研究项目：JBoss架构分析

Jenny Liu
School of Information Technologies
University of Sydney
jennyliu@cs.usyd.edu.au

熙可集团 系统分析员:李剑华翻译
huihoo.org Allen整理,制作
English

摘要

JBoss是一个免费的开放的J2EE实现。它的架构是基于高标准的模块化和插入式设计。JBoss使用工业标准的JMX来管理， JBoss组件和为EJB提供服务。基于我们以前的开发经验，我们发现了不同的J2EE应用服务器间的存在着巨大的性能和可扩展性差异。我们相信架构的设计是决定类似于性能和可扩展性等质量指标的重要因素。分析和展现JBoss架构模型有助于我们了解其内部行为并帮助我们创建一个精确的最终性能模型。在这个项目中，我们分析JBoss应用服务器架构的四个特殊部分，JBoss EJB 容器、JBossNS、JBossTX以及JBossCMP，逆转工程工具能使我们通过源代码来分解组件/子系统。无论是三个JBoss子系统的概念模型或实际模型都将被我们用来讨论JBoss 架构模块设计风格。

Table of Content

图示列表

---

1. 介绍

1.1 JBoss是什么?

JBoss是免费的，开放源代码J2EE的实现，它通过LGPL许可证进行发布。它提供了基本的EJB容器以及EJB(好像应该是J2EE)服务，例如：数据库访问JDBC、交易(JTA/JTS)、消息机制(JTS)、命名机制(JNDI)和管理支持(JMX)。目前的JBoss发布版2.2.4实现了EJB 1.1和部分EJB 2.0的标准、JMS 1.0.1、Servlet 2.2、JSP 1.1、JMX 1.0、JNDI 1.0、JDBC 1.2和2.0扩充(支持连接池 (Connection Polling))、JavaMail/JAF、JTA 1.0和JAAS 1.0标准，JBoss是100%纯Java实现能运行于任何平台。

1.2 动机

这个项目的动机是我们想分析一下中间件基础系统的性能。基于我们以前的开发经验，我们知道不同J2EE应用服务器在性能和可扩展性方面有着极大的差异，并且相信架构的设计是决定类似于性能和可扩展性等质量指标的重要因素，我们想通过分析这个系统来了解架构设计究竟对于性能和可扩展性具有着怎样的影响。无论概念性模型的局限性或实际模型中对于系统运行期行为的Reflect(反射机制)应用，他们还是能提供

给我们一个对于整个系统的全面架构的了解的视点和符合基本境况的分析模型的构建的前提。

1.3 方法论

大型软件系统的架构分析可以分为两个层面：概念性架构和实际架构。概念性架构通过将子系统的"捆绑式"分析和子系统间的分析描述了这个系统的架构。每一个子系统具有清晰的有意义的方法和他们包含了整个系统的特殊的架构风格。实际架构和概念性架构比起来具有较少的层次关系。它表述了实际的编程规划/模型的实际体现，它和想象的概念性架构有很多不同。在这个项目中，我们将JBoss的概念性架构和实际架构进行了分割。想象的概念性架构模型通过参考资料来分割和获得，我们自己的经验来自配置应用系统和JBoss的在线论坛。眼下，我们先关心一下每个组件在模块层面上的功能性，他们彼此不相关。实际架构模型是综合的。我们使用逆转工程工具Together 5.5以便于将源代码翻转成为类(class)图和序列(sequence)图并使他们在一个子系统模型中综合。Together 5.5支持应用设计，实施，配置和JBoss的逆转工程。它可以通过Java文件和class文件来获得类图。更深一层，我们通过使用Together选择相应的特殊的方法来获得序列图。有两个工具可以帮助我们来了解组件行为，最终实际模型和概念性模型的比较。在实际模型中以外的模块、组件和其它部分也将被讨论。

1.4 组织

这份报告将按照以下次序进行组织：第二部分将介绍JBoss架构的整体设计和主要的组件。第三部分讨论JBoss子系统的概念性模型，即：容器框架和它的插件。JBoss命名服务(JNDI)，JBoss容器持久性管理(CMP)和JBoss 交易服务。第四部分，我们将挖掘JBoss子系统的实际模型和被提及的组件间的相应稳固的关系。第五部分，我们来评价一下JBoss 的架构风格和在性能、可更改性、可扩展性等一系列质量指标上的表现。在第六部分我们将讨论我们将来的工作和提出我们报告的最终结论。

2．2．JBoss 服务器架构概述

JBoss的构架和其他J2EE应用服务器的构架有着巨大的不同。JBoss的模块架构是建立在JMX底层上的，下图展现了JBoss主要组件和JMX的联

系。

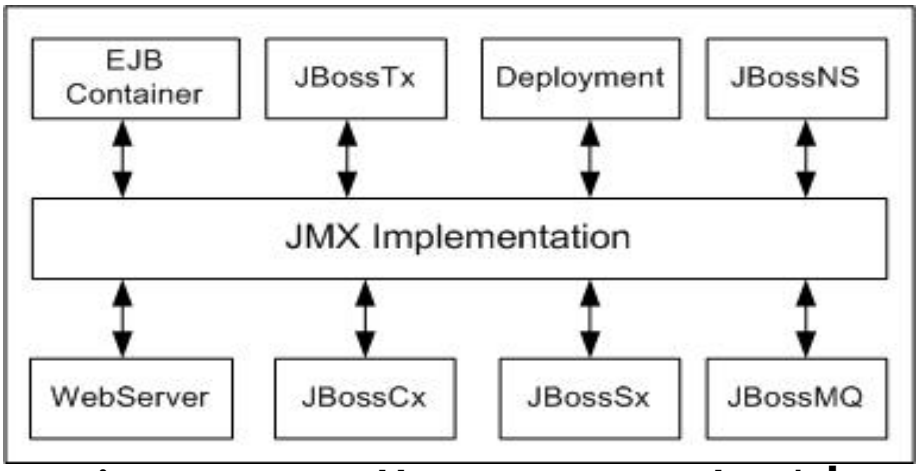

Figure 1-1 Overall JBoss Conceptual Model

2.1 JMX - *层次*

JMX是一个可复用框架，它为远程**(Remote)**和本地**(Local)**管理工具扩展了应用。它的架构是层式架构。他们是实现层**(instrumentation layer)**、代理层**(agent layer)**和发布层**(distribution layer)**。其中，发布层还在等待未来的标准化。简要的表述是，用户使用管理**Bean**，**MBean**来提供获得相应资源的实现方法。实现层实现相关的特性资源并将它发布于**JMX**相关应用中，它的代理层控制和发布相应的注册在**MBeanServer**代理上的管理资源。**.**

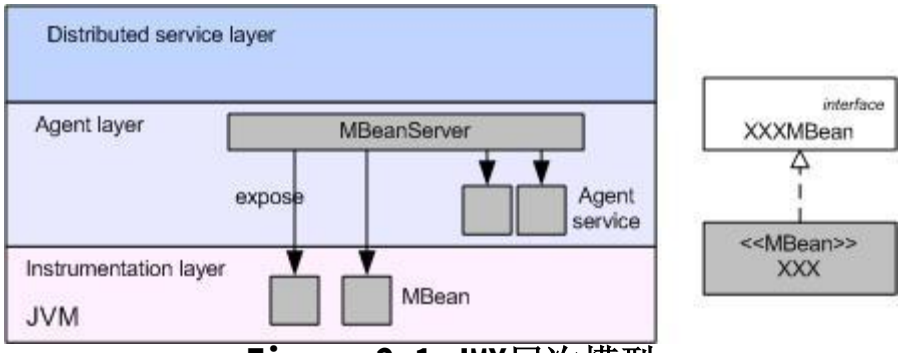

Figure 2-1 JMX层次模型

2.2 JBoss主要模块

主要的**JBoss**模块是在**MeanServer**上的可管理**MBean**。 **[2]. 1.JBoss EJB**容器是**JBoss**服务器的核心实现。它有两个特性，第一是在运行期产生**EJB** 对象的**Stub**和**Skeleton**类，第二是支持热部署。
2.JBossNS是**JBoss**命名服务用来定位对象和资源。它实现了**JNDI J2EE**

规范.
3.JBossTX 是由JTA/JTS支持的交易管理控制.
4.部署服务支持EJB(jar)、Web应用文档(war)和企业级应用文档(ears)的部署。它会时刻关心J2EE应用的URL情况，一旦它们被改变或出现的时候将自动部署。
5.JBossMQ使Java 消息规范(JMS)的实现。
6.JBossSX支持基于JAAS的或不支持JAAS机制的安全实现。
7.JBossCX实现了部分JCA的功能。JCA制订了J2EE应用组件如何访问基于连接的资源。
8.Web服务器支持Web容器和Servlet引擎。JBoss 2.4.x版本支持Tomcat 4.0.1，Tomcat 3.23和Jetty 3.x服务.

2.3 他们是如何工作的?

当JBoss被启动，它的第一步是创建一个MBean服务器的实例。一个基于管理机制的MBean组件通过在Mean Server中的注册而被插入JBoss中。JBoss实现了动态类装载 M-Let 服务，它是代理服务，M-let允许MBean被注册到MBean服务器上。通过基于文本文件的配置文件中的配置，相应MBean将被装载。
JMX MBean服务器实际上本身并没有实现很多功能。它的工作类似于一个MBean中联系的微核聚合组件，通过Mbeans取代JMX MBean 服务起来提供相应的功能，换而言之，真正起作用的是MBean。JBoss的整体架构并不是依循Garlan和Shaw文件中的架构风格严格分类的，代替它的是一个组件插入式的框架。MBean的接口是一个连接器。
在这封报告的余下部分，我们选择了JBoss架构中的JBoss EJB容器、JBossNS、JBossTX和JBossCMP子系统来加以学习。虽然JBossCMP，实体Bean的容器管理持久层是容器架构的一部分，但我们还是将它分开讨论，因为他们有自己的构架。我们只在这个项目中介绍三个部分是因为它们是我们关心的JBoss应用服务器的性能问题的关键点。在这个项目中我们使用的方法学可以扩展到更多有用的子系统的学习中去。

---

3. 概念架构模型

3.1 容器的概念性架构模型 - *插入式*

JBoss EJB容器是JBoss服务器的核心实现。图3-1展示了EJB容器的概念

性模型。我们发现JBoss容器的架构并不是一个严格意义上的层，决大多数的独立件是双向管理，容器依赖于更多的低层次组件。实际上，容器和它的插件、实例池(instance pool)、实例缓存(instance cache)、拦截器、实体持久管理、有状态会话持久管理，都是基于插入式框架来为特定的EJB提供相应的EJB服务。

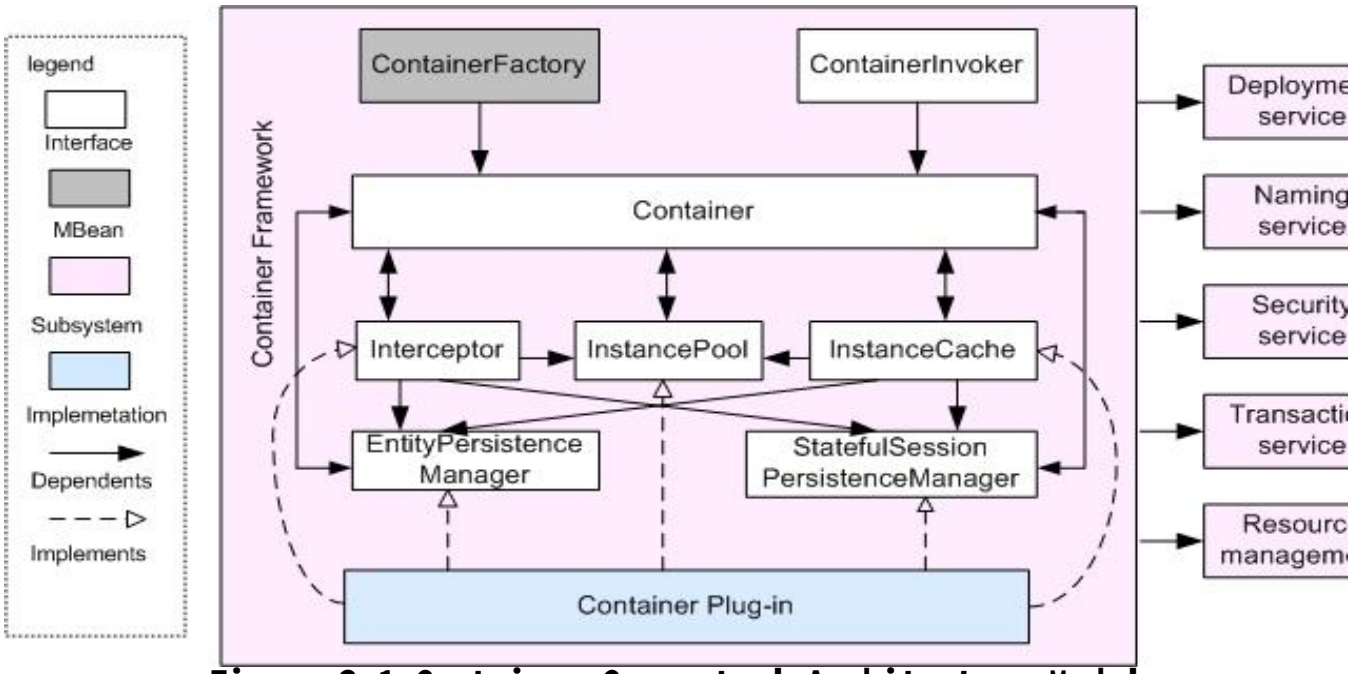

Figure 3-1 Container Conceptual Architecture Model

3.1.1 主要的组件和接口

客户端不可以直接访问EJB实例而是要通过Home(EJBHome)和容器提供的远程对象 (EJB Object) 接口。Container类是依循客户端的调用来提供Bean实例并实现操作。Container类的责任来实现插件的交互，为插件提供信息来实现操作并管理Bean的生命周期。Container类有四个子类(四种Bean的类型)，分别是：StatelessSessionContainer、StatefulSessionContrainer、EntityContainer和MessageDrivenContainer。它们是由ContainerFactory通过相应的Bean类型在部署期中被创建和初始化的。 ContainerFactory被用来创建EJB容器和在容器中部署相应的EJB。ContainerFactory 被作为一个MBean实现。这意味着JBoss服务器启动的时候其相应的服务也被启动。它会对EJB-jar的XML定义文件获得相应的URL。ContainerFactory使用EJB-jar XML中的元数据产生一个容器实例并使他们处于可被调用状态，在部署期中，ContainerFactory的功能包括：

- 通过在部署描述文件中的EJB类型来创建Container类的子类，即四个子类中的一个。
- 通过jboss.xml和standardjboss.html文件创建容器属性。
- 通过定义在standardjboss.html文件中的内容创建和添加容器拦截器。
- 使用应用对象来联系相应的容器。

ContainerInvoker 是一个Java RMI 服务器对象。正如他名字所表述的，ContainerInvoker 通过客户端的请求 (request)方法来调用相应的容器。可以看作客户端请求和容器间的接口，它利用RMI来获得自身的有效调用，无论这个调用来自其他JVM上的远程客户或是来自同一JVM上同一EJB应用的其他Bean。ContainerInvoker工作在通讯层面上，通过特殊协议进行通讯。如果想在JBoss服务器上实现新的协议，第一是需要提供一个该协议的ContainerInvoker实现。JBoss 通讯层回复是通过Sun RMI的JRMP，ContainerInvoker的RMI版本是JRMContainerInvoker。一个ContainerInvoker在EJB中实现分割相应的通讯协议，这种设计增加了系统的可更改性。JBoss EJB容器中所使用的协议可以在相应的服务器配置文件中定义。

EJB对象实例被放入InstancePool中以减少在运行期中创建它们的开销。在Instance Pool中的实例不能和其他的EJB对象交流，它们由Instance Pool来管理。

有状态会话Bean和实体Bean实例将被缓存化，在生命周期中它们拥有相应的状态。一个缓存实例通过实例池获得，他们和特殊的对象相关联并具有相应的标示。其状态由InstanceCache控制，例如在缓存中的实例状态和第二方存储介质中的对象的同步。

EntityPersistenceManager 对于实体Bean的持久性起作用。

StatefulSessionPersistenceManager 对于有状态会话Bean的持久性起作用。

拦截器通过容器获得相应的方法调用。在容器配置文件standardjboss.html中，被方法调用的拦截器必须被依次定义在其中。图3-2展示了通过拦截器的方法调用的逻辑执行顺序。

图3-2 通过拦截器管道的方法调用

这个设计遵循了David Garlan和Mary Shaw的"管道和过滤"("pipe and filter")架构定义，在此定义的过滤原型是一个组件，其包括了数据流和类似于输出总是发生在输入流被完全读取之后等方面的计算输出的增强，拦截器是过滤器而方法调用是连续拦截器中的连接器。拦截器是整个构架中的优势部分：

- 它能够知道每一个阶段管道的行为。
- 它通过模块来支持重用和扩展，不同拦截器之间具有明显的一个功能性区别。添加一个新的拦截器，第一是需要实现拦截 器的接口并将他定义到相应的容器配置文件中
- 支持同步.
- 具有快速的容错语义表述.如果在拦截器处理中产生错误或违例,就会出现相应表述。

3.1.2 依赖性

本质上，InstancePool，InstanceCache，EntityPersistenceManager，StatefulSessionPersistenceManager，都是容器插件的接口。容器的插件是这些接口的实现对象的集合。JBoss容器并不做太多的复杂的工作，它只是提供了一个联系不同插件的框架。

- 容器向拦截器发出方法调用信息。
- 拦截器依赖于InstancePool来访问会话Bean实例，或是依赖于InstanceCahce访问实体Bean实例
- 最好还是为了一个自由Bean时，InstanceCache会和InstancePool进行互动
- InstanceCache依赖于EntityPersistenceManager(或StatefulSessionPersistence Manager) 通过数据源进行实例状态初始化。.
- 拦截器依赖于EntityPersistenceManager(或StatefulSessionPersistenceManager)通过数据源进行实例状态的同步。

当执行客户端请求时，容器的框架依赖于外部的其他的服务功能块、命

名服务、交易服务、安全服务和资源管理。举个例子，当客户端请求一个交易信息将更新数据库内容，容器会通过命名服务获得相应的数据源和资源管理提供的相应的数据源驱动。整个交易过程在容器内进行，交易管理器和资源管理器由交易服务进行控制。

不象传统的分布式系统构架，EJB容器在部署描述文件中声明了外部属性。虽然容器担当的是和元数据信息的通讯作用，它在部署服务中显示出的外在独立性和其他的服务还是有点不同的。这意味着它的信息在部署期时就被放入容器中了。

3．2 JBoss命名服务的概念性模型

3.2.1 主要的JNDI API

JNDI提供了为数众多的命名服务。主要的JNDI API是javax.naming.Name，javax.naming.Context以及javax.namingInitialContext。根本上命名系统是一个对象的集合并且每个对象都有独立的名字。Context是用户端访问命名服务的接口。InitalContext 实现了Context。JBoss 命名系统是JBoss JNDI的提供者。源代码在org.jnp包中，就像我们在第二部分中提到的一样，JBoss命名系统被实现成为MBean。图3-2 展示了JBoss命名系统的概念性模型。

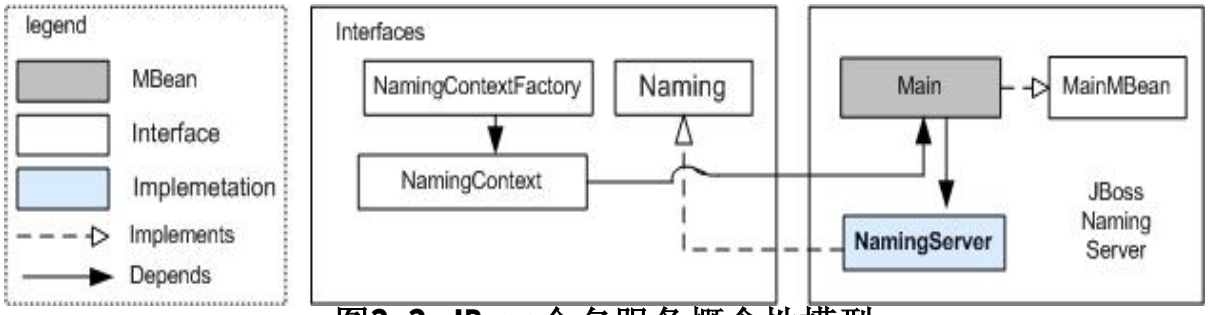

图3-3 JBoss命名服务概念性模型

3.2.2 主要的组件和接口

Org.jnp.server包包含了命名服务的MBean，Main包装了Main NamingServer并发布它。NamingServer的工作是进行"命名-对象" 一对对的序列编排。
Org.inp.interface包继承/实现了javax.naming.*接口，这是个J2EE规范。这个接口可以通过客户端远程访问。它使得Main可以在Naming

Server中和命名服务进行交互，NamingContext 实现了javax.naming.Context接口，它是客户端和JBoss命名服务之间的接口。

3.2.3 依赖

JBossNS没有更多的外部依赖.

3.3 JBossCMP 概念性模型

3.3.1 主要组件和接口

JBossCMP通过扩展JAWS来支持内存中Java对象和关系型数据库基本储存之间的映射(是一个O/R Mapping的概念)。 JBossCMP包含了支持EJB 1.1容器持久性管理(CMP)模型的组件。在CMP的实体Bean模型中，EJB实体的持久性状态的性能是由容器决定的。容器访问数据库是实体Bean的行为。图3-4展示了JBoss CMP服务的概念性模型。

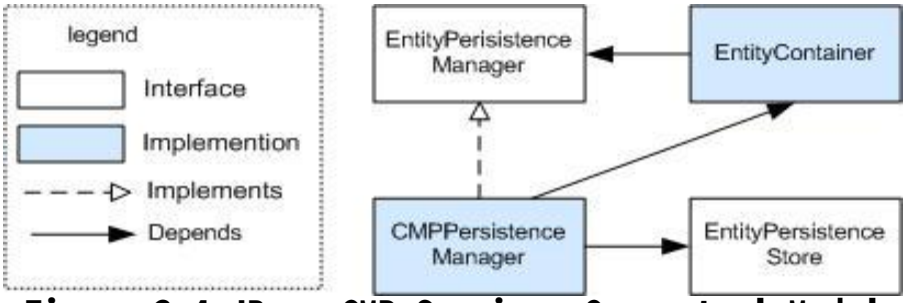

Figure 3-4 JBoss CMP Services Conceptual Model

EntityContainer 依赖EntityPersistenceManager接口为持久性管理的实体Bean。
CMPPersistenceManage实现了EntityPersistenceManager接口。就像我们前面提到的，容器管理了实例的状态。EJB1.1CMP的语义有回调方法、 ejbLoad、ejbStore、ejbActivate、ejbPassivate、ejbRemove提供了实例状态观点的容器表述。实际上，正是CMPPesistenceManager在做一项工作：底层的数据库和缓存实例状态的同步。举个例子：当一个Bean的缓存数据被装载，CMPPersistenceManager将会在Bean实例中调用容器的回叫方法ejbLoad。当缓存数据需要更新数据库，ejbStore方法将被调用来准备相应的缓存数据，这样CMPPersistenceManager 将关注于更新数据库。
EntityPersistenceStore接口的实现关注的是具体的物理储存细节。CMPPersistenceManager授权于EntityPersistenceStore进行实体持久

性内容的实际储存。注意EntityPersistenceStore是一个接口，它留着持久层储存实现的客户化空间，e.g。基于文件的储存或数据库存储。

3.3.2 依赖性

JBossCMP 并不是和JBossNS一样被实现成为MBean服务。实际上， 它被包含在EJB容器包org.jboss.ejb中通过容器和其他的插件进行交互。表面上，JBossCMP是依赖于JbossNS来获得相应的数据源 调用并在Bean实例中存放持久性数据。

3．4 JBossTX 概念性模型

3.4.1 主要组件和接口

JBossTX 构架可以使用的是任何实现了JTA规范的交易管理。 在分布式交易中主要的参与者包括：：

1.交易管理器：它提供了相应的服务和管理方法来支持交易划分、交易资源管理、同步和交易内容传播等功能。使用javax.transaction.TransactionManager接口以便于可以通过RMI来输出交易管理。

2.应用服务器：一个应用服务器(或TP监视器)提供了基础结构来支持运行期环境的交易状态管理应用。应用服务器的例子是:EJB服务器/容器。

3.资源管理器：资源管理器提供进入其他资源的功能。资源管理器的例子是：关系型数据库服务器。

4.交易内容：交易内容确定一个特定的交易。

5.交易客户端：交易客户端在单个交易中可以调用一个或多个交易对象。

6.交易对象：交易对象的行为是由交易内容中的运作来决定的.绝大多数的EJB Bean是交易对象。

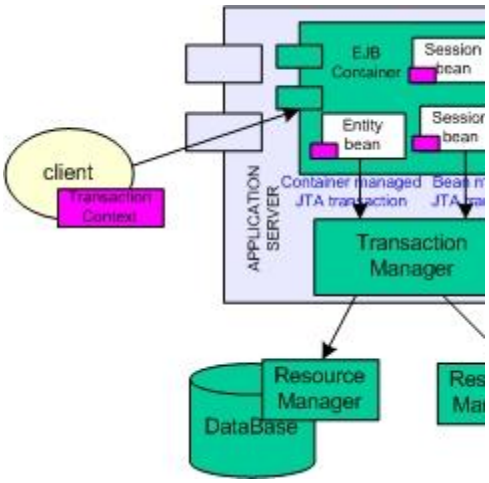

Figure 3-5 JBossTX 概念性

3.4.2 依赖

JBossTX架构被设计成为可以使用任何的实现了JTA javax.transaction. TransactionManager接口的交易管理， JBoss交易管理将被看作为一个MBean，并可通过RMI当作一个服务输出其自身。

交易管理服务的基本需求是在JBoss服务器服务管理启动的时候通过JNDI命名目录绑定它的实现。因此，JBossTX看上去是依赖于 JBossNS的。

4．4.实际架构模型

实际架构模型是通过工程逆转工具通过JBoss源代码获得的。我们选择了TogetherSoft 公司的 Together 5.5。Together 5.5 是一个Case IDE工具，它可以通过源代码或jar文件来生成相应的类图。它还支持跟踪相应的类图以创建出相应的序列图的方法，用户可以选择可视化界面。这里是一个它如何工作的概述。
当开发者注意到在包中的一组文件具有相同的功能性，我们发现源代码包的层次性非常类似于概念性模型。我们将不同包中的源代码依循我们的概念性模型进行了相应的组重排，并将它们导入了Together 5.5。
实际架构模型和我们所期待对于概念模型的认识有着本质上的区别。一些组件和关联的表现是相当特殊的，并且有一些不做任何表示。这是因为实际模型和概念性模型比较起来更接近于运行期行为的实现观念。在这个章节中，我们将讨论容器、命名服务、容器管理持久性(CMP)服务和交易服务的实际模型。

4.1 容器实际模型

4.1.1 获得综合实际模型的方法

图4-1 展示了容器和其插件的独立的图。我们从Together 5.5获得的实际构架模型和我们脑中固有的对概念模型的认识有着很大的区别，这是一个层构架模型，在顶层实现的是插入式实现组件，容器位于中间层，插入接口位于最底层。拦截器和持久性管理之间的关系是不可见的，它是基于JBoss文档的计算本能。我们找到它归因于Together 5.5的关系型可视化的陷阱。在同一层面上的组件关联都被忽略，举个例子，实体实例拦截器和实体实体。为了解决这个问题，我们使用下面的步骤：

1.重排容器的源代码和它的插件接口，依据Bean类型来实现。将它

们导入**Together 5.5**项目。
2.分析组件的依赖关系和固有联系。
3.为关键组件的选择关键方法，追踪方法调用来生成消息视图。
4.重复**1**，**2**，**3**步骤以合成容器架构的实际模型。
图 **4-1**、**4-2**、**4-3**

Figure 4-1 container dependecy diagram

Figure 4-2 container inherence diagram

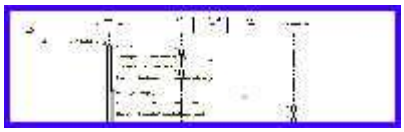

Figure 4-3 method invocation message diagram

4.1.2 非正规组件和依赖

在这里我们只讨论实体容器的构架模型。在附录中我们展示了 **StatelessSessionContainer** 和 **StatefulSessionContainer** 实际架构模型。

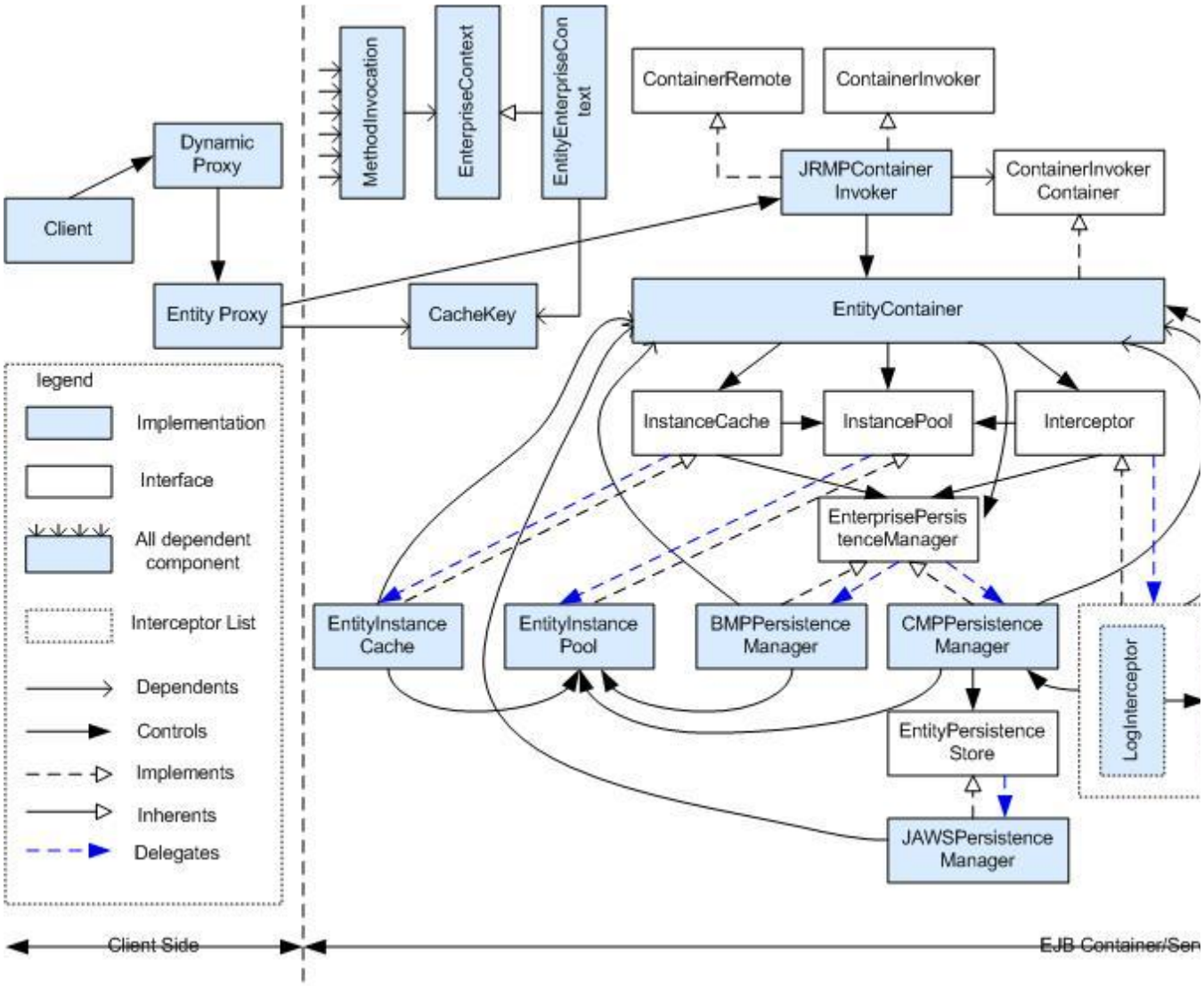

Figure 4-4 实体容器的实际构架模型

- *基于容器的动态代理设计*

在**EJB**规范中，**EJBHome**实现了**Bean**的本地接口，**EJBObject**实现了**Bean**的 远程接口。**EJBObject** 和**EJB**的客户视图进行交互。容器提供者的责任是产生**javax.ejb.EJBHome**和**javax.ejb.EJBObject**。**JBoss EJB** 容器的设计观点是在这里更本没有**EJBHome**和 **EJBObject**的对象实现，这里通过动态代理机制来获得**EJBHome**和**EJBObject**的角色。 （**JBoss**是基于**Dynamic Proxy**机制而**Auspic**是基于自动代码生成机制的。）动态代理机制是一个对象，它可以在运行期中实现特定的 一系列接口通过**java**反射机制来实现。
在部署期**ContainerFactory**创建和初始化相应的容器。**Home**对象是通过动态装载机制使用**JRMP ContainerInvoker**来创建的，我们会在下面的文章中讨论它。**InvocationHandler**被**HomeProxy**类所取代，现在他可以

在部署描述文件中定义并被作为一个特定的JNDI名称绑定在JNDI命名树上。

```
Public static Object newProxyInstance(java.lang.ClassLoader loader,
    java.lang.Class[] interfaces, java.lang.reflection.InvocationHandler h) throws
    IllegalArgumentException
```

代理器是可序列化的，这意味着它可以通过网络被发送到远程客户那里去。当一个客户端通过JNDI寻找EJBHome，本地代理器实例被序列化。在客户端，本地代理器实例被反序列化。因为代理器实现了Bean的本地接口，它将可以被当作本地接口来捕获和使用。
当客户端利用本地对象引用reference来请求相应的EJBObject.EJBObject的动态代理会通过上面的代码自动生成。InvocationHandler 会被某一个EJB对象代理器所取代，即，基于Bean的类型的StatefulSessionProxy，StatelessSessionProxy和EntityProxy来获得。本地代理的共同之处，EJB 对象代理也可以被序列化并通过网络发送到远程客户端。
最后，客户端获得EJB对象的句柄，并使用它去调用在服务器端部署的Bean的方法。本地和对象的动态代理通过客户端的 InvocationHander调用。

- ***EJB 代理器***

StatefulSessionProxy，StatelessSessionProxy和EntityProxy实现了InvocationHandler接口。就像他们名称所指出的一样，在容器中他们的工作类似于一个代理器。他们首先着力处理客户端请求的 方法。如果方法是由部署在服务器端的远程方法实现所调用的话，调用将被转换为RemoteMethodInvocation对象，接着将其包装成为MarsharlledObject并通过RMI传递给JRMPContainerInvoker。

- ***JRMPContainerInvoker***

回到3.1.1章节，ContainerInvoker是一个容器传输句柄，JRMPContainerInvoker实现了RMI/JRMP传输协议。它通过接受实现ContainerRemote接口的远程方法的调用来输出其自身， 实现ContainerRemote接口继承了java.rmi.Remote.JRMPContainerInvoker，通过执行invoke()和invokeHome()两个方法来获得优化。 第一个使用MarshalledObject参

数(值调用)和其他的使用方法反射类型的参数(Reference调用)，如果调用器来自于容器内的同一VM， 调用器会选择Reference调用而忽视MarshalledObject序列化，否则，方法调用信息会被MarshalledObject通过最初的EJB方法调用 (例如安全和交易内容)的属性进行解包。接着，它被转换成文MethodInvocation并通过容器传递。

- ***EntityContainer 和它的拦截器***

在部署期中容器类型是通过Bean类型而被指定。 ContainerFactory创建并初始化它。此外，对于每一个Bean类型，standardjboss.xml都会定义使用这个容器和他们调用次序的 拦截器。拦截器都可以被创建和初始化，每一个容器维护着一组拦截器。它促使了针对于序列中第一个拦截器方法调用， 这是通过invoke()方法实现的。拦截器首先处理相应的调用并通过调用自身的invoke()方法来触发下一个拦截器，最后的一个拦截器 是ContainerInterceptor，它的工作是将方法调用委托给Bean实例。

- *依赖性*

我们重新排列了实际构架模型以便于更容易和概念性模型进行比较。象我们在3.1.1部分介绍 的InstancePool、InstanceCache、Intercptor和PersistenceManager都是继承于ContainerPlugin的接口。在实际模型中，容器的插件 层被实际实现组件所替代。这些组件实现了插件的接口，因此EntityContainer在插件接口层面的控制流被当前委托到实际实现组件。 插件接口中的关系，比如InstanceCache和InstancePool在当前的实际实现组件中的控制流,, EntityInstance 和 EntityInstancePool。
一个实体Bean的持久性管理有两种类型，Bean管理持久性(BMP)和容器管理持久性(CMP)。作为BMP的程序员你需要非常关注Bean缓存状态和底层数据库之间的同步。使用CMP，容器会产生相应的代码来处理这种事情，这将使程序员变得轻松点。它们之间性能的影响不属于这篇文章讨论的内容，我们会在以后讨论JBossCMP的架构。
每一个拦截器都实现了拦截器接口，他们都是容器的插件。和实体Bean一样， EntityInstanceInterceptor，EntityLockInterceptor 和 EntitySynchronizationInterceptor使用BeanLockManager进行Bean实例的当前控制。
每一个ContainerPlugin的实现都和EntityContainer相关，它将知道哪个容器正在被setContainer()和getContainer()方法调用。

ContainerInvoker和Container交流是必需的，举个例子一个会话Bean可以调用实体Bean的方法。ContainerInvoker是容器的一个接口，它使用Containernvoker.JRMPContainerInvoker和ContainerInvokerContainer有相关性，ContainerInvokerContainer是实际上的 EntityContainer。
MethodInvocation通过网络在所有的组件中通行。所有的组件都和它有关联.。MethodInvocation依赖于EnterpriseContext，EnterpriseContext并且和生命期实例中的Bean实例相关联。在实体Bean模型中，实体Bean是一个Java对象和底层数据库的表现。
CacheKey是对于PrimaryKey组件的封装，EntityEnterpriseContext依赖于CacheKey去获得实体对象而EntityProxy通过CacheKey作为一个实体容器的参数。

- *总结*

和概念性模型相比，一些特殊的组件和它们之间的关系展示了JBoss EJB容器的特殊之处。 动态代理机制的设计和预编译容器设计之间有着本质上的不同，预编译技术使用在很多被测试过的EJB容器实现上 (例如：WebLogic等)。附录-3展示了EJB容器测试设备的概念性结构模型。JBoss文档中宣称，基于EJB容器的动态代理设计使容器 设计变得容易。它会对质量属性有影响，例如不能在实际模型中直接观测到的性能问题。它必须经过严格的质量测试，这将是我们 将来的工作。
EJB容器的插入式框架使得EJB容器变得灵活性和扩展性更好。如果需要改变只需要写一个特定结构的新的实现即可。

4.1.3 实体Bean容器的示例和它的执行方法调用的插件

图4-5 展示了客户端和EJB容器/服务器交互 的情况.我们分析EJB容器的实际模型可以帮助我们理解容器的行为。

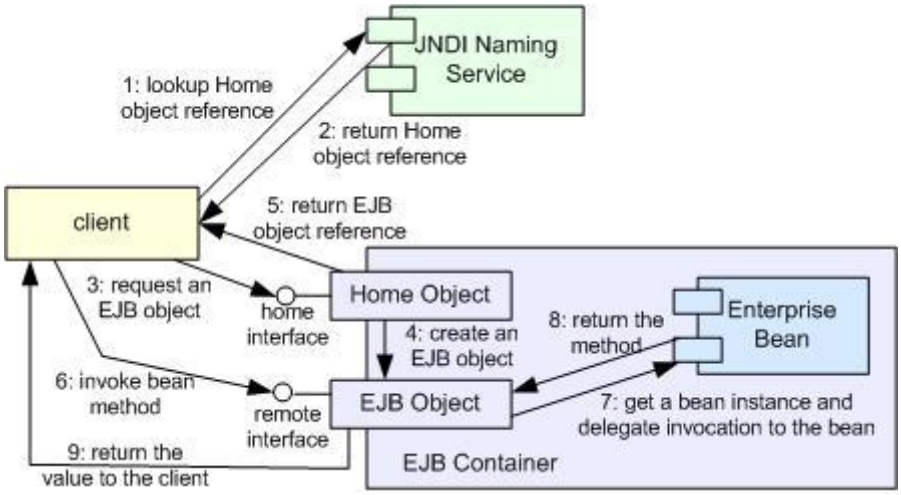

图4-5 客户端和EJB容器的交互示意

1.在部署期，EJB本地对象将被捆绑在JBoss命名服务的JNDI树上，并被分配一个JNDI名称。
2.客户端第一次接触JNDI命名服务以获得EJB本地对象。
3.客户端通过使用EJB 本地对象的Reference来向EJB对象发出请求。
4.EJB 本地对象创建(或寻找)一个EJB对象并将它的reference 返回给客户端。
5.客户端获得EJB 对象的reference，并在远程接口中调用相应方法。
6.容器拦截下方法的调用并将其委派给Bean实例，相应的实例将通过远程接口向客户端返回结果值。
7.LogInterceptor纪录下调用的日志。
8.TxInterceptor通过XML部署描述，依循调用方法来决定如何进行管理交易。
9.SecuirtyInterceptor通过XML部署描述来验证调用是否可以执行。
10.容器在他调用Bean的商务方法的时候必须有一个实例，EntityInstanceInterceptor通过给予一个主键来调用InstanceCache以此来获得相应的实体Bean实例。
11.如果缓存中没有和所提供的主键相一致的实例，它会通知InstancePool获得一个空闲的实例来和主键相关联。
12.InstanceCache 现在要调用PersistenceManager，它通过调用ejbActivate()方法来获得已被激活的实例。
13.EntitySynchronizationInterceptor被EntityInstanceInterceptor调用，用来处理实例和数据库的同步.

它有几个选项，每一个选项定义了一个拦截器，**loadEntity()**方法将在**EntityPersistanceManager**中被调用。
14.ContainerInterceptor是在整个链中最后一个拦截器，它是通过容器本身添加的而不是容器工厂，业务方法的调用现在已经被委托给了**EJB**实例。
15.实例实现了一些工作并返回了结果。
16.EntitySynchronizationInterceptor选择了将目前的实例状态储存进数据库，**PersistenceManager**的**storeEntity()**方法将被调用。

17.实例将被返回入缓存。当交易在运行时被调用，实例会和这个交易锁定以便于在这个交易期间没有别的交易可以使用这个实例。
18.TxInteceptor依循交易设置处理相应的方法并针对目前的交易选择提交或是回滚。
19.容器激活返回机制向客户端返回结果。

4.2 JBoss 命名服务实际模型

4.2.1 特殊组件和相互关系

图 **4-2** 是**JBoss**命名服务实际架构模型。作为容器的实际模型， 这里有着特殊组件和组件间的依赖关系，类似于**NamingService MBean**以及它和**org.jnp**子系统的关联，在下面的部分， 我们会展示其中的细节。

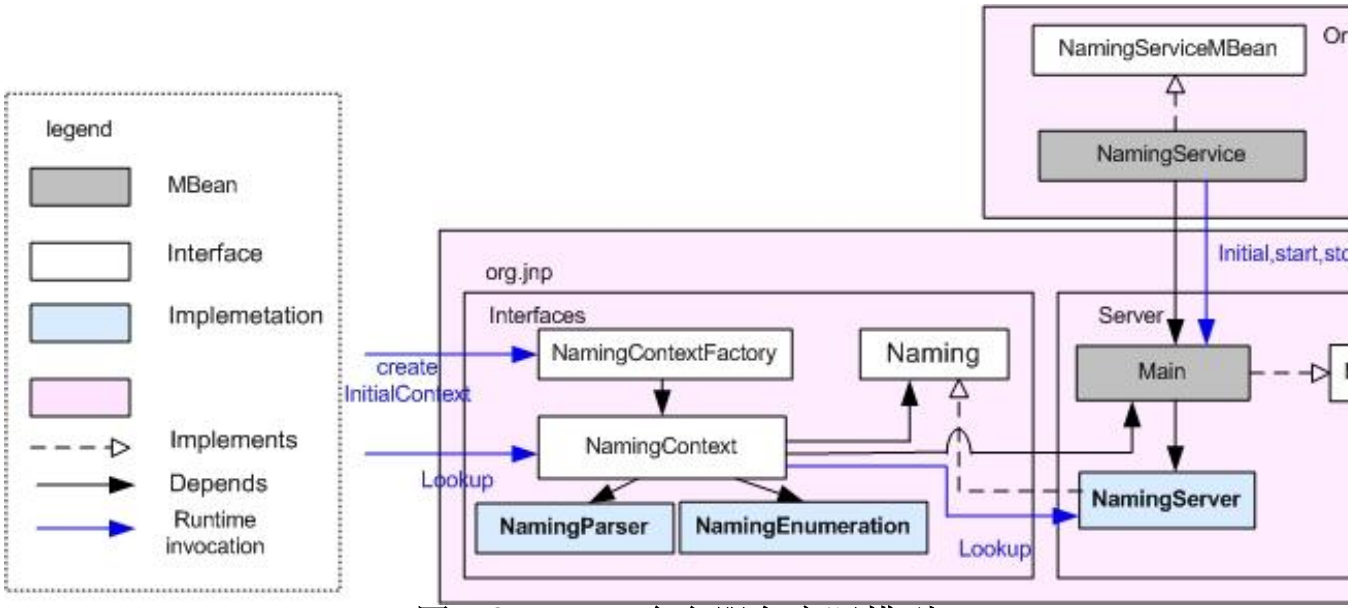

图4-2 JBoss 命名服务实际模型

Org.jboss.naming.NamingService被作为一个**MBean**实现。它提供了

JNDI命名服务，这个NamingService创建了Main MBean并且管理他们的状态。当一个NamingService启动，它将初始化并启动Main MBean。NamingService将相应的功能都为委托给了Main MBean。复制MBean符合我们的概念性模型，可是，在表象后面的是JBoss中的JNDI命名服务被实现成了一个独立的应用。NamingService MBean 通过创建一个新的实例插入Main MBean。这种设计的好处是如果JNDI VM 和JBoss Server VM一致的话，JNDI操作将会通过socket连接以减少相应的开销。

4.2.2 客户端获得EJB 本地对象的例子

为了更好的了解JBoss命名服务架构，我们给出了一个例子并追踪整个调用过程。举个例子，当一个客户端想去调用一个Bean的方法，它不得不先定位Bean Home，因为它是为客户端需求而创建Bean实例的句柄。本地对象的JNDI名字是在部署期中被定义在部署描述文件中的。客户端在运行期中通过使用JNDI名字访问相应的命名服务来获得对象。

1.当JBoss服务器启动，NamingService MBean将被注册并等候调用。
2.NamingService初始化Main MBean并使得命名服务通过侦听相应的Socket端口准备被调用。
3.一个客户端提供命名服务的环境配置。
4.一个客户端创建一个新的IntialContext，它是NamingContextFactory用来创建新的NamingContext的促发器。
5.一个客户端通过提供JNDI名字并在JNDI命名服务中定位相应的本地对象。NamingContext连接相应的命名服务器来进行相应的定位。

4.3 JBossCMP 实际模型

图4-6展示了我们通过Together 5.5 获得得JBossCMP的关联和固有的内容。JBossCMP实际模型非常类似我们的概念性模型. 这个特殊的子系统是JWAS。

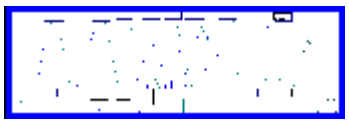

图 4-6 JBossCMP依赖和关联图

4.3.1 特殊组件和联系

Org.jboss.ejb.plugin.jwas 包包括了 JBoss中CMP实体Bean O/R Mapping工具的默认实现。它使用JDBC数据库作为它的持久层存储。EntityPersistenceStore的默认实现是JAWSPersistenceManager。

CMP实体Bean持久层存储的客户化实现了EntityPersistenceStore的接口方法。JAWSPersistenceManage通过调用它的子类中的 execute()方法来实现，举个例子，JDBCActivateEntityCommand通过它的JMPActivateEntityCommand的接口方法， JAWSPersistenceManager基于数据库的存储.如果有其他的存储介质，它需要提供一个实现了JMPXXXCommand接口的包。

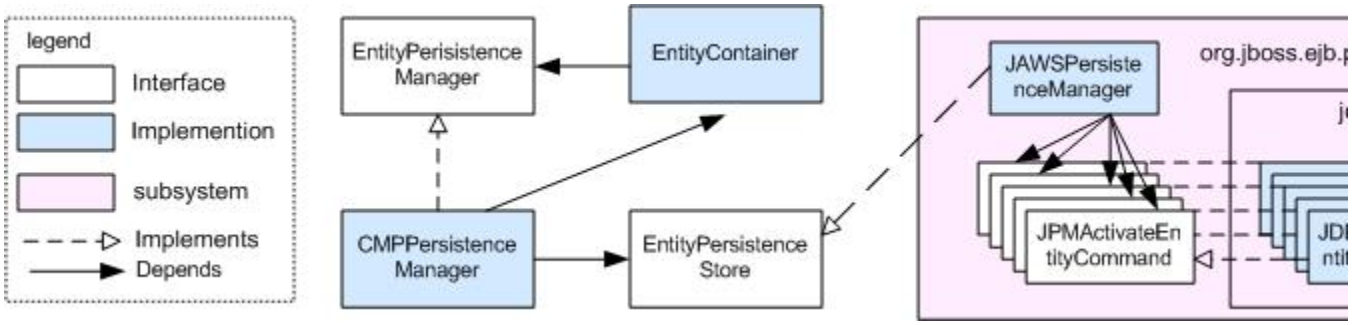

图4-7 JBossCMP 实际模型

表 4-1 显示了EntityPersistenceStore的接口方法，在org.jboss.ejb.plugin.jwas包中的接口被实现了实现了EntityPersistenceStore的JAWSPersistenceManager使用，以及在那些接口中是比较低层次的实现。

EntityPersistenceStore中的方法	接口名称	实现名称
createEntity	JMPCreateEntityCommand	JDBCCreateEntityCommand
findEntity	JMPFindEntityCommand	JDBCFindEntityCommand

findEntities	JMPFindEntitiesCommand	JDBCFindEntitiesCommand
activateEntity	JMPActivateEntityCommand	JDBCActivateEntityComma
loadEntity	JMPLoadEntityCommand	JDBCLaodEntityCommand

loadEntities	JMPLoadEntitesCommand	JDBCLoadEntitesCommand
storeEntity	JMPStoreEntityCommand	JDBCStoreEntityCommand
passivateEntity	JMPPassivateEntityCommand	JDBCPassivateEntityComm

removeEntity	JMPRemoveEntityCommand	JDBCRemoveEntityCommand

4.4 JBoss 交易管理实体模型

图4-8 展示了我们从Together 5.5 中获得的JBossTx的关联和固有信息图。到目前为止，我们并没有从UML图中惊奇的发现 一些特殊的组件和关联，我们再次排列关键组件和关系的关联以及集合以便我们从中得出综合的实际模型。

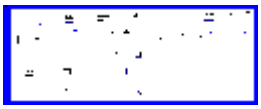

Figure 4-8 JBossTx dependency and inherency diagram

4.4.1 特殊组件和关联

我们关注JBossTx的概念性模型，JBossTx被实现成为MBean通过RMI被发布。 通过 RMI/JRMP的交易内容传播被设计成为两个接口，TransactionPropagationContextImpoter接口，它的实现被用来向交易管理 期导入交易传播内容，TransactionPropagationContext Factory接口，它的实现被用来从客户端获得交易传播内容。当交易管理 器被启动，它的第一个工作是在熟悉的JNDI位置上绑定TransactionManager， TransactionPropagationContext Impoter和TransactionPropagationContextFactory。
TxManager实现了javax.transaction.TransactionManager和其上层的两个接口。它由 TransactionManagerService进行管理。 TxManager依靠TransactionImpl来实现交易操作，类似于声明交易的开始、提交、回滚方法。有趣的是，TransactionImpl只是一个轻量级的前置TxCapsule。TxCapsule是通过TransactionImp的方法调用而被控制的。TxCapsule 控制着关于一个交易的所有信息。在这个类中实现了回调和同步，它通过XidImpl来分辨不同交易。
在会话Bean中通过管理JTA交易Bean调用概念性模型。在这个模型中，

javax.transaction.UserTransaction是必需的。 **JBossTx**由一个子系统实现了**UserTransaction** 接口，位于**org. jboss.tm.usertx**包中。**Usertx** 被分割成为两个子系统，客户端和服务器端，他们通过接口进行交互。
这是一个非常纯的层构架。**ClientUserTransaction**是客户端**UserTransaction**的实现。它将通过**UserTransactionSession**接口委托所有的**UserTransaction**去调用相应的服务器。
UserTransactionSessionImpl在服务器端实现了**UserTransactionSession**接口。这是以便作为不在同一个地点的交易管理的**VM**的远程客户端。当客户端执行的是在同一服务器上的**VM**，**ServerVMClientUserTransaction**是一个客户端的**Usertransaction**实现，它将所有的**UserTransaction**调用委托给了服务器端的**TransactionManager**。和**JBoss**中的大多数服务一样，**UserTransaction**的被作为一个**MBean**来实现和管理。它通过**JNDI**名称 **UserTranaction**被绑定在**JNDI**的相应位置上。

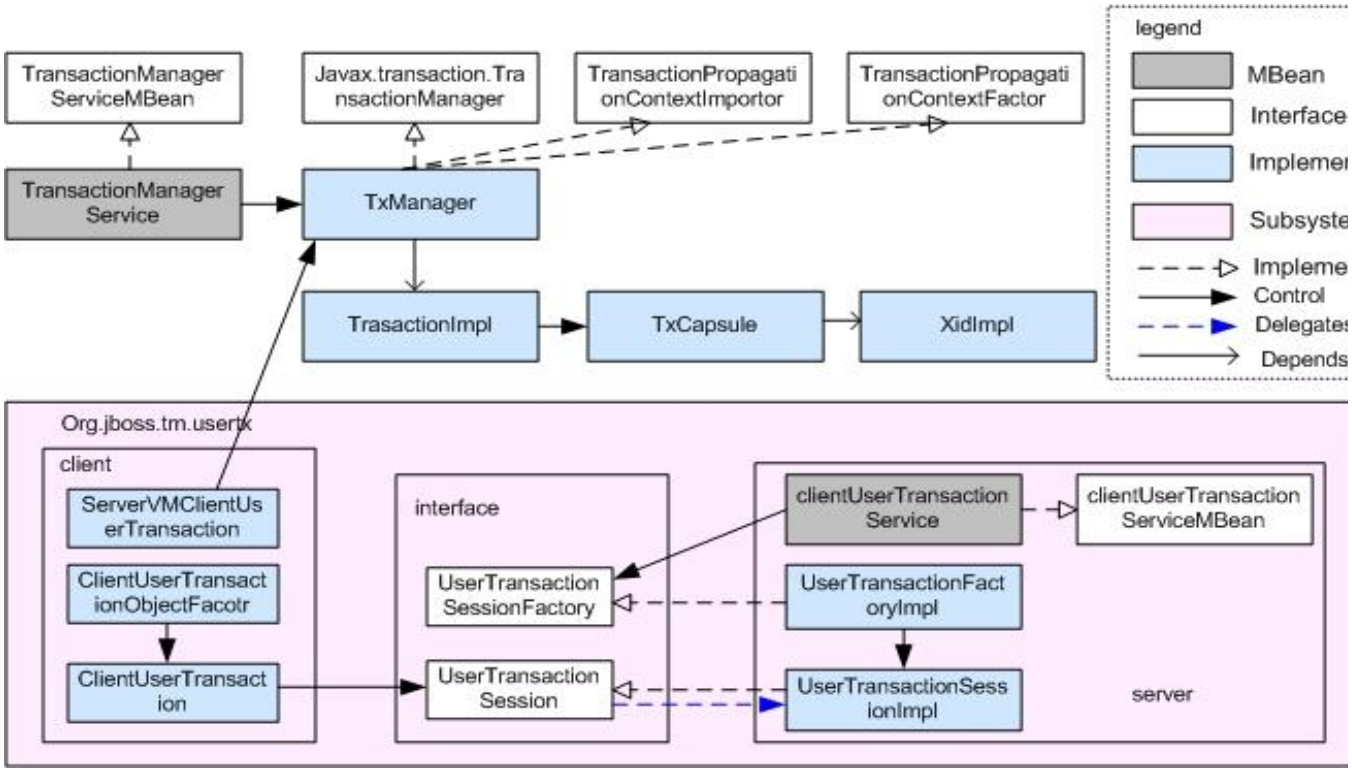

图4-9 JBossTx 实际模型

5.JBoss 架构的可扩展性

通过上面对于**JBoss**架构的讨论，我们可以看出在**JBoss**架构设计中的两

个重要的特性，第一是使用JMX作为一个软件总线垂直的贯穿其所有的服务，通过将新的服务组件遵循JMX规范挂接上"总线"，使得系统扩展现有的服务变得容易。可插入式框架被广泛的运用于服务的实现。开发者可以选择他们需要的服务并编写他们所需要的相应实现，通过定义在部署描述文件中，让JBoss服务器知道。另一个是容器被设计成为动态代理机制，这样使容器的实现变得简单和使开发者避免费劲的将jar文件进行预编译以获得stub和skeleton代码。但是这样做潜在的问题是性能和可测性，因为我们知道java反射机制会引起性能的损失。JBoss中存在着相应的优化方案并且在将来的研究中我们会论述该优化方法在什么时候工作并且是如何工作的。

6．结论

在这篇文章中，我们讨论了JBoss概念性架构模型和实际架构模型。我们通过使用逆工程工具和人工追踪源代码的方法创建了一个综合的实际模型，我们发现实际模型和基于文档的概念性模型有着较大的差异。这就是为什么实际模型是处于实现层面的东西它更接近于 "真实的故事"。实际模型展示了JBoss应用服务器的独特的、特殊的设计。我们尝试着将这个方法提供给J2EE产品的检测中去。不幸的是，源代码并不都是有效的而且逆工程结果也不完全有效的，这将导致误导，我们只能做到JBoss架构模型和概念性模型的比较。

虽然，在这篇文章中我们分析JBoss架构的方法非常的令人兴奋，但这种接触是有限的。我们依然没有获得一些架构对于质量因素影响的结论，比如说性能，因为这取决于运行期内组件和子系统的多样性。从实际模型中我们获得的是静态分析，一个具有可能性的解决方案是分析相应的源代码并在运行期中跟踪其组件并测试它的性能。在这里，概念性架构模型将和软件实现模型相映射，相应的节点将表现软件的功能组件，相应的流程将表现控制流。实际架构模型将被映射成为系统执行模型，它将体现出类似于网络队列的关键计算机资源。因此，这是一个基于架构分析的可行的JBoss性能测试方法。我们的工作是分析JBoss架构以使我们更了解系统。接下来要做的是，通过这次分析来获得JBoss应用的初步的解析性模型，分析代码是可行的测试方法并可测试它的性能。

7．参考资料

1.JBoss Home Page http://www.jboss.org/
2.JBoss Documentation http://www.jboss.org/doco.jsp
3.JBoss Quick Techincal Overview
http://www.ejbean.com/resources/free-open/jboss. html
4.Prof. Richard C. Holt's Home Page
http://plg.uwaterloo.ca/~holt/
5.Ivan T. Bowman, Richard C. Holt and Neil V. Brewster, Linux as a Case Study: Its Extracted Software Architecture, ICSE 99, Los Angeles, May 99.
6.How do I deploy Enterprise JavaBean to JBoss
http://otn.oracle.com/products/jdev/
howtos/appservers/deploy_to_jboss.html
7.The art of EJB deployment
http://www.javaworld.com/javaworld/jw-08-2001/jw- 0803-ejb_p.html
8.Superior app management with JMX
http://www.javaworld.com/javaworld/jw-06- 2001/jw-0608-jmx_p.html
9.DynaServer: System Support for Dynamic Content Web Servers
http://www.cs.rice. edu/CS/Systems/DynaServer/
10.MTE Project http://www.cmis.csiro.au/adsat/mte.htm
11.TogetherSoft Home Page
http://www.togethersoft.de/downloads/down_index.html
12.David Garlan and Mary Shaw, An Introduction to Software Architecture, CMU Software Engineering Institute Technical Report, 1994
13.Loyd G. Williams, Connie U. Smith, Performance Evaluation of Software Architecture, WOSP 98, Santa Fe.N.M
14.Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau,Technical Concepts of Component-based Software Engineering, Technical Report, CMU/SEI-2000-TR-008 ESC-TR-2000-007

---

8. 数据字典

[J2EE] Java 2 Enterprise Edition from Sun MicroSystem. It is a web operationg system

[JMX] the Java Management eXtension (TM) to offer standard interfaces to the management of its components as well as the applications deployed on it.

[JAWS] Just Another Web Storage

[JCA] The J2EE Connector Architecture

[JNDI] Java Naming and Directory Interface

[RMI] Remote Method Invocation. It is Sun's object request broker(ORB).

[JRMP] Java Remote Method Protocal.

[JTA] Java Transaction API.

Appendix-1

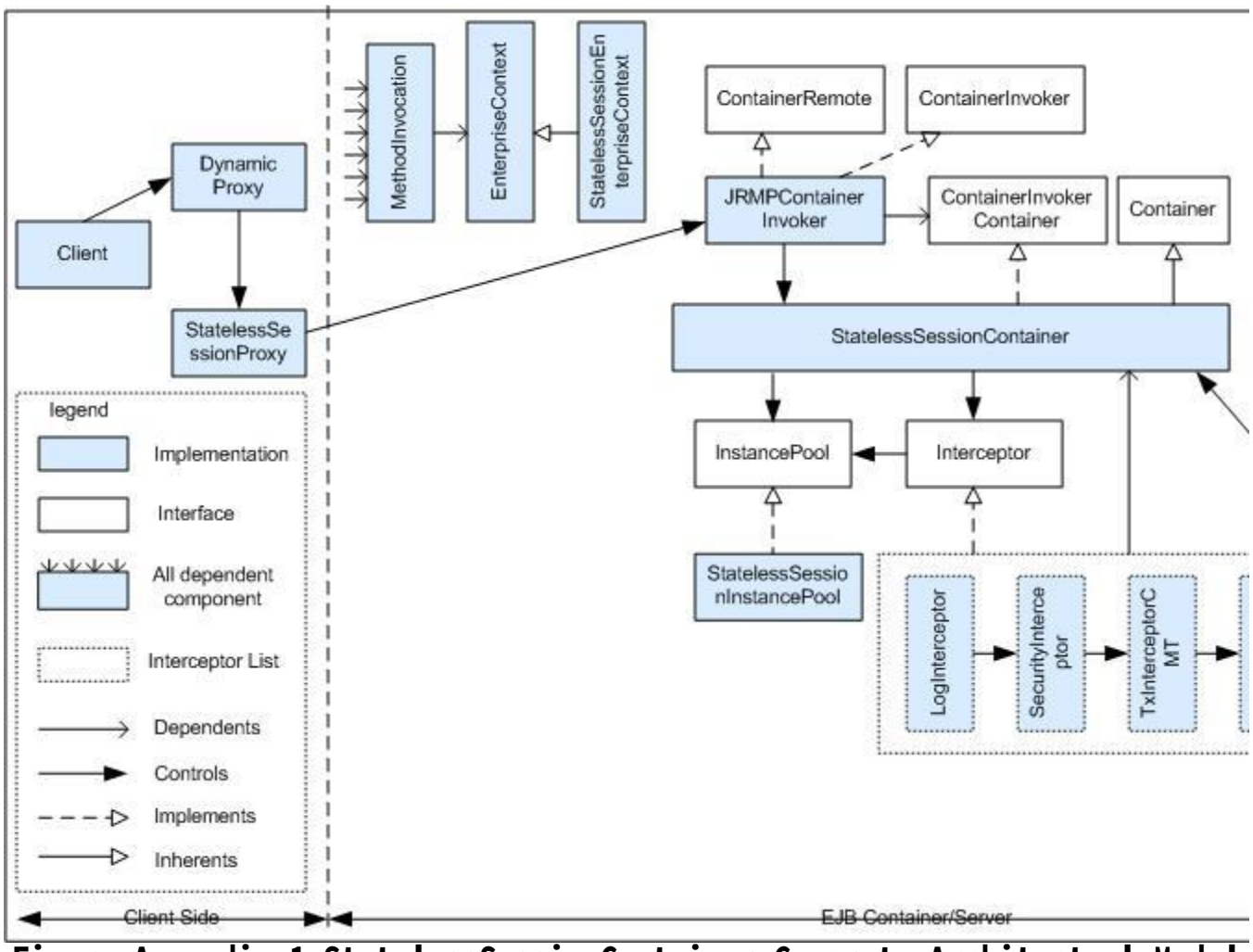

Figure Appendix-1 StatelessSessionContainer Concrete Architectual Model

Appendix-2

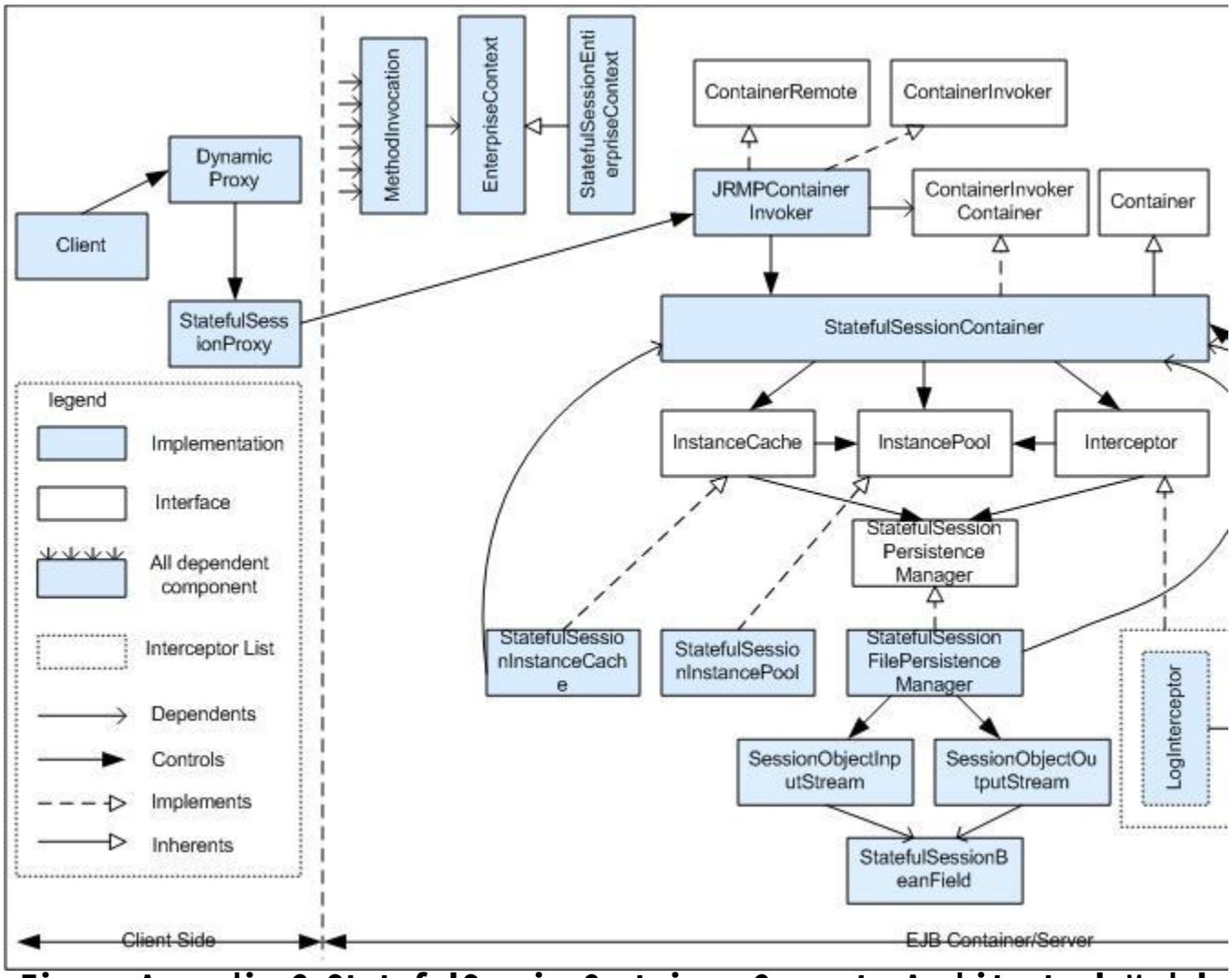

Figure Appendix-2 StatefulSessionContainer Concrete Architectual Model

Appendix-3

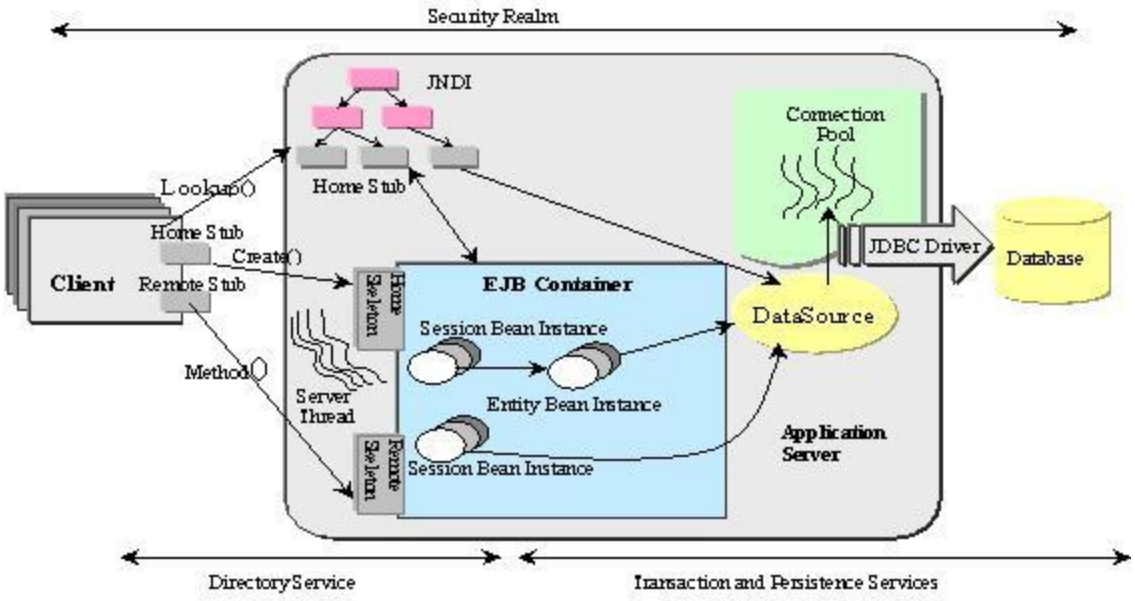

Figure Appendix-3 A COTS EJB Container Conceptual Architecture Model

---

Submitted on April 29,2002. This work is done in 15 working days. You are more than welcome to write to the author about this work. Any of your comment is greatly apprieciated.
