支持 EJB 动态分布的组件迁移模型与算法*

范国闯+, 魏 峻, 钟 华, 冯玉琳

(中国科学院 软件研究所 软件工程技术中心,北京 100080)

Migration Model and Algorithms for Dynamic Redistribution of Enterprise Java Bean Components

FAN Guo-Chuang+, WEI Jun, ZHONG Hua, FENG Yu-Lin

(Technology Center of Software Engineering, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)

+ Corresponding author: Phn: +86-10-62630989, E-mail: fanguo@otcaix.iscas.ac.cn, http://otcaix.iscas.ac.cn

Received 2003-08-21; Accepted 2003-12-08

Fan GC, Wei J, Zhong H, Feng YL. Migration model and algorithms for dynamic redistribution of enterprise Java bean components. *Journal of Software*, 2004,15(3):404~413.
http://www.jos.org.cn/1000-9825/15/404.htm

Abstract: Web application servers (WASs) provide a web computing infrastructure for distributed components. The component structure of statically configured distribution prevents web applications from being adaptive to the changing environmental conditions at runtime. To meet the requirement of dynamic redistribution, WASs should provide the capability to support component migration. The most challenging problem is to maintain component consistency during such a component migration. To resolve this inconsistency problem, some kinds of component migration constrains (CMC) are defined in this paper. A component migration model for J2EE (Java 2 platform enterprise edition) application servers is proposed, and SLB_Copy, SFB_Copy and EB_Copy component migration algorithms are presented. It is proved that SLB_Copy, SFB_Copy and EB_Copy migration algorithms all satisfy the CMC constrains. At present, the migration model is implemented in a J2EE application server, referred to as WebFrame 2.0, and these algorithms are applied to provide numerous services such as the adaptive load balancing service and the failover service.

Key words: Web application server; component migration model; component migration algorithm; component migration constrains; dynamic redistribution

摘 要: Web 应用服务器是 Web 计算环境下的新型中间件,为基于组件的分布式 Web 应用提供了基础运行平台.组件静态分布限制了事务性 Web 应用在运行期间适应执行环境变化的能力.为了满足 Web 应用的动态分布需

* Supported by the National High-Tech Research and Development Plan of China under Grant Nos.2001AA113010, 2001AA414020, 2001AA414310 (国家高技术研究发展计划(863)); the National Grand Fundamental Research 973 Program of China under Grant No.2002CB312005 (国家重点基础研究发展规划(973))

作者简介: 范国闯(1974－),男,湖南娄底人,博士生,助理研究员,主要研究领域为网络分布计算,软件工程技术;**魏峻**(1970－),男,博士,副研究员,主要研究领域为网络分布式计算,软件中间件技术,软件形式化描述与验证方法,面向组件和面向 Agent 的软件工程;**钟华**(1971－),男,博士,副研究员,主要研究领域为网络分布计算,软件工程技术;**冯玉琳**(1942－),男,博士,研究员,主要研究领域为软件工程,形式化方法,分布对象计算.

求,Web 应用服务器需在底层为组件提供一种动态迁移的能力.如何维持组件迁移前后的一致性是组件迁移中最棘手的问题之一.为解决此问题,定义了组件迁移一致性约束 CMC(component migration constrains),并给出了在 J2EE(Java 2 platform enterprise edition)应用服务器中支持 EJB(enterprise Java Bean)动态分布的组件迁移模型和 SLB_Copy,SFB_Copy,EB_Copy 3 个迁移算法.分析得出 SLB_Copy,SFB_Copy 和 EB_Copy 均满足 CMC 约束.迁移模型和算法已在自主研制的 Web 应用服务器 WebFrame2.0 中实现,并已应用到自适应负载平衡、失效恢复等多个方面.

关键词: Web 应用服务器;组件迁移模型;组件迁移算法;组件迁移约束;动态分布

中图法分类号: TP311 **文献标识码**: A

Web 应用服务器是 Web 计算环境下的新型中间件,为创建、部署、运行、集成和管理 Web 应用提供一系列运行时服务(如消息、事务、安全、应用集成等)[1],被认为是自关系型数据库以来最令人激动的企业应用技术,在工业界已得到广泛应用.将这些 Web 应用分布在多个服务器,构建分布式 Web 应用是一种可靠性和可用性强且有效的解决办法.通过一种透明的组件间远程通信机制,大多数 Web 应用服务器均支持基于组件(如 CORBA CCM,COM/DCOM 和 EJB(enterprise Java bean))的分布式 Web 应用.但由于组件的位置是在 Web 应用运行前指定的,一旦绑定后,组件就不能移动,从而不能对分布的结构进行调整[2],即 Web 应用的这种分布支持是静态的.

静态分布限制了 Web 应用在运行期间适应其执行环境变化的能力,不能满足 Web 应用在任意运行时刻都能对组件进行重新布局的要求:(1) 为了满足 Web 计算环境下大规模用户的并发访问,Web 应用组件需要根据系统运行状态信息,如 CPU 使用率、内存使用率、连接数等进行负载分配的决策,对系统负载的变化具有动态的调整能力.在运行期间,组件从一个负载重的 Web 应用服务器动态迁移到负载轻的应用服务器,从而提高了系统整体性能和吞吐量[3].(2) 为了支持灵活性、可扩展性等特性,Web 应用服务器需提供在运行期间可动态重配置(reconfigurable)的能力[4~6].重配造成的变化可能促使 Web 应用的结构进行重建(restructuring),如将两个处理节点合并或分离、集成独立的分布组件或引入组件复件(replica)等.(3) 有时需要关闭计算机,定期对 Web 应用服务器进行维护或临时将 Web 应用服务器作为它用.在这种情况下,如果能将组件迁移到另外一台计算机,保持其原有的执行状态.当原服务器可用时,再将组件迁回,从而提高组件的可用性,改进系统管理.(4) 在移动计算环境下,需将组件迁移到被访问数据所在的服务器,即数据访问本地化,降低网络访问的开销[7,8].

为了满足 Web 应用的上述动态分布需求,Web 应用服务器需提供一种组件迁移的能力.组件迁移问题可以细分为 3 个子问题:1) 何时迁移;2) 如何选择被迁移组件与迁移目的地;3) 如何迁移.本文结合自主研发 Web 应用服务器 WebFrame 的实际需求,重点解决如何在 J2EE(Java 2 platform enterprise edition)应用服务器中进行 EJB 组件迁移.

1 EJB 组件迁移模型

EJB 组件模型是 J2EE 规范中最重要的一部分[9],定义了多层 Client/Server 结构下用 Java 语言编写服务端分布组件的规范.如图 1 所示,一个 J2EE 应用服务器由多个 EJB 服务器构成,每个 EJB 服务器上运行多个 EJB 组件容器,每个组件容器为多个 EJB 组件实例提供执行环境,提供事务、安全、RMI 等基本服务.根据使用目的不同,EJB 组件可以分为无状态会话组件(stateless session EJB)、有状态会话组件(stateful session EJB)、实体组件

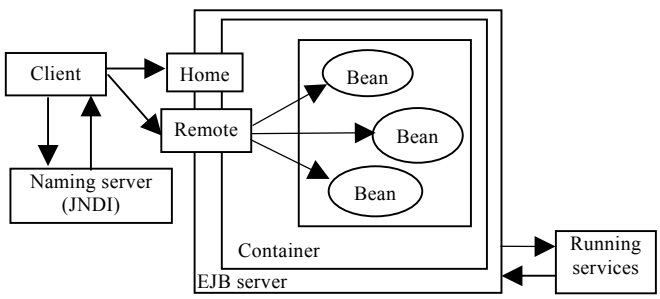

Fig.1 EJB component model

图 1 EJB 组件模型

(entity EJB)和消息驱动组件(message-driven EJB).

考虑到组件的执行依赖于 EJB 服务器,我们将组件迁移定义为组件实例在运行期间从一个服务器进程迁移到另一个服务器进程的行为.EJB 组件迁移模型如图 2 所示,首先抽取组件状态数据,然后将状态数据传输到目标进程,最后在目标进程上创建组件实例,恢复其状态.组件状态数据包括代码数据、会话状态数据和执行状态数据.组件代码数据是指,组件所对应的二进制代码、组件的接口、类型信息和结构信息(如组件的元数据等).会话状态数据包含组件的成员变量、属性、Java 对象引用等.组件执行状态数据包括组件局部变量值、函数参数值、线程状态、方法调用堆栈等.从移动范型来讲,组件迁移也可以分为强迁移和弱迁移.强迁移是指包含代码数据、会话状态数据和执行状态数据的迁移,弱迁移是指包含代码数据、会话状态数据的迁移.强迁移开销大,比较复杂.此外,由于受 Java 虚拟机的限制,实现难度更大.而弱迁移其移动开销小,执行效率高,虽然改变了迁移后的执行语义,但能满足 J2EE1.3 规范中会话状态的要求.鉴于此,本文主要研究 EJB 组件的弱迁移.为了较严格地描述 EJB 组件迁移,我们给出如下约定和定义:

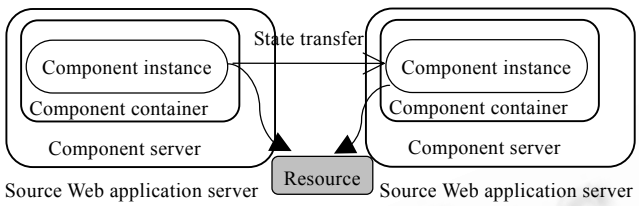

Fig.2 EJB component migration model

图 2 EJB 组件迁移模型

约定. 考虑到消息驱动组件是一种无状态组件,所以将消息驱动组件和无状态会话组件统称为无状态 EJB 组件,记为 SLB.有状态会话组件记为 SFB,实体组件记为 EB.

定义 1. 组件服务器进程集合 $P=\{P_i\}$,其中 $i=1,\ldots,N$,N 为服务器进程数目.

定义 2. 资源定义为 *Resource*=(*id*,*object*,*url*,*crbtype*,*rmbtype*).其中 *id* 为资源的惟一标识符;*object* 为运行时间环境中的资源对象;*crbtype* 为组件与资源绑定类型,*crbtype*∈{*BYID*,*BYVAL*,*BYTYPE*},其中 *BYID* 表示引用标识绑定,*BYVAL* 表示值绑定,*BYTYPE* 表示类型绑定;*rmbtype* 为资源与机器的绑定类型,*rmbtype*∈{*UNATTACHED*,*FASTENED*,*FIXED*},其中 *UNATTACHED* 表示独立资源,*FASTENED* 表示捆绑类型,*FIXED* 表示固定资源.

定义 3. 组件实例定义为 $C=(name,id,I,SD,SD_0,location,R,t,state)$,其中,*name* 为 EJB 组件实例的 JNDI 名字,由字符串构成,客户通过该名字获得组件的引用;*id* 为 EJB 组件的惟一标识,由 EJB 组件容器创建组件实例时指定;*I* 为组件对外的接口集合;*SD* 为组件状态数据集合,$SD=CS\cup SS\cup ES$,其中 *CS* 为代码数据集,*SS* 为会话状态数据,*ES* 为组件执行状态数据;SD_0 为组件初始状态;*location* 表示所在位置,*location*∈*L* 为组件位置集合,根据组件迁移的定义,*L*=*P*;*R* 为组件所引用的资源集合;*t* 为 EJB 组件实例的类型,*t*∈*T*,*T* 为 EJB 组件类型集合 *T*={*SLB*,*SFB*,*EB*};*state* 表示组件实例的迁移状态,*state*∈*S*={*Active*,*Passive*,*Frozen*},其中 *Active* 表示组件实例处于正常执行状态,*Passive* 表示组件实例处于钝化状态,,*Frozen* 表示组件不接受也不处理任何客户请求只接受但不处理客户请求.

定义 4. 组件容器定义为 *Container*=(*name*,*C*,*Services*,*Context*,*Queue*),其中 *name* 为组件容器的名字;*C* 为运行在该组件容器内的组件实例集合,$C=\{C_i|C_i.name=Container.name\}$,$i=1,\ldots,N$,$N$ 为组件实例的数目;*Services* 是容器为组件实例提供的运行时服务集合,如消息、事务、安全、日志,负载平衡服务等;*Context* 是组件实例运行时的上下文环境;*Queue*=*WaitingQueue*∪*BusyQueue*,其中 *WaitingQueue* 表示等待处理的客户请求队列,*BusyQueue* 表示为正在处理的请求队列.

定义 5. 在 Web 应用服务器中,存在两种关系 RPC 和 RCC,其中 $RPC=\{\langle P_i,Containerj\rangle|P_i\in P,Containerj\in Container\}$,*Containerj* 运行在 P_i 进程空间中},表示进程和组件容器之间的关系;$RCC=\{\langle Containeri,Cj\rangle|Containeri\in Container,Cj\in C\}$,*Containeri* 为 *Cj* 提供驻留环境和服务},表示组件容器和组件实例之间的关系.

定义 6. 迁移模式集合 *MODE*={*NO_MIGRATION*,*WITH_RESOURCE*,*WITHOUT_RESOURCE*},其中 *NO_MIGRATION* 表示组件实例不可迁移,*WITH_RESOURCE* 表示迁移组件实例时需迁移所引用的资源,*WITHOUT_RESOURCE* 表示组件实例可以迁移,但其所引用的资源不可迁移.

定义 7. 组件实例的迁移为映射 $migrate$:$C\times L\times MODE\to C$,满足:

(1) $\forall C_{src}\in C,L_{dest}\in L$,则 $migrate(C_{src},L_{dest},NO\_MIGRATION)=C_{src}$;

(2) $\forall C_{src}\in C,L_{dest}\in L,mode\in MODE$,若 $L_{dest}\neq C_{src}.location$ 且 $mode\neq NO\_MIGRATION$,则 $migrate(C_{src},L_{dest},mode)=Cdest$,其中 $C_{dest.}location=L_{dest}$ 且 $C_{dest}.S=C_{src}.S$;

(3) $\forall C_{src}\in C,L_{dest}\in L,mode\in MODE$,若 $L_{dest}=C_{src}.location$ 则 $migrate(C_{src},L_{dest},mode)=C_{src}$.

定义 8. 如果组件实例 c 正在处理的队列 $BusyQueue_c$ 为空,则称 c 是迁移安全的,记为 $safe(c)$.

定义 9. 如果队列 queue 的处理线程处于挂起状态(suspended),并且任何队列操作都失效,对客户的请求都停止响应,则称队列 queue 是被钝化的,记为 $suspend(queue)$.

定义 10. passivate(ejbinstance)表示组件实例 ejbinstance 处于 passive 状态,如果$\forall m\in BusyQueue_{ejbinstance}$满足 $result(m)\wedge safe(ejbinstance)$,其中 $result$ 表示消息是否被处理并返回结果.

定义 11. frozen(ejbinstance)表示组件实例 ejbinstance 处于 frozen 状态,如果

$$passivate(ejbinstance)\wedge suspend(waitingQueue_{ejbinstance}).$$

2 组件迁移操作原语与约束

组件迁移的操作原语有两部分,一部分是 J2EE 规范中一系列标准的 EJB 操作原语如 Create,Remove 等;另一部分是根据组件迁移的语义.操作原语包括 ejbCreate,createContainer,transferResource,writeState,commitState,transferState,loadState,readState,passivateInstance,quiescenceInstance,activateInstance,transferRequest,ejbRebind,ejbRemove 和 updateAllReferences 等.其中 writeState 表示将 EJB 实例的会话状态保存到序列化文件中,而 commitState 表示将 EJB 实例的状态提交到数据库中;passivateInstance 使组件进入 Passive 状态,而 quiescenceInstance 使组件进入 Frozen 状态.其他操作原语其含义如其英文所示,如 ejbCreate 表示创建 EJB 组件实例.

EJB 组件有 3 种迁移状态:Active,Passive 和 Frozen.passivateInstance,quiescence Instance 和 activateInstance 等原语可改变组件的迁移状态,其状态转换关系如图 3 所示.

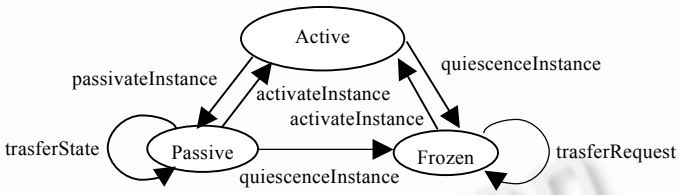

Fig.3 The state diagram of EJB migration

图 3 EJB 组件迁移状态图

组件迁移为 Web 应用提供了一种动态分布的能力,Web 应用服务器需管理这种动态变化,最小化对其他正在运行的组件的影响,尽可能维持迁移前后组件的一致性.有 3 个基本问题会导致迁移前后的不一致.下面以图 4 和图 5 为例加以说明.

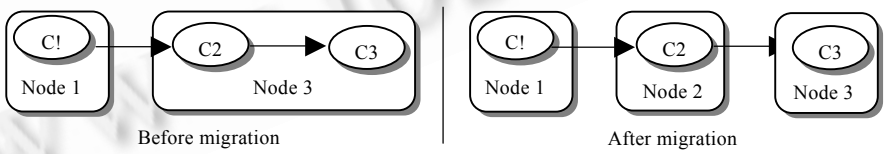

Fig.4 Component migration illustration

图 4 组件迁移示意图

(1) 引用一致性问题.如图 4 所示,组件 C1 引用位于 Node 3 上的组件 C2,但当组件 C2 从 Node 3 迁移到 Node 2 后,如果没有及时更新 C2 的引用,则对组件 C2 的引用将会失效,从而导致它们之间不能进行通信,C1 就无法访问组件 C2.

(2) 状态一致性问题.在组件 C2 从 Node 3 迁移到 Node 2 的过程中,如果组件 C2 继续处理客户的请求,那么组件 C2 在 Node 3 上的状态可能发生变化,从而导致 C2 在 Node 2 和 Node 3 上迁移前后状态不一致,组件 C2 在 Node 2 上继续响应客户请求的执行结果可能与在 Node 3 上继续执行的结果不一致.

(3) 客户请求丢失问题.如图 5 所示,在组件 C2 从 Node3 迁移到 Node2 的过程中,客户可能正在向组件 C2

传送请求消息 m3,但由于 C2 已迁移到另外的节点上,C2 可能会丢失该消息.

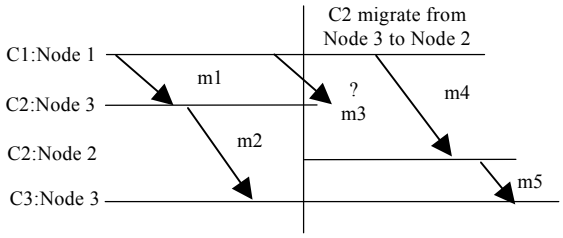

Fig.5 Illustration of lost invocation messages

图 5 客户请求丢失示意图

针对上述一致性问题,下面给出组件迁移约束:

定义 12. 引用一致性约束(reference constraint)为

$$RC\equiv\forall r\in Reference_{Csrc}:$$

$$(rebind(C_{src})\Rightarrow referenceUpdated(r)),$$

其中 $Reference_{C_{src}}$ 为 C_{src} 的所有引用集合,*rebind* 和 *referenceUpdated* 都是谓词,分别表示绑定和引用更新是否成功.

定义 13. 设 $t1$ 为迁移开始时刻,$t2$ 为迁移结束时刻,$t1<t2$.状态一致性约束(state constraint)为

$$SC\equiv migrated(C_{src}, SD_{C_{src}}^{t1})\Rightarrow SD_{C_{src}}^{t1}=SD_{C_{dest}}^{t2},$$

其中 $SD_{C_i}^t$ 表示组件实例 C_i 在 t 时刻的状态数据集,且 $SD_{C_i}^t\subseteq(CS_{C_i}^t\cup SS_{C_i}^t)$,*migrated* 谓词表示组件实例是否迁移成功.

定义 14. 消息一致性约束(message constraint)为

$$MC\equiv remove(C_{src})\Rightarrow(\forall m\in queue_{C_{src}}, result(m)),$$

其中 $queue_{C_i}$ 表示组件 C_i 请求队列中消息的集合,谓词 *result* 表示消息是否被处理并返回结果.

定义 15. 组件迁移一致性约束(compont migration constraint)为

$$CMC\equiv RC\wedge SC\wedge MC.$$

为了满足上述迁移一致性约束 *CMC*,一些迁移操作需满足某前提条件(precondition)和后置条件(posteffect).下面给出一些操作及其前、后置条件的描述.

(1) writeState(ejbinstance)

precondition ≡ *passivate*(*ejbinstance*);

posteffect ≡ *stateCommited*(*ejbinstance.name*);

(2) transferState(ejbinstance,target)

precondition ≡ *passivate*(*ejbinstance*)∧*stateCommited*(*ejbinstance.name*);

(3) loadState(ejbname,ejbinstance)

precondition ≡ *stateCommited*(*ejbinstance.name*)∧*passivate*(*ejbinstance*);

(4) transferRequest(srcinstance,targetinstance)

precondition ≡ *frozen*(*srcinstance*)∧*ejbExist*(*targetinstance*);

posteffect ≡ $(\forall m\in queue_{C_{src}}, result(m))$;

(5) ejbRebind(ejbname,oldinstance,newinstance)

precondition ≡ $pre_{transferRequest(oldinstance,newinstance)}\wedge(\forall r\in Reference_{oldinstance}\ referenceupdated(r))$;

(6) ejbRemove(ejbinstance)

precondition ≡ $rebind(ejbinstance)\wedge(\forall m\in queue(C_{src}), result(m))$;

(7) updateAllRefernce(ejbinstance)

posteffect ≡ $(\forall r\in Reference_{ejbinstance}: referenceUpdated(r))$.

3 组件迁移算法

文献[10,11]介绍了进程迁移方面所做的工作,研究人员提出了很多算法.虽然部分进程迁移技术可以应用到组件迁移,但进程迁移策略和算法在组件迁移中不一定可行.与进程迁移相比,组件迁移具有一些其自身的特点:1) 组件迁移的粒度小,不需要将整个进程状态信息(如进程控制信息、地址空间等)进行迁移;2) 组件迁移的目标有多种类型,分为无状态和有状态两种,而有状态的又分为会话状态和永久状态两种.由于不同类型的组件对状态的要求不一样,所以不同的组件类型应该有不同的迁移算法;3) 由于组件的执行需要组件容器,所以组

件迁移到目标进程之前,组件容器和其所提供的服务必须在运行期间动态地启动.针对各类型组件的特点,下面分别给出满足上述迁移一致性约束 CMC 的 3 个迁移算法:无状态组件迁移算法(SLB_Copy)、有状态组件迁移算法(SFB_Copy)和实体组件迁移算法(EB_Copy).

3.1 无状态组件迁移算法(SLB_Copy)

Stateless Session Bean 和 Message Driven Bean 都是无状态的 EJB 组件,在方法调用间不保存任何客户的状态,不需要迁移任何会话状态,从而不存在迁移前后状态的一致性.其迁移算法描述见表 1.

由于不需要迁移组件实例的会话状态,SLB_Copy 算法实现起来就相对简单,不需要在迁移安全的前提条件下就可迁移,进入迁移状态后,首先迁移组件所需的二进制软件包,然后根据定义 7 的方式迁移应用的资源,并在目标进程内创建组件实例所需的组件容器,启动组件容器所需的底层服务如事务、安全等.然后才创建迁移后的组件实例 C_{target},并激活 C_{target}.但考虑到客户对 C_{src} 的已有的引用,所以在删除组件实例 C_{src} 前需将客户对它的引用更新.为此,首先让 C_{src} 直接进入 Frozen 状态,然后停止对所有客户的响应,将等待队列中的请求全部转发到 C_{target}.当 C_{src} 进入迁移完毕后,才更新所有的客户引用,并将 JNDI 名重新绑定 C_{target},删除 C_{src},这样,整个迁移过程结束.值得注意的是,SLB_Copy 算法在转发请求前才进入 Frozen 状态,在迁移软件包、资源和创建 C_{target} 时还可以响应客户请求.而 C_{target} 创建后,进入 Active 状态,就可以直接响应客户请求.

Table 1 SLB_Copy Algorithm

表 1 SLB_Copy 算法

```
SLB_Copy (Csrc, Ptarget, mode) {
    if (mode == NO_MIGRATION) return;
    if (Csrc.location == Ptarget ) return;
     try{
          transferEJBJar(Csrc,Ptarget);
          if (mode ==WITH_RESOURCE){
              resources= Csrc.getAllRefResources();
                 forall res in resources{
                        transferResource(Csrc,res,Ptarget);
                 }
              }
          loadService(services(Csrc),Ptarget);
          createContainer (Csrc.name,Ptarget);
          Ctarget =ejbCreate(Csrc.name,Ptarget);
          activateInstance(Ctarget);
          quiescenceInstance(Csrc):
          transferRequest(Csrc,Ctarget);
          updateAllReferences(Csrc):
          ejbRebind(Csrc.name,Csrc,Ctarget):
          Remove(Csrc);
      }catch(Exception e){
              MigrationRollBack();
      }
}
```

3.2 有状态组件迁移算法(SFB_Copy)

SFB_Copy 算法迁移的对象是有状态会话组件(stateful session bean).该类型组件在同一会话内的方法调用间保持会话状态,但不是对所有客户持久保存其状态,所以 SFB_Copy 算法需要维持组件迁移前后在同一会话内状态的一致性.其迁移算法描述见表 2.

不同于 SLB_Copy 算法,由于 SFB_Copy 算法需要维持迁移前后状态的一致性,在序列化会话状态前,使组件 C_{src} 进入迁移安全状态后才迁移状态,从而确保在组件会话状态迁移期间,不会有客户改变组件 C_{src} 的状态,实现组件 C_{src} 迁移前后状态的一致.最后停止对客户的响应,将组件 C_{src} 上的客户请求透明地转发到 C_{target} 后,才使 C_{target} 进入 Active 状态,响应同一会话中客户的请求,保证了客户请求的处理顺序.同样,与 SLB_Copy 算法一样,也可以通过预先启动服务,创建组件容器,生成组件复件等方式提高算法的效率.

3.3 实体组件迁移算法(EB_Copy)

EntityBean 状态具有持久性,对所有客户均有效,其状态一般都保存在与之连接的数据库中.与 SFB_Copy 算法不同,EB_Copy 算法可以通过数据库的事务特性维持迁移前后状态的一致性.其迁移算法描述见表 3.EB_Copy 算法类似于 SLB_Copy 算法,但区别是不需要传输 C_{src} 的状态,而是直接将状态提交到数据库中,其状态的一致性由数据库事务维持.创建 C_{target} 后从数据库中将状态加载即可获得组件的状态数据.

Table 2 SFB_Copy algorithm
表 2 SFB_Copy 算法

```
SFB_Copy(C_src,P_target,mode){
    if (mode = = NO_MIGRATION) return;
    if (C_src.location= = P_target ) return;
      try{
        transferEJBJar(C_src,P_target);
        passivateInstance(C_src);
        if (mode ==WITH_RESOURCE){
              resources=C_src.getAllRefResources();
              forall res in resources{
                  transferResource(C_src,res,P_target);
               }
        }
        loadService(services(C_src),P_target);
        createContainer (C_src.name,P_target);
        C_target =ejbCreate(C_src.name,P_target);
        writeState(C_src);
        TransferState(C_src.name,P_target);
        readState(C_src.name,C_target);
        quiescenceInstance(C_src):
        transferRequest (C_src,C_target);
        activateInstance(C_target);
        updateAllReferences(C_src):
        ejbRebind(C_src.name,C_src,C_target):
        Remove(C_src);
    }catch (Exception e){
         MigrationRollBack();
    }
}
```

Table 3 EB_Copy algorithm
表 3 EB_Copy 算法

```
EB_Copy(C_src,P_target,mode){
    if (mode = = NO_MIGRATION) return;
    if (C_src.location= = P_target ) return;
    try{
        transferEJBJar(C_src,P_target);
        passivateInstance(C_src);
        if (mode = =WITH_RESOURCE){
            resources= C_src.getAllRefResources();
            forall res in resources{
                transferResource(C_src,res,P_target) ;
            }
        }
        loadService(services(C_src),P_target);
        createContainer (C_src.name,P_target);
        C_target =ejbCreate(C_src.name,P_target);
        //通过事务控制实体组件的状态
        CommitState(C_src);
        loadState(C_src.name,C_target );
        quiescenceInstance(C_src):
        transferRequest (C_src,C_target);
        activateInstance(C_target);
        updateAllReferences(C_src):
        ejbRebind(Csrc.name,C_src,C_target):
        Remove(C_src);
    }
    catch (Exception e){
         MigrationRollBack();
    }
}
```

4 算法分析与实现

4.1 算法一致性约束分析

下面简要分析 SLB_Copy,SFB_Copy 和 EB_Copy 所满足的特性.为了描述方便,操作 Operation 的前提条件记为 $pre_{operation}$,后置条件 posteffect 记为 $post_{operation}$.

性质 1. SLB 组件在任何时候都满足状态一致性约束 SC.

根据 J2EE1.3 规范的定义,无状态组件 SLB 在方法调用间不保存任何客户的状态.设 $t1$ 为迁移开始时刻,$t2$ 为迁移结束时刻,则对于任意 SLB 组件实例 C_{src},满足:

$$SD_{C_{src}}^{t1} = SD_{C_{src}}^{t2} \tag{1}$$

由于 SLB 并没有进行状态迁移,所以

$$SD_{C_{src}}^{t2} = SD_{C_{dest}}^{t2} \tag{2}$$

结合式(1)和式(2),得出

$$SD_{C_{src}}^{t1} = SD_{C_{dest}}^{t2} .$$

所以,$SC \equiv$true,即满足状态一致性约束 SC,即 SLB 组件在任何时候都满足状态一致性约束 SC.

性质 2. SLB_Copy,SFB_Copy 和 EB_Copy 均满足组件迁移一致性约束 CMC.

SLB_Copy 算法在重新绑定前,调用了 UpdateAllReferences 操作,根据 UpdateAllReferences 的后置条件,得出

$$\forall r \in Reference_{C_{src}}, referenceUpdated(r) \tag{3}$$

即满足引用一致性约束(RC).

SLB_Copy 算法在 Remove 前,调用了 transferRequest 操作,而 quiescenceInstance 使 transferRequest 的前提条件得到满足.根据 transferRequest 的后置条件,得出

$$\forall m \in queue_{C_{src}}, result(m) \tag{4}$$

即满足消息一致性约束(MC).

此外,由性质 1,状态一致性约束(SC)满足

$$SC \equiv \text{true} \tag{5}$$

由式(3)~式(5)得出满足组件迁移一致性约束(CMC).

所以,SLB_Copy 算法满足组件迁移一致性约束.

同理,SFB_Copy 和 EB_Copy 也满足引用一致性和消息一致性约束,但与 SLB_Copy 不同,SFB_Copy 和 EB_Copy 算法迁移的对象是有状态的 EJB 组件.

下面以 SFB_Copy 为例,分析算法的状态一致性.设 $t1$ 为状态迁移开始时刻,$t2$ 为状态迁移结束时刻.

SFB_Copy 算法调用了 passivateInstance 操作,使得 C_{src} 完成 BusyQueue 中的所有请求后,不再处理任何请求,只将请求放在 WaitingQueue 中,C_{src} 进入 Passive 状态,所以

$$passivate(C_{src}) = \text{true}.$$

调用 writeState 操作后,transferState 的前期条件成立,即 $pre_{\text{transferState(ejbinstance,target)}} = \text{true}$.

由于在状态迁移过程中,C_{src} 一直处于 Passive 状态,所以

$$SD_{C_{src}}^{t1} = SD_{C_{src}}^{t2} \tag{6}$$

由于 C_{dest} 装载了 C_{src} 在 $t2$ 时刻的状态,所以

$$SD_{C_{src}}^{t2} = SD_{C_{dest}}^{t2} \tag{7}$$

由式(6)和式(7)得,$SD_{C_{src}}^{t1} = SD_{C_{dest}}^{t2}$,即满足状态一致性约束.

同理,EB_Copy 算法亦满足状态一致性约束满足.

4.2 算法效率分析

SLB_Copy,SFB_Copy 和 EB_Copy 这 3 个算法的效率主要由二进制软件包、引用资源、服务的启动、容器的创建、组件的部署以及客户请求的转发几个方面决定.客户请求的转发其效率主要受系统当时在队列中等待的请求数影响,是无法预见的,但相对于前 5 个方面,影响较小,而状态数据的迁移、服务的启动、容器的创建、组件的部署等占用的时间开销较大,影响了算法的效率.实际上,可以将创建组件实例所需的软件包、资源文件预先存放目标进程 P_{src} 中,在组件迁移时就可以直接使用目标进程 P_{src} 中的软件包和资源,从而提高组件迁移的效率.如果 Web 应用服务器支持集群,每个服务器事先启动服务、创建组件容器,生成组件实例的复件(replica),在组件迁移时,只需将客户请求的透明转发到该复件即可,节省了组件以及其容器创建的开销,提高了算法效率.

4.3 算法实现

SLB_Copy,SFB_Copy 和 EB_Copy 这 3 个算法均已在我们自主研制的 J2EE 应用服务器 WebFrame2.0 中实现,较好地满足了事务性 Web 应用的动态分布要求.为了保证消息的一致性,WebFrame 在 RMI 通信基础上提供了一个基于任务队列的 RMI 通信框架,请求连接线程将 EJB 客户的请求先放在等待队列(WaitingQueue)中,后台工作线程再从等待队列中获取任务,所有后台工作线程处理的任务都在 BusyQueue 中.在迁移状态前,先将 BusyQueue 中的任务处理完,当停止后台工作线程后,组件的状态就不会改变,保证迁移安全的特性.当迁移请求时,先停止请求连接线程,才将等待队列的任务迁移到目标组件实例,从而确保客户请求都能得到响应,满足消息一致性约束.

WebFrame 通过 Java 对象序列化方法实现了 EJB 组件实例状态的持久性,由持久管理器

(PersistenceManager)完成.目前,WebFrame2.0 实现了文件式、数据库和 In-Memory 这 3 种传输状态的方式.对于一些处于打开状态的资源(如 Socket、数据库连接、数据结果集等)的迁移,WebFrame 提供了 IResourceMigration 接口.该接口有两个重要的方法:ejbBeforeMigration 和 ejbAfterMigration.为实现这些资源的迁移,组件开发人员需在 ejbBeforeMigration 方法中关闭这些资源,而在 ejbAfterMigration 方法中重新打开它们.

由于不可能直接更新所有客户对组件的引用,WebFrame 在客户端应用动态代理设计模式.当客户使用迁移前的引用发送请求时,动态代理会捕获“引用不存在”例外,然后从 ComponentRepository 中查找组件实例目前所在的位置,根据所在的位置信息,更新客户端的引用.

WebFrame 通过集群提高组件迁移的效率.在集群环境下,组件迁移的时间开销主要在状态复制和请求转发两个方面.见表 4,实验数据表明,3 种不同类型的请求转发相差不大,平均时间大约为 122ms,而状态保存、传输、恢复的时间则相差较大,与组件的类型和大小有关.对于同一类型的 EJB 组件来说,状态保存和状态恢复的所用时间差不多,但超过了状态传输所用的时间.

Table 4 The cost of EJB migration

表 4 EJB 组件迁移开销

(ms)

	Stateless EJB	Stateful EJB	Entity EJB
State preserving	0	36.6	28.4
State transferring	0	15.5	0
State loading	0	37.5	30
Request forwarding	114.5	138.1	112.2

5 相关工作比较

在动态分布支持方面,文献[12,13]在已有组件系统实现了 OMG 定义的 Object Trading 服务.该服务查找属性值满足某种条件的特定类型的对象实例,通过 export-import-execute 模型提供服务的动态发布和后期绑定的能力[14].虽然 Trader 返回已有实例或新创建实例的引用,但一旦绑定,实例就不能移动.虽然 OMG 使用一种消息转发机制支持 CORBA CCM 组件迁移[15],GIOP 协议定义了组件绑定后改变其位置时的消息转发行为语义,文献[16]亦对 CCM 组件迁移问题进行了探讨,给出了具体的迁移步骤,但均没有定义一种实际可行的组件迁移机制确保迁移安全和状态一致性.而 WebFrame 则在满足组件迁移一致性约束的前提下,允许 EJB 组件的动态绑定和迁移,为分布式应用提供了一种在运行期间重构的能力,满足 Web 应用动态分布的更高需求.

在 EJB 组件迁移方面,文献[17]扩展了 JNDI 名字服务,在固定的应用服务器列表中按 JNDI 名字搜索组件,如果存在,则将组件所需的部署文件包(.jar 文件)迁移到本地,然后在本地重新部署.该迁移服务实际上只迁移了组件的代码状态,是一种很弱的迁移,并没有考虑组件的会话状态.而 WebFrame 具备了在运行期间动态迁移 EJB 组件的能力,不仅能迁移组件的代码状态,而且能在满足客户请求不丢失、迁移前后状态保持一致的情况下,迁移会话状态,提供了一种较强的组件迁移能力.文献[18]将 Eager_Copy 和 Lazy_Copy 等进程迁移算法简单地应用到 EJB 组件迁移,但由于没有考虑到组件迁移的自身特点,很难实用.此外,进程迁移中的状态对所有客户均有效,而组件的状态则不同,其有效范围可能仅在一个会话内,所以不能像进程迁移一样,只将状态迁移到目标节点.组件迁移还需保持迁移前后状态的一致性以及客户引用的有效性和透明性.Lazy_Copy,Post_Copy 算法都存在剩余依赖问题(residual dependecy),不是迁移安全的,很难满足组件迁移一致性约束.而本文提出的 SLB_Copy, SFB_Copy,EB_Copy 迁移算法结合不同组件类型的特点,较好地满足了组件迁移一致性约束(CMC).

目前主流 Web 应用服务器如 Oracle9iAs,IBM Websphere,BEA Weblogic 和 JBOSS 等都通过状态复制提供组件冗余服务,当系统崩溃时,能将客户的请求透明地转发到备用组件,从而实现失效恢复,但不具备在运行期间动态迁移 EJB 组件的能力,不能满足组件动态分布的要求.

6 结束语

本文结合我们自主研发的 J2EE 应用服务器 WebFrame 的实际需求,定义了 EJB 组件迁移模型、迁移原语

和约束,并针对各类组件类型的特点,设计了 SLB_Copy,SFB_Copy,EB_Copy 迁移算法.这 3 种算法均满足组件迁移一致性约束(CMC),避免了组件迁移前后的不一致.这 3 个组件迁移算法已在我们自主研制的 J2EE 应用服务器 WebFrame2.0 中实现.模型与算法已应用在自适应负载平衡、失效恢复以及 Web 应用的动态重配 3 个方面.从实际应用情况看,WebFrame 中的 EJB 组件迁移服务较好地满足了 Web 应用的动态分布需求.

进一步的工作包括 3 个方面:1) 如何改进上述 3 种迁移算法满足更强的一致性约束,尽可能降低对其他应用的影响,优化算法的性能;2) 如何形式地验证迁移前后的一致性;3) 如何将组件迁移模型和算法应用到移动计算环境,满足客户移动的需求.

References:

[1] Fan GC, Zhong H, Huang T, Feng YL. A survey of Web application servers. Journal of Software, 2003,14(10):1728~1739 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/1728.htm

[2] Richmond M. Component migration with enterprise JavaBeans. In: ACM SIGPLAN, ed. Proc. of the Conf. on Object-Oriented Pprogramming, Systems, Languages, and Applications. New York: ACM Press, 2000. 79~80.

[3] Fan GC, Zhu H, Huang T, Feng YL. Towards adaptive load balancing services for Web application servers. Journal of Software, 2003,14(6):1134~1141 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/1134.htm

[4] Astley M, Sturman DC, Agha GA. Customizable middleware for modular distributed software. Communications of the ACM, 2001,44(5):99~107.

[5] Blair GS, Coulson G, Robin P, Papathomas M. An architecture for next generation middleware. In: Proc. of the IFIP Int'l Conf. on Distributed Systems Platforms and Open Distributed Processing. New York: Springer-Verlag, 1998. 191~206.

[6] Truyen E, Jørgensen BN, Matthijs F, Joosen W, Verbaeten P. Component architecture for dynamic reconfiguration of object request brokers. In: Sventek J, ed. Proc. of the IFIP/ACM Middlware 2000, Workshop on Reflective Middleware. New York: Springer-Verlag, 2000. 14~17.

[7] Forman GH, Zahorjan J. The challenges of mobile computing. IEEE Computer, 1994,27(4):38~47.

[8] Litiu R, Prakash A. DACIA: A mobile component framework for buildinh adaptive distributed applications. Operating Systems Review, 2001,35(2):31~42.

[9] SUN Microsystems. Enterprise java beans 2.1specification, 2001. http://java.sun.com/products/ejb

[10] Miloicic DS, Douglis F, Paindaveine Y, Wheeler R, Zhou SN. Process migration. ACM Computing Surveys, 2000,32(4):241~299.

[11] Richmond M, Hitchens M. A new process migration algorithm. ACM SIGOPS Operating Systems Review, 1997,31(1):31~42.

[12] PrismTech Corporation. OpenFusion Trading Service white paper, 2001. http://www.prismtechnologies.com/English/Products/CORBA/CORBA_services/trading/whitepaper/1_Trading_may01.html

[13] ExoLab Group. The OpenORB Trading Service. 2001. http://dog.intalio.com/trading.html

[14] Object Management Group. CORBAservices: Common Object Services Specification. 1998.

[15] Object Management Group. The Common Object Request Broker: Architecture and Specification Minor Revision 2.3.1. 1999.

[16] Villoldo EJ. Component migration supported. http://moriarty.dif.um.es/pipermail/ccm/2002-July/000153.html

[17] Frénot S, Avin MS, Almasri N. EJB components migration service and automatic deployment. Technical Report, 2002. http://www.inria.fr/rrrt/rr-4480.html

[18] Richmond M. Support for dynamic distribution in component systems. In: Landherr S, ed. Proc. of the Workshop on Component-Oriented Software Engineering '98 in Conjunction with Australian Software Engineering Conference (ASWEC'98). Adelaide: IEEE Computer Society Press, 1998. 5~8.

附中文参考文献:

[1] 范国闯,钟华,黄涛,冯玉琳.Web 应用服务器研究综述.软件学报,2003,14(10):1728~1739.http://www.jos.org.cn/1000-9825/14/1728.htm

[3] 范国闯,朱寰,黄涛,冯玉琳.Web 应用服务器自适应负载平衡服务.软件学报,2003,14(6):1134~1141.http://www.jos.org.cn/1000-9825/14/1134.htm