

文章编号:1001-9081(2006)08-1909-03

基于组件的动态软件更新

鲍春健,吴俊敏,许胤龙,李黄海,王文韬

(中国科学技术大学 计算机科学技术系,安徽 合肥 230027)

(baochj@mail.ustc.edu.cn)

摘要:在分析影响动态软件更新的关键问题的基础上,从组件模型的设计出发来寻求系统架构对动态更新的支持,并利用动态插入拦截器来获取更新的安全点,从而实现了动态组件的热交换,其主要过程包括插入拦截器、阻塞调用、状态转移和请求重定向。基于该思想实现的集群高可用应用开发平台具有更好的可用性、可扩展性和运行性能。

关键词:动态更新;基于组件;拦截器

中图分类号:TP311.53 **文献标识码:**A

Component-based dynamic updating software system

BAO Chun-jian, WU Jun-min, XU Yin-long, LI Huang-hai, WANG Wen-tao

(Department of Computer Science and Technology, University of Science and Technology of China, Hefei Anhui 230027, China)

Abstract: Based on the analysis of the key issues that affect dynamic updating, a component-based dynamic updating mechanism which obtained a safe updating point by interposing interceptor dynamically was presented, and the hot-swapping of dynamic components was achieved. The main procedure included interposing interceptor, blocking invocation, transferring state and redirecting requests. This design makes the application has better feasibility, expansibility and operation performance.

Key words: dynamic updating; component-based; interceptor

0 引言

传统的软件更新步骤包括停止需要更新的系统,执行更新程序,然后再重新启动程序。然而,很多的应用要求连续运行,例如银行和通讯软件的停工代价就很大,每年仅有几分钟的停工时间。而对于航空交通管理和生命支持系统来说,停止运行会产生更为严重的后果,所以这些系统一般来说是不允许停止服务的。动态软件更新是指软件程序在更新其部分的时候不需要停止运行服务,且被更新部分对于其他部分而言是透明的,它可以使得程序的维护成本下降,减轻软件公司繁琐的维护工作,提高了工作效率。

软件更新的原因多种多样,根据其更新目的可以分成如下几种:1)纠正软件错误。用一个新组件替换掉原来运行时出现错误的组件,且这个新组件提供与原组件相同的功能;2)适应新执行环境。为了适应一些外部环境的变化而做的更新,如硬件、操作系统和虚拟机等的变化;3)扩展软件功能。为了满足新的需求而增加新的组件来扩展原来应用程序的功能,增加了原来应用程序的生命期;4)完善应用程序。即使是应用程序运行正确,也可以使用更好的(例如效率更高、界面更友好、速度更快)组件来替换原来的组件,从而不断增强应用程序的鲁棒性。

允许系统软件对环境变化做出积极反应的传统做法是使用适应性代码^[1],但它有三个主要缺点:需要预知各种运行环境、较高代码复杂度和较高性能开销。本文在分析影响动态更新的关键问题的基础上,利用动态插入拦截器技术实现

了基于组件的动态更新,基于该思想实现的应用软件具有更好的可用性、可扩展性和运行性能。

1 影响动态更新的关键问题^[2]

要实现软件系统的动态更新,就必须在软件的设计过程中预先考虑一些特定的限制,而且这些限制不能影响到系统最基本的性能。

1)确定最基本可更新单元。选取的更新单元要有良好的模块性且模块边界清晰。根据支持更新的类型和系统实现,这种可更新单元可能是一个代码模块,或者是代码和被封装数据。该单元必须有清晰的外部接口,外部代码必须通过这些接口来访问代码和数据,代码或数据不能被直接访问。

2)获取更新最佳时机。如果动态更新发生在系统执行过程中的关键时期,那么系统就会崩溃,这个和并发系统中的资源竞争有着相同的原因。例如:当系统的状态正在发生改变的时候,更新就不应该发生。因此,决定何时更新可以安全执行至关重要。然而,一般来说这个更新是无法知道的,所以必须要有系统架构的支持。通常的解决方案包括要求软件系统对于更新点是可编程的,或者可以检测到什么时候系统是空闲的或者休眠的。

3)状态转移机制。在更新过程中,对于有状态的更新单元需要进行状态转移。除非系统的可更新单元中不包含状态,否则必须有一种机制来传送当前的状态,以便更新单元能够由它的替代单元从某个状态来继续执行。

4)请求重定向。在新的单元被安装且状态转移完毕之

收稿日期:2006-02-13;修订日期:2006-05-15

基金项目:中国科学技术大学青年基金资助项目(KA1125);中国科学院高水平大学建设基金资助项目(KY2706)

作者简介:鲍春健(1980-),男,安徽枞阳人,硕士研究生,主要研究方向:分布式系统; 吴俊敏(1973-),男,安徽太湖人,讲师,博士,主要研究方向:并行计算机体系结构、分布式系统; 许胤龙(1963-),男,安徽庐江人,教授,博士生导师,主要研究方向:并行计算、组合优化算法、网络路由算法; 李黄海(1981-),男,江苏启东人,硕士研究生,主要研究方向:并行与分布式系统; 王文韬(1982-),男,安徽六安人,硕士研究生,主要研究方向:计算机网络技术。

后,系统必须保证所有未来的对原单元的调用重定向到新单元上去。要实现这种引用重定向,一个最简单的办法就是实现组件之间访问的间接引用。

2 基于组件的动态更新

随着软件复用技术的日臻成熟,利用可复用组件构造软件系统已成为软件开发的主要手段。组件是指具有某种特定功能的软件模块,拥有清晰的边界,透过接口与外界沟通,能够在系统中部署和替换。因此,选取组件作为动态软件更新的最基本可更新单元是可行的。本节从组件模型的设计出发,寻求软件系统框架对组件动态更新的支持,并利用动态插入拦截器来获取更新的安全点,从而实现了动态组件的热交换,其主要过程包括插入拦截器、阻塞调用、状态转移和请求重定向。

2.1 组件模型的设计

在图1中,组件在软件系统框架上进行服务注册,调用者通过提供服务接口名来询问软件系统框架,从而获取一个已注册服务的访问。在该组件模型中,每个组件被划分为管理器和功能对象两个部分。功能对象为组件的可替换部分,组件的更新过程就是替换其功能对象部分。管理器是软件支持组件技术的主要机制,它为组件提供独立运行环境,使组件的开发和管理更加灵活方便。作为组件功能对象和访问者之间的中间层,管理器负责对组件的间接访问、生命周期管理、事务协同、持久性维护和身份认证等。

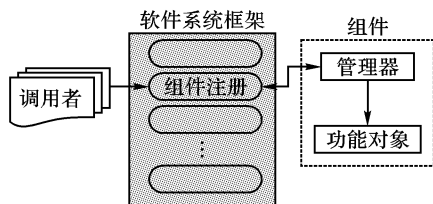


图1 支持动态更新的组件模型

另外,管理器需要记录当前正在访问组件的引用计数,主要作用有:1)在更新过程中为更新程序提供组件被引用信息,使其进入更新的不同阶段;2)当组件注销时,若该组件的引用计数非零,就需要注册一个回调函数,等待引用计数为零时调用并释放组件,这样可以避免造成悬空引用。

2.2 拦截器的动态插入

作为被广泛采用的设计模式^[4],拦截器用来拦截和处理客户和服务器之间的消息,多个拦截器被组成一个拦截器队列,系统根据应用程序的预配置策略逐个处理消息。拦截器开始被用于执行安全服务,很快就被发现,它可以被用于限制需求的许多其他实现。对于需要进行动态更新的软件系统,使用拦截器技术就很容易获取更新的安全点。

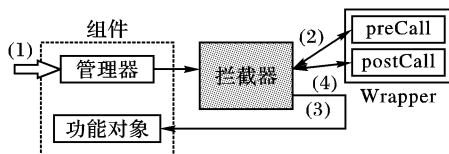


图2 拦截器的插入

基于2.1节组件模型,在组件的管理器和功能对象之间插入拦截器,如图2所示。Wrapper是一个标准的C++对象,preCall将在原对象函数调用之前被调用,postCall在原对象函数调用之后被调用,这样可以在功能对象调用过程中,收集相关数据信息,维护函数调用状态。例如,记录当前通过拦截器访问组件的引用计数,更新时阻塞调用等。

```
//interceptor
public interface Interceptor
{
    ...
    void intercept( invokeRequest, component)
    { insert( invokeRequest, queue); } //intercept invoke threads
    void resume( component)
    { foreach ( item in queue)
      { invoke( component, invokeRequest); }
    } //resume all invoke threads
}
```

2.3 组件的动态更新过程

如图3所示,更新过程包括六个阶段:a)更新之前,调用者通过组件管理器调用旧版本的组件接口;b)插入拦截器,它将跟踪随后的所有调用(虚线箭头表示),并继续传递到旧版本,直到拦截器插入之前的所有调用均返回(实线箭头表示);c)一旦拦截器插入之前的所有调用完成,就表明该组件当前的所有调用都被跟踪,开始阻塞新的调用,并等待已经被跟踪的调用完成;d)一旦之前的所有调用完成,组件进入了静止状态,调用状态转移函数将旧版本状态信息传送给新版本(由于不同组件的数据集和数据形式不同,很难提供一种统一的状态转移协议,本文将协议的具体实现留给组件开发者);e)将组件管理器中的组件指针指向新版本,随后的调用将直接访问新版本组件,并唤醒之前被阻塞的调用请求重定向到新版本上;f)销毁拦截器和旧版本组件。

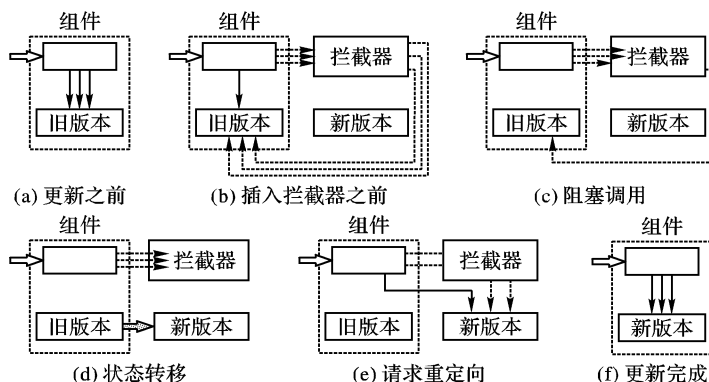


图3 组件的动态更新过程

3 可动态更新的高可用开发平台

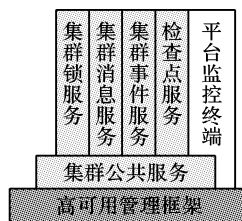


图4 高可用应用开发平台架构

3.1 体系结构

基于以上设计思想,兼容服务可用性论坛SAF(Service Availability Forum)的应用接口规范AIS(Application Interface Specification)^[5],实现了一个可动态更新的集群高可用应用开发平台。该平台提供一个灵活的高可用编程环境,使得应用程序的开发只需要集中于具体的业务流程上,而不需要过多地考虑其他高可用特性。

如图4所示,平台底层为高可用管理框架,上层各具体服务模块通过向其进行组件注册提供服务。集群公共服务包括底层组通信、集群成员管理、命名服务、安全服务等,它为上层具体服务模块提供基本服务支持。

3.2 动态更新的实现

为了在高可用平台上实现组件动态更新,在其基础上增加了平台监控终端。平台监控终端负责系统配置、动态监控和发起动态更新。在上述体系结构中,将平台的每个服务看成一个组件,将每个组件分成组件管理器和功能对象两部分。

在高可用平台上实现动态更新主要有两个问题:1)更新时机的选择,采用引用技术的方式来解决,即组件的更新时机是在旧组件的引用计数为0的时刻开始。若组件是有状态组件,则执行状态转移操作,若组件是无状态组件,则执行新旧组件的替换操作。2)拦截器的插入,采用接口注入的方式,为每个要更新的组件通过配置文件的形式来插入拦截器,并使用命令的方式来完成拦截器的动态插入;当不再需要拦截器时,可将其从配置文件中删除,并通过命令来通知组件。

组件更新过程的伪代码如下:

```
//component
class component
{ ...
    void addInterceptor( interceptor)
    { this. interceptor = interceptor; }
    void doUpdate( compEntity)
    { saveCpStateInfo();
      this. compEntity = compEntity;
      restoreCpStateInfo();
      this. interceptor. resume( this);
    }
    void service();
}
//update thread
void main()
{ ...
    while( true)
    { if ( component needs to be updated)
      { component. addInterceptor( interceptor); }
      while( component. referCount!=0);
      component. doUpdate( newCompEntity);
      component. state = NORMAL;
    }
}
```

可动态更新使得该集群高可用应用软件具有更好的可用性、可扩展性和运行性能:1)当平台监控终端检测到安全漏洞、软件错误和性能异常时,基于组件的动态更新可以很容易对当前系统中已部署组件进行动态修改,而不需要停止整个高可用开发平台,提高了软件的可用性;2)在具体服务模块中,不同算法适用于不同的运行软硬件环境。传统适应性策略通过实时监控来选取现有算法集合中的最优者,而基于热交换的组件动态更新将各种适应性算法模块化,从而降低了组件设计的复杂度且允许算法运行时更新和扩展;3)平台监控终端能够在面对系统异常时提供相关信息,有利于系统的维护和管理,但这也为正常情况下引入了不必要的开销,若能提供动态监控,就可以减少这种开销。动态更新中的插入技术允许监控代码在合适的时间和地点被加入,在不需要的时候删除。

4 相关工作

文献[6]提出了一种基于软件体系结构的动态组件更新方法 SOFA/DCUP (SOFTware Appliances/Dynamic Component UPdating)。在该方法中,组件被划分为永久部分和可替换部分,相应地提供控制操作和功能操作,更新组件就是在运行时替换其可替换部分。该方法的优点在于其强大的分级模型和对象之间的重定向机制,简化了组件之间相互引用的修改。其缺点是:功能对象没有隐藏到组件提供的接口的后面,即组件边界不容易确定,违反了信息封装原则;更新的粒度不明确,若当全局组件的子组件需要调整时,整个全局组件都将受到影响,且其下所有可替换组件都要重新部署。

开放服务网关促进会 OSGi (the Open Service Gateway Initiative)^[3] 制定了一个支持运行时更新的 Java™ 组件框架。在 OSGi 中,所有组件的管理信息都由系统框架来维护和管理。组件通过注册插入到该框架中,因此,应用程序可以在运行时增量式开发。在 OSGi 中,组件被称为束,可动态地安装、激活、停止和卸载。OSGi 的优点在于它提供了一个高效的开发环境,组件可以在运行时被增加和更新。其缺点是在更新的过程中编程者必须能够有效地停止正在运行的对象并释放资源,容易导致悬空引用。另外,更新过程中 OSGi 没有考虑组件的状态。

文献[7]提出了一种可动态更新的组件系统 DUCS (Dynamically Updatable Component-based System),支持分布式组件应用程序的动态更新。DUCS 有一个层次的体系结构,运行于一个扩展的虚拟机上,该虚拟机支持对象的卸载和替换。DUCS 是一个层次框架,对应用程序的实现影响很小。它支持动态组件替换,组件间的状态转移,接口的修改和更新的自动传播。其缺点是这种扩展的虚拟机的实现较为复杂。

本文提出的模型使用组件作为更新对象,通过组件注册实现运行时的增量开发和动态更新。将组件分为管理器和功能对象,使功能对象对调用者来说是透明的,满足了信息封装的原则。利用动态插入拦截器技术,使编程者不必关心运行组件的激活/停止和相关资源的释放,避免了悬空引用,也使得组件的动态更新对于组件编程者来说也是透明的。利用拦截器技术可以明确确定动态更新发生的时刻,为实时监控动态更新过程和状态转移过程提供了技术支持,从而更为简单地实现了组件更新、状态转移、接口修改和更新的自动传播。

5 结语

动态更新技术在软件设计中的应用广泛,最典型应用包括在线补丁、优化适应性算法、软件测试和动态监控等。在线补丁使得软件系统能够在运行时弥补安全漏洞,修复软件错误和性能异常;基于组件的动态软件更新使得传统适应性代码能更好地适应各种变化的运行时环境;动态插入能够在软件测试过程中减少异常情况代码的负面影响,简化测试过程;动态监控可以避免正常情况下引入的额外开销。本文在分析影响动态更新技术的关键问题的基础上,利用拦截器技术实现基于组件的动态软件更新,使得应用软件具有更好的可用性、可扩展性和运行性能,增强了软件系统的自适应能力。

参考文献:

- [1] GLASS G, CAO P. Adaptive Page Replacement Based on Memory Reference Behavior[A]. Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems[C]. Seattle, WA, ACM Press, New York, 1997. 115 - 126.
- [2] SOULES CAN, APPAVOO J, HUI K, *et al.* System support for online reconfiguration[A]. Proceedings of the 2003 USENIX Technical Conference[C]. San Antonio, TX, USA, 2003. 141 - 154.
- [3] Open Service Gateway Initiative[EB/OL]. <http://www.osgi.org>, 2006.
- [4] OMG. corba23_book[EB/OL]. <http://www.omg.org>, 1999 - 10.
- [5] SA Forum Application Interface Specification B.01.01[EB/OL]. [http://www.saforum.org/specification/AIS Information/](http://www.saforum.org/specification/AIS%20Information/), 2006.
- [6] PLASIL F, BALEK D, JANECEK R. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating[A]. Proceedings of ICCDS'98[C]. Annapolis, Maryland, USA, IEEE CS Press, 1998.
- [7] BIALEK RP. The architecture of a dynamically updatable, component-based system[A]. Workshop on Dependable On-line Upgrading of Dist. Systems in conjunction with COMPSAC 2002[C]. Oxford, England, 2002.