ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.9, September 2008, pp.2167−2180
DOI: 10.3724/SP.J.1001.2008.02167
© 2008 by Journal of Software. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于反模式的中间件应用系统性能优化*

兰 灵, 黄 罡+, 王玮琥, 梅 宏

(北京大学 信息科学技术学院 软件研究所,北京 100871)

Anti-Pattern Based Performance Optimization for Middleware Applications

LAN Ling, HUANG Gang+, WANG Wei-Hu, MEI Hong

(Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

+ Corresponding author: E-mail: huanggang@sei.pku.edu.cn

Lan L, Huang G, Wang WH, Mei H. Anti-Pattern based performance optimization for middleware applications. *Journal of Software*, 2008,19(9):2167−2180. http://www.jos.org.cn/1000-9825/19/2167.htm

Abstract: This paper presents an approach to optimizing performance of middleware applications based on anti-pattern. This approach has three major features: First, a meta-model is offered to build more understandable and formalized representation of anti-patterns; second, the detection of anti-patterns is based on both the static and the dynamic information, which is retrieved at runtime; third, refactorings operate without interrupt the running systems, and is completed in an automated way with the help of the middleware. A prototype based on J2EE has been developed and an e-bookstore is used as a running example to illustrate the ideas introduced in this approach.
Key words: anti-pattern; middleware; performance optimization; detection; refacotring

摘 要: 提出了一种基于反模式的中间件应用系统的性能优化方法.该方法的主要特点包括:建立了反模式元模型以使得反模式的表示更加准确;综合利用系统的静态信息和动态信息以检测运行系统中存在的反模式;系统重构在不中断应用系统运行的前提下在线执行,并在中间件的辅助下自动/半自动地完成.在 J2EE 平台上实现了支持该方法的原型系统,并通过网上书店的实例展示了该方法的有效性.
关键词: 反模式;中间件;性能优化;检测;重构
中图法分类号: TP391 **文献标识码**: A

模式是人们所熟知的一种经验的总结,它描述了对一些可重复出现问题的有效解决方案[1].作为对模式概念的一种扩展,反模式描述的是一个普遍采用的不良解决方案,该方案将会给应用系统带来负面的影响[2].为了去除反模式导致的负面影响,通常需要对应用系统进行重构,即在不改变系统行为的前提下,改变系统内部结构[3].完整的反模式描述中,最重要的部分是两个解决方案:会带来负面影响的反模式解决方案;相应的重构方案.反模式可以指导人们提高软件的质量属性,包括可扩展性、可维护性以及软件性能等.特别地,它可以帮助人们查找系统中导致性能低下的根源并提供相应的优化方案.文献[4−8]介绍了一些与性能相关的反模式.

* Supported by the National Natural Science Foundation of China under Grant Nos.90612011, 90412011, 60403030 (国家自然科学基金); the National Basic Research Program of China under Grant No.2005CB321800 (国家重点基础研究发展计划(973)); the Fok Ying Tong Education Foundation (霍英东教育基金)

Received 2007-03-26; Accepted 2007-06-30

© 中国科学院软件研究所 http://www.jos.org.cn

目前,对于反模式的研究主要停留在反模式的收集和整理阶段,缺少有效的方法或者工具来指导或帮助人们正确、高效地使用反模式.人们一般按照如下的步骤检测反模式并重构应用系统:首先,通过对源代码以及设计文档的分析,检测出应用系统中存在的反模式;然后,确定重构方案;最后,修改应用系统的源代码,并重新编译、部署应用以使得重构生效.然而,通过上述的步骤对应用系统进行性能优化时,会存在如下一些问题:

- 反模式的检测与系统重构仅仅依赖于源代码.首先,在对应用系统的维护与优化过程中,并不能够保证得到源代码.在目前的方法中,没有源代码将很难检测到应用系统中的反模式或者对应用系统进行重构.但是随着软件构件技术的发展,目前的许多应用系统都采用了基于构件的开发技术,并通过组装不同厂商提供的商用构件(commercial off-the-shelf,简称 COTS)进行开发.这意味着获得应用系统完整的源代码或者设计文档非常困难.其次,仅限于对源代码的分析将无法检测到许多目前已知的性能反模式.从性能优化的角度来看,由于软件性能是一个运行时刻的概念,因此,大部分性能相关反模式的检测都需要考察某些运行时刻的信息,如构件之间的交互频率、构件实例消耗的内存等.然而,这些运行时刻信息无法通过分析源代码得到;最后,基于源代码的重构可能会引发其他问题.一般来说,性能优化会使应用系统变得更加复杂,使其更难以理解,在提高性能的同时导致应用系统其他质量属性降低.
- 系统重构往往是离线执行的.目前的重构是修改源代码,而不是直接修改运行时刻的应用系统,要使重构生效,应用系统需要经过“停止-更新-重启”的步骤,我们称这样的重构方式为离线重构.对于目前具有高可用性的应用系统而言,能够提供 7(天)×24(小时)的不中断服务已经成为很普遍的需求.然而,由于对应用系统的离线重构将不得不中断应用系统的正常运行,因此无法满足应用系统的高可用需求.
- 系统重构需手工完成.在确定重构方案以后,需要系统开发或管理人员手工地完成重构工作.尽管目前有一些开发工具能够在一定程度上减少手工重构的工作量,但重构依然会给开发管理人员带来不小的负担.此外,已经进行的重构并不能持续保证正确有效.由于管理人员的失误,也许会实施不合适的重构方案;或者由于运行环境或者用户需求的变化,以前进行的重构方案已经不再适合当前的具体情况.在这种情况下,可能需要对重构进行回滚,即让应用系统恢复到该次重构以前的状态.但是对于手工进行的重构而言,要进行重构回滚会非常困难.

综上所述,软件性能的特性以及目前应用系统的实际需求要求能够在系统的运行时刻应用反模式:对反模式的检测不应该仅依赖于源代码的分析,而应该更多地考虑运行时刻的信息;对应用系统的重构,应该在不中断正常运行的前提下完成,即实现运行时刻的在线重构.为了满足上述需求,底层平台需要提供相关的支持,以监测应用系统的状态并且在线调整其行为.在目前大型分布式应用系统的开发过程中,将应用系统搭建在中间件之上已经成为一种最普遍的解决方案,中间件主要解决的是分布式通信以及屏蔽底层平台的异构性问题.而随着中间件技术的发展,现在的中间件除了完成其基本功能外,还能够在运行时刻提供应用系统的动态信息,并在一定程度上调整其行为.由此可见,中间件可以作为运行时刻应用反模式的支撑机制.

针对目前方法的局限性,本文将中间件技术引入到反模式应用中,提出了一种运行时刻基于反模式的中间件应用系统性能的优化方法,并在 J2EE 应用服务器 PKUAS[9]上实现了一个原型系统.与传统的反模式应用方式相比,该方法主要有 3 点改进:1) 通过分析运行时刻的信息而不是仅仅分析源代码来检测应用系统中的反模式;2) 由中间件在运行时刻进行系统在线重构,无须修改应用系统,由此也不会导致中断应用系统的正常运行;3) 反模式的检测和系统重构由中间件自动/半自动地完成.

本文第 1 节概述整个方法框架.第 2 节介绍反模式检测.第 3 节介绍应用系统的在线重构.第 4 节演示该方法的使用以及性能优化效果.第 5 节与相关工作进行比较.最后一节总结全文并展望将来的工作.

1 基于反模式的性能优化过程

图 1 展示了该方法的过程图.该方法一共包括两个子过程:一个是管理反模式的过程,另一个是进行性能优化的过程.

© 中国科学院软件研究所 http://www.jos.org.cn

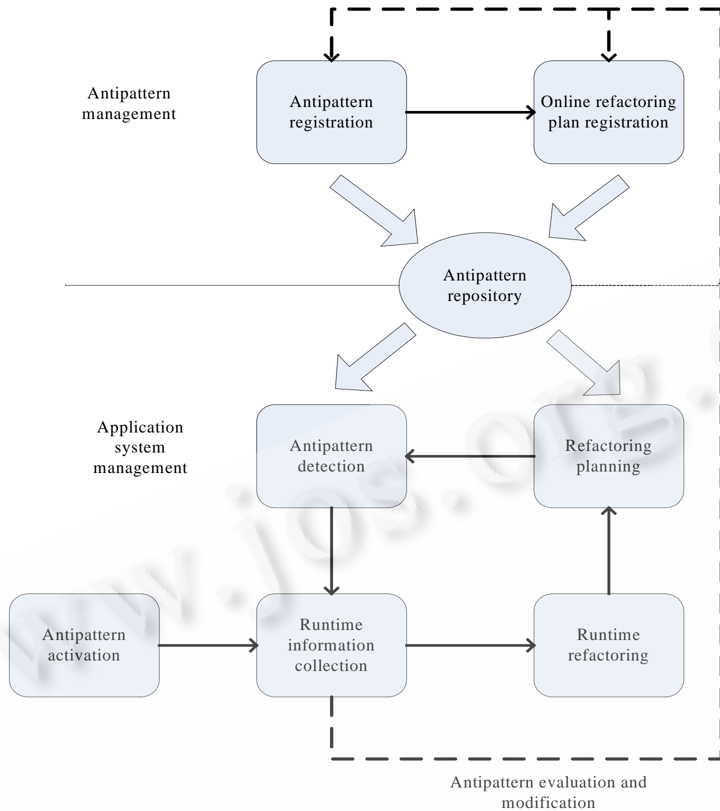

Fig.1 Processes of anti-pattern based optimization

图 1 基于反模式的性能优化过程

本节首先介绍反模式的管理过程,它主要分为两个步骤:

1. 反模式注册:为了检测应用系统中存在的反模式并根据该反模式提供的重构方案对系统进行重构,首先必须注册这些待检的反模式.对一个反模式的定义需要包括如下一些部分:判断规则,用于检测运行系统中指定的反模式;重构方案,用于指导系统的重构工作以提高性能.这些反模式定义将被保存在一个反模式库中,以便在进行性能优化过程中使用或更新.

2. 在线重构方案注册:在第 1 个步骤中注册的重构方案是离线的重构方案,通过对应用本身的调整来完成重构,该步骤一般是由领域专家来完成的.我们的方法不仅支持运行时刻的反模式检测,还支持运行时刻的系统在线重构.与离线重构相比,在线重构直接作用在运行系统之上,而不是作用在源代码之上,这些重构工作大部分可以在中间件的帮助下自动或者半自动地完成.在线重构模型应该由对中间件熟悉的专家完成,并添加到反模式库中.

该方法中第 2 个过程是基于反模式的性能优化过程,它分为如下几个步骤:

(1) 反模式激活:在实际的性能优化过程中,考虑到反模式检测的代价,如采集运行时刻信息导致的系统资源消耗等,我们的方法并不会尝试去检测所有已经注册的反模式.在反模式库中,反模式有两种状态:激活与去活状态.注册的反模式的默认状态是去活状态,用户可以设置反模式在这两种状态之间进行切换.在之后的反模式检测步骤中,系统只会尝试去检测已激活的反模式.在这个步骤中,用户的任务就是选择需要检测的反模式.用户可以根据系统当前的运行状态或者工作负载来决定需要检测哪些反模式.举例来说,我们可以在工作负载很低、系统空闲的情况下,对所有的反模式进行检测.

(2) 运行时刻信息采集:在对应用系统的维护与优化过程中,并不能保证得到完整的源代码以及设计文档.

© 中国科学院软件研究所 http://www.jos.org.cn

因此,本方法中的反模式检测综合分析了静态信息与运行时刻的动态信息.静态信息主要是指应用部署包中的一些部署描述信息.运行时刻的动态信息包括构件之间的方法调用、远程客户端的 IP 地址等.这些信息可以在不修改应用系统的前提下,由中间件自动地进行采集.

(3) 反模式检测:在采集到足够的信息以后,就可以根据反模式库中的反模式定义进行反模式检测了.在检测之前,已经采集到的原始运行信息需要进行一定的预处理.与已激活的反模式相关的信息需要进行萃取与重新组织.反模式检测的相关内容将在第 2 节进行更详细的介绍.

(4) 重构规划:在反模式被检测出来以后,并不是马上就进行系统重构,在此之前还需要进行一些权衡工作.一些情况下,一个反模式可能有不止一种重构方案,此时,我们就应该根据一些因素,比如重构方案的代价、用户的具体需求来选择最适合的重构方案.另外,在同一个运行系统中,如果同时检测到多个反模式,这些反模式的重构方案之间可能会存在冲突.在这种情况下,我们需要在这些反模式以及与它们相关的重构方案作一个权衡,以得到尽量优化的解决方案.

(5) 运行时刻系统重构:通过中间件可以在一定程度上调整应用系统的行为,我们的方法支持的是通过中间件的帮助,在运行时刻对应用系统进行重构.重构完成后,通过对当前应用系统的运行情况监测,有可能会进行重构回滚,使应用系统恢复到重构之前的状态.关于重构相关的内容将在第 3 节进行更详细的介绍.

(6) 反模式评估和修正:在重构完成以后,性能优化的过程并没有结束,需要通过系统重构前后的性能比较,对反模式库中的反模式以及重构模型进行评估,并在需要的情况下进行修正,使以后的性能优化达到更好的效果.

2 反模式检测

2.1 反模式模型

一个完整记录的反模式应该包含如下一些内容:应用系统的状态、造成该状态的原因、该状态引发的不良后果、相应的重构方案等.在进行反模式检测时,需要检查当前应用系统的状态是否与反模式定义的状态一致,从而确定该应用系统中是否存在该反模式.我们通过对目前已知的性能相关的反模式进行调研,将检测反模式所需的应用系统状态信息分为如下 3 类:

- 件信息:包括构件类型(如 EJB/Servlet/POJO 等)、构件方法(如方法名称、方法个数等)、构件配置(如实例池大小等)、构件操作(如构件回调方法等);
- 件交互信息:包括方法调用类型(如同步/异步调用等)、方法调用参数(如参数大小等)、方法调用序列;
- 境信息:包括中间件服务配置信息(如事务过期时间、线程池大小等)与底层硬件信息(如内存使用情况、网络拓扑结构等).

在上述信息中,除了某些与应用逻辑相关的构件操作以外,其余信息都是可以由中间件自动获取的,因此,大部分的反模式都可以自动检测.但是,目前的反模式几乎都是使用自然语言来描述的.使用这种描述方法使得人们很难精确地表示它,更不可能实现自动化检测.我们给出了一个基于 MOF[10]的元模型,它可以表示上述这 3 类的应用状态信息.与自然语言相比,使用这个元模型可以更加准确地表示反模式.而且,用户可以使用图形化的方式对反模式模型进行定义、编辑,使得记录反模式更加方便,图形化的反模式模型也更利于理解.图 2 展示了该元模型的简化版本,它一共包含 5 类实体,其中:Component 代表构件;Call 代表构件之间的调用关系;Method 代表调用的方法;Service 代表中间件服务;Host 代表物理主机.这 5 类实体还包含若干相应的属性,而实体之间可能存在着各种关联关系,由于篇幅所限,这里不再进行详述.目前,我们提供的反模式元模型可以描述 J2EE 领域的反模式.

© 中国科学院软件研究所 http://www.jos.org.cn

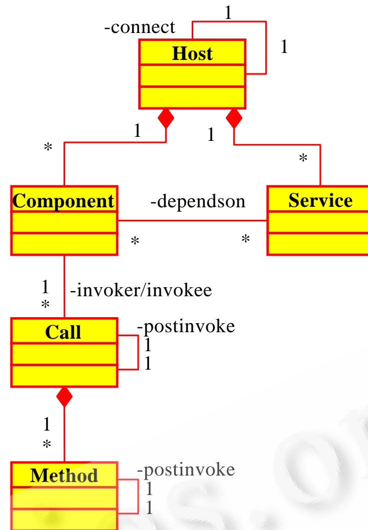

Fig.2 Meta-Model of anti-pattern

图 2 反模式元模型

2.2 反模式检测

在进行反模式检测之前,需要先采集应用系统相关的信息,包括静态信息与运行时刻的动态信息.用户将根据待检测的反模式模型来确定需要采集当前应用系统的哪些状态信息.应用的静态信息,比如构件名称、构件类型等可以从部署描述符里得到.而应用的运行时刻信息,我们通过给中间件添加截取器的方式来进行采集.截取器会把用户关心的运行时刻信息写到日志文件中.这些截取器实现了分布式系统中常见的截取器模式[11],可以在不被应用感知的情况下添加/删除,而且可以动态地触发.通过这种方式,我们就可以很方便地定制采集的信息量以及时间段.

在采集到应用系统信息之后,就可以进行反模式的检测工作:将应用系统信息与反模式模型进行匹配,以检测出当前运行系统中存在的反模式.对于构件信息与环境信息的匹配,可以通过简单的遍历来查找与模型匹配的状态信息,比如,为了找出 public 方法个数少于 2 个的 EJB,可以通过遍历所有的构件完成.而对于构件交互信息匹配,比如找出存在指定调用序列的两个构件,由于一般情况下构件之间的调用数目都很庞大,无法通过遍历来完成反模式检测,因此我们采用了数据挖掘技术来完成这项任务.通过使用相关规则算法[12],可以查找出符合一定条件的调用序列,从而完成此类反模式的检测[13].

3 系统重构

3.1 系统重构模型

与检测反模式时所需的状态信息类似,反模式对应的重构动作也可以分为如下 3 类:

- 构件重构:包括添加/删除构件、修改构件属性、修改构件类型等;
- 构件交互重构:包括修改调用类型、修改调用参数、重构调用序列等;
- 环境重构:包括修改中间件服务属性等.

系统重构模型同样使用第 2.1 节中介绍的元模型进行构建,它描述了系统经过重构后的状态.用户需要定义一组重构操作从反模式模型演化为重构模型.重构操作可以分为两类:一类是原子操作,包括 ADD,DEL,MOD 这 3 个基本的重构操作,它表示对模型中的实体或者实体之间的关系进行添加或删除,以及实体属性的修改,比如添加一个构件、删除两个构件之间的调用、修改服务的配置等;另一类是复合操作,它由一组预设的原子操作组成,而通过这一组原子操作可以为系统中引入一些可以提高性能的模式.此外,针对一些重构操作,我们给

© 中国科学院软件研究所 http://www.jos.org.cn

出了重构约束以保证这些操作的正确性.重构约束分为前置条件与后置条件两类,分别对系统重构前后的状态进行约束,只有满足这些条件,重构操作才可以执行.

上述的重构模型是离线的重构模型,它可以用于指导用户对应用系统进行重构.但是,该类重构需要应用系统通过“停止-更新-重启”的步骤才能使所做的重构生效.为了达到不中断应用系统运行的目的,需要底层机制支持进行在线的重构.目前的中间件不仅为应用系统提供了一个运行平台,而且还提供了调整机制来完成应用系统的某些重构工作,它可以为在线重构提供支持.但是,由于重构模型中的重构动作是对应用自身的调整,而中间件的调整机制是在不修改应用的前提下对包括中间件在内的整个系统进行调整,因此,这二者并不是完全一致的.为了使重构模型指导中间件完成重构,需要将模型重构动作映射为中间件调整机制,生成在线重构模型.在线重构模型会根据底层的具体调整机制,引入一些中间件特定的实体与关系.由于不同的中间件会拥有不同的调整机制,因此,在线重构模型是基于特定中间件产品的,我们目前提供了基于 J2EE 应用服务器 PKUAS 的在线重构模型.

3.2 重构规划

在检测出反模式以后,需要确定如何进行重构.一般来说,每个反模式都有对应的重构方案,但是在具体的应用系统中,是否进行重构、如何重构,都需要进行事先的规划,才能保证性能优化效果尽可能地好,而进行重构的代价也尽可能地小.

我们给出了一组指导原则,通过这些原则,能够指导如何进行系统重构工作(AP 表示检测出来的反模式,R 表示重构方案):

- $AP \rightarrow R$:如果一个系统中只检测出一个反模式且该反模式只对应一个重构方案,则选择该重构方案.
- $AP \rightarrow \{R_1,R_2,\ldots,R_n\}$:如果一个反模式对应多个重构方案,则从中选择一个重构代价最小的方案.
- $AP_1 \rightarrow R_1,AP_2 \rightarrow R_2,\ldots,AP_n \rightarrow R_n$:如果系统中存在不止一个反模式,而它们对应的重构方案之间又存在着冲突.则应该综合考虑反模式对性能的影响程度以及重构方案的代价,找出最适合的重构方案.

反模式对性能的影响主要需要从以下几个方面考虑:

1) 性能影响程度:不同的反模式对于应用系统的性能影响也不同,应该优先考虑去除对性能影响严重的反模式;
2) 影响条件:某些系统虽然存在反模式,但是只有在特定条件下才会对性能造成影响.例如,某些反模式消耗了过多的系统资源,但是只会在系统负载较重的情况下才会造成应用系统的性能降低,而在轻载时不会影响性能.对于此类反模式,应根据应用系统的实际运行情况来判断是否去除;
3) 出现频率:某些反模式虽然存在,但出现的次数很少.例如,某反模式造成了某方法调用的响应时间增大,但是在实际运行中,如果该方法很少被调用,那么,该反模式对应用系统的影响也很小.重构时应优先考虑去除出现更为频繁的反模式.

重构方案代价需要从以下两方面考虑:

1) 平台限制:一些重构操作虽然满足重构约束,但是,由于底层平台的限制导致实际无法执行.例如,某重构操作需要增大一个 EJB 的实例池大小以应对频繁的并发访问,但是,如果该 EJB 的实例池大小已经达到底层平台规定的上限,那么,该重构操作将无法执行.在进行规划的过程中,应该去除这一类的重构方案;
2) 重构开销:任意一个重构操作都需要消费一定的时间,而对于在线重构来说,重构消费的时间越多,对运行中的应用系统影响也就越大.因此,应该尽量选择重构开销比较小的方案.

对于重构规划中的影响因素,包括反模式对性能的影响以及重构方案代价,我们根据不同的反模式以及重构动作都给出了默认值,使得重构规划可以自动完成.用户也可以根据具体的情况为这些因素进行赋值,实现对重构规划的定制.

© 中国科学院软件研究所 http://www.jos.org.cn

3.3 重构执行

在确定了重构方案以后,就可以根据重构模型进行应用系统重构.我们将目前的重构分为 3 个类别,分别是:

- 自动重构:该重构可以由中间件自动完成,不需要用户过多的参与.自动重构包括环境重构以及部分对构件的重构,这些重构主要是修改构件或者环境的属性值,中间件可以在运行时刻自动完成这些修改,将属性值调整为重构方案中预设的值;
- 半自动重构:这类重构也可以在中间件的帮助下完成,但需要用户进行一些工作;构件交互的重构一般属于该类别,它通常需要用户添加一些应用特定的辅助类,再由中间件借助这些辅助类完成重构工作;
- 手工重构:对此类重构,我们的方法只能给出重构方案,而所有的重构工作将由用户自己完成,中间件无法提供更多的帮助.部分构件重构属于该类,这些重构牵涉到修改构件内部的应用逻辑,这类重构必须通过修改应用源代码才可能完成,因此需要用户全手工完成.

在重构执行的过程中,将不可避免地对正在运行的应用系统造成一定的影响,需要尽可能地将这类影响降到最低.重构有可能影响到正在进行的应用系统行为,包括正在处理客户请求的构件或中间件服务需要重构,以及正在执行的调用需要进行重构.为了保证应用逻辑的一致性,不应该对这些正在进行的应用系统行为进行重构.因此,本方法中采用的在线重构只针对重构开始执行后的应用系统行为,包括构件或中间件服务接收到的新请求、构件之间新发起的调用.

4 实例研究

4.1 反模式描述

为了更好地演示如何检测 J2EE 应用中存在的反模式并进行相应的重构,我们开发了一个存在反模式的应用系统.该应用系统是一个网上书店的部分实现,主要用于客户端从服务器端获取图书的信息,并将这些信息显示在 Web 页面上.该应用主要包括一个叫 BookEJB 的实体 bean 以及一些 Web 端的构件,它们被部署在不同的机器上.该应用中存在着这样一个反模式:如图 3 所示,当客户端需要从服务器端获取一本书的包括书名、作者、价格在内的一组信息时,需要对服务器端的 BookEJB 执行一系列的方法调用.由于客户端和服务器部署在不同的机器上,因此,这些调用都必须通过网络,调用参数与返回值都需要进行序列化与反序列化的操作,而且还会花费大量的时间通过网络传输数据.这种采用过多次远程调用获取一组属性值的方法会极大地降低性能,是应用中存在的一个性能反模式.它是反模式“细粒度远程调用(fine-grained remote calls)”[5]中的一种.

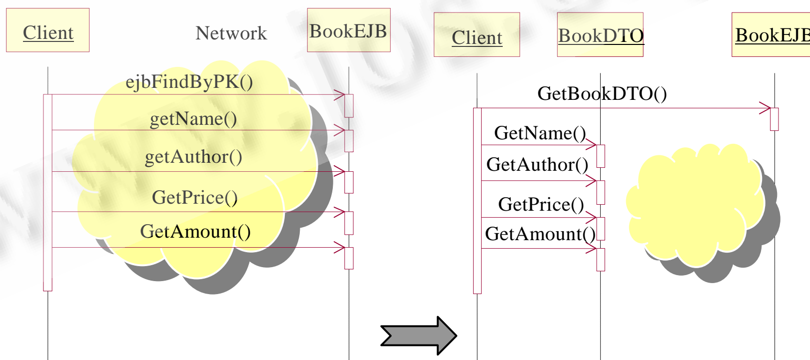

Fig.3 Anti-Pattern of fine-grained remote calls and the refactoring

图 3 “细粒度远程调用”反模式及重构示意图

为了去除该反模式带来的性能影响,需要对网上书店系统进行重构.一种重构方案是通过一次远程调用就

© 中国科学院软件研究所 http://www.jos.org.cn

将客户端所需的所有属性全部获取,而不是对每个属性都进行一次远程调用.可以使用经典的EJB设计模式“数据传输对象(data transfer object,简称DTO)”支持该重构[14].如图3所示,我们使用DTO对网上书店进行重构,当接收到客户端的请求时,BookEJB会构建一个名叫BookDTO的DTO类,BookDTO中包含客户端需要获取的所有属性值.通过这样的重构,所有的属性值就可以通过一次远程调用返回给客户端.自从EJB2.0规范中添加了EJB的本地接口以后,在一些情况下,该反模式将不再带来性能的损耗[15].当客户端通过EJB的本地接口获取属性值时,将不再进行序列化与反序列化这些冗余的操作.但是,EJB本地接口并不适用于所有运行环境,如在我们的例子中,由于客户端与服务器构件被部署到不同的机器上,客户端无法使用EJB的本地接口.而在当前的实际应用中,客户端与服务器构件几乎都是分别部署的.此外,EJB不一定会提供本地接口,而且没有经验的开发者在开发应用中也有可能不会使用本地接口.在这些情况下,该反模式都会对应用的性能造成影响,需要对应用进行重构.

4.2 检测与重构

为了检测上述的反模式,首先应该建立反模式模型.图4展示该反模式的模型.该模型描述了如下的反模式:如果一个实体Bean与客户程序之间存在一个调用序列,该调用序列中第1个调用的是ejbFindBy*()方法(ejbFindBy*表示方法名以ejbFindBy开始的方法),紧接着调用至少*n*个get*()方法,而且这些方法调用都是远程调用.那么,该应用中存在“细粒度远程调用”反模式.在激活该反模式以后,这个模型可以用于指导运行时刻的信息采集,需要采集的运行时刻信息包括应用系统中所有实体Bean的方法调用,所调用方法的名称以及调用的类型(是远程调用还是本地调用).信息采集结束以后,则可以进行反模式检测,通过数据挖掘技术找出是否存在符合模型中规定的调用序列[13].如果有,则当前应用系统中存在该反模式.

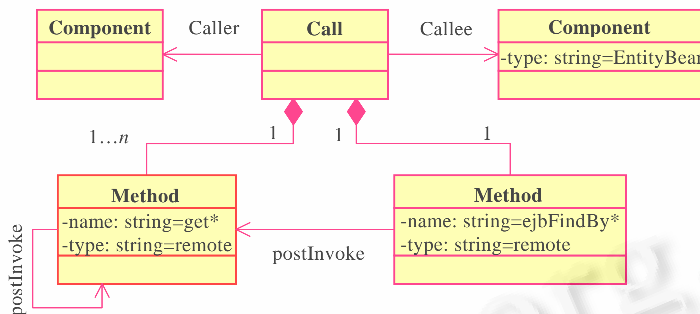

Fig.4 The model of the anti-pattern

图4 反模式模型

检测出反模式以后,应用系统需要进行重构.开发者可以不需要中间件的协助,直接通过修改应用程序源代码完成重构.图5(a)展示了离线重构模型.实体Bean中将添加新的方法getDTO(),而该方法将返回一个DTO对象.客户端将不再调用ejbFindBy*()方法,而是首先调用getDTO()方法得到返回的DTO对象,然后,客户端将调用DTO中的get*()方法,而不是调用实体Bean中的相应方法.对比反模式模型可以发现,该次重构对应用系统作了很多修改:需要添加DTO对象,而且实体Bean中也需要实现getDTO()方法.对于开发者而言,他们需要花费一定的时间与精力来完成这些修改.此外,由于调用序列的改变,不仅服务器端的代码需要进行相应的修改,客户端的代码也需要.但是,在一个分布式环境中,有时很难做到修改所有的客户端.在这种情况下,重构对于某些客户端将不会生效,该反模式对于它们依然存在.

从离线重构模型可以看出,这个例子中最重要的工作就是进行了构件之间调用的重构.实际上,中间件是可以在不修改应用的前提下完成这类重构的.当调用远程服务器上的EJB时,由于客户端与EJB运行在不同的机器上,它其实并不会去直接调用该EJB,而是调用EJB接口的一个实现,该实现被称为一个客户桩(stub).当接收到客户端发来的请求后,Stub负责将该请求转发给远程的EJB.Stub由中间件根据EJB接口自动生成,存在于中

© 中国科学院软件研究所 http://www.jos.org.cn

间件内,并不为应用可见,它的主要任务就是帮助应用系统完成网络通信.而通过对 Stub 的修改,向 Stub 里添加一些额外的功能,则可以达到重构构件之间调用的目的.图 5(b)展示了在线重构模型.在该模型中有两个视图:实线部分是应用视图,仅仅描述了应用的信息;而虚线部分是中间件视图,描述的是中间件上运行的应用系统实际状态.新添加的实体 StubCall 是为了指出来自客户端的请求被转发给哪个构件.新添加的关系 smartReturn 表示 ejbFindBy*()方法返回的不是一个普通的 Stub,而是经过修改附加有额外功能的 Stub.所有的 get*()方法调用全都被转发给了 DTOStub,而不是实体 Bean.这个模型可以指导中间件自动完成系统重构:smartReturn 表示需要为实体 Bean 生成新的 Stub,而 smartReturn 所关联的构件将指导中间件生成什么样的 Stub.在这个例子中,smartReturn 关联了一个名字为 DTOStub 的构件,因此,中间件会启动相应的机制来自动生成 DTOStub.相关的重构机制细节将在下一小节详细加以介绍.

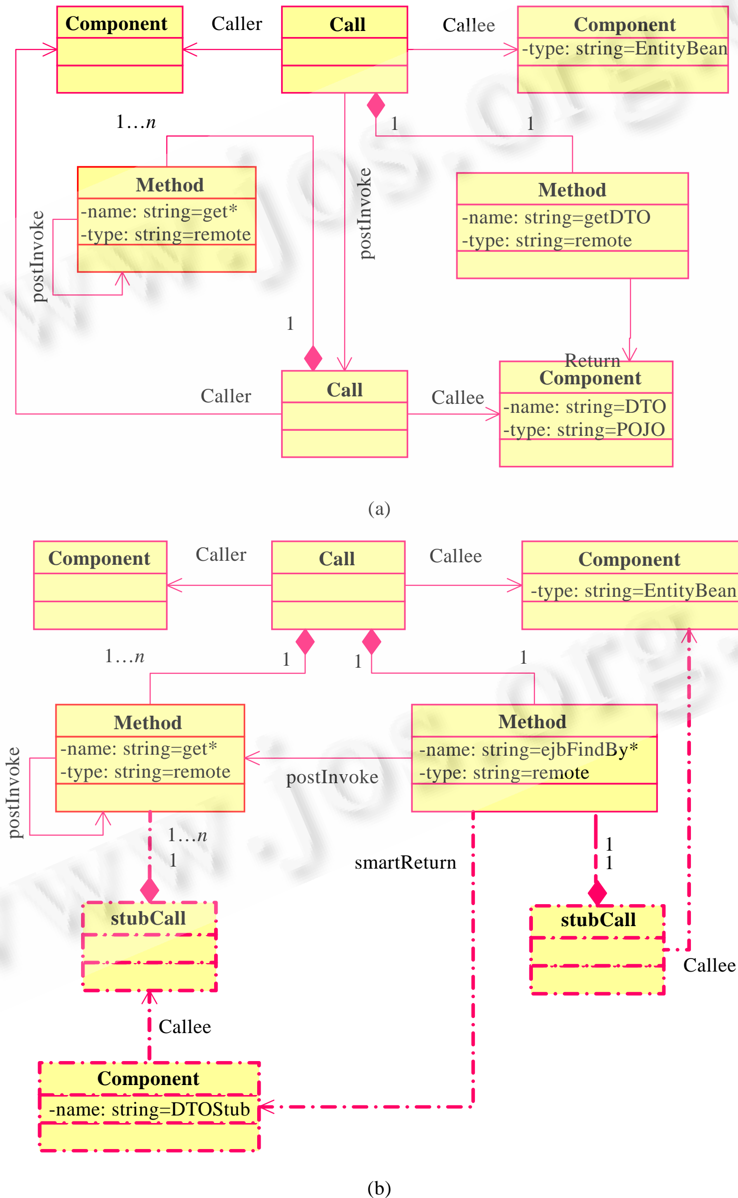

Fig.5 The models of refactoring

图 5 重构模型

© 中国科学院软件研究所 http://www.jos.org.cn

从模型对比上看,与离线重构相比,中间件支持的在线重构有两个优势:

首先,离线重构对于没有修改的客户端将不会生效,而在线重构则不存在这个问题.从图 5(b)可以看出,与最初的反模式模型相比,在线重构模型中的应用视图没有任何的改动,所有的改动都在中间件视图范围内.这意味着所有的改动对于应用都是不可见的,重构是在不修改应用的前提下完成的.因此,在线重构不需要客户端作任何修改就可以生效.

其次,尽管离线重构与在线重构都是在应用中使用了 DTO 模式,但在线重构是由中间件自动完成的,这极大地降低了开发者的工作量.

4.3 重构机制

我们目前实现的原型系统是基于 J2EE 应用服务器 PKUAS 实现的,它为应用系统重构提供了丰富的调整机制[16,17].本节将详细介绍在网上书店的应用中,如何通过 PKUAS 的支持在线地进行应用重构.根据如图 5(b)所示的重构模型的指导,中间件将进行如下的操作:

(1) 在 BookEJB 的容器中添加 DTO 控制器(DTO controller):在 PKUAS 的 EJB 容器中,可以动态地添加或者删除各式控制器,而这些控制器可以用于修改 EJB 的行为.在线重构模型中,与方法 ejbFindBy*()关联的关系 smartReturn 表示需要在容器中添加一个控制器,该控制器负责实例化一个附带额外功能的 Stub,并用它取代普通的 Stub 返回给客户端.由于与 smartReturn 关联的构件名为 DTOStub,中间件将在 BookEJB 容器中添加预定义的 DTO 控制器,在不被应用感知的前提下为该应用实现 DTO 模式.

(2) 生成 DTOStub 类:在线重构模型中的 DTOStub 构件表示需要在普通的 Stub 类基础上生成 DTOStub 类.与普通的 Stub 比较,DTOStub 中添加了 BookEJB 中的一些属性,并且修改了 Stub 中获得属性值的 get*()方法.在普通的 Stub 中,get*()方法会将客户端的请求转发给服务器端的 BookEJB 以获得属性值,而在 DTOStub 中,修改后的 get*()方法将直接返回当前 DTOStub 实例中对应的属性值.具体需要在生成的 DTOStub 中添加哪些属性,以及需要修改哪些 get*()方法,将由实际的运行情况来决定.生成 DTOStub 以后,将编译该类,并由 BookEJB 的类装载器进行装载,使得 BookEJB 可以使用新生成的 DTOStub.

在 PKUAS 中允许对 Stub 进行定制,用户可以在普通的 Stub 中添加额外的功能.一般情况下,定制的 Stub 将由用户自己完成,PKUAS 则负责将普通的 Stub 动态地替换为用户定制的 Stub.在本例的重构工作中,由于相关的修改信息都可以在运行时刻获取,因此,添加的 DTOStub 将可以自动生成.用户需要开发 DTO 模式控制器以及 DTOStub 的自动生成程序,但它们并不是特定于应用的,因此可以在需要在应用中添加 DTO 模式时得到有效的复用.

上述两个操作完成以后,重构工作就已经完成,这些操作可以在用户不参与的情况下由中间件自动完成.由于上述的操作都需要耗费一定的时间,因此重构开始时,服务器将会把接收到的客户端发来的 ejbFindBy*()请求放置于一个缓冲池中,在重构完成以后,再应答这些请求.对于重构开始时已经开始处理的请求,为了不破坏应用逻辑的一致性,对它们将不进行任何处理.

在重构完成以后,应用系统依旧保持不变,但是实际的运行流程却已经被中间件改变了,细粒度的远程调用已经被去除.图 6 展示了重构完成后客户端从服务器获取一组属性值时的过程图.该过程是从 EJB 容器接受到 ejbFindBy*()请求开始的:

1. 操作名为 ejbFindBy*()的请求到达 BookEJB 的容器;
2. 容器初始化一个 BookEJB 实例,该实例将从数据库中查找相应的记录.这两个步骤与重构之前是完全一样的.在正常的流程中,这两个步骤完成以后,一个包含该 BookEJB 实例相关信息的对象将被返回给客户端;
3. 此时,DTO 模式控制器发生作用,它将中断正常的流程并初始化一个 DTOStub 实例;
4. 生成的 DTOStub 实例从 BookEJB 实例获取属性值.在接收到 DTOStub 的请求后,BookEJB 将调用 EJBLoad()方法从数据库中获得需要的属性值.最后,DTOStub 将更新自己属性的值;
5. 序列化 DTOStub,通过网络将其返回到客户端;

© 中国科学院软件研究所 http://www.jos.org.cn

6. 客户端可以通过动态下载技术获得 DTOStub 类,而服务器端返回的 DTOStub 实例可以在客户端通过反序列化重新生成.客户端则通过这一次的远程调用就可以得到所需要的全部属性值;
7. 当客户端调用 get*()方法时,实际上调用的是 DTOStub 中的同名方法并返回 DTOStub 实例中的属性值.如果客户端调用其他方法,比如给某个属性赋值,该请求则与普通流程一样会转发到服务器端.

从以上步骤可以看出,经过重构后的应用系统中,细粒度的远程调用实际上已经被去除了.

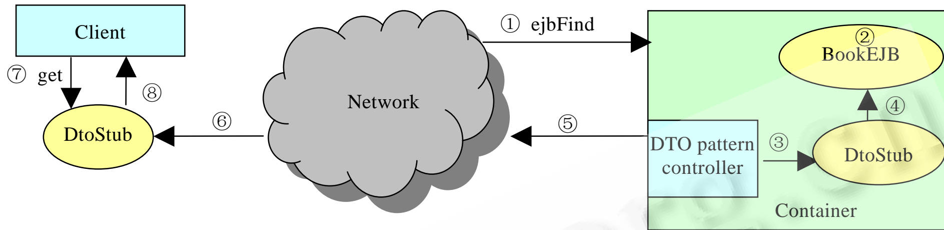

Fig.6 The actual flow of the call

图 6 方法调用流程

4.4 重构效果评估

在我们的方法中,在运行时刻重构整个应用系统以去除反模式.这种方法与传统的离线重构方式不同,但是会有相当接近的效果.本节将对重构效果进行评估.整个应用系统的运行环境如下:部署 BookEJB 的机器配置有奔腾 4 2.8G Hz 的处理器以及 512M 内存;部署 Web 端构件的机器配置有奔腾 4 2.4G Hz 的处理器以及 256M 内存;两台机器之间的网络带宽为 100M.

图 7 展示了网上书店在重构前、经过离线重构、经过在线重构 3 种状态下的性能对比.在这里,我们以客户端获取一组属性值的平均响应时间作为评估性能的指标.其中,客户端需要获取的属性值个数,即调用 get*()方法的次数是影响响应时间的重要因素,我们对 3 种状态下客户端获取 1~5 个属性值的响应时间分别进行了测试.

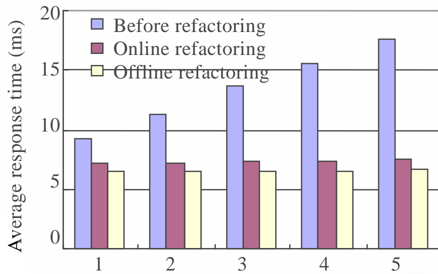

Fig.7 Performance before/after refactoring

图 7 重构前后性能对比

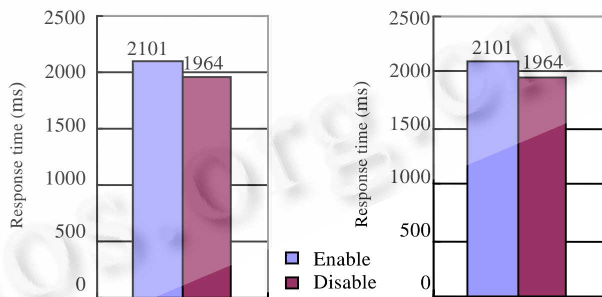

Fig.8 Performance effects of run-time information collection

图 8 运行时刻数据采集对性能的影响

从图 7 展示的结果可以看出,在线重构会降低操作的响应时间.在进行重构之前,每当客户端需要多获取一个属性值,即多进行一次 get*()操作时,响应时间会增加 2ms 左右.这是因为每执行一次 get*()操作就会导致一次远程调用,相比本地调用,远程调用将会把大量的时间花费在网络相关的操作上,主要包括建立网络连接、传输数据,在本应用中,执行一次这些操作的时间是 1.9ms 左右.系统进行了在线重构以后,当客户端需要获取的属性数目增加时,响应时间增幅非常小,每多获取一个属性值,响应时间只增加约 0.1ms.这是由于在经过重构以后,无论客户端需要获取多少个属性的值,都只将进行一次远程调用.在重构以后,无论执行多少次 get*()操作,都不再有建立网络连接的操作.此外,在该应用中,客户端获取更多的属性值并不会造成该次远程调用中传输数据的大幅增加,当仅获取一个属性值时,通过网络传输的数据量是 1 200 字节,此后,每多获取一个属性值,数据量只增加大约 40 字节.由此可见,在获取属性数目不是特别大的情况下,数据传输时间不会有明显变化.因此,相比重构前,

© 中国科学院软件研究所 http://www.jos.org.cn

客户端需要多获取一个属性值时,网络操作所消费的时间将不再存在.

在系统经过离线重构的状态下,响应时间会比在线重构后更低.这是由于在线重构后系统的过程中多进行了一次数据库操作.如图 6 所示,整个过程中执行了两次访问数据库的操作:在步骤 2 中,从数据中查找相应的记录,在这个步骤中只读取了记录中的主键的数据;在步骤 4 中,再次访问数据库以提取相应记录中的其他属性值.而在离线重构中,由于开发者可以直接修改源代码,则可以在 EJB 类中实现 getDTO()方法,只通过一次数据库访问的操作就完成所有数据的提取.但是,由于 getDTO()方法是应用特定的,无法进行复用,因此,需要开发人员直接进行改写.而且两种重构方案的效果非常接近,在线重构后的响应时间仅比离线重构后多 0.7ms(约 9%).更为重要的是,在线重构的优势在于它可以在不中断当前应用运行的基础上完成.

运行时刻信息采集会对应用系统的性能带来负面的影响.我们采用的信息采集机制是可配置的,并不会对应用系统造成连续的影响.为了明确运行时刻信息采集会对应用系统造成多大的负面影响,我们在目前的原型系统上进行了性能测试.我们采用了一个 J2EE 性能基准测试集 ECPerf[18],并分别进行了采集运行时刻信息和不采集时候的性能测试.性能测试结果如图 8 所示,在采集运行时刻信息的情况下,应用系统的平均响应时间会增加大约 7%,而吞吐量会降低 5%左右.从测试结果可以看出,我们的运行时刻信息采集机制不会对应用系统的性能造成严重的负面影响.

4.5 适用性评估

为了评估本方法的适用性,我们对现有的性能相关的反模式应用了本方法.通过对中间件应用系统中可能出现的反模式进行调研,我们共总结出 38 个性能相关的反模式.图 9 展示了在实际应用中,本方法针对这些反模式的适用程度.

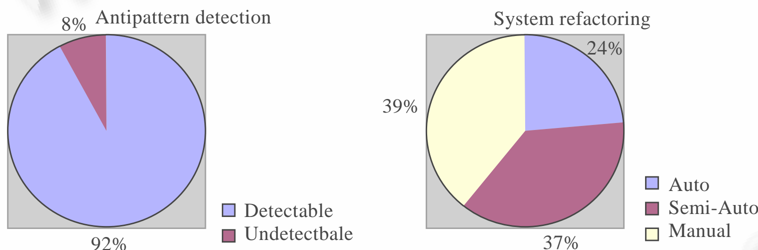

Fig.9 Applicability of the approach

图 9 方法适用范围

在反模式检测阶段,使用本方法可以检测出绝大部分的反模式(35 个,92%).仅有极少数反模式(3 个,8%)无法使用本方法进行检测,对于该类反模式的检测,需要获得构件内部应用逻辑的相关信息,而此类信息目前无法在系统运行时刻通过中间件获取,因此无法使用本方法进行检测.

在系统重构阶段,本方法可以对大部分的重构(23 个,61%)提供支持.其中,9 个重构方案(24%)可全自动地完成,这些重构方案全是修改构件或者环境的属性值,中间件可以在运行时刻进行调整;14 个重构方案(37%)可以半自动地完成,它们通常是修改构件之间的调用序列或者调用参数,该类重构的特点是需要用户添加一些应用特定的辅助类,再由中间件完成重构工作.特别地,一些辅助类可以根据运行时刻信息自动生成,例如,本文实例研究中的重构方案.因此,事实上这些重构也可以全自动地完成.剩余的 15 个重构方案(39.5%)需要修改构件内部的应用逻辑,对于此类重构,本方法只能向用户给出重构方案,并由用户手工完成重构工作.

综上所述,本方法适用于构件与环境的配置、构件之间调用相关的反模式及其相应重构,而对于涉及构件内部应用逻辑的情况,则不在本方法的适用范围内.

5 相关工作

目前有许多成熟的性能管理工具[19−21],它们通常能够检测出某些编程上的失误,比如内存泄露等.与这些工具相比较,我们的研究更倾向于在反模式的指导下检测到设计阶段的错误.对于分布式系统的性能问题,目前的研究主要集中在能够提供性能优化支持的机制[22,23].与这些研究相比,我们的工作更多地关注性能优化的策

© 中国科学院软件研究所 http://www.jos.org.cn

略、如何借助中间件的帮助更好地将反模式应用到性能优化工作中.

文献[24]介绍了软件性能工程(software performance engineering,简称 SPE)的概念、方法与关键技术等.软件性能工程是控制性能的一种综合方法,在软件的整个开发过程中,系统地规划和预测正在开发的软件的性能.该方法中的一项关键技术就是通过应用性能模式与反模式,在体系结构和设计方案中做出最优的选择,从而控制软件达到性能目标.与软件性能工程着眼于在软件设计开发时刻应用反模式相比,我们的方法更强调在运行时刻应用反模式.我们的方法可以作为软件性能工程的一个补充,对于未在设计开发过程中充分考虑性能因素的应用系统,可以在运行时刻检测到反模式并且进行在线重构.

文献[25]提出了一个可以检测性能反模式的框架.该框架以一组规则表示一个反模式,通过综合分析,应用系统的静态信息与运行时刻信息检测出性能反模式;然后使用性能模型来评估这些反模式带来的影响;最终将该反模式以图形化的方式展现给用户,使得用户更容易理解该反模式.我们的工作与这个框架类似,但是研究得更加深入.在反模式检测方面,我们通过对检测反模式所需应用状态信息的总结,给出了基于 MOF 的元模型来表示反模式.这种表示方式更加通用化,也更易于理解和扩展,同时也支持了自动化的反模式检测.更为重要的是,我们的方法支持完整的反模式应用,不仅包括反模式的检测,还包括之后相应的重构工作.我们的工作充分地利用了中间件的调整机制,在检测出反模式以后,通过中间件的支持,自动化地对应用系统进行在线重构.

6 结束语

本文介绍了一种基于反模式的中间件应用系统性能优化方法.在该方法中,使用一个基于 MOF 的元模型来描述反模式,这使得反模式表示更加准确,也更易于理解.在反模式模型的指导下,反模式的检测以及相应的重构工作将以一种自动化的方式进行.反模式的检测是基于运行时刻信息的,而这些信息由中间件自动采集,不需要对应用进行任何修改.在检测出反模式以后,将对应用系统进行在线重构以去除反模式.与传统的离线重构方式相比,在线重构可以在不中断应用系统运行的前提下完成,而且许多重构工作可以由中间件自动完成.

今后的工作重点包括:在更多的中间件产品上实现该方法;提高整个方法的自动化程度;与设计开发阶段的方法更好地结合.

References:

[1] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[2] Brown WJ, Malveau RC, McCormicm HW, Mowbray TJ. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. New York: John Wiley and Sons, Inc., 1998.

[3] Fowler M. Refactoring: Improving the Design of Existing Code. Addison Wesley Longman, Inc., 1999.

[4] Dudney B, Asbury S, Krozak JK, Wittkopf K. J2EE Antipatterns. Wiley Press, 2003.

[5] Tate B, Clark M, Lee B, Linskey P. Bitter EJB. Manning Publications. 2003.

[6] Smith CU, Williams LG. More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot. CMG, 2003.

[7] Smith CU, Williams LG. New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot. CMG, 2002.

[8] Smith CU, Williams LG. Software Performance Antipatterns. In: Proc. of the 2nd Int'l Workshop on Software and Performance. New York: ACM, 2000. 127−136.

[9] Mei H, Huang G. PKUAS: An architecture-based reflective component operating platform, invited paper. In: Proc. of the 10th IEEE Int'l Workshop on Future Trends of Distributed Computing Systems. Los Alamitos: IEEE Computer Society, 2004. 26−28.

[10] OMG. Meta object facility (MOF) specification. Version. 2.0. 2003.

[11] Schmidt D, Stal M, Rohnert H, Buschmann F. Patterns-Oriented Software Architecture. Wiley Press, 2000.

[12] Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. In: Proc. of the '93 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM, 1993. 207−216.

© 中国科学院软件研究所 http://www.jos.org.cn

[13] Parsons T, Murphy J. Data mining for performance antipatterns in component based system using run-time and static analysis. Trans. on Automatic Control and Computer Science, 2004,49(63):49−63.

[14] Marinescu F. EJB Design Patterns: Advanced Patterns, Processes, and Idioms. John Wiley and Sons, Inc., 2002.

[15] Sun Microsystems. Enterprise JavaBeans spec. Version 2.0. 2001.

[16] Huang G, Mei H, Yang FQ. Runtime recovery and manipulation of software architecture of component-based systems. Int'l Journal of Automated Software Engineering, 2006,13(2):251−278.

[17] Huang G, Mei H, Wang QX, Yang FQ. Research on architecture-based reflective middleware. Journal of Software, 2003,14(11): 1819−1826 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/1819.htm

[18] Sun Microsystems. ECPerf kit Version 1.1 final release. 2003.

[19] Quest Software. JProbe performance profiler. 2006. http://www.quest.com/jprobe

[20] IBM. WebSphere studio profiling tool. 2006. http://www.javaperformancetuning.com/tools/websphereprofiler/

[21] Borland. Optimizeit enterprise suite. 2007. http://www.javaperformancetuning.com/tools/optimizeit

[22] Steigner C, Wilke J. Performance tuning of distributed applications with CoSMoS. In: Proc. of the ICDCS. Washington: IEEE Computer Society, 2001. 173−180.

[23] Kandasamy N, Abdelwahed S, Khandekar M. A hierarchical optimization framework for autonomic performance management of distributed computing systems. In: Proc. of the ICDCS. Washington: IEEE Computer Society, 2006. 9−18.

[24] Smith CU, Williams LG. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addision Wesley, 2002.

[25] Parsons T. A framework for detecting performance design and deployment antipatterns in component based enterprise systems. In: Proc. of the 2nd Int'l Middleware Doctoral Symp. New York: ACM, 2005. 1−5.

附中文参考文献:

[17] 黄罡,王千祥,梅宏,杨芙清.基于软件体系结构的反射式中间件研究.软件学报,2003,14(11):1819−1826. http://www.jos.org.cn/1000-9825/14/1819.htm

兰灵(1978－),男,北京人,博士生,主要研究领域为中间件技术,软件工程.

王玮琥(1986－),男,主要研究领域为中间件技术,软件工程.

黄罡(1975－),男,博士,副教授,CCF 会员,主要研究领域为中间件技术,软件工程.

梅宏(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程,软件复用和软件构件技术,分布对象技术.

© 中国科学院软件研究所 http://www.jos.org.cn