String message) {
 logger.debug(message);
}

@Override
public void finest(String message) {
 logger.debug(message);
```

```
 }

 @Override
 public void severe(String message) {
  logger.error(message);
 }
}
```

pku.as.jpa.toplink.PKUASLogger覆盖了oracle.toplink.essentials.logging.!DefaultSessionLog中的日志输出方法，在这些日志输出方法中，将日志信息转发到PKUAS中的org.apache.commons.logging.Log。需要注意的是，oracle.toplink.essentials.logging.!DefaultSessionLog使用了OFF、SEVERE、WARNING、INFO、CONFIG、FINE、FINER、FINEST八级日志，而org.apache.commons.logging.Log使用了OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL七级日志，在转发日志时，需要考虑从oracle.toplink.essentials.logging.!DefaultSessionLog的一个级别转发到org.apache.commons.logging.Log的哪一个级别。pku.as.jpa.toplink.PKUASLogger采用的方案是：OFF-OFF、SEVERE-ERROR、WARNING-WARN、INFO-INFO、CONFIG-DEBUG、FINE-DEBUG、FINER-DEBUG、FINEST-DEBUG。

同样，在实现了自定义的日志实现类之后，也需要为JPA应用配置使用该日志实现类。JPA应用的日志实现类也是在toplink essentials的持久化单元定义中通过属性指定的，因此可以按照配置事务控制器的方式来配置日志实现类，也就是在pku.as.jpa.toplink.!PersistenceProvider类中为持久化单元指定属性值，相关代码如下：

```
package pku.as.jpa.toplink;

public class PersistenceProvider extends oracle.toplink.essentials.PersistenceProv
 ...

 @Override
 public EntityManagerFactory createContainerEntityManagerFactory(PersistenceUnitIn
  ...
  props.put("toplink.logging.logger", "pku.as.jpa.toplink.PKUASLogger");
  props.put("toplink.logging.level", "FINEST");
  ...
  return super.createContainerEntityManagerFactory(unitInfo, props);
 }
}
```

这里添加了两个属性。toplink.logging.logger为日志实现类的类名，而toplink.logging.level为日志输出级别。将日志输出级别设为FINEST，是为了转

发所有toplink essentials的日志，实际的日志输出级别由PKUAS中的日志服务决定。

12.3. 应用与接口

下文介绍了如何在EJB中获取与使用持久化服务。

12.3.1. 引用持久化上下文

JavaEE组件支持对资源引用的概念。引用是一个资源的命名链接，可以在运行时从应用代码内部动态地解析它，或者在组件实例被创建的时候由容器自动地解析。声明一个引用，会使用以下资源引用注释之一：
@Resource、@EJB、@PersistenceContext或@PersistenceUnit。这些注解可以被旋转在类、数据成员或者setter方法上。不同位置的选择决定了引用的默认名，以及是否服务器会自动解析这个引用。

在JavaEE环境中，持久化单元的EntityManagerFactory可以用@PersistenceUnit注释来引用，以实现实体管理器工厂。unitName用来标注其名字。

```
@PersistenceUnit(unitName="my_emf");
EntityManagerFactory emf;
```

而@PersistenceContext注释可以被用来声明一个持久化上下文的依赖，并使这个持久化上下文的实体管理器自动被获取。

```
@Persistence
EntityManager em;
```

12.3.2. 容器管理的持久化上下文类型

通过@PersistenceContext注释获得的实体管理器叫做容器管理的(Container-Managed)，其持久化上下文有两种类型。 最常见的叫做事务范围的(Transaction-Scoped)，由type=PersistenceContextType.TRANSACTION指定，缺省时默认该值，事务范围意味着实体管理器管理的持久化上下文由活动的JTA事务决定。每次在实体管理器上调用操作，它都会检查是否有持久化上下文与事务关联。如果有，那么实体管理器就会使用这个持久化上下文，否则会创建一个新的持久化上下文，并将其与事务关联起来。当事务结束时，这个持久化上下文也会消失。另一种类型叫做扩展的(Extended)，由type=PersistenceContextType.EXTENDED来设定，该类型的实体管理器需要有状态的会话Bean，在有状态会话Bean被创建时创建持久化上下文，并一直存在，直到Bean被删除。

12.3.3. 实体管理器操作

表 12.2.

API	描述
public void persist(Object entity)	持久化一个实体，让它成为托管的。但并不意味着它会被立刻持久化到数据库中，需要等待同步操作
public void remove(Object entity)	从持久化上下文中删除实体
public <T> T find(Class<T> entityClass, Object primaryKey)	通过主键获取实体
public void flush()	刷新以使持久化上下文与数据库同步
public void refresh(Object entity)	以数据库中的状态信息覆盖持久化上下文中的被托管实体
public void clear()	清理持久化上下文
public boolean contains(Object entity)	检查某实体是否由当前的实体管理器托管
public Query createQuery(String qlString)	创建JPQL查询语句

第 13 章 日志服务

13.1. 目标与需求

应用服务器需要某种机制，在软件开发中提供排错服务。并且需要解决传统的系统输出不能解决的问题，如：输出信息的控制，输出目的地的选取，需要部分输出……

日志就是为了满足开发过程中的这种需求。首先，它能精确地提供运行时的上下文，一旦在程序中加入了日志代码，它就能自动的生成并输出日志信息而不需要人为的干预。另外，日志信息的输出可以被保存到一个固定的地方，以备以后研究。除了在开发过程中发挥它的作用外，一个性能丰富的日志记录软件包还能当作一个审计工具使用。

考虑到复用性、复杂性等方面，要求日志服务应该满足以下需求，达成相应目标：

1. 日志记录与输出；
2. 支持灵活的日志分类记录方式；
3. 支持多线程：日志服务会在多线程环境中使用，需要确保线程安全性；
4. 稳定性：日志服务必须保持高度的稳定性，不能因为内部错误导致业务代码的崩溃；
5. 高性能：日志服务需要提供高速的日志记录功能以应对大请求流量下业务系统的正常运转。

在满足以上需求的情况下，通过PKUAS的日志服务，可以记录系统和应用在运行过程中各个模块的详细信息以及应用产生的各种异常。对于开发者，可以通过查看日志内容进行系统的排错及维护；对于一般用户，可以通过查看日志内容了解和追踪应用运行的过程和状况，借助日志来发现应用在运行过程中出现的问题。

13.2. 方案与设计

13.2.1. 体系结构

PKUAS日志服务采用的是log4j与commons-logging二者结合的模式。下面首先对log4j，commons-logging进行介绍，然后具体介绍在PKUAS-OSGI-2010中的日志服务体系情况。

13.2.1.1. Log4j简介

Log4j是Apache的一个开放源代码项目，通过使用Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI组件、甚至是套接口服务器、NT的事件记录器、UNIX Syslog守护进程等；我们也可以控制每一条日志的输出格式；通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程。这些可以通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。

Log4j主要由三大类组件构成：

1）Logger

负责输出日志信息，并能够对日志信息进行分类筛选，即决定哪些日志信息应该被输出，哪些该被忽略。Loggers组件输出日志信息时分为5个级别：DEBUG、INFO、WARN、ERROR、FATAL。这五个级别的顺序是：

```
DEBUG<INFO<WARN<ERROR<FATAL
```

如果设置某个Logger组件的级别是P，则只有级别比P高的日志信息才能输出。Logger是有继承关系的，最上层是rootLogger，定义的其他Logger都会继承rootLogger。

2）Appender

定义了日志输出目的地，指定日志信息应该被输出到什么地方。输出的目的地可以是控制台、文件或网络设备。

3）Layout

通过在Appender的后面附加Layout来实现格式化输出。

一个Logger可以有多个Appender，每个Appender对应一个Layout。

13.2.1.2. commons-logging

Commons-logging在org.apache.commons.logging包中，它是对多种日志APIs的简单封装包。该包为服务器端程序的日志处理提供API以使用多种不同的日志系统。包括如下已经实现的：

- Log4J: Apache Jakarta 项目。每个Log的实例都对应于一个Log4j Category类。
- JDK: Logging API JDK1.4及后续版本中。每个Log的实例都是一个java.util.logging.Logger实例。
- LogKit: Apache Jakarta 项目。每个Log的实例都对应于一个LogKit Logger类。
- NoOpLog: 简单地接受将所有的Log实例的日志输出，。
- SimpleLog: 将所有的Log实例的日志输出到 System.out中。

13.2.1.3. commons-logging和Log4j的结合

由于Commons-logging的目的是为“所有的Java日志实现”提供接口，它提供了简捷、统一的接口，不需要额外配置，但它自身只实现了一个简单的SimpleLog，如果仅仅单独使用，则提供的日志功能很弱；Log4j的功能全面、强大，能从多方面满足对于日志功能的需求。

PKUAS使用commons-logging，集成了log4j作为其日志服务，采用了二者结合的模式，综合利用到两者各自的优点。这样的设计方案简化了使用和配置，同时，日志功能作为一项服务，加载在PKUAS上的，使得用户可以根据对系统性能的需求或者系统运行中所关注的不同内容来选择日志功能的启停。

13.2.2. 处理流程

13.2.2.1. 日志实现系统的选择流程

PKUAS日志服务的设计方案，为用户提供了一个统一的日志接口，简化了操作，同时避免项目与某个日志实现系统紧密耦合。它可以自动选择适当的日志实现系统，其处理流程及机制如下：

1) 首先在classpath下寻找commons-logging自己的配置文件commons-logging.properties，如果找到，则使用其中定义的Log实现类；

2) 如果找不到commons-logging.properties文件，则再找是否已定义系统环境变量org.apache.commons.logging.Log，找到则使用其定义的Log实现类；

3) 否则，查看classpath中是否有Log4j的包，如果发现，则自动使用Log4j作为日志实现类；

4) 否则，使用JDK自身的日志实现类；

5) 否则，使用commons-logging自己提供的一个简单的日志实现类SimpleLog。

可见，自动选择的结果总是能找到一个日志实现类，并且尽可能找到一个“最合适”的日志实现类。这样的机制给开发者带来方便，因为：

1）可以不需要配置文件；

2）自动判断有没有Log4j包，有则自动使用之；

3）最悲观的情况下也总能保证提供一个日志实现（SimpleLog）。

可以看到，commons-logging对编程者和Log4j都非常友好。

为了简化配置，PKUAS不使用commons-logging的配置文件，也不设置与commons-logging 相关的系统环境变量，而是选择Log4j作为具体的日志实现系统。

13.2.2.2. 日志的使用流程

一般而言，在需要输出日志信息的类中做如下工作：

1、导入所需的commons-logging类。

2、在使用日志的类中定义一个org.apache.commons.logging.Log类的私有静态类成员。

该成员将获取日志记录器，这个记录器将负责控制日志信息。这将通过指定的名字获得记录器，如果必要的话，则为这个名字创建一个新的记录器。

3、使用org.apache.commons.logging.Log类的成员方法输出日志信息。

当上两个必要步骤执行完毕，就可以轻松地使用不同优先级别的日志记录语句插入到想记录日志的任何地方。

13.3. 日志配置方法

13.3.1. 概述

PKUAS采用的是xml格式的配置文件来进行日志配置，遵从实现定义好的文档类型定义文件，即DTD文件。针对日志配置的xml文件的语法定义，可以在log4j的发布包中找到。

DTD文件主要包含下列元素：

- Configuration-由可选的renderer，appender，categories，和可选的root等元素组成
- Renderer-允许用户自定义从消息对象到String类型的转换
- Appender-可以包含1个error handler，1个layout，可选的param以及filter元素
- ErrorHandler-可以指定为任何一个class
- Priority-默认为定义级别的类
- Level-默认为定义级别的类
- Category，Logger，Root-用于具体定义日志记录器的元素

其中，每个元素还有各种属性可以设置，下一小节将进行具体说明。

13.3.2. 配置参数详解

本小节对xml文件中对于日志服务的各个配置参数进行详细解释。

13.3.2.1. 配置日志信息的输出通道Appender

Appender负责控制日志记录操作的输出，其语法为：

```
<!ELEMENT appender (errorHandler?, param*, layout?, filter*, appender-ref*)>
<!ATTLIST appender
  name   ID  #REQUIRED
  class  CDATA #REQUIRED
>
```

1. name属性用于设置通道名称
2. class属性用于设置具体的通道输出方式，目前提供的通道有以下几种：
   - ConsoleAppender（控制台）
   - FileAppender（文件）
   - DailyRollingFileAppender（每天产生一个日志文件）
   - RollingFileAppender（文件大小到达指定尺寸的时候产生一个新的文件）
   - WriterAppender（将日志信息以流格式发送到任意指定的地方）
3. ErrorHandler用于设置处理错误的类。
4. Filter元素用于设置过滤器级别。
5. Appender-ref元素用于设置对其他Appender的引用。
6. Layout元素的作用及设置见下一小节。
7. Param元素可以设置以下参数：
   - File-设置输出日志文件的文件名
   - MaxFileSize-设置输出日志文件的大小上限
   - MaxBackupIndex-设置保存备份文件个数
   - Threshold-设置日志输出级别
   - DatePattern-设置日期格式
   - Append-指明是覆盖或接着输出到原日志文件
   - ……

13.3.2.2. 配置日志信息的格式器Layout

Layout 负责格式化Appender的输出，其语法为：

```
<!ELEMENT layout (param*)>
<!ATTLIST layout
  class  CDATA #REQUIRED
>
```

1. class属性用于选择具体的格式，包括：

- org.apache.log4j.HTMLLayout（以HTML表格形式布局）
- org.apache.log4j.PatternLayout（可以灵活地指定布局模式）
- org.apache.log4j.SimpleLayout（包含日志信息的级别和信息字符串）
- org.apache.log4j.TTCCLayout（包含日志产生的时间、线程、类别等等信息）

2. Param元素可以设置以下参数：

- ConversionPattern-用于自定义格式化日志信息
- ……

其中，格式化日志信息采用类似C语言中的printf函数的打印格式格式化日志信息，打印参数如下：

- %m 输出代码中指定的消息
- %p 输出优先级，即DEBUG，INFO，WARN，ERROR，FATAL
- %r 输出自应用启动到输出该log信息耗费的毫秒数
- %c 输出所属的类目，通常就是所在类的全名
- %C输出所在类的全名
- %t 输出产生该日志事件的线程名
- %n 输出一个回车换行符，Windows平台为“rn”，Unix平台为“n”
- %d 输出日志时间点的日期或时间，默认格式为ISO8601，也可以在其后指定格式，比如：%d{yyyy MMM dd HH:mm:ss,SSS}，输出类似：2002年10月18日 22：10：28，921
- %l 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数
- %L输出日志事件的发生位置在代码中的行数
- %F输出日志事件的发生的文件名

13.3.2.3. 配置日志信息的记录器Logger

任何一个记录器的使用都有两个步骤：

Step1. 在配置文件中定义相应的日志记录器

Step2. 在代码中调用Logger类的取得生成器方法取得相应的记录器对象

在配置文件中定义记录器的格式有三种。

定义一般记录器的语法为：

```
<!ELEMENT logger (level?,appender-ref*)>
<!ATTLIST logger
  name  ID #REQUIRED
  additivity (true|false) "true"
>
```

1. name属性用于设置类别记录器名称
2. additivity属性用于指出是否叠加输出日志，如果是false，则在该记录器中的日志不会被其它满足条件的记录器输出
3. level元素用于指明要输出的日志级别
4. appender-ref元素用于设置对其他Appender的引用

定义类别记录器的语法为：

```
<!ELEMENT category (param*,(priority|level)?,appender-ref*)>
<!ATTLIST category
  class         CDATA   #IMPLIED
  name  CDATA #REQUIRED
  additivity (true|false) "true"
>
```

1. class属性用于定义类别，通常可以以包作为分类标准
2. name属性用于设置类别记录器名称
3. additivity属性用于指出是否叠加输出日志，如果是false，则在该记录器中的日志不会被其它满足条件的记录器输出
4. priority|level元素用于指明要输出的日志级别
5. appender-ref元素用于设置对其他Appender的引用
6. param元素可以设置相应参数

定义根记录器的语法为：

```
<!ELEMENT root (param*, (priority|level)?, appender-ref*)>
```

1. priority|level元素用于指明要输出的日志级别

2. appender-ref元素用于设置对其他Appender的引用

3. param元素可以设置相应参数

4.

13.3.3. 配置文件示例

13.3.3.1. 基础配置——利用预置简单布局输出到控制台的单个Appender

以下示例是利用简单输出格式的单个Appender设置

例 13.1. 基础配置——利用预置简单布局输出到控制台的单个Appender

```
<?xml version="1.0" encoding="GB2312" ?>

<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>


    <!-- 设置通道STDOUT和输出方式：org.apache.log4j.ConsoleAppender  -->

 <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">

     <!-- 设置输出格式为预置的SimpleLayout -->
           <layout class=" org.apache.log4j.SimpleLayout ">
           </layout>
 </appender>

 <!-- 定义根记录器接受上述通道 -->
    <root>
    <priority value ="debug" />
       <appender-ref ref="STDOUT" />
 </root>

</log4j:configuration>
```

13.3.3.2. 设置多个Appender

以下示例是设置多个场景下的Appender。

例 13.2. 设置多个Appender

```
<?xml version="1.0" encoding="GB2312" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

<!-- 设置通道file和输出方式：org.apache.log4j.RollingFileAppender -->
<appender name="FILE"  class="org.apache.log4j.RollingFileAppender">

  <!-- 设置File参数：日志输出文件名 -->
<param name="File" value=" all.output.log" />

<!-- 设置是否在重新启动服务时，在原有日志的基础添加新日志 -->
  <param name="Append" value="true" />

  <!-- 设置备份文件 --
  <param name="MaxBackupIndex" value="10" />

<!-- 设置输出格式为预置的SimpleLayout -->
  <layout class=" org.apache.log4j.SimpleLayout ">
  </layout>
 </appender>

<!--设置控制台输出方式-->
 <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">

<!-- 设置输出格式为预置的SimpleLayout -->
  <layout class=" org.apache.log4j.SimpleLayout ">
</appender>

<!—定义根记录器，设置接收所有输出的通道 -->
<root>
  <priority value="info" />
  <appender-ref ref="FILE" />
  <appender-ref ref="STDOUT" />
</root>

</log4j:configuration>
```

13.3.3.3. 格式化输出

以下示例展示了自定义格式输出的程序片段。

例 13.3. 格式化输出

```
<appender name="FILE"  class="org.apache.log4j.RollingFileAppender">
  <param name="File" value=" all.output.log" />
  <param name="Append" value="true" />
  <param name="MaxBackupIndex" value="10" />

<!-- 设置输出格式 -->
  <layout class="org.apache.log4j.PatternLayout">
   <param name="ConversionPattern" value="%p (%c:%L)- %m%n" />
  </layout>
 </appender>

 <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">

<!-- 设置输出格式 -->
  <layout class="org.apache.log4j.PatternLayout">
   <param name="ConversionPattern"
    value="%-4r [%t] %-5p %c %x - %m%n" />
  </layout>
 </appender>
```

13.3.3.4. 日志级别控制

以下示例展示了用两种方式控制日志级别的程序片段。

例 13.4. 日志级别控制

```
<appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
  <!-- 设置输出格式为预置的SimpleLayout -->
  <layout class=" org.apache.log4j.SimpleLayout ">
  </layout>

<!—用filter参数设置输出的级别-->
<filter class="org.apache.log4j.varia.LevelRangeFilter">
    <param name="levelMin" value="info" />
    <param name="levelMax" value="info" />
    <param name="AcceptOnMatch" value="true" />
  </filter>
 </appender>

<appender name="FILE"  class="org.apache.log4j.RollingFileAppender">
  <param name="File" value=" all.output.log" />
  <param name="Append" value="true" />
  <param name="MaxBackupIndex" value="10" />

<!-- 设置输出格式为预置的SimpleLayout -->
  <layout class=" org.apache.log4j.SimpleLayout ">
  </layout>

<!—用Threshold参数设置输出的级别-->
  <param name="Threshold" value="info" />
</appender>

<!—用Threshold参数设置输出的级别-->
```

13.3.3.5. 设置类别记录器

以下示例展示定义了类别记录器的程序片段。

例 13.5. 设置类别记录器

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>

 <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
          <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
      value="%d %-5p [%t] %C{2} (%F:%L) - %m%n"/>
          </layout>
 </appender>

 <!—设置类别记录器，输出指定类包中的日志-->
 <category name="org.apache.log4j.xml">
   <priority value="info" />
 </category>

<!—设置根记录器，专门输出相应包中的日志-->
 <root>
    <priority value ="debug" />
       <appender-ref ref="STDOUT" />
 </root>

</log4j:configuration>
```

13.4. 日志应用现状与扩展技巧

13.4.1. 概述

目前，在编译生成PKUAS-OSGI-2010的可运行版本后，执行run.bat进行启动，其启动过程中的日志将显示在控制台中，同时可以以文件形式记录在%PKUAS_HOME%\var\logs下（PKUAS_HOME为可运行版本解压的根目录）。本部分介绍PKUAS日志服务的具体应用现状、方法及扩展。

13.4.2. 日志使用方法

一般而言，在需要输出日志信息的类中做如下工作：

1、导入所需的类

PKUAS中的例子：

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

2、得到记录器(Logger)

获取日志记录器，这个记录器将负责控制日志信息。这将通过指定的名字获得记录器，如果必要的话，则为这个名字创建一个新的记录器。Name一般取本类的名字，PKUAS中的例子，在要使用日志服务的类中，声明:

```
public final class Main {
private static Log logger = LogFactory.getLog(Main.class);
……
}
```

注意这里定义的是static成员，以避免产生多个实例，LogFactory.getLog()方法的参数使用的是当前类的class，这是目前被普通认为的最好的方式。（注意不写作LogFactory.getLog(this.getClass()，因为static类成员访问不到this指针)

3、输出日志信息

当上两个必要步骤执行完毕，就可以轻松地使用不同优先级别的日志记录语句插入到想记录日志的任何地方，其语法如下：

```
logger.debug ( Object message ) ;
logger.info ( Object message ) ;
logger.warn ( Object message ) ;
logger.error ( Object message ) ;
logger.fatal(Object message);
```

这里的logger是第2步中定义的类成员变量，其类型是org.apache.commons.logging.Log，通过该类的成员方法，我们就可以将不同性质的日志信息输出到目的地（目的地由配置文件指定，可能是stdout，也可能是文件，还可能是发送到邮件，甚至发送短信到手机……）

13.4.3. 日志的扩展技巧

如果在pkuas中需要更灵活地对日志服务进行一些自定义配置，则可以通过修改pkuas.xml文件中的日志配置部分来实现。下面讲讲几种主要的配置方法。

1、自定义logger

通过建立一个单独的logger，来输出需要的特定信息。如下例子：

首先定义appender，指明输出方式、级别、格式等：

```
……
<appender name="myTestAppender" class="org.apache.log4j.RollingFileAppender">
<param name="File" value="../var/logs/myTest.log"/>
    <param name="Append" value="false"/>
    <param name="Threshold" value="debug"/>
 <param name="MaxFileSize" value="300KB"/>
    <param name="MaxBackupIndex" value="10"/>
    <layout class="org.apache.log4j.PatternLayout">
     <param name="ConversionPattern"
                value="%d{HH:mm:ss} %l [%t]%c(%p) %m%n"/>
</layout>
</appender>
……
```

接着定义一个单独的logger，如下：

```
……
<logger name="myTestLog" additivity="false">
   <level class="org.apache.log4j.Level" value="debug"/>
    <appender-ref ref="myTestAppender"/>
  </logger>
……
```

这样，便能在代码中需要使用该logger的地方引用这个logger了。获得logger的方法如下：

```
private static Log logger = LogFactory.getLog(“myTestLog”);
```

2、往远程主机写日志

可以使用Socket Appender。如下示例：

```
……
<appender name="socketAppender" class="org.apache.log4j.RollingFileAppender">
<param name="RemoteHost" value="localhost"/>
    <param name="Port" value="5001"/>
   <param name="LocationInfo" value="true"/>
    <layout class="org.apache.log4j.PatternLayout">
     <param name="ConversionPattern"
                  value="[start]%d{DATE}[DATE]%n%p[PRIORITY]%n%x
[NDC]%n%t[THREAD]%n%c[CATEGORY]%n%m[MESSAGE]%n%n"/>
</layout>
</appender>
……
```

3、使用自定义的格式

可以通过继承Log4j的Layout类及Parser类，来自定义新的日志布局器和解析器。

如，为了定位到bundle信息，定义了Log4j的Layout类及Parser类的子类，分别为UserBundlePatternLayout及UserBundlePatternParser，放在pku.as.logger包下。这两个类可以针对以下特定的格式输出bundle信息。在定义格式时，用%.n#可以输出所在bundle名称，其中，n为一整数，用于容纳包名，再根据包名映射为bundle名。一般n设为50就合适。

例：

```
……
<layout class="pku.as.logger.UserBundlePatternLayout">
<param name="ConversionPattern"
           value="%d{HH:mm:ss} [BUNDLE:%.50#] - %m%n"/>
</layout>
……
```

4、建议使用的模板

下面以建立控制台输出的appender为例，给出一个建议的模板。如下：

```
……
<appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
<param name="Threshold" value="info"/>
 <layout class="pku.as.logger.UserBundlePatternLayout">
  <param name="ConversionPattern"
         value="%d{HH:mm:ss} %5p [THREAD:%t] (LOCATION:%F:%4L) [BUNDLE:%.50#]
- %m%n"/>
</layout>
</appender>
……
```

控制台输出示例：

```
11:57:53  INFO [THREAD:FelixStartLevel] (LOCATION:Main.java:  84) [BUNDLE:SERVICE_
11:57:53  INFO [THREAD:FelixStartLevel] (LOCATION:Main.java: 377) [BUNDLE:SERVICE_
```

说明：输出的每一行信息依次为：时间、日志级别、所在线程、位置（文件以及文件中的行数）、所在bundle以及相应的具体日志信息。

13.4.4. 日志服务配置现状

执行run.bat启动编译生成的可运行版本，其启动过程将被记录在%PKUAS_HOME%\logs\下的pkuas.log和pkuas-web.log这两个相应的日志文件中（PKUAS_HOME为可运行版本解压的根目录）。如下显示的是pkuas.log文件示例：

图 13.1. %PKUAS_HOME%\logs\pkuas.log文件概要示例

13.4.5. 日志服务的配置示例

在PKUAS-OSGi-2010中，日志服务和其他功能模块的配置都被放到了pkuas.xml文件中。

查阅%PKUAS_HOME%\conf\下的pkuas.xml文件，概略如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
-<pkuas>
 -<Kernel>
  +<Properties>
  -<Services>
   +<Service name="log">
   +<Service name="codebase">
   </Services>
  </Kernel>
 +<ServiceManager start="true">
 </pkuas>
```

可以发现，log和codebase是作为内核的两个服务进行配置的。

下面详细看看其中对log服务的配置情况。

```
      <param name="ConversionPattern" value="[%d{HH:mm:ss}][%.50#][
    </layout>
  </appender>
- <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
    <param name="Threshold" value="info" />
  - <layout class="pku.as.logger.UserBundlePatternLayout">
      <param name="ConversionPattern" value="[%d{HH:mm:ss}][%.50#][
    </layout>
  </appender>
- <appender name="PkuasLogFile" class="org.apache.log4j.RollingFileAppe
    <param name="File" value="../var/logs/pkuas.log" />
    <param name="Append" value="false" />
    <param name="Threshold" value="debug" />
    <param name="MaxFileSize" value="300KB" />
    <param name="MaxBackupIndex" value="10" />
  - <layout class="pku.as.logger.UserBundlePatternLayout">
      <param name="ConversionPattern" value="[%d{HH:mm:ss}][%.50#][
    </layout>
  </appender>
- <appender name="WebLogFile" class="org.apache.log4j.RollingFileAppen
    <param name="File" value="../var/logs/pkuas-web.log" />
    <param name="Append" value="false" />
    <param name="Threshold" value="info" />
    <param name="MaxFileSize" value="300KB" />
    <param name="MaxBackupIndex" value="10" />
  - <layout class="org.apache.log4j.TTCCLayout">
      <param name="DateFormat" value="ISO8601" />
    </layout>
  </appender>
- <appender name="ErrorLogFile" class="org.apache.log4j.DailyRollingFileA
    <param name="File" value="../var/logs/error.log" />
    <param name="Append" value="false" />
    <param name="Threshold" value="error" />
    <param name="DatePattern" value="'.'yyyy-MM-dd'.log'" />
  - <layout class="pku.as.logger.UserBundlePatternLayout">
      <param name="ConversionPattern" value="[%d{HH:mm:ss}][%.50#][
    </layout>
  </appender>
- <root>
    <priority value="debug" />
    <appender-ref ref="dummy" />
  </root>
- <category name="pku" additivity="false">
    <priority value="debug" />
    <appender-ref ref="STDOUT" />
    <appender-ref ref="PkuasLogFile" />
    <appender-ref ref="ErrorLogFile" />
  </category>
- <category name="org.apache" additivity="false">
    <priority value="debug" />
    <appender-ref ref="STDOUT" />
    <appender-ref ref="WebLogFile" />
    <appender-ref ref="ErrorLogFile" />
  </category>
</log4j:configuration>
```

说明：

以上示例文件中，一共指定了5个输出器通道（appender）：

1.dummy

输出到控制台。

2.STDOUT

输出到控制台（同dummy没有区别）。

3.PkuasLogFile

RollingFileAppender, 即滚动日志文件，具体记录在../logs/pkuas.log文件中；文件大小到达指定尺寸300KB的时候产生一个新的文件；最多有10个备份；TTCCLayout，包含日志产生的时间、线程、类别等等信息；在重新启动服务时，不会在原有日志的基础添加新日志，而是覆盖掉原有内容；

4.WebLogFile

RollingFileAppender，同上，滚动日志文件，具体记录在../var/logs/pkuas-web.log文件中；在重新启动服务时，不会在原有日志的基础添加新日志，而是覆盖掉原有内容；大于等于info级别的日志将被输出文件大小到达指定尺寸300KB的时候产生一个新的文件；最多有10个备份；TTCCLayout，包含日志产生的时间、线程、类别等等信息。

5.ErrorLogFile

DailyRollingFileAppender，每日生成滚动文件，具体记录在../var/logs/error.log文件中；在重新启动服务时，不会在原有日志的基础添加新日志，而是覆盖掉原有内容；大于等于error级别的日志将被输出；每日的文件名格式遵循后缀.yyyy-MM-dd.log；使用自定义输出格局UserBundlePatternLayout。

接着指定了以下几个Logger，包括rootLogger和两个包的category：

1.rootLogger

debug级别，设置接收上面命名为dummy的输出通道，即输出到控制台。

2.pku

Debug级别，输出指定包pku中的日志，指定这些日志被输出到STDOUT和PkuasLogFile这两个输出通道;additivity指示是否叠加输出log，此处设置成false，则不会再被其它满足条件的logger输出(比如root)。

3.org.apache

debug级别，输出指定包org.apache中的日志,指定这些日志被输出到STDOUT，WebLogFile和ErrorLogFile这三个输出通道；同样的，additivity设置成false，不会被其它满足条件的logger输出。

第 14 章 PKUAS应用开发指南

14.1. 引言

本开发指南面向使用PKUAS来设计、开发应用系统的人员，目的是指导有关人员开发、组装和部署基于PKUAS的应用系统。

阅读本开发指南需要首先了解和掌握J2EE/EJB应用的开发方法，本指南对EJB、JSP/Servlet等实际开发不做具体介绍，这方面内容请参看SUN公司有关文档和其它参看书籍。

本开发指南将按照一般的开发流程，从准备工作开始，逐步介绍EJB、WEB module的开发、应用系统的组装和部署，之后对高级特性的使用进行介绍。

14.2. 应用服务器知识概述

14.2.1. 什么是应用服务器

一般的，软件可以被划分为三类：系统软件、支撑软件和应用软件。

其中，系统软件主要是指操作系统等与系统资源密切相关的软件，常见的如MS Window系列和Unix系列OS，都可以被看作是系统软件；支撑软件是指软件开发工具等支持开发过程中的软件，常见的IDE就属于这种软件；这两种软件可以合称为基础软件。应用软件指特定于应用领域的专用软件。

我们所要探讨的应用服务器是一种为方便应用软件开发与维护而逐步形成的基础软件：应用服务器是网络环境中应用系统的高层运行平台；它使应用系统的代码更为简单，使开发人员的精力可以更加集中于系统的逻辑部分。（《应用服务器原理与实现》）

14.2.2. 应用服务器的出现环境

网络软件（分布在网络环境中的软件系统）不同于单机环境中的软件：它要求分布在网络中的各个节点能够相互协作从而共同完成目标任务，因此网络软件必需解决网络环境里分布性、可靠性、安全性等问题。解决这些问题的一条策略就是：提取软件共性成分，屏蔽系统低层的复杂度，从而在高层保持复杂度的相对稳定。采用了这一策略的典型就是操作系统，数据库管理系统，和应用服务器。

根据前述理由可以知道，一个具体的应用系统，如网络软件，直接从操作系统层开始开发是不合理的，而应该以一个相对高层的平台为起点。这个平台为应用系统的开发、维护提供基础。应用服务器将网络软件的共性成分从应用系统中剥离出来，构成相对完整、独立的系统软件；因其屏蔽了低层网络细节，是开发者能够将精力集中于目标系统的业务逻辑上，从而简化应用系统的开发和维护的过程。是不是很像操作系统？操作系统屏蔽了低层硬件细节，为计算机使用者提供的是一系列应用接口，展示了计算机在逻辑层面的应用。

纵向来看，应用服务器位于操作系统之上，应用系统之下；横向来看，应用服务器位于软件的表示层和数据层之间的业务逻辑层。

14.2.3. 应用服务器的功能

具体来说，应用服务器有以下功能：

- 通过构件容器为构建提供基本的运行环境，包括管理构件生命周期、管理构件的实例、管理构建的信息等；
- 提供高层通信服务，以满足网络环境中的构件之间的互操作，包括业务层与表示层之间的通信、业务层与数据层之间的通信、业务层内部公共服务与应用层之间的通信；
- 提供公共服务，包括查找服务、事务服务和安全服务。

14.3. 准备工作

开发者在开发J2EE应用前需要作相关的准备工作。包括JDK、应用服务器和数据库的安装、设置等。J2SE和J2EE的API文档可以给开发者提供很多帮助。此外，开发者可能还需要选择一个合适的集成开发环境来提高工作效率。

14.3.1. 系统需求

PKUAS运行系统的最低配置为：奔腾166MHZ或同等性能机器，带有48兆以上内存。安装Java 2 SDK需要至少70M的硬盘空间，推荐使用200M以上的空闲磁盘空间。

PKUAS可以在装有Microsoft Windows 95,98(1st or 2nd Edition), NT4.0 with Service Pack 5,ME,2000 Professional,2000 Server,2000 Advanced Server, XP Operating System和类Unix环境(Linux,Solaris)的机器上运行。

运行PKUAS必须预先安装Java 2 SDK 1.4以上版本并配置环境变量，并对PKUAS进行正确配置。

14.3.2. 数据源的配置

PKUAS支持MySQL、Oracle、SQL Server、Cloudscape等多个数据库的使用。实际上所有JDBC支持的数据库在PKUAS中都可以使用。

以MySQL为例，在安装好MySQL后，可以在pkuas.xml文件中加入以下小节来配置数据源：

```
<!-- Datasource (JNDI name: jdbc/petstore) for our petstore application -->

<Service Class="pku.as.datasvc.PoolDataSource"   Name="jdbc/petstore">

        <Attribute Name="expirationTime" Value="300" />

        <Attribute Name="minCapacity" Value="10" />

        <Attribute Name="user" Value="hgj61" />

        <Attribute Name="password" Value="hgj61" />

        <Attribute Name="URL"  Value="jdbc:mysql://192.168.4.137:3306/petsto

        <Attribute Name="driverClassName"  Value="com.mysql.jdbc.Driver" />

        <Attribute Name="maxCapacity" Value="1000" />

</Service>
```

其中Class="pku.as.datasvc.PoolDataSource" Name=" jdbc/petstore"是这个数据源的名字， jdbc/petstore是开发者在开发J2EE应用时要用到的数据源字符串，根据需要可以任意取名。用户名、密码、URL等属性值可根据使用者的情况填写。

"URL"属性的值供JDBC建立连接时用，开发者只需要将MysqlServerAddress换成实际的mysql所处的机器地址，后面的EJBTEST是准备用的数据库名。这样数据源就配置好了。

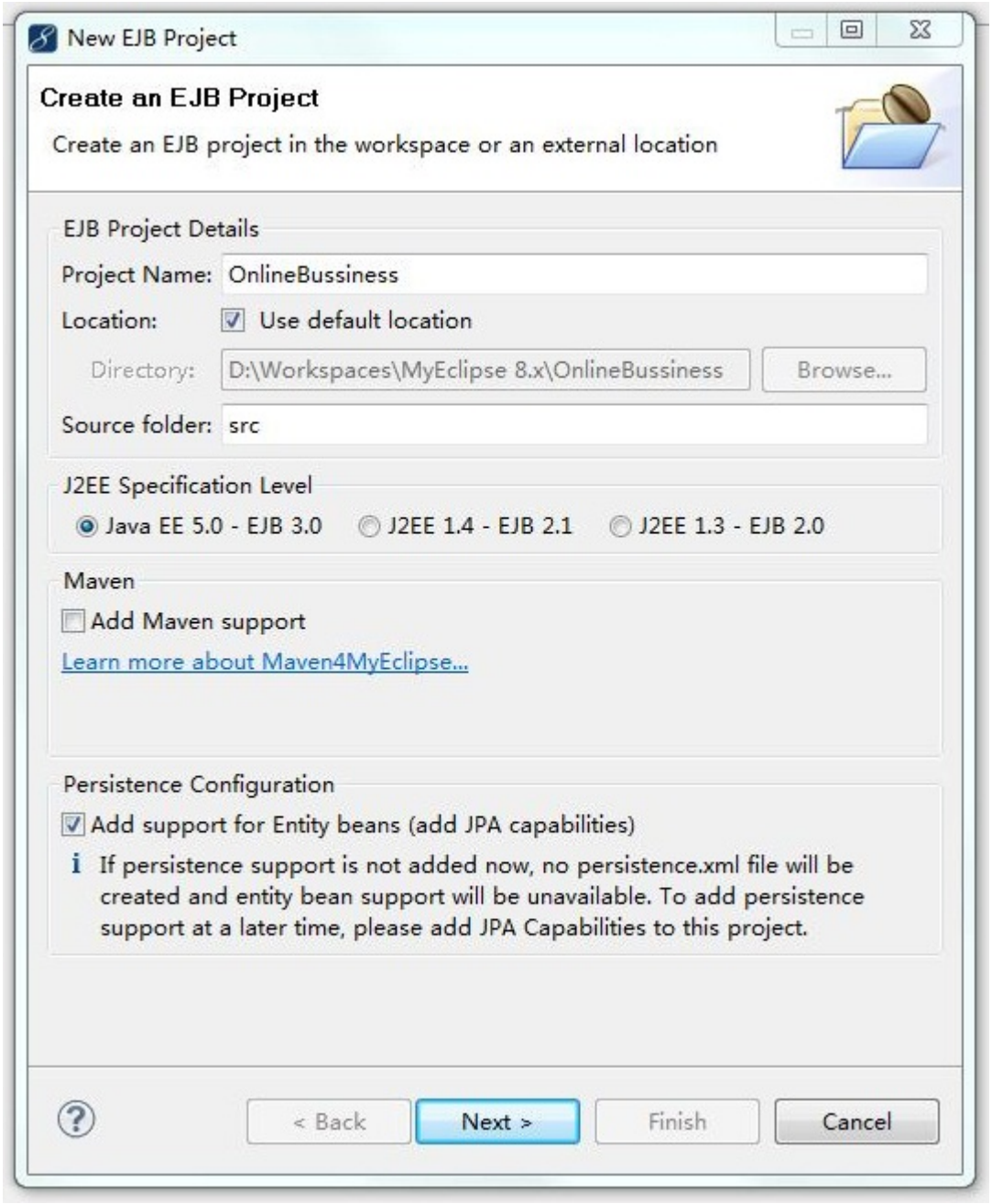
14.4. EJB模块的开发

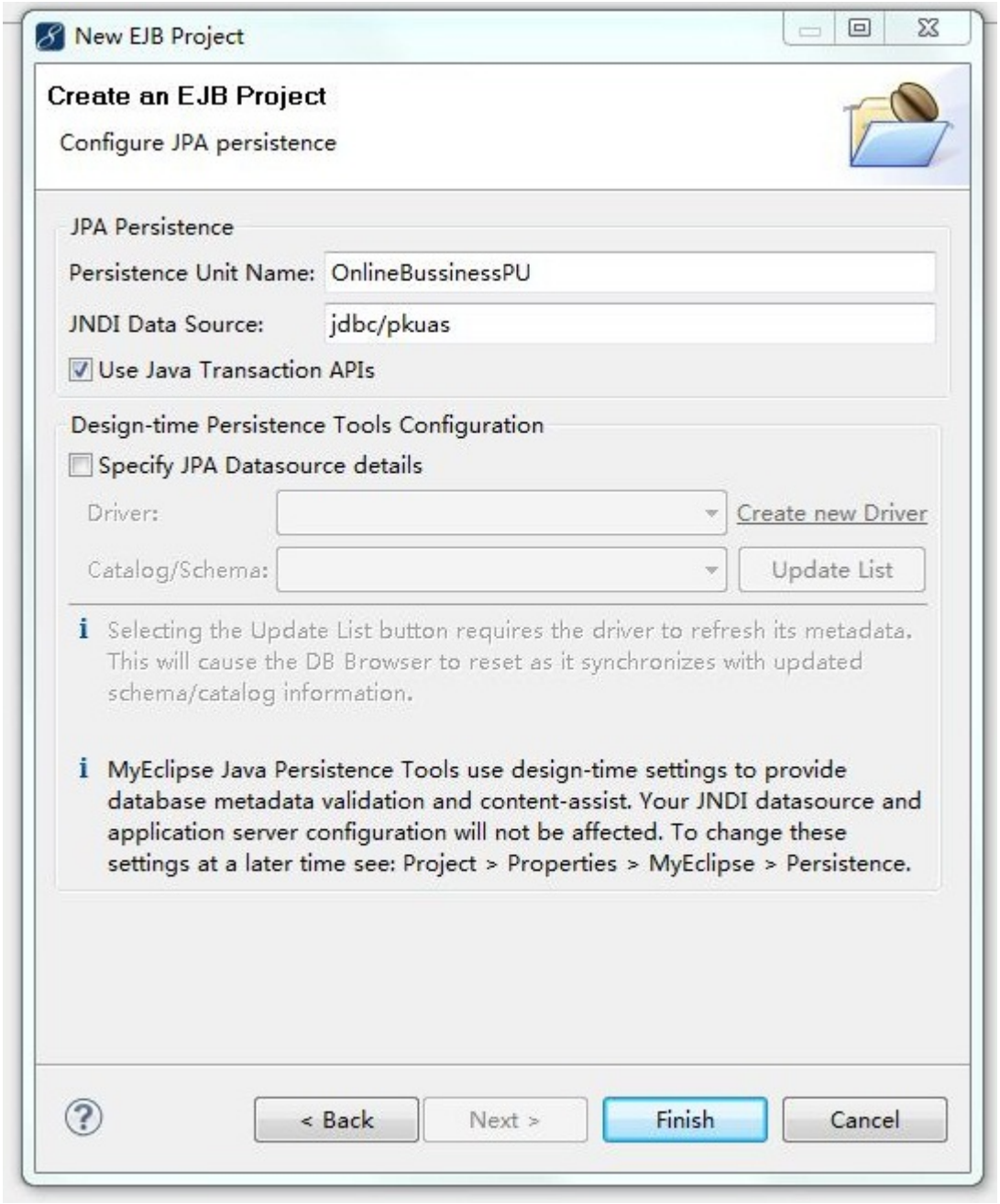
EJB可以分为三种：Session Bean（会话Bean），Entity Bean（实体Bean），和Message-Driven Bean（消息驱动Bean）。

本手册将以一个网上购物的应用为例，展示在pkuas中开发这几种Bean的方法。开发环境是MyEclipse8.0GA。

14.4.1. 创建Project

File->New->EJB Project


New EJB Project
Create an EJB Project
Create an EJB project in the workspace or an external location
EJB Project Details
Project Name: OnlineBussiness
Location: Use default location
Directory: D:\Workspaces\MyEclipse 8.x\OnlineBussiness
Browse...
Source folder: src
J2EE Specification Level
Java EE 5.0 - EJB 3.0
J2EE 1.4 - EJB 2.1
J2EE 1.3 - EJB 2.0
Maven
Add Maven support
Learn more about Maven4MyEclipse...
Persistence Configuration
Add support for Entity beans (add JPA capabilities)
If persistence support is not added now, no persistence.xml file will be created and entity bean support will be unavailable. To add persistence support at a later time, please add JPA Capabilities to this project.
< Back
Next >
Finish
Cancel



Finish

14.4.2. 开发无状态会话Bean

14.4.2.1. 概述

无状态会话Bean是具有类级别注解@Stateless的任意标准Java类。它由业务接口和业务方法两部分构成。

为了演示无状态会话Bean的使用，我们将创建SearchFacade会话Bean，它为客户提供关于现有的酒类的搜索。

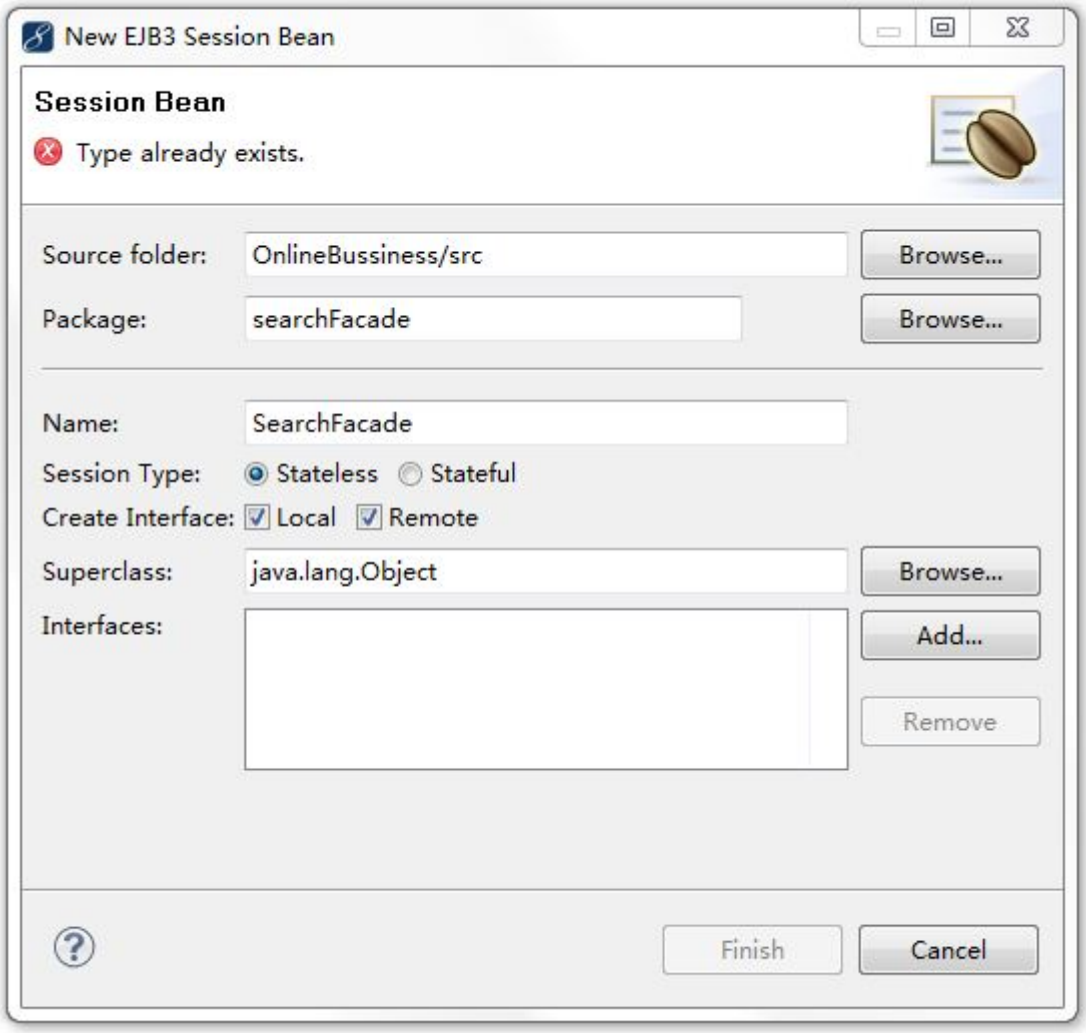
14.4.2.2. 开发流程

无状态会话Bean的业务接口分为：

- 本地业务接口——@Local，默认，用于客户端和EJB容器在同一个JVM上的情况
- 远程业务接口——@Remote，用于客户端和EJB容器不在同一个JVM上的情况

下面演示具体的步骤。

14.4.2.2.1. 创建会话Bean：File->New->EJB3 SessionBean



14.4.2.2.2. 编写业务接口

本地业务接口：SearchFacadeLocal.java

```
package searchFacade;

import java.util.List;

import javax.ejb.Local;

@Local

public interface SearchFacadeLocal {

        List<String> wineSearch(String wineType);

}
```

远程业务接口：SearchFacadeRemote.java

```
package searchFacade;

import java.util.List;

import javax.ejb.Remote;

@Remote

public interface SearchFacadeRemote {

        List<String> wineSearch(String wineType);

}
```

14.4.2.2.3. 编写业务方法

SearchFacade.java

```
package searchFacade;

import java.util.ArrayList;

import java.util.List;

import javax.ejb.Stateless;

@Stateless(name="SearchFacade")

public class SearchFacade implements SearchFacadeLocal, SearchFacadeRemote, java.io.Seriali

        public SearchFacade(){}

        public List<String> wineSearch(String wineType) {

                List<String> wineList = new ArrayList<String>();

                if(wineType.equals("Red")) {

                        wineList.add("Bordeaux");

                        wineList.add("Merlot");

                        wineList.add("Pinot Noir");

                }

                else if (wineType.equals("White")) {

                        wineList.add("Chardonnay");

                }

                return wineList;

        }

}
```

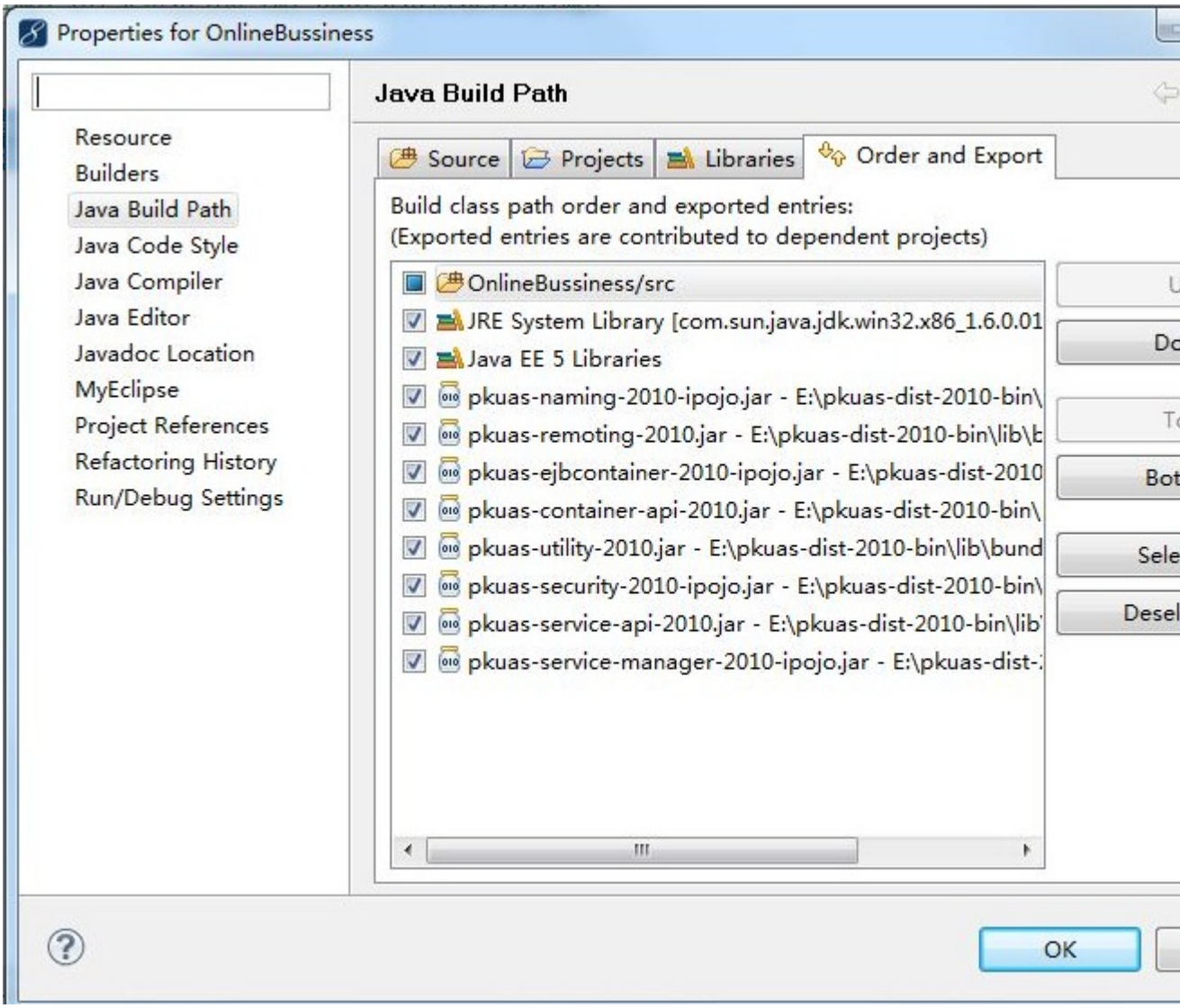
14.4.2.2.4. 配置jndi

```
java.naming.factory.initial=pku.as.naming.SmartCtxFactory

java.naming.provider.url=rmi\://localhost\:2001
```

这个文件要放在工程的根目录下。

14.4.2.2.5. 配置Build Path

工程名右键->Build Path->Configure Build Path



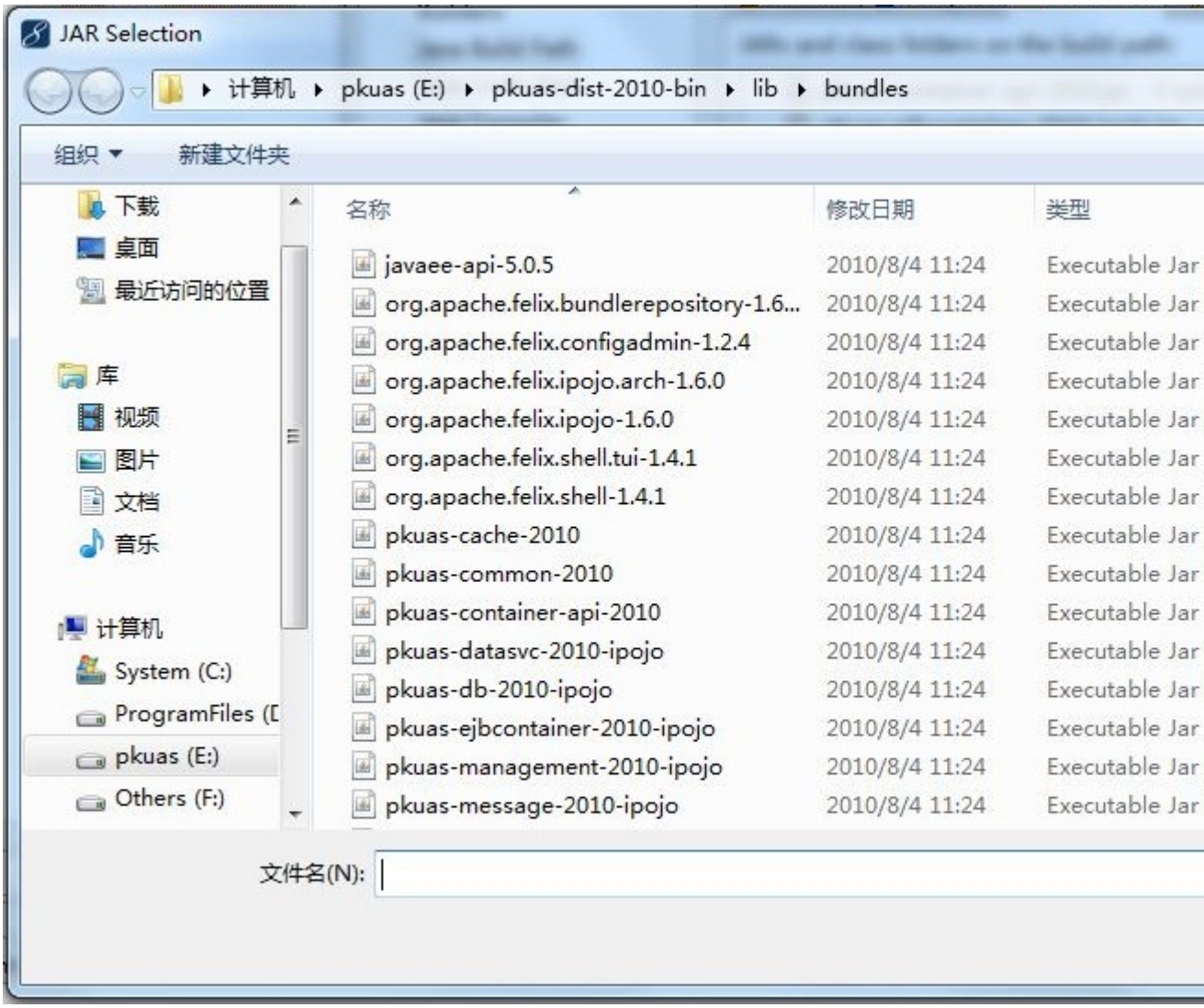
Libraries->Add External JARs

在pkuas的lib目录下选择依赖包，本例所依赖的包是

- pkuas-naming-2010-ipojo.jar
- pkuas-remoting-2010.jar
- pkuas-ejbcontainer-2010-ipojo
- pkuas-container-api-2010.jar
- pkuas-utility-2010.jar
- pkuas-security-2010-ipojo.jar
- pkuas-service-api-2010.jar
- pkuas-serviece-manager-2010-ipojo.jar

将这些包添加进来后，再在Order and Export中选中添加即可。

当然，也可以一次性将pkuas所有依赖包都添加进来



14.4.2.2.6. 启动PKUAS

pkuas目录下bin目录下启动run.bat.

控制台显示"Application Manager is started!"信息表示pkuas启动成功。

14.4.2.2.7. 打包部署

将SearchFacade包打包成JAR，并将这个JAR包放在pkuas目录下的application目录里。

这是pkuas控制台提示"###The app [SearchFacade.jar] deployed###"，表示这个EJB已经成功部署。

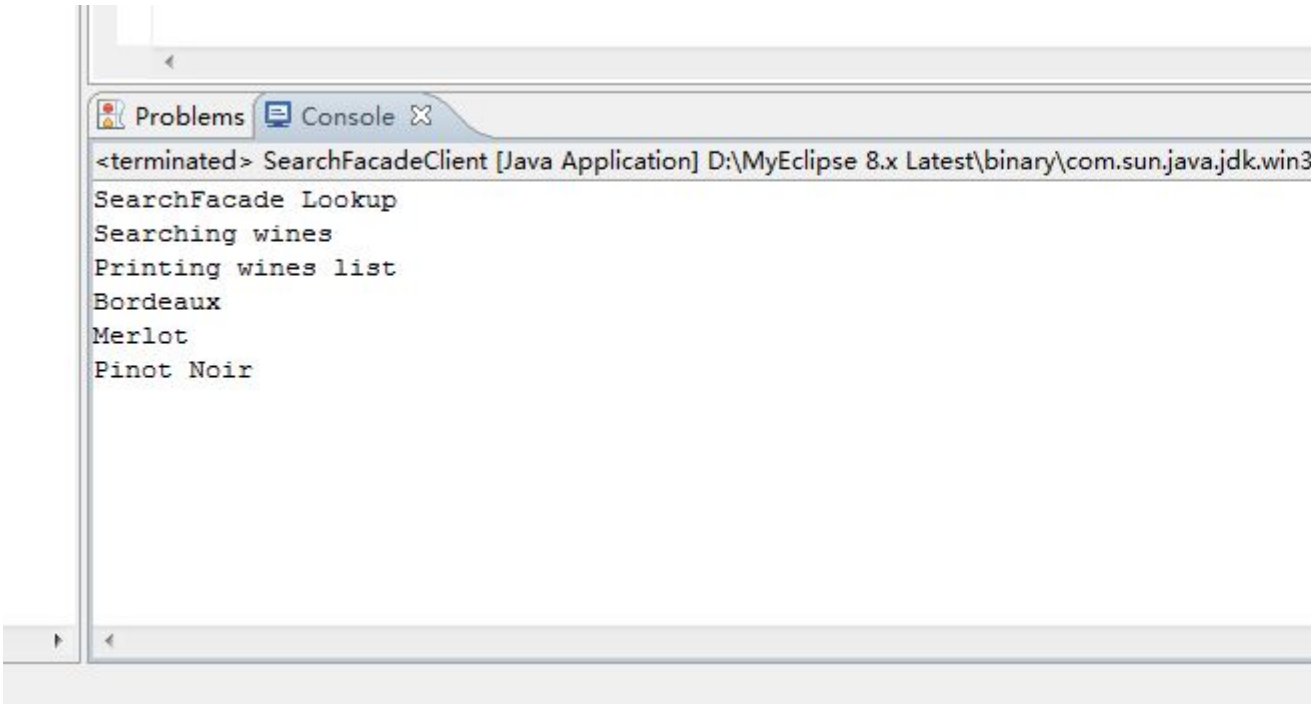
14.4.2.2.8. 在客户端测试运行EJB

为了成功演示EJB的执行结果，还要编写一个客户端来调用这个EJB。在本工程下创建另一个包Client，在包里添加如下java文件：

SearchFacadeClient.java

```
package client;

import searchFacade.*;

import java.util.List;

import javax.naming.*;

public class SearchFacadeClient {

	public SearchFacadeClient(){}

	public static void main(String[] args) throws NamingException {

		Context ctx = new InitialContext();

		SearchFacadeRemote response = (SearchFacadeRemote)ctx.lookup("SearchFacade"

		System.out.println("SearchFacade Lookup");

		System.out.println("Searching wines");

		List<String> winesList = response.wineSearch("Red");

		System.out.println("Printing wines list");

		for(String wine:(List<String>)winesList) {

			System.out.println(wine);

		}

	}

}
```

运行这个客户端程序，控制台显示如下信息，表示我们编写的会话Bean已经成功被调用了：



14.4.3. 开发有状态会话Bean

14.4.3.1. 概述

有状态会话Bean是具有类级别注解@Stateful的任意标准Java类。和有状态会话Bean一样，它也由业务接口和业务方法两部分构成。

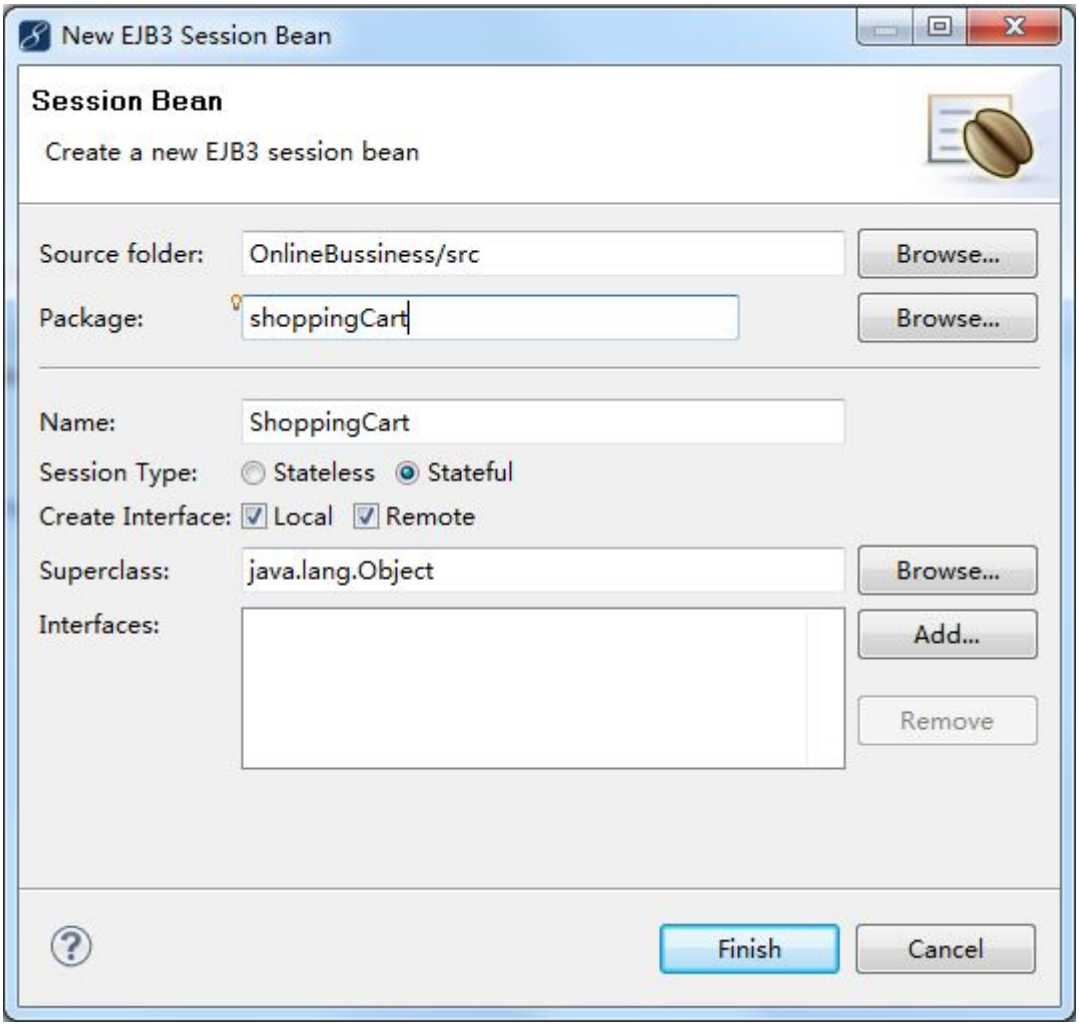
每个有状态会话Bean在bean实例的生命周期内都只服务于一个客户端；它就像是客户端的代理。不同与无状态会话Bean，有状态会话Bean的实例不会在EJB对象之间进行切换，也不会被置于实例池内。一旦有状态会话Bean被实例化，并被指派给了某个EJB对象，他在整个生命周期内只会关联于该EJB对象。

为了演示有状态会话Bean的使用，我们将创建ShoppinCart会话Bean，它跟踪添加到用户购物车的项目以及它们各自的数量。

14.4.3.2. 开发流程

下面演示具体的步骤。

14.4.3.2.1. 创建会话Bean：File->New->EJB3 SessionBean



14.4.3.2.2. 编写业务接口

本地业务接口：ShoppingCartLocal.java

```
package shoppingCart;

import java.util.ArrayList;

import javax.ejb.Local;

@Local
public interface ShoppingCartLocal {

        public void addWineItem(String wine);

        public void removeWineItem(String wine);

        public void setCartItems(ArrayList<String> cartItems);

        public ArrayList<String> getCartItems();

}
```

远程业务接口：ShoppingCartRemote.java

```
package shoppingCart;

import java.util.ArrayList;

import javax.ejb.Remote;

@Remote
public interface ShoppingCartLocal {

        public void addWineItem(String wine);

        public void removeWineItem(String wine);

        public void setCartItems(ArrayList<String> cartItems);

        public ArrayList<String> getCartItems();

}
```

14.4.3.2.3. 编写业务方法

ShoppingCart.java

```
package shoppingCart;

import javax.annotation.*;
import javax.ejb.Remove;
import javax.ejb.Stateful;

import java.util.ArrayList;

@Stateful(name="ShoppingCart")
public class ShoppingCart implements ShoppingCartLocal, ShoppingCartRemote, java.io.Serializable {

        public ShoppingCart(){};

        public ArrayList<String> cartItems;

        @PostConstruct
        public void initialize() {
                cartItems = new ArrayList<String>();
        }

        @PreDestroy
        public void exit() {
                // items list into the database
                System.out.println("Saved items list to database");
        }

        @Remove
        public void stopSession() {
                // the method body can be empty
                System.out.println("From stopSession method with @Remove annotation");
        }

        public void addWineItem(String wine) {
                cartItems.add(wine);
        }

        public void removeWineItem(String wine) {
                cartItems.remove(wine);
        }

        public void setCartItems(ArrayList<String> cartItems) {
                this.cartItems = cartItems;
        }

        public ArrayList getCartItems() {
                return cartItems;
        }
}
```

14.4.3.2.4. 配置jndi

同无状态会话Bean

14.4.3.2.5. 配置Build Path

同无状态会话Bean

14.4.3.2.6. 启动pkuas

同无状态会话Bean

14.4.3.2.7. 打包部署

将ShoppingCart包打包成JAR，并将这个JAR包放在pkuas目录下的application目录里。

这是pkuas控制台提示"###The app [ShoppingCart.jar] deployed###"，表示这个EJB已经成功部署。

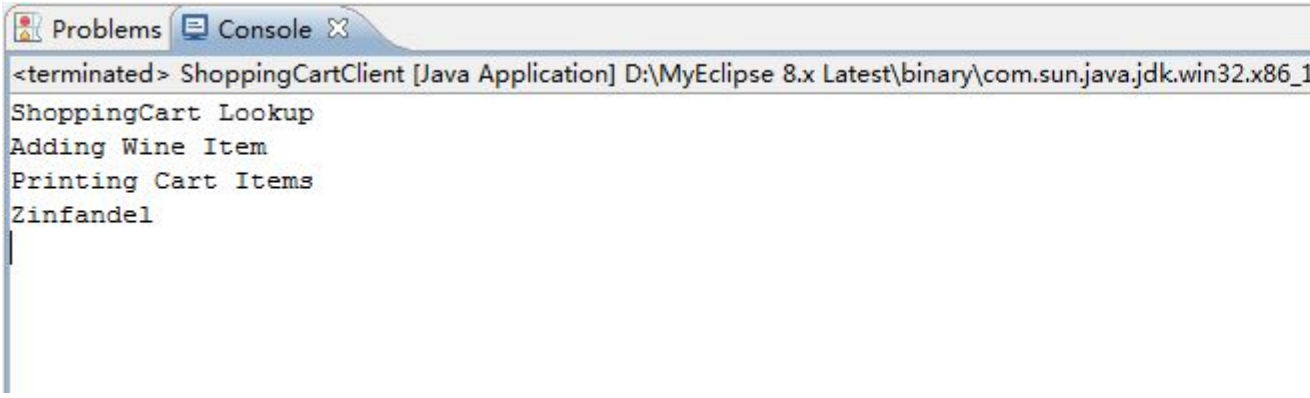
14.4.3.2.8. 在客户端测试运行EJB

为了成功演示EJB的执行结果，还要编写一个客户端来调用这个EJB。在本工程下创建另一个包Client，在包里添加如下java文件：

ShoppingCartClient.java

```
package client;

import shoppingCart.*;

import java.util.*;

import javax.naming.*;

public class ShoppingCartClient {

	public ShoppingCartClient() {}

	public static void main(String[] args) throws NamingException{

		Context ctx = new InitialContext();

		ShoppingCartRemote response = (ShoppingCartRemote)ctx.lookup("ShoppingCart");

		System.out.println("ShoppingCart Lookup");

		System.out.println("Adding Wine Item");

		response.addWineItem("Zinfandel");

		System.out.println("Printing Cart Items");

		ArrayList<String> cartItems = response.getCartItems();

		for (String wine : (List<String>) cartItems) {

			System.out.println(wine);
		}
	}

}
```

运行这个客户端程序，控制台显示如下信息，表示我们编写的会话Bean已经成功被调用了：



14.5. WEB模块的开发

PKUAS集成了Tomcat（目前集成的版本是5.5）作为Servlet引擎。Web模块的打包结构与在Tomcat上面单独部署基本上是一样的。应用的结构为：

JSP/html表示这个Web中用到的静态页面和JSP页面；WEB-INF目录下面的web.xml是Web应用的描述文件。classes目录下面按包目录放了这个Web模块使用的JavaBeans和Servlet的类。而lib目录下面放了Web应用中使用到的JAR包。

WEB模块可以直接在PKUAS上部署，或作为一个企业应用（Enterprise Application，以EAR的形式部署到PKUAS上）的组成部分：

作为独立的WEB应用部署:

- 直接将WEB 模块的目录放到PKUAS下面的application目录下
- 将WEB 模块打包为WAR文件后将WAR放到application目录下面

这两种方式部署的WEB 模块在启动PKUAS后可以直接访问。

作为一个企业应用的一部分:

- 将WEB 模块打包为WAR文件，和整个应用一起部署。在下一章中，将介绍组装和部署的过程。
- 将WEB 模块打包为WAR的过程与上面打包EJB相似，也是使用jar命令，这时后面的参数中，打包文件的扩展名为war。

用户除了使用jar命令外，还可以使用maven、ant等工具来自动实现编译和打包，另外Myeclipse以及安装了web开发插件的eclipse也可以将web项目导出为war包或ear包。如果用户都没有安装这些的话，还可以通过解压工具（如WinRAR,7zip等）将web工程目录压缩为zip格式，同时将后缀名改为war或ear，需要注意的如果

选择这种方式，对于web项目所需要的web.xml和ear项目所需要的application.xml需要注意审查，因为手动打包不会检查这些文件的正确性.

对独立部署的Web 模块，如果它还需要调用其它EJB，则需要将这些EJB的接口定义以及PKUAS为其生成的Stub/Skeleton都放到classes目录（以class文件的形式）或者lib目录（以JAR包形式）下面。

14.6. 应用的组装和部署

应用的组装就是将EJB 模块和WEB 模块进行组合并配置为一个完整应用系统的过程；应用的部署是将组装得到的系统放入特定的运行环境、并根据实际环境进行必要的配置，使系统可以运行的过程。从本质上讲，组装和部署应该是两个独立的过程。但在实际开发过程中，这两个过程的界限很难划分，通常是同时进行的。因此，本文将两个过程合并一同说明。

另外，当直接在PKUAS上部署EJB模块时，可以将这个EJB模块看成只包含一个EJB模块的应用。因此，在下面的把EJB模块看成特殊的应用，与一般企业应用一同介绍。

14.6.1. 对Web模块的描述文件作必要的修改

在组装和部署应用的时候，也需要对Web模块的描述文件进行修改和增加。在PKUAS中，对Web模块的描述文件主要为J2EE和JSP/Servlet规范定义的web.xml；另外，在web.xml中也会有一些关于Web模块的特定于PKUAS的信息，这部分将在后面进行介绍。

14.6.1.1. web.xml

web.xml中的绝大多数内容都在开发Web模块的时候写好，这里只需要对其中的信息做一点补充。

在web.xml中，用ejb-ref元素来描述Web module中要使用的EJB信息，例如：

```
<ejb-ref>

    <ejb-ref-name>ejb/cart/Cart</ejb-ref-name>

    <ejb-ref-type>Session</ejb-ref-type>

    <home>com.sun.j2ee.blueprints.shoppingcart.cart.ejb.ShoppingCartHome</home

    <remote>com.sun.j2ee.blueprints.shoppingcart.cart.ejb.ShoppingCart</remote

    <ejb-link>CartEJB</ejb-link>

  </ejb-ref>
```

这里描述了一个在程序中使用的名字为CartEJB的EJB，其EJB类型为会话（Session）EJB，Home和Remote接口分别是

com.sun.j2ee.blueprints.shoppingcart.cart.ejb.ShoppingCartHome和com.sun.j2ee.blueprints.shoppingcart.cart.ejb.ShoppingCart。

引用的EJB名字为CartEJB。在实际的开发过程中，被引用的EJB可能在另一个存档文件中；一般地，用jar包路径接EJB名字的格式给出，中间以#标示。

如：

```
<ejb-link>EmployeeRecord</ejb-link>

<ejb-link>../products/product.jar#ProductEJB</ejb-link>
```

14.6.2. 编写描述文件

描述文件中保存了应用的配置信息。在PKUAS上面部署的应用，需要编写与应用相关的部署描述文件：

application.xml：由J2EE规范规定，对应用结构进行描述。

web.xml：特定于PKUAS的配置文件，对web应用的部署信息进行详细描述。

下面将分别介绍。

14.6.2.1. application.xml

application.xml主要描述应用本身及其包含的模块的静态信息。目前主要包括：

应用的信息：应用的名字display-name应用的描述descriptionEJB

模块信息：每个EJB模块中可以包含多个EJB，这里只描述EJB模块对应的物理文件（*.jar）。Web 模块信息：每个Web模块中包含一个可以通过Web方式访问的应用，这里描述Web对应的物理文件（*.war）及其启动后的路径（在URL中的信息）。

14.6.2.2. web.xml

web.xml位于每一个war包中，描述了部署在PKUAS上的web应用的一些具体信息，主要是外部资源引用。整个xml文件的信息结构如下：

- web：整个XML文件的最外层唯一元素，下面所有的元素都是它的子元素
  - resource-ref
    - 同pkuas-ejb.xml文件
  - resource-env-ref
    - 同pkuas-ejb.xml文件
  - login-config：供single-sign-on使用
    - auth-method
    - form-longin-config
      - form-login-page：声明用于给用户登录的页面
      - form-error-page：声明安全认证失败的页面

14.6.3. 打包应用

打包是指将前面开发的EJB、Web module以及组装、部署描述文件组合成一个可以在PKUAS中可以运行的文件。目前，PKUAS中支持在其之上部署EJB应用（只包含EJB，以JAR的形式部署）、Web应用（只包含Web页面、JSP/Servlet、Javabeans等）以及企业应用（包含EJB、Web module）。在本节中，将会分别介绍这三种应用的打包过程。

1．企业应用的打包过程：企业应用以EAR文件的形式部署到PKUAS上面。一个完整的EAR文件应该包括：

- 任意多个EJB的JAR
- 任意多个Web的WAR
- 组装、部署描述文件

2．EJB应用的打包过程：

打包后的应用就可以进行部署了。

EJB应用以JAR文件的形式部署到PKUAS上面。一个完整的JAR文件应该包括：

- 任意多个EJB：这个JAR为一个EJB模块，其中可以包含任意多个EJB，JAR中包含了EJB所需要的类文件。

3．Web应用的打包过程：与EAR应用的打包类似。

14.6.4. 应用的部署

在PKUAS中，部署即将打包好的应用放到PKUAS中应用部署目录中。具体的说，就是将EAR文件放到PKUAS安装目录下面的applications目录下。PKUAS会自动检测到应用的更新，如果是一个新的应用，那么PKUAS会直接部署，如果是一个已存在的应用的较新版本，那么PKUAS会将旧的应用卸载，然后部署新的应用。

还有一种方法就是通过网页端管理工具JSR77 Management Tools，在ear应用管理中，可以远程部署新的应用。

部署时，服务器为应用在cache文件夹中建立相应的目录，然后调用应用管理器对应用的各个部分进行解析和部署。

14.7. 应用的启动

目前，PKUAS中的应用启动与服务器启动同步，即启动PKUAS的同时，在部署目录（application）下面的所有应用都会被自动启动。

PKUAS提供自动部署功能，服务器会自动检测application目录，发现新的应用包或者应用包有所改动时，会重新部署该应用。

14.8. 高级特性

14.8.1. 使用事务

开发者在开发EJB构件的时候，可以利用应用服务器提供的事务服务来提高构件运行的可靠性。在构件开发过程中访问事务服务使得构件内部的操作具有ACID性质。根据J2EE规范，应用服务器给用户提供两种访问事务服务的途径：一种是编程式访问；另一种是声明式访问。

14.8.1.1. 声明式

Session Bean，Entity Bean和MessageDriven Bean都可以使用声明式事务服务。构件开发者只需要在应用的每一个构件的描述文件中声明事务服务的需求，不必关心服务器端具体的实现代码是怎么样的。例如，在eshop应用中有一个构件叫Order(Order是一个实体Bean)，开发者希望Order构件的create()方法（对应Bean类中的ejbCreate()方法。）在事务环境中执行，也就是希望在开始执行create()方法前必须已经启动事务，执行完create()后结束事务。由事务服务的支持来保证create()方法中的数据库操作的可靠性和一致性。

事务属性有以下几种：Required, RequiresNew, Supports, Mandatory, NotSupported, Never。对每一种事务属性，应用服务器会做出相应的反应策略来满足用户的需求，参见下图：

TRANSACTION ATTRIBUTE	CLIENT'S TRANSACTION	BEAN'S TRANSACTION
Required	none	T2
	T1	T1
RequiresNew	none	T2
	T1	T2
Supports	none	none
	T1	T1
Mandatory	none	error
	T1	T1
NotSupported	none	none
	T1	none
Never	none	none
	T1	error

从表中可以看到Required属性的含义是如果客户端没有发起一个事务，那么服务器会在调用Bean的方法前启动一个新的事务；如果客户端已经发起了一个事务，那么这个事务会传播到被调用端，Bean的方法就在客户事务上下文中执行。其他事务属性的含义都可以从表中得知。客户根据自己不同的需要可以在相应构件的ejb-jar.xml文件中设置合适的事务属性。在构件描述文件中声明事务服务需求也支持通配符表示构件的所有方法，以及支持对同一方法名，不同参数的方法的事务声明。

14.8.1.2. 编程式

Standalone客户，Web客户和Enterprise Bean中的Session Bean及MessageDriven Bean还可以使用编程式访问事务服务。客户通过字符串”java:comp/UserTransaction”利用名字服务得到javax.transaction.!UserTransaction这个JTA接口的实现，然后调用!UserTransaction的begin()方法启动一个事务，commit()方法提交一个事务，rollback()方法回滚一个事务以及其他一些方法来访问和控制一个事务。Enterprise Bean中的访问流程如下所示：

```
Context ctx = new InitialContext();

UserTransaction utx= (UserTransaction)ctx.lookup("java:comp/UserTransaction");

utx.begin();

Connection conn = getConnection();

PreparedStatement ps = conn.prepareStatement("insert into accounts values(?,?,?)

ps.setString(1,"123");

ps.setString(2,"456");

ps.setDouble(3,789);

ps.executeUpdate();

utx.commit();
```

对于Session Bean和MessageDriven Bean除了通过名字服务得到UserTransaction接口外，还可以通过EJB Context得到这个接口：

```
protected SessionContext ctx;

……

UserTransaction utx = this.ctx.getUserTransaction();

……

utx.begin();

//do something

utx.commit();
```

14.8.2. 使用安全

14.8.2.1. 安全机制

EJB规范定义了完整的安全机制来控制用户对每一个EJB方法的访问。EJB规范不赞成EJB开发者将安全控制硬编码到EJB程序中，而鼓励遵循规范在部署文件中配置安全访问权限，这样可以使EJB能够灵活地部署到各种操作环境中。

为了简化部署者(Deployer)的工作，应用程序装配者(Application Assembler)可以定义安全角色(security roles)。一个安全角色是为假定的用户设置的一组访问权限，应用程序装配者可以在安全角色中详细定义对每 一个EJB方法的访问权限。部署者在部署EJB时，只需要将用户或组简单地对应到安全角色上。

14.8.2.2. 定义安全角色

使用security-role元素来定义安全角色

用role-name元素给角色命名

使用description描述角色(可选)

定义的安全角色在ejb-jar整个文件范围有效，即在同一个ejb-jar文件的所有的ejb都可使用这些安全角色。

14.8.2.3. 方法授权许可

定义安全角色后，可以通过method-permission元素指定相应角色所能进行的方法调用。

method-permission元素包括下列子元素：

- role-name：角色名称
- method：授权调用的方法
- ejb-name：方法所在的构件
- method-name：方法名称

14.8.2.4. 定义用户

第三节定义的角色需要委派给服务器的用户，因此，首先必须在服务器中创建用户。注意和大多数应用服务器不同，PKUAS中没有组的概念。这主要源于这样的考虑：安全角色本身就是一个抽象的用户组，一个安全角色可以委派给多个用户。对于大多数应用而言，通过将角色委派给用户已经可以满足需求。

PKUAS通过JAAS机制实现用户认证，目前服务器提供的缺省认证模块为pku.as.securety.login.UsersRolesLoginModule。该认证模块对用户名、密码进行认证。

PKUAS提供了一个命令行节目创建用户。通过运行PKUAS安装目录下bin目录下面的useradmin.bat（windows平台）或useradmin.sh（linux/unix/Solaris平台），实现用户管理功能。

14.8.2.5. 将角色委派给用户

创建了角色、用户，还必须将不同的角色委派给相应的用户，这必须在（pkuas.xml） application.xml中添加项目security-realm进行“角色委派”：

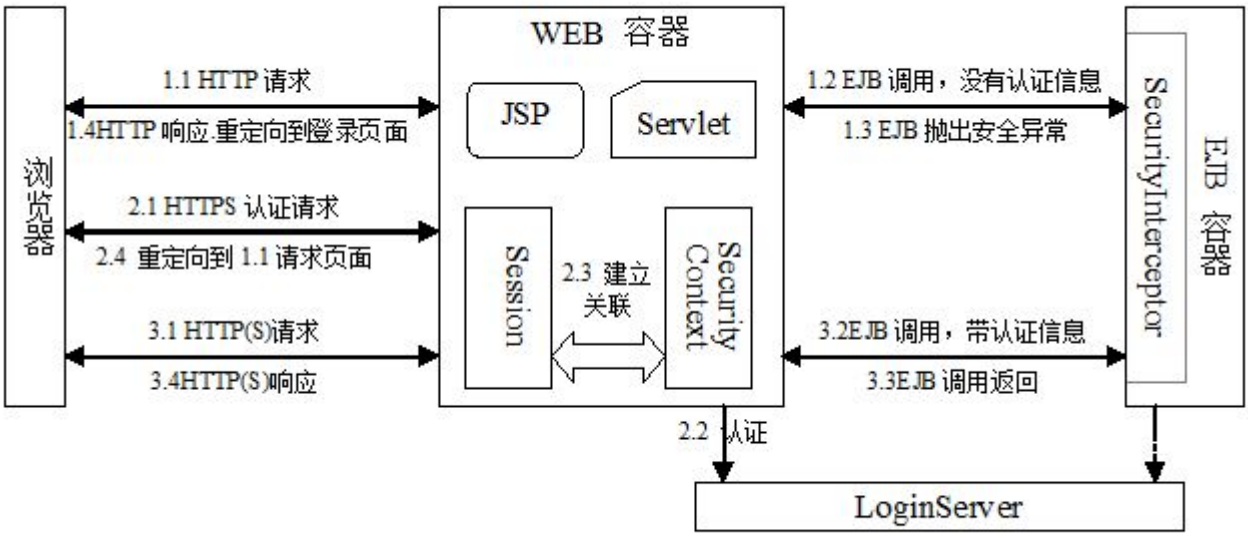
security-realm包括下列子元素：

- realm-name：安全域名称
- mapping：将哪个角色委托给哪个用户

14.8.2.6. 使用PKUAS的“单次登录（Single signon）”

14.8.2.6.1. pkuas中的WEB访问流程及“单次登录”

PKUAS对Servlet规范中定义的标准安全性进行了扩展。如果应用开发者没有进行如图3所示的页面保护配置，则意味着安全检查延迟到对EJB构件访问时进行。WEB容器会捕获对EJB构件访问时引发的安全异常：如果安全异常的原因是用户没有登录，则将请求重定向到登录页面。WEB容器获得用户的身份凭证信息然后，到认证服务器认证，并建立本次会话的安全上下文。WEB容器以后发出的EJB调用，都携带该安全上下文。



上图描述了用户通过WEB方式对PKUAS中受保护的EJB构件进行访问的一次流程：

用户对某JSP页面发出HTTP请求（1.1），WEB容器接收到该HTTP请求，并进行分析；

如果该页面没有被保护，JSP或者Servlet对象将调用后台的EJB构件，进行EJB调用（1.2）；如果该页面被保护，则转到4）；

由于EJB调用（1.2）访问的EJB构件受保护，而此时用户没有登录，因此EJB调用请求（1.2）中没有安全认证信息，EJB容器负责安全检查的SecurityInterceptor将引发安全异常。该异常被WEB容器捕获后，转4）；

WEB容器将HTTP请求（1.1）重定向到登录页面，要求用户输入身份认证信息。登录页面的访问通过安全传输协议HTTPS进行；

用户输入用户名/密码，WEB容器将到认证服务器LoginServer进行认证（2.2），得到认证服务器返回的认证令牌后，建立安全上下文；并建立HTTP会话和该安全上下文的关联（2.3）；最后重定向到原来的HTTP请求（1.1）指向的页面；

用户开始正常的HTTP(S)请求（3.1），当调用EJB构件时，根据本次会话信息，得到安全上下文，随后进行的调用请求（3.2）中，将携带相关认证信息；

EJB容器进行安全检查。通过后，返回本次调用结果（3.3）；

浏览器得到响应（3.4），本次调用结束。

除非用户关闭本次会话，否则该浏览器随后发出的对当前应用的HTTP调用，都不必让用户输入安全认证信息，这就是所谓的“单次登录”（Single Sign-on）。WEB容器负责对一次会话进行管理。

从上面的WEB接入流程可以看出，应用服务器在WEB容器和EJB容器内分别进行了安全检查。在PKUAS中，浏览器和WEB服务器之间的传输安全由集成的Tomcat WEB服务器负责，WEB容器和EJB容器使用统一的安全认证设施。

14.8.2.6.2. 使用“单次登录”

在PKUAS使用单次登录，需要在WEB端的部署描述信息中加入有关安全的信息，以及提供用于用户登录和登录失败后显示的两个页面。

在（pkuas.xml）application.xml中描述安全信息入下所示：

```
<web-module>

       ……

       < auth-method>form</auth-method>

<form-longin-config>

              <login-page>login.jsp</login-page>

              <error-page>error.jsp</error-page>

       </form-login-config>

</web-module>
```

form-longin-config元素中声明用于登录的JSP页面和登录失败后返回的错误页面。

一个简单的登录页面如下所示：

```
<html>

<head>

<title>Login Page for Eshop</title>

<body bgcolor="white">

<form method="POST" action='<%= response.encodeURL("j_security_check") %>' >

  <table border="0" cellspacing="5">

    <tr>

      <th align="right">Username:</th>

      <td align="left"><input type="text" name="j_username"></td>

    </tr>

    <tr>

      <th align="right">Password:</th>

      <td align="left"><input type="password" name="j_password"></td>

    </tr>

      <td align="right"><input type="submit" value="Log In"></td>

      <td align="left"><input type="reset"></td>

    </tr>

  </table>

</form>

</body>

</html>
```

其中，包含的form的action必须为response.encodeURL("j_security_check")。

附录 A. 附录

本规范参考以下文档:
[1]AAAA

[2]BBBB

[3]BBBBBBB