软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(11):2652−2667 [doi: 10.3724/SP.J.1001.2011.03917]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

事务控制的面向服务系统的动态更新协调*

王德俊+, 黄林鹏, 徐小辉

(上海交通大学 计算机科学与工程系,上海 200240)

Coordinating of Dynamic Updating Controlled by Transaction in Service- Oriented System

WANG De-Jun+, HUANG Lin-Peng, XU Xiao-Hui

(Department of Computer Science and Engineering, Shanghai Jiaotong University, Shanghai 200240, China)

+ Corresponding author: E-mail: wangdejun@sjtu.edu.cn

Wang DJ, Huang LP, Xu XH. Coordinating of dynamic updating controlled by transaction in service-oriented system. *Journal of Software*, 2011,22(11):2652−2667. http://www.jos.org.cn/1000-9825/3917.htm

Abstract: This paper presents a group of strategies based on transaction mechanism for the coordination of the multi-services' dynamic updating in a service-oriented distributed system. These strategies included a selection strategy for choosing of security updating time-point and a 2PC (two-phase-commit) strategy for controlling the multi-services' updating. With the analysis of time, needed by related updating operation steps, this paper suggests that the basic updating operations (including creation of service instances, orderly execution of multiple service instances' runtime state-convertion, redirection of service-request-message and activation of new services) should be controlled in a short updating transaction, and multiple persistent data conversions should be controlled orderly under a long transferring transaction (every persistent data conversion should be controlled in a short transaction). The goal is that with the help of transaction mechanism, the world will expect to ensure system state's ACID properties before and after updating it as efficiently as possible. Furthermore, to make of the most of the correct updating of system, this paper researches about automatically producing and updating transaction, data conversion transaction, and serializing among application transactions and conversion transactions. In the last section, this paper gives a prototype validation of the strategies based on Apache CXF-DOSGi.

Key words: dynamic update; coordination strategy; transaction; distributed system; service-oriented

摘 要: 就面向服务的分布式系统中多服务动态更新的协调问题,提出基于事务控制的更新协调策略,包括安全更新时机的选择策略、使用 2PC(two-phase-commit)协调多服务更新的控制策略.根据更新实施的各步操作时间开销,提出分别采用更新短事务控制基本更新操作(包括创建服务实例、顺序实施运行时状态转换、重定向服务请求和新服务的激活操作)和转换长事务控制多组持久数据转换操作(其中每组持久数据转换由一个相应的短事务控制)的更新实施策略,以期借助于事务的控制机制尽可能高效地保证更新前后系统运行状态的 ACID 特性.就事务控制策略的自动实施做了一定的研究工作,包括:根据更新描述脚本自动生成更新短事务操作序列、根据持久数据关联构件的依赖关系分析自动生成各个持久数据转换短事务序列组成的长事务以及对应用事务和各转换事务的序

* 基金项目: 国家自然科学基金(60970010); 国家重点基础研究发展计划(973)(2009CB320705); 高等学校博士学科点专项科研基金(20090073110026)

收稿时间: 2009-11-12; 修改时间: 2010-04-14; 定稿时间: 2010-07-06

列化处理.最后,通过基于 Apache CXF-DOSGi 的原型实现对给出的控制策略进行了可行性验证.

关键词: 动态更新;协调策略;事务;分布式系统;面向服务

中图法分类号: TP311 **文献标识码:** A

目前,在大规模分布式在线服务系统中,系统的可靠性、可用性、可信性以及可维护性逐步成为计算机系统必须考虑的重要因素,尤其在一些关键服务应用领域,如金融数据处理系统、医疗服务系统、远程通信系统、电信在线计费系统等[1,2].

此类系统在软件的生命周期中必须保持不断的进化,以修正软件故障、扩展服务功能、提高系统性能.因此,需要相应的机制支持系统在持续运行的同时能够演变和扩展功能,同时又不影响现存的服务,即实现分布式系统的动态更新支持机制.

而对于一个大型的应用服务系统,可能具有上千个终端,用到上百台计算机,需要提供绝对没有停机时间的服务,对于这种系统的设计、实现和操作,需要用一种综合的观察角度和方法来解决分布式的问题.事务提供了这样一种集成的概念框架,没有事务,分布式系统不能满足典型实际应用的需要[3].而对于在系统上作的任何软件动态更新,都可能需要同时更新许多相关组件.如要给屏幕增加一个字段来改变屏幕,很可能要修改客户软件和服务器界面,也可能要向表里增加一列.所有这些更新必须尽可能一起进行(作为一个 ACID 单元).如果更新没有完成,那么它们则需要全部被复原,以使系统返回到更新前的状态.手工实施这样的更新无疑会带来灾难.

本文首先分析了在“24×7”面向服务的分布式系统中支持动态更新需要解决的关键问题,并就其中的两个更新协调相关问题:更新时机的选择和多服务更新的协调展开深入的研究,提出用基于事务的控制策略来协调多服务的更新,从而尽可能保证系统更新前后的 ACID 特性.

本文第 1 节分析分布式系统软件动态更新需要解决的协调问题.第 2 节分析提出基于事务控制的协调更新策略,包括:安全动态更新时机的选取策略、使用 2PC(two-phase-commit)来协调更新多个服务的控制策略.第 3 节进一步提出自动实施更新策略的关键处理方法,包括更新事务操作序列的自动生成、基于服务构件依赖关系的状态转换顺序确定以及应用事务和转换事务交叉的序列化.第 4 节简要分析策略的实施对系统的 ACID 特性影响.第 5 节通过原型设计来验证相关策略的合理性.第 6 节对相关工作作分析.第 7 节总结全文.

1 分布式系统软件动态更新协调问题

随着服务成为开放网络环境下资源封装与抽象的核心概念,通过动态地组合服务实现资源的灵活聚合,是面向服务计算的核心技术,成为近年来的研究热点.事务机制是保证服务组合可靠性的重要手段,然而,服务组合的复杂性、动态性、长期运行性以及服务之间的异构性,使得传统事务的 ACID 特性必须被放松.就我们所了解,统一的事务理论还没有发展起来,扁平事务最终成为在商业数据库中使用的事务模型;同时,它也被应用到操作系统和通信系统中.很多事务模型是针对某个特定的应用,根据需要给传统的 ACID 事务模型增加一些特性来支持特定的应用.比如,短事务的 ACID 属性由数据库或应用服务器实现,而长事务的 ACID 属性是通过短事务来保证,并通过补偿事务来完成的;基于普适计算的应用为了保证以服务为单位的应用程序执行的可靠性,需要在服务组合和执行层面上的事务模型和上下文感知的事务管理功能[4].

现有的商用事务处理系统能够较好地支持分布式事务,并且系统的开放性也正在得到改进,如 IMS,CICS,Guardian,ACMS,X/OPEN 等,且多数都支持事务型远程过程调用机制(TRPC).因此,本文以 Gray 提出的关于事务处理系统的计算模型[3]为基础,分析并提出分布式系统上的动态更新协调控制方法.该模型包括以下 4 部分内涵:

(1) 资源管理器:系统有一系列的事务资源管理器,资源管理器协同工作提供保持 ACID 特性的事务.

(2) 持久数据:应用设计者把应用状态表示为资源管理器所存储的永久数据.

(3) 事务程序:应用设计者把应用查询和状态转换表示为用常规的或专门的编程语言写成的程序.

(4) TRPC:TRPC 允许应用去调用本地和远程资源管理器,也允许应用设计者通过 TRPC 把在不同计算机

上互联的应用分解为请求服务和提供服务的服务进程或线程.

此外,访问远程服务时,本地的服务和服务器被绑定在调用者的地址空间,而远程的服务和服务器则是运行在独立的进程或者是远程计算机系统上.整个计算在一个单独事务内发生.事务管理器监视着事务的进展,把客户连接到服务器并且协调着事务的提交和回滚.

因此,本文考察动态更新所基于的分布式系统环境的运行特征是:面向服务的、且是以事务机制来保证分布式系统上服务运行的状态一致性.根据我们已有的工作[5,6]以及上述分布式系统中服务运行的事务处理系统的计算模型,面向服务的分布式系统的动态更新需要解决的关键问题包括如下 4 个(其中的问题(1)和问题(4)是关于动态更新的协调问题,也是本文所侧重的):

(1) 更新时机的确定:一旦旧服务接收到服务更新消息后,需要在有限的时间范围内能够停止当前已经被启动的服务,选择在安全的时机将其替换成新的服务.

(2) 创建新的服务,进行服务状态的传递转换:创建已经装载的新的代码实体,并在本地和远程服务器登记注册新服务.根据系统配置人员提交的更新请求描述文件,服务器产生旧服务到新服务的映射,并支持服务状态的运行时传递转换.

(3) 重定向服务请求:在等待更新时机到来期间,需要缓存更新期间到来的但不能响应的服务请求,更新服务后,将缓存的服务请求进行重定向.

(4) 多个服务的协调更新:针对多个有依赖关系的服务更新,需要采用一定的协调机制来确保这多个服务的更新被原子地、一致地执行,以保证更新前后系统状态的一致性.包括:选取在适当的时机按照一定的顺序创建多个新的服务、实施多个服务的运行时状态数据的转换、实现多个服务的重定向和多个服务对应的持久数据的转换等,在必要的时候撤销或确认更新操作.

2 基于事务控制的更新协调策略

2.1 安全动态更新时机的选取策略

处理动态更新时机的模型基本分成两种:调用模型和中断模型[7].调用模型比较简单、直接,但没有被广泛采纳,因为它要求应用在被开发时就确定更新点.中断模型是根据相应的更新消息选择适当的更新时机来触发更新操作,更自动、更灵活,但也更复杂.理论上是可以在任意时间更新,是调用模型的超集,但事实上,运行时检查工作非常复杂.一个好的运行时检查应该保证在安全的时机进行动态更新,且使用的方法不会过度保守[8],能够在保证更新前后功能系统的状态合法、一致的前提下尽快提供用户持续的服务.

Gupta 已经证明,并不是在程序运行的任何时候都能进行动态更新[9].为保证更新前后的系统状态的一致性,Kramer 等人提出的静止状态是基于构件的动态配置方法所公认的动态配置安全状态,即在此状态到达时更新目标构件,对于系统是安全的[10].

为了保证面向服务的分布式系统状态在更新前后的一致性,本文要求服务在动态更新前必须进入服务静止状态(quiescent state),即认为服务静止状态的到达为动态更新的安全时机.

服务静止状态需要满足如下 4 个条件:

(C1) 服务当前没有参与自行启动的事务,即自行启动的事务已经执行完成.

(C2) 服务将来不会自行启动新的事务.

(C3) 服务当前没有参与由其他服务启动的事务.

(C4) 服务将来不会参与由其他服务启动的事务.

当条件 C1、条件 C2 满足时,服务不存在自行启动的事务,只会被动参与由其他服务启动的事务,称为被动状态(passive state).根据 Gray,已有事务总是能够在有限时间内执行完成[3],因此服务的 Passive 状态是可达的.

需要说明的是,在面向服务的计算中,如果采用完全动态的绑定服务,则其系统的执行效率较低,一般采用的处理办法是:在引用远程服务时,分布式系统的服务查找是基于消息传递的松散耦合,在找到匹配的服务后,则采用的多不是非阻塞式访问,而是阻塞式,从而提高服务的响应效率,即当远程服务被调用时,此时系统往往

提供一个机制使得可跳过服务中心的查找匹配工作(无须再做查找匹配工作)直接向另一端的服务发起调用.当调用时,可以走某种通信机制(如代理机制)将请求服务的接口、方法以及参数传递至中心服务器,在事务管理器上登记后,进一步将服务请求消息发给相应的服务提供端,由服务端的构件运行在容器中提供服务;当服务提供端处理完毕后,继续走这个通信机制把处理的结果返回至调用端.本文基于的分布式系统 Apache CXF-DOSGi,是通过建立在 proxy bundle 机制之上的远程服务访问来实现远程服务访问的透明性.

鉴于上述远程服务访问方法,可以证明服务静止状态是可达的.

对单个需要更新的服务 Q 来说,定义服务 Q 的被动集合(passive set)为 $PS(Q)$,包括 Q 和已有事务中所有依赖服务 Q 的服务.确切地说,$PS(Q)$中包含的依赖 Q 的服务 S,并不是任何可能依赖 Q 的服务,而是事务运行过程中对服务的访问中曾经绑定访问过的服务 S,S 依赖 Q,即

$PS(Q)=\{Q\}\cup\{S|S$ 是当前已有事务 T 中正在运行或曾经访问过的服务,且 S 依赖并绑定到 $Q\}$.

可以证明以下定理:

定理 1. 当 $PS(Q)$中的所有服务都进入 Passive 状态时,服务 Q 进入静止状态.

证明:

首先,依赖服务 Q 的服务 SA 有 3 种可能:(1) 已经直接和间接地绑定到 Q 上、依赖 Q 服务的服务 S;(2) 已经加入事务的服务 $SO1$,但还没有绑定到 Q 上、可能在将来会绑定到 Q 上;(3) 还没有加入到事务中,但将来会加入到事务中,且可能会绑定到 Q 上的服务 $SO2$.

对以上 3 种情况,采用严格以已启动事务 T 的完成为更新的安全时机是可以的,但如果 T 是短事务,则所有依赖 Q 的服务均会很快进入 Passive 状态,显然条件 C3、条件 C4 成立,服务 Q 进入静止状态.

如果 T 是长事务,那么等待 T 完成后再更新将是不实际的.为此,需要做一个约定:在服务 Q 更新请求到来后,不再被未曾绑定过的服务 SO 绑定,除非 SO 被情况(1)中的 S 依赖.因此有:对情况(1)中的 S,Q 将继续响应依赖服务的请求直至 S 和 Q 均进入 Passive 状态,对于情况(2)、情况(3)中的 $SO1$ 和 $SO2$,不再绑定到 Q 上,除非是因为情况(1)中的 S 需要进入 Passive 状态,而 S 依赖的 $SO1$/$SO2$ 需要将来绑定到 Q.但这种情况事实上只要 S 和 Q 均进入 Passive,则 $SO1$/$SO2$ 也进入 Passive 状态.因此,当 $PS(Q)$中的所有服务均进入 Passive 状态时,显然条件 C3、条件 C4 也成立,因此服务 Q 进入静止状态.证毕. □

从定理 1 可知,服务的静止状态可达,从而可以得出驱动服务 Q 进入静止状态的算法.通过驱动 $PS(Q)$中除 Q 以外的所有服务进入被动状态,将在禁止其他服务继续启动需要 Q 参与的事务或者有条件地禁止未曾绑定到 Q 上的服务绑定 Q 后,驱动等待 $PS(Q)$中的服务全部进入 Passive 状态.

因此可以看出,为了能够确定 $PS(Q)$,需要记录事务和其访问服务的关联信息,从而支持协调控制到达安全的更新时机.如果 Q 参与的应用事务是短事务,则以事务的完成为安全更新时机;如果 Q 参与的应用事务是长事务,则当 $PS(Q)$中的所有服务都进入 Passive 状态时为安全的更新时机.

对一组包含多个需要更新的服务来说,存在有依赖关系和没有依赖关系之分,如果没有依赖关系,则可以进一步通过并行控制执行各自的更新操作而提高更新效率.但考虑到多服务更新操作需要的原子性,本文将这一组多个服务的更新时机均到达时作为此组服务的更新时机,并将其更新操作控制在一个更新事务下.如图 1 中的 $Service1$,$Service2$ 和 $Service3$ 需要更新,且在其参与的事务中已经有 $Service1$ 绑定依赖 $Service2$,$Service2$ 绑定依赖 $Service3$,如果需要更新此 3 个服务,则需要 3 个服务均到达安全更新时机.根据前面所述的更新时机选择策略,当 $PS(Service1)\cup PS(Service2)\cup PS(Service3)$中的服务均进入 Passive 状态时为此组服务更新的安全时机.如果该事务只有图 1 中的服务和相应依赖关系,则

- $PS(Service1)=\{Service1\}$;
- $PS(Service2)=\{Service1,Service2\}$;
- $PS(Service3)=\{Service1,Service2,Service3\}$.

因此,$PS(Service1)\cup PS(Service2)\cup PS(Service3)=\{Service1,Service2,Service3\}$.即,如果此 3 个服务要同时更新,则其更新的安全时机是 3 个服务均处于 Passive 状态时,可以启动控制更新操作的相关事务.具体在下一节阐述.

需要说明的是,由于在事务中可能一个服务对另一个服务的访问请求有多次,因此,图 1 中只要 *Service*1 没有进入 Passive 状态,它的运行就可能使之前已经完成服务请求的 *Service*2 再次被激活以响应 *Service*1 的请求,执行完成后再次进入 Passive 状态.因此如图 2 所示,在 3 个服务均到达 Passive 状态前,即到达安全的更新时机前,有些服务需要多次进入 Passive 状态.但由于事务总是会在有限时间内完成的,所以相关服务也都是能在有限时间内到达 Passive 状态.

Fig.1 A set of services with dependency

图 1 一组有依赖关系的服务

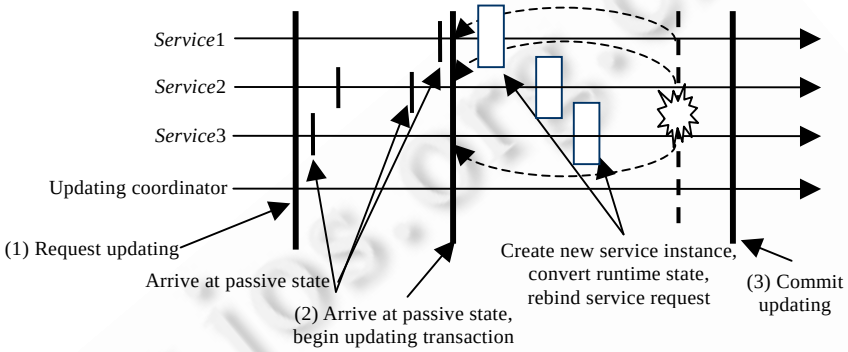


Fig.2 Update coordination chart

图 2 更新协调图

2.2 多个服务的协调更新策略

在更新时机到达后,系统可以开始一系列的更新操作以完成服务的更新,这一系列操作包括创建新的服务实例、新旧服务运行时状态传递和转换、服务请求的重定向、服务的激活和持久数据转换操作.其中,持久数据的转换操作可能需要很长时间才能完成,系统不可能等所有的持久数据转换执行完才提交确认这一系列更新操作.因此,本文将这一系列操作分为两组,分别用不同的事务进行控制:将除了持久数据转换外的其他操作构成更新基本操作,因为时间开销较短,所以用一个更新短事务(以下称更新事务)来控制实施;而对相关持久数据的转换,则用一个转换长事务来控制执行.即以更新事务的方式实施控制基本操作,从而保证多个有依赖服务更新的原子性;以转换长事务控制持久数据的转换,其中包含 1 个或多个可能有依赖的转换短事务,每个短事务控制一组持久数据转换的原子操作,其执行顺序建立在持久数据转换的依赖关系上.具体转换短事务顺序的确定方法和依据见第 3 节.之所以如此,是希望借助于事务控制基本更新操作和持久数据转换操作,以尽量保证系统更新前后状态的一致性,且使得用 lazy 方式控制运行各个转换短事务成为可能,从而减少系统软件更新期间服务的不可得时间.

对多个服务的基本更新操作用更新事务来组织控制,使得多个服务的基本更新操作要么一次更新完成,要么不更新,以保证多个服务的基本更新操作的原子性.即,由更新协调器(updating coordinator,简记为 UC)在接收到多个服务更新的请求且确认更新服务的依赖关系后,使用 2PC 来控制和协调多个服务的更新操作,对应图 1 中的一组已经产生依赖关系的服务 *Service*1,*Service*2,*Service*3 的更新,其协调步骤如图 2 所示,其中:

- 步骤(1)表示由 UC 产生更新请求,并等待各个服务到达安全更新时机;
- 步骤(2)表示启动执行各个服务的更新基本操作(包括创建新的服务实例、旧到新的服务运行时状态的转换、旧到新的服务的重定向),向 UC 确认更新准备工作的完成情况;

- 步骤(3)表示,如果多个服务更新成功,则由 UC 最终确认更新,以激活各个服务端的新服务.

3 策略实施的自动控制

3.1 更新事务操作序列的自动生成

为了提供用户方便、直接地提交更新请求,并由系统自动验证分析接口的兼容,根据服务依赖关系产生更新操作序列,并按照这个序列串行或并发地控制各个服务的更新操作,尽可能保证自动、高效地实施动态更新,本文定义了描述更新请求的脚本语言.根据上下文无关文法,通过定义产生式规则给出更新脚本文件的语言语法如下:

Update→(*BasicUpdate*;(*PersistentDataUpdateList*))

BasicUpdate→([*AddService*;][*DeleteService*;][*ReplaceService*;])

PersistentDataUpdateList→*PersistentDataUpdate*; *PersistentDataUpdateList*|ε

PersistentDataUpdate→*DataSourceName=PathName*; *DataDestinationName=PathName*;
PersistentTransComponentName=Identifiers;

AddService→*Add*(*ServiceList*)|ε

ServiceList→*Service*; *ServiceList*|ε

Service→*ServiceDescription*(*ServiceName=Identifiers*;*ServiceVersion*=[*Number*.]**Number*;
DescriptionFileName=PathName)

DeleteService→*Delete*(*ServiceList*)|ε

ReplaceService→*Replace*(*Old=Service*,*New=Service*); *ReplaceService*|ε

其中,文法 *G* 是一个四元组(*Vt*,*Vn*,*S*,ρ):*Vt* 表示一个终结符集合;*Vn* 表示中间符号集合;*S* 表示起始符,即为 *Update*;ρ表示一个产生式规则集.具体的 *Vt* 和 *Vn* 定义如下:

Vt={*Identifiers*,*PathName*,*Number*};

Vn={*Update*,*BasicUpdate*,*PersistentDataUpdateList*,*PersistentDataUpdate*,*AddService*,*DeleteService*,
ReplaceService,*ServiceList*,*Service*}.

上述产生式中,*Add*,*Delete*,*Replace*,*DataSourceName*,*DataDestinationName*,*PersistentTransComponent-Name*,*ServiceDescription*,*ServiceName*,*ServiceVersion*,*DescriptionFileName*,*Old*,*New* 是关键字.*Identifiers* 和 *Number* 的词法定义如下:

Identifiers→"[[*a*~*z*,*A*~*Z*,0~1]*.]*[*a*~*z*,*A*~*Z*,0~1]*"

PathName→"[[*a*~*z*,*A*~*Z*,0~1]*/]*.[*a*~*z*,*A*~*Z*,0~1]*"

Number→[1~9][0~9]*

根据脚本语法定义,如果对图 1 中的 *Service*3 进行更新,将 *Service*3 从版本 1 更新到版本 2,则更新描述如下:

```
(Replace
(Old=ServiceDescription(ServiceName="Service3";ServiceVersion=1;
    DescriptionFileName="cs/simpleserver/res/S3metadata1.xml"),
    New=ServiceDescription(ServiceName="newService3";ServiceVersion=2;
    DescriptionFileName="cs/simpleserver/res/S3metadata2.xml");
))
```

因为本文的更新对服务接口的绑定是基于子类型匹配的[11],所以名字可以相同也可以不同,但类型要匹配,以保证更新后的接口绑定的类型安全性[5].根据服务间的依赖关系分析产生更新事务控制的操作序列,产生的结果伪代码描述如下:

```
BeginWork
  Test_Interface_Consistent("cs/simpleserver/res/S3metadata1.xml","cs/simpleserver/res/S3metadata2.xml");
```

```
while (wait_to_Passive_State({Service1,Service2,Service3})==0)  //等待更新时机到达前缓存服务请求
        Save_Service_Require(Service3);
Create_Service_instance(newService3);
Transfer_State(Service3,newService3);
Require_Rebind(Service3,newService3);
Activate(newService3);
EndWork
```

如果需要更新多个服务,如需要更新图 1 中的 *Service*1,*Service*2 和 *Service*3,则描述文件需要增加 *Service*1,*Service*2 服务更新前后版本的对应刻画,建立在图 1 上,产生的伪代码中需要增加对单个服务的顺序无关的接口验证、服务请求缓存、服务重定向、服务激活的处理,还要根据依赖关系处理依赖相关的更新基本操作序列,涉及顺序创建实例、运行时状态转换等操作.其实例创建和状态转换伪代码描述如下:

```
BeginWork
…
Create_Service_instance(newService1);
Transfer_State(Service1,newService1);
Create_Service_instance(newService2);
Transfer_State(Service2,newService2);
Create_Service_instance(Service3);
Transfer_State(Service3,newService3);
…
EndWork
```

可以看出,因为有服务依赖关系,*Service*1,*Service*2 和 *Service*3 的实例创建和状态转换需要根据其依赖关系顺序执行,关键是看更新服务的依赖关系,具体的状态转换实施和顺序确定方法将在第 3.2 节详细阐述,而持久数据转换方法的顺序执行控制和运行时状态转换顺序控制方法类似.

3.2 基于依赖关系的状态转换顺序确定方法

本文关于策略实施自动控制的研究是基于面向服务的构件模型(service-oriented component model,简称 SOCM)展开的,且本文实验原型所在的基于 OSGi(open services gateway initiative,简记为 OSGi)的分布式系统也是基于 SOCM 的.因此,本节将首先分析 SOCM 的基本设计原则,并在此基础上提出基于依赖关系的状态转换顺序控制方法.

3.2.1 面向服务的构件模型(SOCM)分析

由于面向服务思想主要专注于动态性和互换性,而面向构件技术则主要专注于创建可复用的软件构件模块,因此,很多流行的软件技术都是将构件模型和面向服务模型结合,形成 SOCM,SOCM 是面向服务和面向构件思想的结合,其设计遵守如下的基本原则[12]:

(1) 服务是可被使用的工作模块:是可重复利用的行为或行为的集合.
(2) 服务被描述成一个契约:契约定义了服务的特征,这些特征可用来组合、交互以及查找.
(3) 构件用来实现契约:通过实现契约,构件实现了服务描述中描述的服务,并在实现的过程中遵守服务描述的约束.
(4) 使用面向服务的交互模式来解决服务之间的依赖关系:服务由构件所提供,并发布在服务注册表中.
(5) 组合用契约来描述:组合是一个契约的集合,可以在这个集合中选择具体的构件来初始化.
(6) 契约是可替代能力的基础,在组合中,任何实现一个给定契约的构件都能被实现同样契约的构件所替代.

基于这个原则,一个服务是服务提供者为其客户提供的功能单元,由 1 个或者多个构件组合实现.这种服务

称为构件服务(component service),而实现服务的构件称为服务构件(service component)[13].一个现存的服务只有在它不被任何服务请求者使用时才可以离开,从服务构件的角度来讲,只有当导出的服务没有被绑定到任何服务请求者上时,一个构件可以被卸载.因此,为支持服务的动态更新,需要构件实例侦听服务的离开或到达、应用程序侦听构件的离开或到达.图 3 反映了在 SOCM 中的服务构件和构件服务的关系,而且也直观地表达了构件组合的概念和有状态构件的含义.

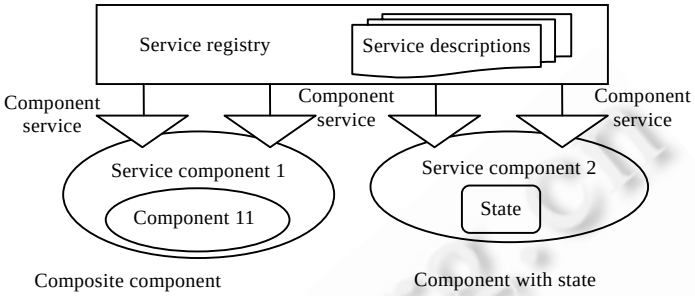


Fig.3 Relation between component service and service component in SOCM

图 3 SOCM 中的构件服务和服务构件的关系

构件服务具有如下特征:

(1) 构件服务只有两种形式:一个是原子(atomic)构件服务、一个是复合构件服务.

(2) 由单个构件通过其接口实现的服务称为原子构件服务,原子构件服务功能独立,其功能通过访问原子服务构件的接口得到.

(3) 复合构件服务是由原子服务构件组合而成的复合构件提供的,这些原子构件服务之间存在功能依赖关系.

为完成构件服务功能,服务构件开发人员需要实现构件设计人员所定义的构件接口,并按照服务构件组合方法将服务构件包部署到构件容器中,生成服务构件实例,并建立起构件服务.

服务构件分无状态构件和有状态构件:对于无状态构件来说,它们没有构件范围的状态;对于有状态构件来说,有一个构件范围的状态,在其提供的服务均已经处于静止状态,到达更新时机后,在升级到新的构件时,需要将原先保存在低版本构件(实例)中的状态值导出并转换传递到高版本的构件(实例)中.

3.2.2 服务构件状态的有序转换方法

根据 SOCM 的设计原则可知,当多个服务需要更新时,则自然需要对多个支持服务的服务构件的更新.由于多个服务构件的状态转换间可能存在依赖关系,为了保证更新前后系统状态的一致性,本文要求构件状态转换方法的执行能够确保新版本的服务构件只能看到相邻老版本服务构件的状态值和接口,以支持开发人员容易在开发时推理更新的正确性.

为了实现这样一个目标,首先需要分析服务构件间的依赖关系.依赖关系主要由两个原因产生:一个是构件服务调用关系产生的依赖;一个是服务构件之间的复合包含关系产生的依赖,如图 4(a)所示.

基于服务构件的两种依赖关系,确定服务构件的运行时状态转换顺序如下:

(1) 对于支持状态转换的多个构件来说,如果多个构件有服务引用关系,则其包含的相应转换方法执行顺序为:先执行请求服务所在构件的转换方法,后执行提供服务所在构件的转换方法.当然,这要求构件之间不能形成服务的递归引用,如图 4(b)所示.

(2) 如果多个构件是基于包含关系产生的依赖,则规定包含构件的转换方法先执行,后执行被包含构件的转换方法.类似地,不能出现构件类型通过引用或者指针类型等方式递归包含的情况.

这个顺序控制可以手工编程实现,也可以根据构件的调用关系和包含关系自动地分析生成构件状态转换操作执行序列,以避免手工编程控制而产生的人为错误.因此,本文规定构件间的包含关系要么是包含,要么是

构件间交集为空.如图 5 所示,图中每个椭圆均表示一个构件,椭圆上的三角形表示构件提供的公有接口.可以看出:被包含的构件是不会直接访问外部构件提供的服务的;同样,构件也不会访问被其他构件包含的构件数据成员.根据上述两种依赖关系,可以自动保证构件状态转换方法按照正确的顺序执行,从而保证系统状态在更新前后的一致性.

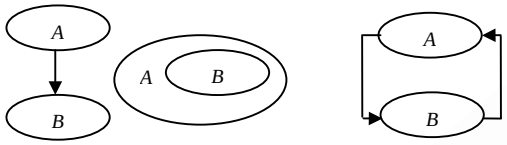


Fig.4 Component service request dependency relation classification

图 4 构件服务请求依赖分类

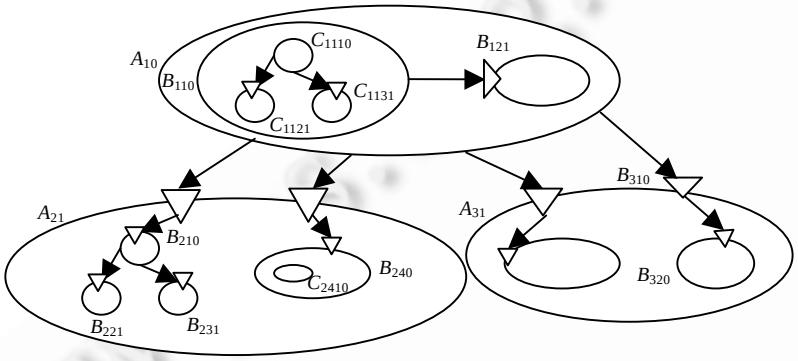


Fig.5 Dependency relation between the components

图 5 构件间的依赖关系

对图 5 中描述的依赖关系进行形式化描述如下:

按照构件的包含关系,记最外层包含构件内部标识名为 A_i(i 取值 1,2,…),其相邻包含的内层构件标识名为 B_{ij}(i 表示包含于 A_i,j 作为区分不同构件的下标,取值 1,2,…),再内层的构件标识名为 C_{ijk}(i 表示包含于 A_i,ij 表示包含于 B_{ij},k 作为区分同层不同构件的下标,取值 1,2,…),依此类推命名包含的内层构件.

构件间的调用依赖关系信息建立在上面对构件的标识基础上,在构件的下标加入调用层次信息来刻画.在同一个包含层次中,有调用关系的构件调用层次信息从顶层为 0 开始标记,从顶向下依次递增,即调用层次依次为 0,1,2,…将调用层次信息加到上面的构件标识的下标上则可以看出,构件的标识信息中就包含了调用或包含关系信息.

很容易有定理 2:

定理 2. 构件 M 依赖构件 N,则需满足如下条件之一:

(1) M 的包含关系下标组序列是 N 的包含关系下标组序列的前端序列;

(2) 存在构件服务访问路径经过 M 到达 N.

证明:

(1) 根据构件的编号方式,假设 M 的下标编组为 $m_1m_2…m_sL_m$,N 的下标编组为 $n_1n_2…n_tL_n$,其中,$m_1m_2…m_s$,$n_1n_2…n_t$ 分别为 M,N 的包含信息层次编组,L_m,L_n 分别为构件 M 和构件 N 的调用层次编号.条件描述的是:$m_1m_2…m_s$ 是 $n_1n_2…n_t$ 的前端序列,即 $t \geqslant s$ 且 $m_1m_2…m_s=n_1n_2…n_s$,则根据构件包含关系的编号方式,显然构件 M 包含构件 N,所以有 M 依赖 N.

(2) 根据依赖关系的定义,显然有结论(2)成立. □

显然,上面的定理论证过程也给出了判定服务构件依赖关系以及据此生成转换操作序列的方法.但需要说明的是,根据上面的规定,首先不会有内层被包含构件调用外层构件的情况出现.因此,此种调用依赖只可能有两种情况:一种是同包含层次(即构件的下标编组除去最低两位的编组序列)的构件间调用关系,这在本文的实现原型中由 dependency manager 来维护;一种是从外层包含构件到内层包含构件的调用,这可以根据结论(1)来判断.

3.3 持久数据转换事务和应用事务的序列化

在基于 SOCM 设计的软件系统中,就我们所知,持久数据的访问通常是通过相应的关联构件实施的,构件的数据成员和持久数据的各项信息(如数据表的字段或者 XML 文件的各个标记属性字段)基本对应,以此支持系统运行中的持久数据的读入和写出.因此,持久数据状态转换也由其关联的构件实施.

然而,当多组持久数据均需进行转换,且其转换有依赖关系时,首先,大量持久数据的转换不会经常发生,但一旦需要,其所需执行时间就可能会很长.比如,因为软件更新升级,某实时数据库中压缩的实时数据需要从最初存储的 Access 数据库转储到 Oracle 中,且转储时加入相应的压缩算法信息,以便需要时解压显示.由于数据量巨大而使得转换存储的时间很长,如果等待全部数据转换完成后再响应用户可视化终端对数据的请求,则系统的服务不可得时间过长.为了减少更新期间系统服务的不可得时间,本文采用 lazy 方式运行每个持久数据转换短事务(以下简称为转换事务).即,并不等待所有转换事务执行完后再支持应用事务的运行,而是根据更新需求生成多个转换事务后将转换事务暂时挂起.首先保证服务构件的应用事务的运行,以尽可能地减少系统的不可得时间,并在需要时触发相应转换事务执行以实现相应的持久数据更新.为了简化问题,本文假设系统中同时只可能存在两种版本.

为保证 lazy 方式更新的正确性,需要采用相应的机制对转换事务和应用事务进行序列化处理.因为 lazy 方式运行转换事务的触发时机有两种,一种是应用事务触发转换事务,一种是被依赖的转换事务触发依赖的转换事务的执行,因此,本文需要对这两种 lazy 触发交叉现象进行序列化分析.

为保证转换的正确性,在序列化之前首先做一个假设:

假设 1. 转换方法 TF 是 well-behaved.

所谓一个转换方法 TF 是 well-behaved 的是指,如果 TF 除了修改所提供构件关联的持久数据以外,不修改任何它在执行中遇到的其他构件关联的持久数据,包括它的参数持久数据和任何从它可达的持久数据,也即每个构件中负责转换的数据之间交集为空.之所以作这个假设,是因为转换方法的执行顺序是系统自动分析确定的,所以它选择的任何顺序都必须是正确的,否则,表面上不依赖的两个构件的转换方法运行会互相产生影响.

序列化 1. 当系统运行到一个应用事务 A 时,A 执行某个新构件方法,该方法需要访问某个持久数据表,而此时数据表尚未更新,因此,这个表数据首先需要被更新,即会触发对应关联的构件转换方法所在事务运行.所以,需要在这一点上中断应用事务 A,先运行对应的转换事务 T,即事务 T 必须在 A 之前运行.如果 T 需要访问老版本的持久数据,而该数据又被 A 修改过了,本文仍旧提供这个访问.因为此时 A 还没有提交(commit),所以,老版本的持久数据仍旧存在,并没有被修改.等 T 一执行完成,系统就可以提交 T,然后系统可以继续执行 A.但是如果 T 在此过程中修改了 A 已经读过的某个持久数据表(当然不会是 T 对应转换的数据表),则 A 需要回滚恢复重新执行.但是根据假设,T 是 well-behaved,那么它将不会修改它运行中可达的其他持久数据表,因此 A 可以继续执行.

序列化 2. 当运行转换事务 T_2 时,发现有依赖 T_2 转换的转换事务 T_1,根据前面的约定,T_1 只能读取 T_2 对应转换的数据的旧版本,因此,此时中断 T_2(就像前面序列化 1 中中断 A 事务一样)来运行挂起的转换 T_1.这里可能存在 T_1 修改了 T_2 对应的持久数据,但因为此前的假设 T_1 和 T_2 均是 well-behaved,因此不存在这种情况的可能性.

4 更新策略对系统的 ACID 特性影响分析

从第 1 节的背景分析可以看出,本文假设了动态更新策略所基于的分布式系统环境是支持事务处理的,且是面向服务的.然而,服务组合的复杂性、动态性、长期运行性以及服务之间的异构性使得传统事务的 ACID 特性必须被放松.本文根据长短事务的运行时间特征和更新相关操作的时间开销,提出了基于事务控制的动态更

新协调策略,以实施多服务的动态更新协调.策略的基本出发点是,以短事务的一致性能够保证长事务的一致性,从而能够在以短事务为 ACID 单元的基础上保证更新前后系统的 ACID 特性:

(1) 一致性 C 保证:通过确定选取相关依赖服务的静止状态为更新时机,保证更新时被更新服务是处于静止状态的,是可更新的安全状态;为支持正确更新,不妨假设新版本服务程序是正确的,对应转换方法也是正确的,策略在此基础上通过自动生成更新事务和持久数据转换事务序列,从而保证正确控制依赖的各服务构件的更新,以此来保证更新前后系统状态的一致性.

(2) 隔离性 I 保证:策略对更新事务和相关依赖的应用事务是处理成串行运行的,而对于转换事务和应用事务是可并发运行的,但对二者的交叉作了序列化处理,因此当应用事务执行时,应用事务读到的所有数据和转换事务的更新是隔离的;并且通过对支持转换的服务构件之间的依赖关系的规定,转换事务要转换的所有数据和由其他转换事务并发的读写也是隔离的.

(3) 永久性 D 保证:一旦提交成功执行,事务的状态转换是持久和公开的,事务执行的中间结果是不外化的.

(4) 原子性 A 保证:使用 2PC 协调多服务更新的控制策略,在提交前的任何一点,更新事务都可以回滚终止.如果事务被终止,那么它对服务构件运行时状态所有的改变都将被撤销.

5 原型实验结果分析

OSGi R4.2 规范为网络设备定义了一个标准的、面向服务构件的计算环境.该规范由 Framework,Standard Services,Framework Services,System Services,Protocal Services 等共同组成.在 OSGi 的核心框架中,Declarative Service 对 SOCM 提出了完整的支持,使得在服务构件单元 bundle 中可以按照构件+服务的方式进行开发[14].

Apache CXF-DOSGi 是分布式 OSGi 规范的一个实现,通过 CXF-DOSGi,能够方便地把运行在一个 JVM 中的 bundle 部署到不同的 JVM 中去.它主要提供了两方面的功能:(1) 将需向远程客户端提供服务的 bundle 暴露出去;(2) 为远程提供服务的 bundle 实现本地的存根.

为了验证更新协调策略的正确性,本文基于 Apache CXF-DOSGi 实现了一个原型系统,其系统架构如图 6 所示.在更新控制台(update console)上,更新部署者通过更新控制台提供软件更新包,并发出服务更新控制命令;更新控制台负责解析更新请求描述文件,并分析确定欲更新服务所在的 bundle 模块;更新控制接口(update control interface)同更新控制台交互,接受来自更新控制台的控制命令,解析控制命令,并将调用更新流程管理(update process manager)提供的服务以完成软件系统的动态更新.更新流程构件(update process component)包含 3 个子构件:流程计划制订构件(process plan maker)、流程执行引擎(process execution engine)和流程监控构件(process monitor),如图 6 所示.流程计划制订构件根据更新请求描述生成更新控制流程,流程执行引擎包含一个更新协调器构件,以事务方式原子地执行更新控制流程、协调多服务的原子更新.流程监控构件负责监控流程执行情况并对其操作作日志记录,在需要的时候支持撤销更新、恢复到更新前的状态.生命周期管理构件(lifecycle manager)的基本功能负责监测、记录和管理构件的运行状态,并和流程监控构件交互以确定更新安全时机的到达.运行时状态转换构件(runtime state transition)的功能是实现服务构件运行时状态从旧版本到新版本的传递转换.持久数据转换构件(persistent data transfer)负责根据模块依赖分析(module dependency analsis)的结果进行持久数据转换短事务的序列化,以支持 lazy 转换持久数据.多 bundles 模块依赖分析工具如图 7 所示,它能获取运行时系统中 bundles 的信息及其间的依赖关系,并能以图形形式表示依赖关系,以此支持多服务 bundles 的顺序协调更新和持久数据的序列化控制.更新校验构件(update checker)的功能是进行更新检查与验证,即在物理上实施更新前,检查更新是否合法,判断更新是否可用,并拒绝不可使用的更新.

需要说明的是,为了支持动态更新,开发人员需要在所开发软件系统的 bundles 模块中增加:

(1) 服务跟踪器(service tracker)和 bundle 跟踪器(bundle tracker):service tracker 用于监控进出 bundle 的服务,bundle tracker 用于监控进出的 bundle 信息,用于跟踪确定服务和服务 bundle 的运行状态[14].

(2) 服务钩子(service hook):利用 service hook,可以拦截服务之间的事件并对其进行过滤,用于分析转发普通服务请求消息和动态更新请求消息.

在原型系统上,本文对基于 Apache CXF-DOSGi 构建的系统网络节点上支持文件数据传输的两个服务的动态更新性能做了测试,涉及数据传输协议支持服务的更新和数据传输服务的更新,由低版本的支持单一协议的带状数据传输方式升级更新为基于节点负载平衡的支持多个协议的多副本带状数据传输[15],数据传输服务依赖数据传输协议支持服务的运行.从图 8 可以看出,数据传输性能有显著提升.同时可以看出,在更新点前后,数据传输性能无明显下降,且更新后是从更新前停止传输的地方继续传输,保存并传递了服务的中间运行结果.实验结果表明,本文提出的策略是合理的,能够支持面向服务的基于 bundle 构件实现的协调动态更新,且不影响分布式系统提供服务的运行性能.

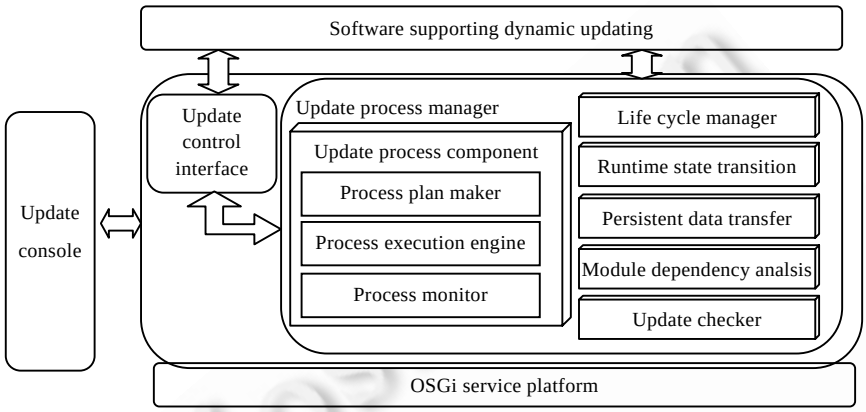


Fig.6 System architecture to support dynamic updating

图 6 动态更新支持系统架构

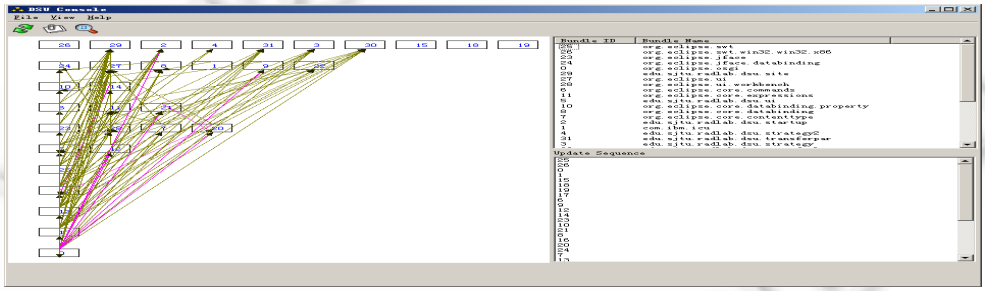


Fig.7 Multiple bundles module dependency analysis tool interface

图 7 多 bundles 模块依赖分析工具界面

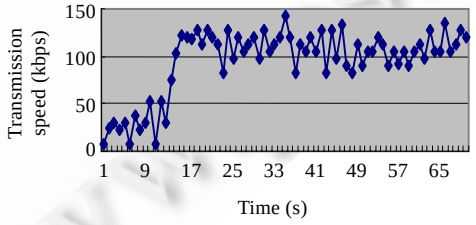


Fig.8 Data transferring performance supporting dynamic update

图 8 支持动态更新的数据传输性能

6 相关工作

目前流行的中间件平台的互操作、分布性、开发方便性、可移植性为开发具有方便、透明特征的分布式动态更新支持系统提供了有很好基础的开发平台.面向服务的构件模型(SOCM)强调面向服务模型和面向构件模型的结合,认为构件(component)是服务(service)的提供者,目前已有面向服务的构件模型的多个实现平台.从支持构件更新的角度来说,研究人员已经做了大量的工作.下面将就基于 JAVA 平台、CORBA 平台的构件模型和基于构件体系结构上的更新已有的工作做一个简单的回顾,而后分析面向服务的分布式系统更新的已有成果.

就 JAVA 的相关平台来说,Solarski 研发了一个基于 C/S 结构的分布式动态更新支持原型框架,且系统构件通信机制基于 RPC/RMI.该框架以 EJB 作为动态更新单元,系统提供了 3 种更新算法以解决有复制的多个 EJB 构件的更新,使用 2PC 来控制有多个复制的构件更新操作[16].王晓鹏等人借助 JAVA 平台的类装载机制,提出对构件化软件进行在线演化的方法,并将演化方案在一个符合 J2EE 规范的应用服务器 PKUAS 上进行了实现[17].

就基于 CORBA 的分布式系统上的动态更新支持来说,Moser 等人描述了在 Eternal 系统中的 CORBA 应用的更新.他们使用服务器对象的复制来提供在更新期间不被中断的服务[18].Bidan 等人为基于 CORBA 的分布式系统 Aster 描述了一个动态可重配置管理器[19].Almeida 等人描述了在 CORBA 里的动态更新,支持多个对象的原子更新,并使用 ORB 实现了较好的透明性[20].Balasubramanian 通过在中间件平台 SwapCIAO 上扩展轻量级 CORBA 构件模型的服务功能来实现构件的动态更新[21].

从基于构件体系结构上的更新来说,一个基于构件应用系统的更新经常被叫做重配置.Bialek 将动态构件体系结构思想(诸如 CORBA,EJB,DCOM 等)和动态软件更新结合起来,提出了一个支持构件结构运行时重配置的更新体系结构[22,23].Oreizy 等人在软件体系结构层次上提出了支持运行时软件更新的方法,高度抽象了运行时更新的处理操作,将有关应用具体的行为和动态更新策略分开,但是所有的构件和 connectors 必须被用 JAVA-C2 框架来写[24].Kramer 和 Magee 等人的工作主要集中在动态更新的管理工作上,如管理构件的增加、删除和重新配置构件间的链接[10,25].

为了选择安全的更新时机,以保证更新前后系统状态的一致性,在 Conic 系统中,一方面是定义构件的静止状态作为更新的安全时机,另一方面将更新具体化为说明性的改变命令(link,unlink,create 和 delete),通过一个配置管理器(configuration manager)将这些命令翻译成一个“更新事务”来执行一致的更新操作[10,25].Bidan 提出的更新机制不是要求对象在更新前是被动静止(passivate),而是要求对象之间的连接是处在被动静止的[19].窦蕾就面向构件的复杂软件系统中动态配置技术在保证分布式系统的一致性方面,给出了系统强弱一致性的概念和相应的保证方法[26].

李长云等人为了使面向服务的架构更适应动态更新的需求,提出了基于体系结构空间、支持动态更新的软件模型 SASM,通过具有因果相联的基层和元层,由可运行的服务构造基层,通过对体系结构空间的在线调整,实现对基层的修改,进而实现系统的非预设动态更新[27].马晓星等人提出了一种面向服务的动态协同架构,以适应底层因特网计算环境和用户需求的变化,设计实现了一个支撑平台 Artemis2ARC,为具有动态调整能力的面向服务应用系统的开发、运行和监控提供了一套可视化的集成环境[28].

OSGi 平台规范提供了开放和通用的架构,使服务提供商、开发人员等可以用统一的方式开发、部署和管理服务[14].张仕等人主要解决了 OSGi 平台上服务实例的动态演化问题,提出了通过间接引用来实现服务的重定向,利用实现和数据分离的办法解决了更新时数据转换的一致性问题[29].Wu 等人为基于 OSGi 的面向服务的软件系统建立了一个基于 ASM 的高层语义模型[30].该模型对 OSGi 平台的运行机理进行刻画,对现存基于 OSGi 的系统进行检测和比较,以形式化地确定是否满足动态更新需求和提供必要的功能.

7 结 论

本文就面向服务的分布式系统动态更新中的协调问题,包括更新时机的选择和多服务更新协调展开阐述,

并借助于事务机制尽可能地保证更新前后系统的ACID特性.提出以待更新服务的静止状态的到达为更新的安全时机;以更新相关操作的时间开销为基础,提出以短事务为单位组织基本更新操作,保证更新操作的原子特性;以短事务为单位控制单组持久数据转换的执行,以长事务为单位组织依赖的多组持久数据的转换执行;借鉴Boyapatihe 和 Liskov 等人对面向对象数据库中对象更新的顺序确定方法[31,32],通过约定关联构件的依赖关系,从而支持自动生成各组持久数据转换短事务的执行序列;并就应用事务和转换事务、转换事务间存在的交叉运行问题序列化,从而基于短事务的一致性基础上,尽可能地保证更新前后系统状态的一致性、隔离性和持久性.

目前就我们所了解,虽然还没有统一的事务处理系统(TP)架构[3],但是,现有构件编程技术和构件运行时容器已经具备安全、事务处理能力,为实现事务控制下的分布式系统、普适计算提供了很好的开发工具和研究参考价值, J2EE 和 CORBA 中的对象事务服务(OTS)以及".NET"框架中的分布式事务协调器(distributed transaction coordinator,简称 DTC)都是具体的例子.Gray 对 TP 做了基本的假设:在未来的系统中,和系统的交互或者系统间的交互都将被植入到事务中,以满足特定应用的一致性需求[3].因此,本文提出的策略是合理、可行且有现实意义的.最后,通过原型系统的实现进一步验证了时机选择策略和协调更新策略的合理性.

但由于分布式系统和动态更新这二者本身的复杂性,使得真正自动、高效、正确地执行分布式系统上的动态更新还需要做大量的理论和实践工作,如本文中:

(1) 新旧版本间的状态数据的转换(包括运行时状态和持久数据的转换)顺序的确定是建立在服务构件的包含、依赖关系的假设前提下,没有处理服务构件的交叉依赖和跨越服务构件边界的服务访问等复杂的情况.

(2) 本文定义的更新描述语言语法需要开发人员严格地遵守,其对应更新事务的生成也是根据定义的语法对应一一生成的,目前还没有成形的语法解释器和事务生成器,这将是本文进一步需要做的工作.

(3) 可以根据更新的服务间是否有依赖关系,协调各节点服务更新的串并执行,以进一步提高系统更新效率.

(4) 本文只是提供了方法来自动控制相关操作的顺序执行,但仍需要编程者提供新旧版本服务状态的转换方法以及持久数据的对应转换方法;进一步可以考虑根据对运行时状态数据成员或持久数据成员新旧版本的分析实现状态数据的自动转换,以便更好地提高开发人员的工作效率.

References:

[1] Strunk EA, Knight JC, Aiello MA. Distributed reconfigurable avionics architectures. In: Proc. of the 23rd Digital Avionics Systems Conf. 2004. 10.B.4-1−10.B.4-10. [doi: 10.1109/DASC.2004.1390803]

[2] Oki B, Pfluegl M, Siegel A, Skeen D. The information bus—An architecture for extensible distributed systems. In: Proc. of the SIGOPS'93. 1993. 58−68. [doi: 10.1145/168619.168624]

[3] Gray J, Reuter A, Wrote; Meng XF, Yu G, *et al*., Trans. Transaction Processing: Concepts and Techniques. Beijing: China Machine Press, 2004 (in Chinese).

[4] Luo JW, Qin X, Chen SG. Context-Based triggered task model in pervasive computing. Mini Micro Systems, 2004,25(8): 1542−1545 (in Chinese with English abstract).

[5] Wang DJ. Research on dynamic updating of service-oriented distributed system [Ph.D. Thesis]. Shanghai: Shanghai Jiaotong University, 2010 (in Chinese with English abstract).

[6] Wang DJ, Huang LP, Xu XH, Wu JK, Zhang S, Wang X. Survey of supporting system for dynamically updating distributed system. Computer Science, 2007,34(11):19−25 (in Chinese with English abstract).

[7] Hicks M. Dynamic software updating [Ph.D. Thesis]. Department of Computer and Information Science, University of Pennsylvania, 2004.

[8] Malabarba S, Pandey R, Gragg J, Barr E, Barnes JF. Runtime support for type-safe dynamic Java classes. In: Proc. of the 14th European Conf. on Object-Oriented Programming. LNCS 1850, London: Springer-Verlag, 2000. 337−361. [doi: 10.1007/3-540-45102-1_17]

[9] Gupta D. On-Line software version change [Ph.D. Thesis]. Kanpur: Department of Computer Science and Engineering, Indian Institute of Technology, 1994.

[10] Kramer J, Magee J. The evolving philosophers problem: Dynamic change management. IEEE Trans. on Software Engineering, 1990,16(11):1293−1306. [doi: 10.1109/32.60317]

[11] Li BX, Wang YF, Li XD, Zhen GL. Type-Subtype analysis and subtyping inference rules for object-orientation. Computer Science, 1999,26(7):23−28 (in Chinese with English abstract).

[12] Cervantes H, Hall RS. Autonomous adaptation to dynamic availability using a service-oriented component model. In: Proc. of the 26th Int'l Conf. on Software Engineering. Washington: IEEE Computer Society, 2004. 614−623. [doi: 10.1109/ICSE.2004.1317483]

[13] Liao Y. A method of QoS-aware service components composition for pervasive computing environments [Ph.D. Thesis]. Beijing: Institute of Software, the Chinese Academy of Sciences, 2005 (in Chinese with English abstract).

[14] OSGi service platform core specification release 4, version 4.2. 2010.

[15] Yang H, Ou JF, Huang LP, Implementation of striped transfer using multiple replicas in grid environments. Computer Applications and Software, 2006,23(11):54−56 (in Chinese with English abstract).

[16] Solarski M. Dynamic upgrade of distributed software components [Ph.D. Thesis]. Berlin: School of Electrical Engineering and Computer Sciences, Technical University of Berlin, 2004.

[17] Wang XP, Wang QX, Mei H. An approach to online evolution of component based software. Chinese Journal of Computers, 2005, 28(11):1890−1897 (in Chinese with English abstract).

[18] Moser LE, Melliar-Smith PM, Narasimhan P, Tewksbury LA, Kalogeraki V. Eternal: Fault tolerant and live upgrades for distributed object systems. In: Proc. of DARPA Information Survivability Conf. and Exposition (Discex 2000). Hilton Head: IEEE Computer Society, 2000. 184−196. [doi: 10.1.1.2.9981]

[19] Bidan C, Issarny V, Saridakis T, Zarras A. A dynamic reconfiguration service for CORBA. In: Proc. of the 4th Int'l Conf. on Configurable Distributed Systems. Annapolis: IEEE Computer Society, 1998. 35−42. [doi: 10.1109/CDS.1998.675756]

[20] Almeida JPA, Wegdam M, van Sinderen MV, Nieuwenhuis L. Transparent dynamic reconfiguration for CORBA. In: Proc. of the 3rd Int'l Symp. on Distributed Objects and Applications (DOA 2001). Washington: IEEE Computer Society, 2001. 197−207. [doi: 10.1109/DOA.2001.954085]

[21] Balasubramanian K, Wang N, Gill C, Schmidt DC. Towards composable distributed real-time and embedded software. In: Proc. of the 8th IEEE Int'l Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003). Guadalajara: IEEE Computer Society, 2003. 226−233. [doi: 10.1109/WORDS.2003.1218087]

[22] Bialek RP. The architecture of a dynamically updatable component based system. In: Proc. of the 26th Annual Int'l Computer Software and Applications Conf. (COMPSAC 2002). Oxford: IEEE Computer Society, 2002. 1012−1016. [doi: 10.1109/CMPSAC.2002.1045139]

[23] Bialek RP. Dynamic updates of existing Java applications [Ph.D. Thesis]. Denmark: Department of Computer Science, Faculty of Science, University of Copenhagen, 2006.

[24] Oreizy P, Medvidovic N, Taylor RN. Architecture-Based runtime software evolution. In: Proc. of the 20th Int'l Conf. on Software Engineering. Washington: IEEE Computer Society, 1998. 177−186. [doi: 10.1109/ICSE.1998.671114]

[25] Magee J, Kramer J. Dynamic structure in software architectures. ACM SIGSOFT Software Engineering Notes, 1996,21(6):3−14. [doi: 10.1145/250707.239104]

[26] Dou L. Reseach on dynamic reconfiguration technology in component-oriented complex software system [Ph.D. Thesis]. Changsha: National Defense Science and Technology University, 2005 (in Chinese with English abstract).

[27] Li CY, Li Y, Wu J, Wu CH. A service-oriented software model supporting dynamic evolution. Chinese Journal of Computers, 2006, 29(7):1020−1028 (in Chinese with English abstract).

[28] Ma XX, Yu P, Tao XP, Lü J. A service-oriented dynamic coordination architecture and its supporting system. Chinese Journal of Computers, 2005,28(4):467−477 (in Chinese with English abstract).

[29] Zhang S, Huang LP. Dynamic service evolving based on OSGi. Journal of Software, 2008,19(5):1201−1211 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/19/1201.htm [doi: 10.3724/SP.J.1001.2008.01201]

[30] Wu JK, Huang LP, Wang DJ. ASM-Based model of dynamic service update in OSGi. ACM SIGSOFT Software Engineering Notes, 2008,33(2):1−8. [doi: 10.1145/1350802.1350815]

[31] Liskov B, Moh CH, Richman S, Shirira L, Cheung Y, Boyapati C, Boyapati R. Safe lazy software upgrades in object-oriented databases. Technical Report, TR-851, MIT Laboratory for Computer Science, 2002. 1−13.

[32] Boyapati C, Liskov B, Shrira L, Moh CH, Richman S. Lazy modular upgrades in persistent object stores. ACM SIGPLAN Notices, 2003,38(11):403−417. [doi: 10.1145/949343.949341]

附中文参考文献:

[3] Gray J, Reuter A,著;孟小峰,于戈,等,译.事务处理概念与技术.北京:机械工业出版社,2004.

[4] 罗俊伟,秦晓,陈思功.普适计算中基于上下文触发的事务模型.小型微型计算机系统,2004,25(8):1542−1545.

[5] 王德俊.面向服务的分布式系统动态更新研究[博士学位论文].上海:上海交通大学,2010.

[6] 王德俊,黄林鹏,徐小辉,伍建焜,张仕,王欣.分布式动态更新支持系统:研究综述.计算机科学,2007,34(11):19−25.

[11] 李必信,王云峰,李宣东,郑国梁.面向对象的类型-子类型分析及推理规则.计算机科学,1999,26(7):23−28.

[13] 廖渊.普适计算环境下一种基于 QoS 的服务构件组合方法[博士学位论文].北京:中国科学院软件研究所,2005.

[15] 杨欢,欧家凡,黄林鹏.网格环境下多副本带状数据传输的实现.计算机应用与软件,2006,23(11):54−56.

[17] 王晓鹏,王千祥,梅宏.一种面向构件化软件的在线演化方法.计算机学报,2005,28(11):1890−1897.

[26] 窦蕾.面向构件的复杂软件系统中动态配置技术的研究[博士学位论文].长沙:国防科学技术大学,2005.

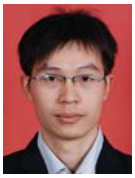
[27] 李长云,李莹,吴建,吴朝晖.一个面向服务的支持动态演化的软件模型.计算机学报,2006,29(7):1020−1028.

[28] 马晓星,余萍,陶先平,吕建.一种面向服务的动态协同架构及其支撑平台.计算机学报,2005,28(4):467−477.

[29] 张仕,黄林鹏.基于 OSGi 的服务动态演化.软件学报,2008,19(5):1201−1211. http://www.jos.org.cn/1000-9825/19/1201.htm [doi: 10.3724/SP.J.1001.2008.01201]

王德俊(1972－),女,江苏江都人,博士,讲师,主要研究领域为分布式系统,程序设计语言,软件动态更新.

黄林鹏(1964－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为分布式计算,程序设计语言.



徐小辉(1979－),男,博士生,主要研究领域为服务计算,软件更新,形式化验证与分析.