中国科学 E 辑：信息科学 2008 年 第 38 卷 第 6 期：901 ~ 920
www.scichina.com info.scichina.com
《中国科学》杂志社
SCIENCE IN CHINA PRESS

基于体系结构的网构软件自适应方法

梅宏[①②*]，黄罡[①②]，兰灵[①②]，李军国[①②]

① 北京大学高可信软件技术教育部重点实验室，北京 100871;
② 北京大学信息科学技术学院软件研究所，北京 100871
* E-mail: meih@pku.edu.cn

收稿日期: 2007-12-24; 接受日期: 2008-03-06
国家重点基础研究发展规划(批准号: 2002CB312000)和国家自然科学基金(批准号: 90612011)资助项目

摘要 作为网构软件的基本特性之一，自适应性是指软件系统在预设策略的指导下自动地监测系统状态信息，并在必要时对自身进行调整，以提供更好的服务. 针对网构软件可适应性、自适应可操作性及适应结果可信性 3 个基本问题，提出了一种以体系结构为中心的网构软件自适应方法，即监测、分析、规划、实施等自适应活动均围绕软件体系结构展开，其中，网构软件运行状态和行为以运行时软件体系结构的方式实时展现和在线调整；自适应相关知识通过软件体系结构记录、组织和加工，以实现系统状态和行为的自动分析和调整规划.

关键词
网构软件
自适应
软件体系结构
中间件

网构软件是Internet环境下一种新的软件形态，具有自主性、协同性、反应性、演化性和多态性等特点[1]. 这些特性在技术上的共性之一可归结为自适应性，即，网构软件在运行过程中能够在合适的时刻、合适的场合、准确捕捉变化并进行合理的适应性调整，以满足功能和质量的需求[2]. 从国内外研究与实践看，自适应技术并不是一个网构软件独有的、新的话题. 然而，不论已有成果和经验能否适用于网构软件，系统化的网构软件自适应需首先解决 3 个基本问题:

■ 网构软件可适应性问题：即，哪些或哪类变化可以被感知或理解，以及哪些或哪类适应性调整可以被实施，这是网构软件实现系统化自适应的首要问题. 理论上，网构软件这种典型的人工制品的所有变化都可以被感知或理解，并实施适应性调整. 但 Internet 环境开放、动态、难控等特点，从实用或实践的角度限定了网构软件自适应的范围.

■ 自适应可操作性问题：即，在给定的可适应范围中，如何操作才能实现自适应. 这是网构软件自适应的核心问题. 以自治计算[3]为例，IBM从产业实践的角度给出了一个以知识为中心、对可管理实体分步骤进行监测、分析、规划和实施的自治计算模型，并按照人参与系统

管理的程度划分了基础级、可管理级、可预测级、自适应级、自治级等 5 个可操作级别. 尽管自治计算模型及其分级给出了一个具有普适意义的自适应可操作性原理, 但仍需针对网构软件进行特化和细化, 如, 知识在每个活动中的表现及应用、活动之间的自动衔接等, 从而形成实际可操作的网构软件自适应基本结构与机理.

■ 自适应结果可信性问题: 主要包括自适应操作是否正确以及是否达到预期效果. 这是网构软件自适应得以实际应用的关键问题. 前者需要评估适应后的网构软件的功能正确性, 后者需要评估网构软件的性能、可靠、安全等质量属性(因为它们是网构软件自适应的主要目标). 这些指标均属于网构软件可信性的范畴, 而可适应范围的有限性则决定了自适应的可信性是网构软件可信性的一个子集.

从技术角度看, 上述 3 个基本问题可分别具体化为“如何看待和理解网构软件基本形态”、“如何监控网构软件并进行分析决策”、“如何评估网构软件功能正确性和主要质量属性”. 我们认为, 软件体系结构(software architecture, SA)为系统化地研究并解决这些基本问题提供了一条切实可行的技术途径, 因为:

■ SA为考察网构软件自适应范围提供了全面、系统、易于理解的模型: SA是一组构件、连接子以及约束所描述的软件系统总体结构[4], 能够自然地描述网构软件的基本形态, 即, 一组分布于Internet 环境下各个节点的、具有主体化特征的软件实体, 以及一组用于支撑这些软件实体以各种交互方式进行协同的连接子[5]. 因此, 网构软件的自适应具体表现为软件实体、软件交互、以及运行环境的自适应, 这些因素可分别建模为SA中的构件、连接子、约束和变化.

■ SA 为网构软件自适应提供了强有力的可操作性支持: 首先, 自适应的广度和深度取决于相关知识的获取(隐性知识转变为显性知识)、组织(劣构知识转变为良构知识)和加工(描述性知识升级为推理性知识). 作为一种模型化、支持分析与决策的高层结构, SA 是一种有效的网构软件自适应知识的载体; 其次, 作为软件工程中分析和控制整个系统功能性与质量的主要手段之一, SA 相关的方法和技术可直接支持监测、分析、规划、实施等自适应主要活动; 进一步地, 围绕 SA 展开这些活动, 为整个自适应过程的无缝衔接提供了自然的支持.

■ SA有助于网构软件自适应结果可信性的评估: 在传统软件系统开发过程中, 软件体系结构是保障目标软件功能性、性能、可靠、安全等系统级质量属性的主要手段, 尤其体现在对这些可信指标的评估方面. 网构软件作为传统软件在Internet环境下的一种自然延伸, 其可信性评估在一定程度上可以复用或受益于已有的SA评估方法与技术. 随着网构软件体系结构模型研究[2]的深入, SA将在网构软件适应结果可信性的评估中起到越来越重要的作用.

基于上述讨论, 本文提出了一种基于体系结构的网构软件自适应方法. 本文余下的部分组织为: 第 1 节概述整个方法; 第 2 节介绍关键技术; 第 3 节以 J2EE 性能基准程序 ECPerf 为例展示方法的可行性与有效性; 第 4 节介绍相关工作; 最后总结全文并展望未来工作.

1 方法概述

所谓以软件体系结构为中心的自适应, 是指自适应知识由 SA 承载, 自适应活动基于 SA 进行. 如图 1 所示的自适应模型, 为了有效支持自适应知识的获取、组织和加工, SA 需具备 4

个特性. 其中:

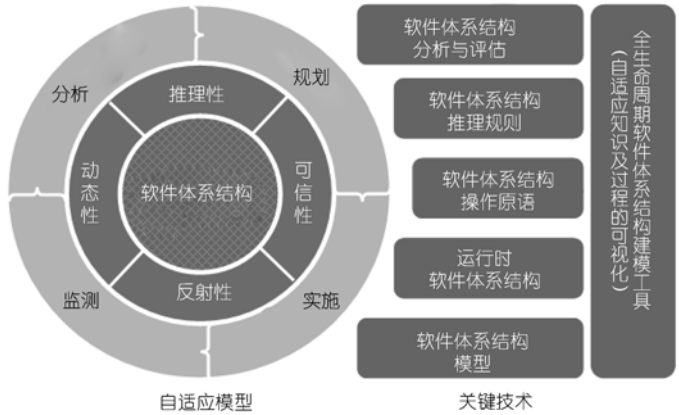


图1 以软件体系结构为中心的网构软件自适应方法概览

■ 反射性: 是指SA与目标系统的运行状态和行为之间具有因果关联关系, 即, 目标系统运行状态和行为的变化会立即导致其SA的相应变化, 反之亦然. 具备反射性的SA称为运行时软件体系结构[6], 它从体系结构层次明确了网构软件自适应的范围, 能够保证自适应环路分析的变化和规划的调整与目标系统实际发生的变化和调整之间的一致性和准确性. 此外, SA反射性是对目标系统所支持的管理操作或监控能力的一种高层抽象, 在一定程度上屏蔽了底层管理机制的细节和异构性, 保证了自适应方法的通用性.

■ 动态性: 是指SA能够全面准确地描述反射性所能感知的变化和实施的调整, 如构件和连接子的增加、删除、以及通过约束增删改而实现的修改. 具有动态性的SA称为动态软件体系结构[7]. 经典SA研究中的动态性往往是由设计人员预测而得, 难以自然描述反射性所感知的正在发生的变化. 同时, 经典SA动态性大多假设通过编程实现, 其描述的调整与反射性运行时可实现的调整之间存在失配的可能, 导致自适应规划难以完整、准确地实施. 因此, 我们定义了一组操作原语以从反射性的角度修订经典SA的动态性.

■ 推理性: 是指 SA 能够在反射性限定的范围内分析变化并推导出相应的适应性调整. 动态性必须显式描述变化及其相应的调整(即 What-to-change 和 How-to-change), 而推理性则可以通过一些规则(即Why-to-change)自动生成动态性描述, 是实现可预测、自适应、自治等高级别自适应的关键. 推理规则有很多种, 我们目前采用了 3 种形式, 包括常用的 ECA 规则(event-condition-action, 对于给定的事件, 根据给定的条件决定具体的动作)、模式(对于给定的模式, 自动检测系统中是否存在该模式并转换成另外一种模式)、风格(对于给定的风格, 自动将使系统的整体或局部符合该风格).

■ 可信性: 是指 SA 所能体现的目标系统的功能正确性、性能、可靠、安全等指标符合用户预期. 本质上, 网构软件自适应就是一种持续追求或保障可信性的过程, 该过程的输入是不可信的目标系统, 输出则是变得可信的目标系统. 换言之, 可信与否是自适应过程启动和终止的判定条件. 技术上, 可信性评估与自适应是两个相对独立的研究领域, 软件体系结构的评

估经过了长期的研究与实践, 我们将复用其中的成果与经验来解决自适应结果可信性的问题.

基于具有上述特性的 SA, 网构软件自适应的主要活动将自动执行. 其中, 监测活动利用 SA 反射性收集网构软件的运行信息, 并将这些信息展现为 SA 的动态性描述; 分析活动对当前系统的 SA 以及监测活动产生的 SA 动态性描述进行可信性评估, 如果评估结果为不可信, 则激活规划活动, 否则终止此次自适应环路; 规划活动首先针对分析活动的可信性评估结果, 找到监测活动的动态性描述中可能导致不可信的变化, 然后利用 SA 推理性得出适应性调整方案, 并生成相应的动态性描述; 实施活动则将规划活动生成的动态性描述转换成反射性操作并执行之, 最终调整实际系统. 这 4 个活动组成一个自适应环路循环往复地进行, 人的参与可能导致自适应过程从任意一个活动开始或结束.

2 关键技术

2.1 软件体系结构模型

尽管目前尚无统一或标准的SA模型, 但多年的研究与实践在SA的基本内涵方面达成了许多共识[4,7]. 本文采用了如图 2所示的ABC/ADL定义的SA模型[8], 原因在于: 首先, 它具有典型的软件体系结构模型的所有要素, 包括完成系统所需计算功能的构件、负责抽象构件之间交互的连接子、对构件之间的交互拓扑结构进行描述的系统配置以及从系统全局角度出发需要考虑的各种约束. 在此基础上, 它还新增了特定于网构软件的特性, 如 1) 区分类型和实例: Asset元素中的Components和Connectors是对类型的定义, Instances元素则是对构件和连接子的描述. 区分类型和实例意味着网构软件体系结构模型是一个类型化的模型, 这样架构师就有可能对设计好的模型进行验证, 以保证组装结果的正确性. 区分类型和实例的另一个好处就是有助于构件类型和连接子类型的复用; 2) 兼顾复用和构造: 新构造的资产与可复用资产具有一个本质的区别, 那就是前者可以根据当前的设计需要随时进行修改, 而后者对当前的架构师而言是一个黑盒的资产, 只能进行恰当的使用而无法修改其定义. 因此, 需在兼容的前提下显式区分新构造资产(newComponent和newConnector)与可复用资产(reusedComponent和reusedConnector); 3) 简洁有效的配置: 配置(Configuration)决定了构件之间的交互拓扑结构, 用松耦合的link来确定构件的交互方式, 以适应网构软件交互方式灵活多样的特点. link的语义可类比为演员(player)扮演某个角色(role). player是对构件计算功能的抽象, role则是对连接子通信能力的抽象. 通过恰当的link定义, 架构师就能指定系统中的每个构件应该在整个系统中扮演怎样的角色才能完成满足目标系统的需求; 4) 可扩展的约束: 软件体系结构之所以受到广大研究人员和实践者的关注就在于它允许架构师在系统开发早期对系统进行分析以便尽早发现潜在的问题. 对系统结构进行分析的基础就是体系结构模型中的约束定义(Constraint), 如安全、事务、响应时间等. 架构师可根据特定的约束扩展已有的描述方式, 以便于约束的定义、转换、保障和评估.

2.2 软件体系结构反射性

反射性主要的实现方式有三种, 其中, 反射式编程语言通过内嵌在编译器或虚拟机中的特殊机制实现对运行程序的反射; 反射式中间件则通过内嵌在中间件的特殊机制实现对上层

应用构件的反射; 反射式构件模型通过定义一组需构件开发者实现的接口来反射构件的内部实现. 三者各有利弊, 如, 反射式编程语言的反射能力理论上最为强大, 但特定于具体的语言, 且反射视图抽象层次偏低、系统化不足; 反射式中间件的抽象层次相对较高, 反射能力也很强大, 但特定于具体的中间件, 且难以反射构件内部的具体实现; 反射式构件模型的抽象层次最为贴近软件体系结构, 但反射能力依赖构件开发者的具体实现, 对全局的反射能力偏弱. 为此, 我们通过运行时软件体系结构集成了上述 3 种反射性实现方式, 且屏蔽了这些技术的异构性以保障自适应方法的通用性.

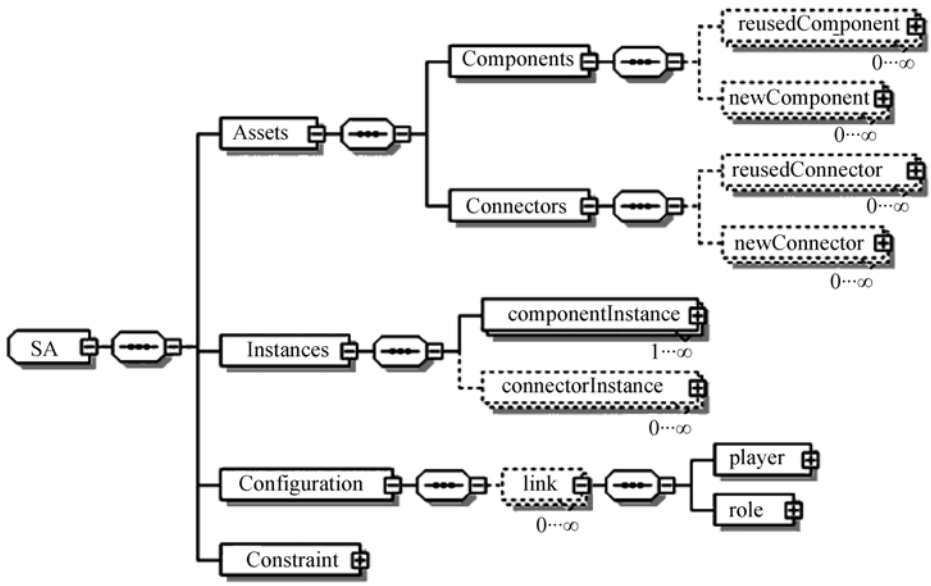


图2 ABC/ADL 定义的软件体系结构基本模型

如图 3 所示, 运行时软件体系结构(runtime software architecture, RSA)的主体是一个内嵌在中间件中的全反射框架[6], 该框架直接利用中间件的反射能力和编程语言的反射能力, 若部署在中间件上的构件实现了反射式构件模型定义的接口, 则可直接调用这些接口以整合构件模型的反射能力. 以中间件为主体是因为中间件是网构软件运行支撑平台的主要形式, 且中间件易于整合编程语言和构件模型的反射性, 以使反射框架尽可能全面地覆盖整个系统. 目标系统中的构件抽象为RSA中的构件; 构件之间若通过中间件交互, 则中间件的互操作功能抽象为RSA中的(简单)连接子; 某些构件可能抽象为RSA中的(复杂)连接子, 这是因为该构件或者是设计阶段SA中的某个连接子, 或者有人显式指定; 中间件提供的安全、事务、持久等功能抽象为RSA中构件或连接子的约束; 反射式构件模型提供的反射能力抽象为RSA中构件或连接子的反射接口; 编程语言(Java)的反射能力则用来实现或增强中间件和构件模型的反射性.

2.3 软件体系结构动态性

按照 SA 模型中元素的种类, SA 的动态性可以分解为实体调整、实体间关系调整、实体属

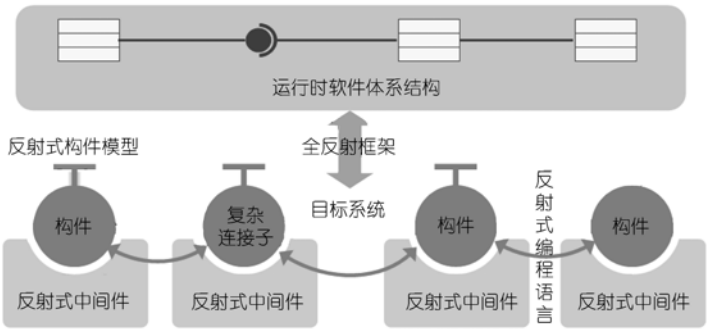


图3　软件体系结构全反射框架

性调整等 3 类操作原语. 定义如下:

■ 一个实体的调整定义为一个三元组, 记为 RE(E, ID, OP), 其中 E 代表被调整的实体的类型, 包括 Host, Service, Component, Call 四类实体, ID 代表被调整实体的唯一标识符, OP 代表对该实体进行的操作, 包括 Add 和 Delete(增加一个实体、删除一个实体)两种操作.

■ 一个实体间关系的调整定义为一个六元组, 记为 RR(E1, ID1, E2, ID2, R,OP), 其中 E1 和 E2 代表被调整的关系所关联的两个实体的类型, ID1 和 ID2 代表这两个实体的唯一标识符, R 代表被调整关系的唯一标识符, OP 代表对该关系进行的操作, 包括 Add 和 Delete(为两实体之间添加关系、删除两实体之间的关系)两种操作.

■ 一个实体属性的调整定义为一个四元组, 记为 RP(E, ID, P, V), 其中 E 代表被调整的实体的类型, ID 代表被调整实体的唯一标识符, P 代表实体中需要进行调整的属性名, V 代表调整后该属性的值.

运行时软件体系结构能够监测的变化和实施的调整均可通过上述原语及其组合来描述. 这组操作原语可以视为一种软件体系结构操作语言(architecture manipulation language), 相应的 SA 动态性描述形成独立的文档, 以保持整个 SA 描述的清晰、简洁、灵活.

2.4　软件体系结构推理性

推理性知识往往可以表示为“如果…那么”, 除了常见的 ECA(event-condition-action)规则, 我们还支持 SA 特有的推理性知识表示方式, 包括模式和风格. 模式抽象描述了 SA 的一个范围较小的局部结构, 如工厂模式、适配器模式等; 而风格则抽象描述了 SA 的整体结构或者一个范围较大的局部结构, 如管道-过滤器风格、三层体系风格等. 模式和风格所抽象的结构可能对可信性带来正面影响、负面影响或无影响, 相应的自适应包括消除具有负面影响的模式和风格、或者实现具有正面影响的模式和风格. 但实际可操作的自适应主要有两种: 其一, 将具有负面影响的模式和风格调整为无影响或具有正面影响的模式和风格; 其二, 通过对整个系统的调整使其直接具备正面影响的风格. 因此, 基于模式和风格的推理性知识包括“如果存在某种模式或风格, 那么调整为另外一种模式或风格”、以及“如果满足某种条件, 那么调整系统使之具备某种风格”.

目前存在多种模式和风格的描述机制, 在不失一般性的前提下, 我们定义了如图 4 所示的

元模型以定义便于推理、可操作的自适应知识. 其中, Component 用于描述应用系统中各种类型的构件; Connector 描述构件之间的连接子, 它的作用是连接构件, 一个 Component 实例可以通过 provide 或 require 关系与一个 Connector 实例关联, 而被连接的 Component 实例分别作为提供者与消费者角色; Service 继承了 Component, Service 实例表示系统中一类特殊的构件, 即中间件提供的各种服务; Host 用于描述分布式环境中的主机及部署在上的中间件, 两个通过 connect 关联的 Host 实例表示这两个主机之间物理上存在连接, 如果一个 Host 实例包含了一个 Component/Service 实例, 表示该构件/服务部署在该主机上. 此外, 元模型中每类实体都有其对应的属性, 以此定义应用系统中对应实体的状态. 管理者不仅可以通过使用 Component 与 Connector 实体及相关的关系描述应用系统中的构件及构件之间关系, 还可以通过使用 Host 与 Service 实体及相关的关系描述系统级的中间件业务能力, 从一个系统化的角度对应用系统进行建模. 该元模型是可扩展的, 管理员可以通过扩展元模型来提高元模型的建模能力, 如定义特殊的构件类型、复杂的连接子类型等. 图 4 中的深色部分举例说明了如何对元模型进行扩展. 其中, Method 用于表示两个构件之间的方法调用, 一组 Method 实例可以通过 postInvoke 关系关联, 表示一个调用序列; Proxy 用于表示中间件提供负责远程调用的代理构件, 而 ProxyConnector 用于表示普通的 Component 实例与 Proxy 实例之间的连接; ExceptionInterceptor 与 Buffer 均是中间件提供的一些辅助机制, 它们分别表示用于捕获应用系统异常与缓冲应用请求的复杂连接子.

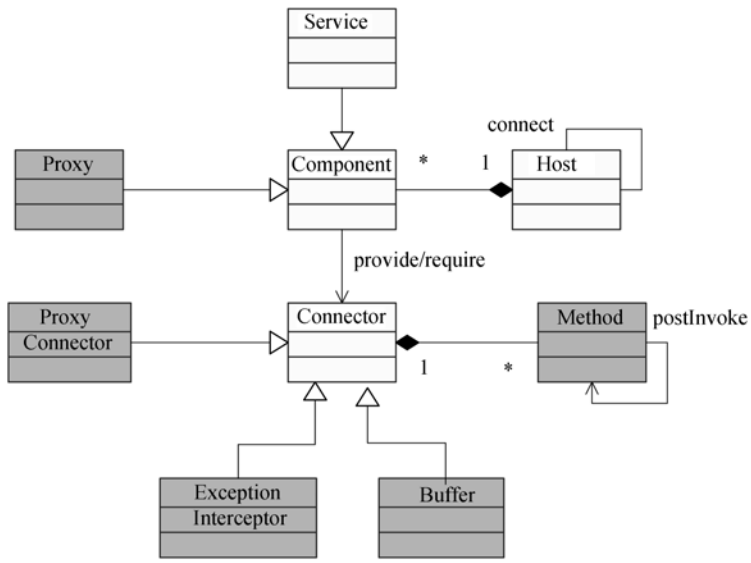


图4 软件体系结构模式和风格元模型

针对该元模型定义出的模式和风格, 可执行的推理包括: 1) 自动检测出目标系统是否具有给定模式或风格; 2) 对于给定的负面影响模式(风格)及相应的无影响或正面影响模式(风格), 可自动生成由前者转换成后者的系统调整方案; 3) 对于给定的正面影响风格, 可自动生成使

目标系统具备该风格的调整方案. 具体细节将在“实例研究”部分论述.

2.5 软件体系结构可信性

2.5.1 正确性评估

网构软件自适应的正确性包含调整后的应用系统正确、以及调整操作可正确执行两个方面. 系统校验(validation and verification)一直是SA研究的重点之一, 主要的方法是在体系结构描述语言中加入形式化的语义模型, 然后使用形式化方法对得到的SA模型进行校验. 本文采用了ABC/ADL的验证机制[9]: 1) 语法层次的校验. 主要是对构件接口的匹配和构件与连接子的拓扑关系进行校验; 2) 实现层次的校验. 根据具体实现的语言和底层平台的规范, 进行类型匹配等实现级的检查(替换构件实现时需要检查其是否在语法上满足构件契约); 3) 语义层次的校验. 利用形式化方法描述构件行为和交互协议, 对系统的整体特性, 如有效性(如, 是否有死锁)和完备性(如, 是否所有请求接口都与合适的提供接口建立了连接等), 进行检查.

自适应操作最终由反射性执行, 在执行的过程中, 受到反射性实现机制的限制, 一些操作不应该或者可能无法执行, 如, 对应用系统的配置超过底层平台规定的上下限, 试图执行底层平台不支持的操作等. 在自适应过程中, 必须保证自适应操作能够执行且不会破坏底层机制设定的各种约束. 调整操作的正确性验证主要分为 3 个部分: 1) 平台与构件、服务之间关系的验证, 必须保证构件或服务可以部署到其指定部署的主机上; 2) 构件与服务之间关系的验证, 必须保证构件可以使用其指定依赖的服务; 3) 属性的验证, 必须保证待系统中实体所有待修改的属性可以设为指定的值.

2.5.2 有效性评估

自适应的目的是为了保障应用系统的质量属性, 在自适应调整之前, 应预先对调整方案进行评估, 以确保其可以满足用户预期效果. 目前的SA研究中存在大量对系统质量属性进行评估的算法, 我们的自适应方法可以复用这些算法. 特别地, 应加入对中间件等运行环境的考虑, 以使得评估算法的输入数据更加丰富, 实现系统化的评估. 例如, 使用现有的可靠性分析算法, 为其输入应用构件与中间件服务的相关信息, 则可对包括中间件在内的整个系统的可靠性进行评估[10].

SA 研究大多在设计时刻对目标系统质量属性进行评估, 往往需要通过预测的方式指定部分输入参数, 如, 评估目标系统的响应时间需要预测构件操作、网络传输等步骤所需的时间, 这难免会造成评估的结果难以真实、准确地反映目标系统的实际运行状况. 而在我们的自适应方法中, 可通过反射性得到实时数据, 从而使得评估算法的结果更为准确.

2.5.3 开销预评估

自适应开销是指执行一个调整方案所需要花费的时间与资源上的消耗. 在自适应调整之前, 应尽可能预测开销, 以确保调整方案的性价比在用户可接受的范围内. 自适应调整是运行时刻执行的, 调整消费的时间越多, 对运行中的系统就影响越大, 因此, 本文将着重考虑自适应调整消费的时间. 执行一个自适应调整方案消费的时间, 等于执行该方案中所有调整操作所消费时间的累加. 不同的方案有不同的执行步骤, 由此也导致执行时间各不相同. 我们在文

献[11]中给出了一个针对不同部署方案计算其部署时间的方法, 重点考虑了软件部署中各个步骤所消费的时间. 在其基础上, 我们给出了一个自适应操作消费时间的计算公式, 它考虑了自适应操作中可能涉及的所有步骤所消费的时间:

$$Time_{refactor} = T_{buffer} + T_{generate} + T_{transfer} + T_{deactivate} + T_{undeploy} + T_{deploy} + T_{activate},$$

其中, $Time_{refactor}$ 表示自适应操作消费时间. 该计算模型将 $Time_{refactor}$ 定义为 7 个步骤所消费时间的累加, 这 7 个步骤消费的时间分别是: T_{buffer} 表示缓存客户请求以及释放缓存所需的时间, 由于调整操作执行需要消费一定的时间, 因此在执行开始前, 需要将客户请求保存在一个缓存中, 当调整完成后再释放缓存中的请求并处理这些请求; $T_{generate}$ 表示生成一些辅助类所需的时间, 例如生成截取器或者代理类等; $T_{transfer}$ 表示将调整操作中需要部署的构件、服务或者辅助类等传输到目标主机上所需的时间; $T_{deactivate}$ 表示去活应用系统中需要替换部分所需的时间; $T_{undeploy}$ 表示卸载需要替换部分所需的时间; T_{deploy} 表示部署新添加的构件、服务或者辅助类所需的时间; $T_{activate}$ 表示激活新添加的构件、服务、辅助类, 或者更新系统配置, 使自适应调整最终生效所需的时间. 该计算公式包括了完成一次自适应调整操作可能涉及到的主要步骤, 在实际应用中, 绝大部分的调整仅执行其中的一部分, 计算开销时应针对不同的自适应操作选择其中不同的步骤. 需要注意的是, 某些步骤在实际的执行过程中是可以并行执行的, 如 $T_{transfer}$ 与 $T_{deactivate}$ 和 $T_{undeploy}$, 即, 在去活以及卸载待替换部分的同时, 传输新添加的类, 这意味着某些情形下实际的自适应开销也许会比预评估的结果更小.

2.6 全生命周期软件体系结构建模工具

由于人工智能、自然语言处理等领域还存在较多尚待解决的问题, 因此, 完全不需要人参与的系统化的自适应目前还很难实现. 正如自治计算分级所体现的, 人适当程度的参与是自适应切实可行的前提. 具体而言, 人的参与主要分为两种: 其一是参与自适应知识的获取、组织和加工; 其二是参与自适应环路的执行. 基于前面的讨论, 我们的自适应方法中的自适应知识通过 SA 描述, 自适应环路均通过 SA 执行, 因此, 以可视化的方式支持 SA 的建模和监控, 是一种便于人参与自适应的用户友好的方式. 为此, 我们采用了 ABCTool, 一种全生命周期软件体系结构建模工具. 一方面, ABCTool 提供了运行时软件体系结构的实时视图, 从而实现自适应环路的实时监控, 如图 5 所示的工具截图; 另一方面, 还可通过 ABCTool 可视化地定义网构软件各个阶段的 SA 基本模型(即经典的 SA 建模)以及动态性和推理性.

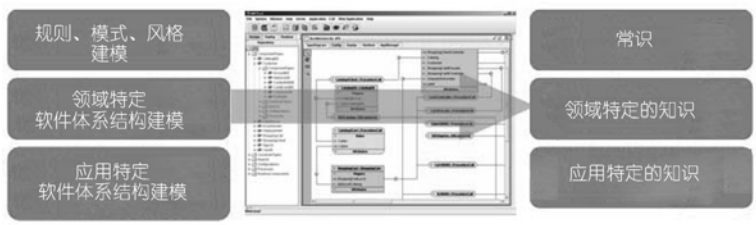


图5 工程化的自适应知识获取

ABCTool的主要功能是支持开发者对网构软件各个阶段SA的建模, 从而规约领域特定的

SA和具体应用的SA. 这些SA模型以及建模的过程均包含有助于自适应的知识. 例如, 在经典的SA设计中, 如果经过评估发现当前SA不满足预期目标, 则需修改SA模型直至通过评估. 传统的SA设计最终产生唯一的满足预期目标的模型, 但网构软件是多目标兼容且目标持续可变, 这意味着某个不满足当前预期目标的阶段性的SA模型, 可能在目标变化后而变得适用, 如果当前SA模型变得不适用, 则需要调整为该阶段性SA模型[12], 这就是一种典型的网构软件自适应. 因此, ABCTool支持一种工程化的自适应知识获取, 即, 将工程化网构软件开发过程及其制品作为自适应知识存储起来. 值得指出的是, 领域特定SA建模产生的自适应知识一般是领域特定的, 而具体应用的SA建模则往往产生特定于该应用的自适应知识. 那些直接使用ABCTool定义的普适性规则、模式、风格则蕴含了常识性的自适应知识.

3 实例研究

ECperf基准测试是测量J2EE应用服务器的行业标准, 主要用来测试基于Java的电子商务软件的可扩展性、性能和总体拥有成本. 本节将以ECPerf为例, 展示如何使用本文提出的SA为中心的自适应方法对其进行调整, 以提高其性能与可靠性. 在实例研究中, 本文使用反射式J2EE应用服务器PKUAS[6]作为底层支撑平台.

3.1 SA 模式驱动的自适应

SA模式是对网构软件中一个范围较小局部结构的抽象, 而这些抽象的局部结构可能会给系统的可信性带来负面影响、正面影响或无影响. 为了提高系统可信性, 需要将具负面影响的模式实例调整为无影响或具正面影响的模式实例. 本文提出的方法支持这种形式的自适应调整[13]: 首先, 管理员使用SA建模工具建立SA模式模型, 包括不良模式模型(描述带来负面影响的模式)和与之对应良好模式模型(描述无影响或带来正面影响的模式); 然后, 通过采集运行时刻信息并与不良模式模型进行匹配, 检测出系统中可能存在的不良模式实例; 通过调整规划得到调整方案, 该方案由本文给出的SA操作原语描述; 最后, 将基于SA模型的调整方案映射为中间件的实际操作, 完成实际的自适应调整. 该过程中, 除第 1 步由管理员手工完成外, 其余步骤均可自动完成.

3.1.1 实例描述

在ECPerf应用中存在如下的不良模式: 如图 6 左半部分所示, 当某个客户需要购物的时候, 它需要若干次调用远程的CartSesEJB中的addItem()方法, 向购物车中添加商品, 最后再通过调用buy()方法提交订单. 由于客户端与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降低性能, 是应用中存在的一个典型的性能反模式, 称为“细粒度远程调用(fine-grained remote calls)”[14].

为去除该不良模式带来的性能影响, 需要对网上书店系统进行调整. 一种调整方案就是通过一次远程调用就将所需要购买的物品信息全部发送给CartSesEJB, 而不是每向购物车中添加一件商品就进行一次远程调用. 可以使用经典的设计模式“会话外观(session facade)”[15]对系统进行调整. 如图 6 右半部分所示, 在服务器端添加一个会话外观构件, 客户端通过一次远

程调用将所需要购买的物品信息全部发送给会话外观构件, 然后再由该构件调用CartSesEJB中的相应方法, 通过一次远程调用完成所有的数据传输.

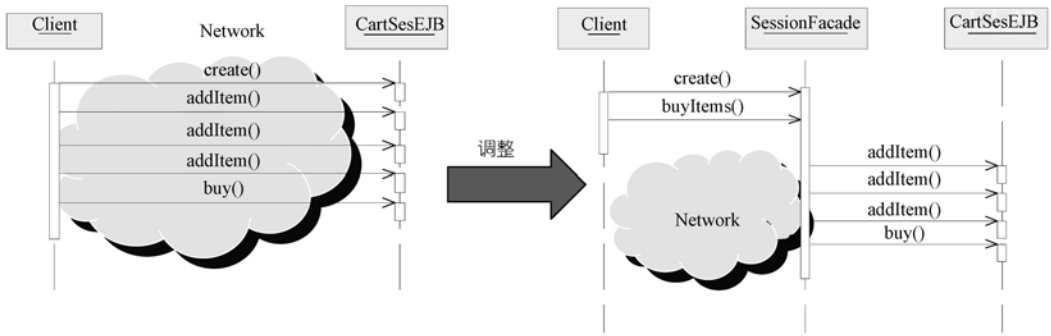


图6 ECPerf 不良模式场景及调整后场景

3.1.2 不良模式检测

为检测该不良模式, 首先需要由管理员建立相应的不良模式模型, 如图 7 所示: 如果一个 EJB 与客户程序之间存在一个调用序列, 该调用序列中第 1 个调用的返回值类型是 EJBObject, 紧接着调用至少 n 个参数不为空的方法, 最后调用一个参数为空的方法, 而且这些方法调用都是远程调用. 那么, 该应用中存在不良模式“细粒度远程调用”的一个实例.

在检测过程中, 该不良模式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 EJB 的方法调用, 所调用方法的返回值、参数以及调用的类型(是远程调用还是本地调用). 信息采集结束以后, 则可以进行不良模式检测. 首先查找出应用系统中所有的 EJB, 然后对 EJB 的调用序列与模型定义的调用序列进行匹配, 最后检测结果为如下的不良模式实例: CartSesEJB 上{create,addItem,addItem,addItem,buy}, 该调用序列存在细粒度远程调用不良模式.

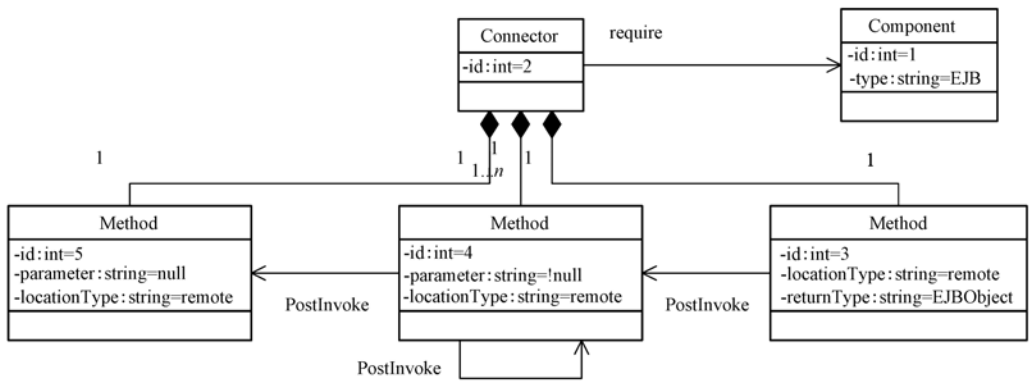


图7 “细粒度远程调用”不良模式模型

3.1.3 建立良好模式模型

图 6 所示的“会话外观”模式需要修改客户端代码才能直接实现, 为了通过中间件自适应

而不是应用代码修改来实现从“细粒度远程调用”不良模式到“会话外观”模式的重构, 我们需要定义一个特殊的良好模式, 如图 8 所示.

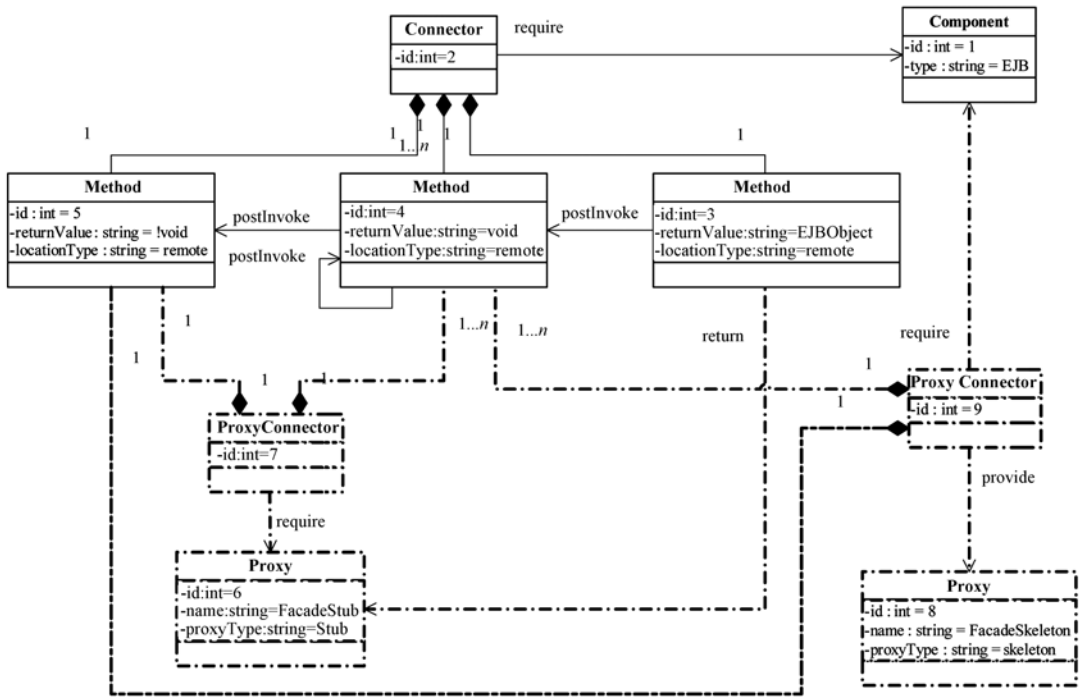


图8 “会话外观”良好模式模型

在分布式环境中, 当位于不同服务器上的构件之间进行方法调用时, 由于它们运行在不同的机器上, 客户端构件其实并不直接调用服务器端构件, 而是调用该构件对应接口的一个实现, 该实现称为一个代理(proxy). 当接收到客户端发来的请求后, 代理构件负责将该请求转发给远程的构件. 服务器端也存在一个代理构件, 它负责接受到远程请求, 对请求信息进行预处理后(例如反序列化调用参数)再对服务器端构件进行调用. 客户端的代理构件一般称为客户桩 Stub, 而服务器端的代理构件一般称为服务器端骨架 Skeleton. 而事实上, 这些代理构件完成了构件之间的通信功能, 它们可以看作连接子的实现. 一般情形下, 代理构件由中间件根据构件的接口自动生成, 以应用无法感知的形态存在于中间件为构件提供的运行环境中, 其主要任务就是帮助应用系统完成网络通信, 允许通过修改代理构件来调整构件之间的交互. 与图 7 所示的不良模式相比, 图 8 的良好模式添加了两个代理构件, 其中, FacadeStub 负责接收客户端发送来的请求并保存这些请求, 当整个调用序列完成后将所有请求的信息发送到服务器端, 它事实上是为客户端添加了 buyItems()方法; FacadeSkeleton 完成会话外观构件的功能, 在接收到 FacadeStub 发送来的多个请求后, 再按照顺序依次将这些请求转发给 CartSesEJB, 当最后的 buy()方法执行完毕后, 再将返回值发送回客户端. 值得指出的是, 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. 在本例中, 良好模式的应用视图没有任何改动, 所有改动都在中间件视图范围内, 这意味着所有改动对应用都是透明的, 即, 自适应调整是在不修改应用代码的前提下完成的.

3.1.4 调整规划与实施

在检测出不良模式实例后, 首先应该根据SA模式模型自动生成调整方案[13]. 通过比较不良模式模型和良好模式模型得到一组调整操作(即两个图之间的差异), 按照操作之间的依赖关系可将这些操作排序, 从而生成图 9 所示的调整方案, 也即SA的动态性描述.

在调整方案生成之后, 需要将其映射为实际的中间件管理操作, 然后由中间件自动完成实际的调整. 本调整方案将映射为 4 个中间件操作:

1: R_E (Proxy, 6, ADD);
2: R_A(Proxy, 6, name, FacadeStub);
3: R_A (Proxy, 6, proxyType, Stub);
4: R_E (ProxyConnector, 7, ADD);
5: R_R (Proxy, 6, ProxyConnector, 7, require, Add);
6: R_R (Method, 3, Proxy, 6, return, Add);
7: R_R(Method, 4, ProxyConnector, 7, composite, Add);
8: R_R (Method, 5, ProxyConnector, 7, composite, Add);
9: R_E (Proxy, 8, ADD);
10: R_A (Proxy, 8, name, FacadeSkeleton);
11: R_A (Proxy, 8, proxyType, Skeleton);
12: R_E (ProxyConnector, 9, ADD);
13: R_R (Proxy, 8, ProxyConnector, 9, provide, Add);
14: R_R (Component, 1, ProxyConnector, 9, require, Add);
15: R_R (Method, 5, ProxyConnector, 9, composite, Add);
16: R_R (Method, 4, ProxyConnector, 9, composite, Add);

图 9 SA 模式驱动的自适应调整方案

1) 生成 FacadeStub 构件: 调整操作 1~3 映射为该操作. 与普通的 Stub 比较, FacadeStub 修改了addItem()与 buy()方法, 不再将 addItem()请求转发给远程的 CartSesEJB, 而是将参数保存并在调用 buy()时将所有保存信息发送到服务器端. 显然, FacadeStub 构件是应用特定的, 但是, 针对细粒度远程调用这一不良模式, 它对于每个函数的修改都是有规律的, 可以由普通的 Stub 自动生成. 因此, 为了能够全自动地完成自适应调整, 管理员在输入了不良/良好模式之后, 还应输入模式相关的一些辅助构件自动生成算法, 这些算法是模式特定的, 可以在不同的应用系统中复用.

2) 添加 Stub 控制器: 调整操作 4~8 映射为该操作. Stub 控制器负责将普通的 Stub 替换为 FacadeStub, 并在客户端调用 Create()方法时返回 FacadeStub.

3) 生成 FacadeSkeleton 构件: 调整操作 9~11 映射为该操作. 与普通的 Skeleton 比较, FacadeSkeleton 修改了 buy()方法, 在该方法中将重新解析由 FacadeStub 传来的多次操作的参数, 并依次对 CartSesEJB 进行调用. 与 FacadeStub 类似, FacadeSkeleton 也可以自动生成, 其自动生成算法是模式特定的.

4) 添加 Skeleton 控制器: 调整操作 12~16 映射为该操作. Skeleton 控制器负责用指定的构件, 即 FacadeSkeleton 替换普通的 Skeleton. 值得一提的是, Stub/Skeleton 控制器提供的是一种通用的机制, 可以在不同的应用或针对不同的模式使用.

3.1.5 自适应效果评估

图 10 展示了自适应前后的性能对比. 整个应用系统的运行环境如下: 部署 ECPerf 应用的机器配置有奔腾 4 2.8G Hz 的处理器以及 1G 内存; 部署 ECPerf 客户端的机器配置有奔腾 4 2.4G Hz 的处理器以及 256M 内存; 安装 ECPerf 应用使用的数据库的机器配置是奔腾 4 2.8G

Hz 的处理器以及 512M 内存; 所有机器之间的网络带宽为 100M. 左图是自适应调整前后进行”addItem*n, buy”调用序列所消费的时间对比, 可以发现用户每多向购物车中添加一个商品, 调整后的优化效果大约为响应时间减少 75 ms. 这是因为每执行一次 addItem()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和传输数据, 在本应用中执行一次这些操作的时间是 70 ms 左右. 而在进行自适应调整之后, 由于使用一次远程调用取代了多次远程调用, 将需要的信息一次发送到服务器端, 减少了网络传输带来的损耗, 因此缩短了响应时间. 右图的性能评测结果是在 ECperf 性能测试结束以后提交的测试报告 Order.summary 中给出的, 对比的性能指标是客户新建一个订单的响应时间 NewOrder Response Time. 可以发现, 在优化后响应时间缩短了 10.5%左右.

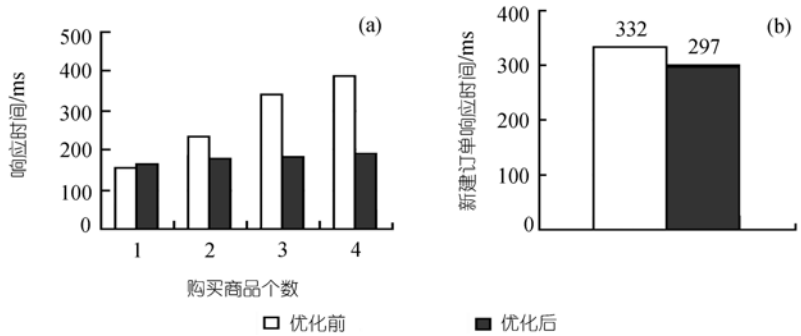


图 10　ECPerf 自适应调整前后性能对比

3.2　SA 风格驱动的自适应

SA 风格是对网构软件整体结构或较大的局部结构的抽象, 与模式类似, 这些抽象的结构可能会给系统的可信性带来负面影响和正面影响或无影响, 为提高系统可信性, 需要将具负面影响的风格实例调整为无影响或具正面影响的风格实例. 与模式不同的是, 风格的全局性意味着可以直接将某种风格作用于运行系统, 本节将以 ECperf 为例介绍如何通过自适应实现容错风格的直接生效.

3.2.1　实例描述

在 ECperf 的客户发出新订单的场景中, 客户首先选择商品的类型和数量, 然后点击浏览器界面中的订购按钮, 生成一个新订单. 服务器端的对应动作是: OrderSes 会话 Bean 调用自己的 newOrder 方法, 在此方法中生成一个与新订单对应的 OrderEnt 实体 Bean 实例. 当这个实体 Bean 被初始化时, 它会首先检查客户资金帐户的余额是否足够支付此次定单、是否为大额订单等, 然后在数据库表中增加一条新记录. 这些活动会调用中间件提供的数据服务, 进行数据库操作. 突发的网络断连或数据库系统瞬时故障会造成管理数据库连接的数据服务发生失效, 数据服务中缓存的数据库连接变为无效连接, 进而导致使用了无效连接的客户请求处理失败.

为降低数据服务失效所产生的失败客户请求的数量, 应使得ECperf具备一定的容错能力. 容错机制大多包含错误检测和恢复两部分, 前者用于发现系统因出现故障而导致的异常状态,

后者则通过状态恢复操作将异常的系统状态恢复至正常状态. 这些容错机制与目标系统的功能正交, 可直接施加到整个系统范围内任何满足某些基本条件的构件之上. 换言之, 这些容错机制可以抽象为容错风格, 以有效提高系统的可靠性和可用性[16,17].

3.2.2 建立容错风格

图 11(a)和(b)分别展示了两种较常见的容错风格, 其中, (a)是用于错误检测的异常捕获风格, 是指基于编程语言的异常处理机制, 对运行时抛出的异常进行截获、分析, 以发现和定位错误; (b)是用于故障恢复的缓冲重启风格, 是指当某一个构件发生失效时, 可以通过重启该构件并将其设置为正确的状态来进行恢复, 在恢复过程中将缓冲所有到达的请求并在恢复成功后重新定向被缓冲的请求. 处理 ECperf 中的数据服务失效需同时进行错误检测和故障恢复, 即, 需同时具备(a)和(b)两种风格, (c)就是这两种风格的混合风格.

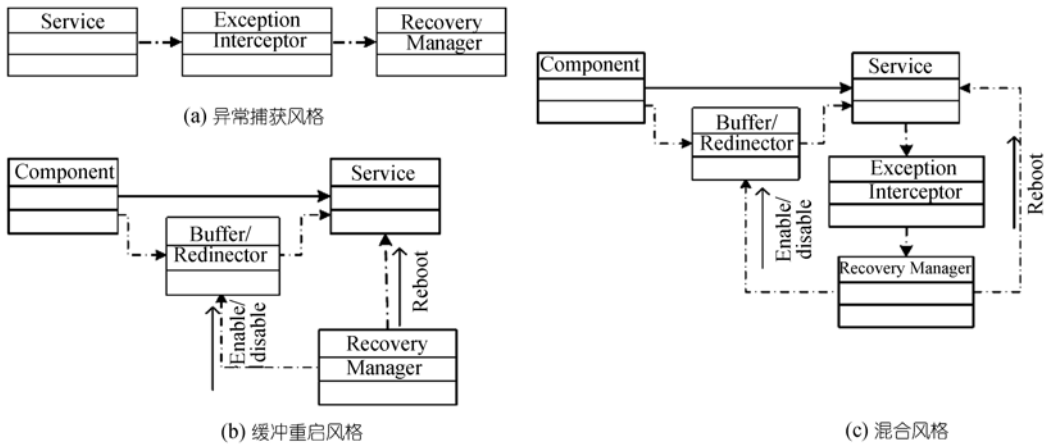


图 11 混合了异常捕获和缓冲重启的容错风格

3.2.3 调整规划与实施

通过对比原始数据服务与满足异常捕获和缓冲重启风格的数据服务, 可以得到一组沟通二者的调整操作, 并得到如图 12 所描述的调整方案. 进一步, 调整方案还要映射为实际的中间件管理操作, 然后由中间件自动完成实际的调整. 首先, 当数据库连接失效并导致数据服务发生故障时, 数据服务会向调用失效数据库连接的实体 Bean 构件抛出异常. 从 SA 的层次来看, 这个异常会沿着数据服务与实体 Bean 构件之间的连接子传递, 可被中间件提供的截取器捕获. 其次, 重启功能可以通过把一个更新数据库连接的截取器插装到数据服务内部来实现, 这个位于数据服务和数据库之间的截取器可以更新已失效的数据库连接. 再次, 在请求转发器和构件实例池之间的截取器可以在数据库连接恢复过程中阻塞调用 OrderEnt 的所有请求, 并在数据库连接恢复成功后解除阻塞. 具体而言, 图 12 所示的调整方案被映射为如下操作:

1) 新增一个用于捕获数据服务异常的截取器 DataServExceptInterceptor, 并加入到原有 EJB 容器的截取器链中(对应调整操作 1~4). DataServExceptInterceptor 截取器实现了 handleException 方法, 所有应用构件抛出的异常要经过这个方法的处理, 并向这个方法提供完

整的异常信息. DataServExceptInterceptor 要对截获的异常信息进行分析、识别, 以确定异常是否来自数据服务.

1: R_E(ExceptionInterceptor, 5, ADD)
2: R_A(ExceptionInterceptor, 5, name, DataServExceptInterceptor)
3: R_A(DataServExceptInterceptor, 5, exceptionType, SQLException)
4: R_R(DataService, 1, DataServiceExceptionMon, 5, thrownSQLException, ADD)
5: R_E(RestartInterceptor, 6, ADD)
6: R_A(RestartInterceptor, 6, name, ConnectionInterceptor)
7: R_R(DataService, 1, ConnectionInterceptor, 6, restart, ADD)
8: R_E(ReqInterceptor, 4, ADD)
9: R_A(ReqInterceptor, 4, name, ReqBufInterceptor)
10: R_R(DataServExceptInterceptor, 5, RecoveryManager, 3, regFaultDetector, ADD)
11: R_R(RecoveryManager, 3, RestartInterceptor, 6, regRecovery, ADD)
12: R_R(RecoveryManager, 3, ReqInterceptor, 4, callee, ADD)

图 12　SA 风格驱动的自适应调整方案

2) 新增一个用于恢复失效数据库连接的截取器 ConnectionInterceptor, 并插装到数据服务中(对应调整操作 5~7). 当数据服务中的数据库连接失效后, 此截取器不断尝试获得新的数据库连接, 并用新的有效数据库连接替换数据服务中的失效数据库连接.

3) 在请求转发器和构件实例池之间增加一个截取器 ReqBufInterceptor, 用于在数据服务恢复过程中阻塞所有欲使用数据服务的新请求(对应调整操作 8 和 9). J2EE 中的任何新请求都要先在实例池中获得一个空闲的实例后才能做下一步处理, 因此, 通过停止分配实体 Bean 实例可以保证不会产生新的数据服务访问请求. 这种有选择的阻塞能力使得那些不使用数据服务的应用构件的执行不受该截取器影响. 当数据服务恢复成功后 ReqBufInterceptor 重新恢复实例分配.

4) 向全局的控制器RecoveryManager中增加一条恢复策略[18](对应调整操作 10~12). 简而言之, 这条恢复策略可以描述为“if(DataServExceptInterceptor发现了数据服务异常)then(调用ConnectionInterceptor恢复数据服务并使用ReqBufInterceptor缓冲新到请求)”. RecoveryManager基于恢复策略协调恢复活动, 并作为中间件的一个服务存在, 其执行过程为: (a) 接收从DataServExceptInterceptor发来的异常信息; (b) 在确定是数据服务发生失效后, 启动ConnectionInterceptor对数据服务进行恢复, 与此同时, 启动ReqBufInterceptor阻塞新到的实体Bean访问请求; (c) 待数据服务恢复正常后, 停止ConnectionInterceptor的执行并解除对实体Bean请求的阻塞.

3.2.4　容错效果评估

为了评估上述自适应效果, 本文使用故障注入方法, 周期性地向 PKUAS 中注入数据库连接失效, 同时在客户端使用开源性能测试工具 OpenSTA 模拟并发的客户请求, 并统计发生据服务失效的情形下客户端生成新订单的请求成功率. 为了比较起见, 分别使用两组恢复配置: (Ⅰ)仅使用数据服务重启动机制; (Ⅱ)同时使用数据服务重启动机制与缓冲/重定向机制. 与此同时, 为评价不同恢复配置对性能的影响, 记录了这两种配置中客户请求的响应时间.

在第 60 和 120 及 180 s处分别注入数据库连接间歇性失效, 实验结果如图 13 所示. 通过

使用风格驱动的自适应为PKUAS的数据服务增加容错能力, 客户端生成新订单的请求成功率从此前的 55.04%[1)]分别提高为 91.13%和 99.38%. 更高的成功率可以通过更复杂的容错风格来实现, 如协同修复[18]等.

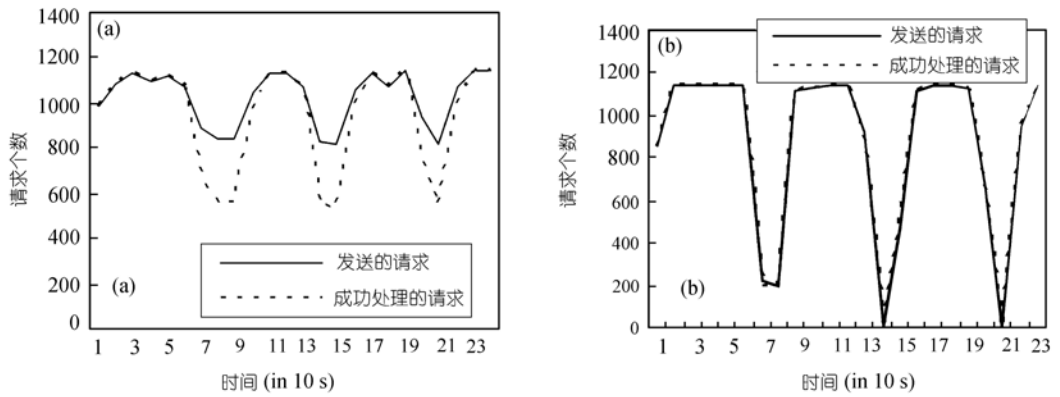


图 13 数据服务自适应的实验结果

(a) 仅使用重启动机制; (b) 使用缓冲重启机制

需要注意的是, 容错机制本身带来了性能损失(如表 1 所示). 当系统无故障运行时, 配置 I 和配置 II 分别造成响应时间延长 1.5%和 1.7%, 这里的性能损失来源于恢复框架. 当系统发生故障时, 上述响应时间分别延长了 12.5%和 12.7%, 此时的性能损失除了包括容错机制本身造成的影响之外, 还有一部分源于对恢复过程中新到请求的处理, 因为当数据服务失效后, 新到达的客户请求会被缓存起来直至数据库连接恢复正常才重新发送出去, 这会增加系统的平均响应时间. 不同的容错配置会得到不同的可靠性和可用性, 但需要付出不同的代价. 这个结果说明, 在网构软件运行环境动态可变的背景下, 需要根据运行时的情形来对容错风格进行动态调整, 也需要将容错与性能调优进行综合考虑.

表 1 不同自适应配置下的平均响应时间/s

	无容错机制	配置 I	配置 II
无故障发生时	0.1021	0.1036	0.1038
有故障发生时	0.1060	0.1193	0.1195

3.3 性能开销

在自适应活动中, 由于对应用系统的监测会增加整个系统的负担, 消耗系统资源, 因此会对应用系统的性能带来负面的影响. 如图 14 所示, 在采集运行时刻信息的情形下, ECPerf 的平均响应时间会增高大约 7%, 而吞吐量会降低 5%左右. 从测试结果可以看出, 运行时刻对应用

1) 对于无任何容错能力的应用来说, 当数据服务第 1 次发生失效后应用就再也无法正常提供功能了, 因此周期性的故障注入对这种情形没有意义. 本文此处给出的请求成功率数据是第 1 个故障注入周期中的请求成功率. 其他不同容错配置下的请求成功率则为多个故障注入周期中的平均请求成功率.

系统的监测不会对应用系统的性能造成严重的负面影响. 更为重要的是, 本文采用为系统构件添加截取器的方式采集运行时刻信息, 因此, 整个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. 此外, 为对应用系统进行更好的管理, 越来越多的应用系统采用了实时的监测方案, 对于这些系统而言, 监测所带来的性能降低已不再是一种开销, 而已经成为了一种不可避免的成本.

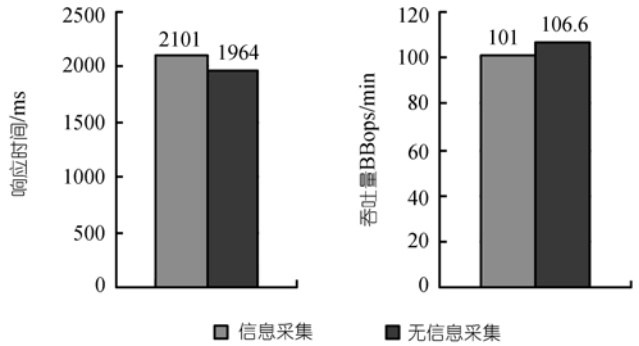


图 14　自适应性能开销

4　相关工作

目前, 产业界对自适应方法与技术的研究非常关注, 并实现了一些具有自适应特性的产品或原型. 其中, IBM研发部门在相关研究上投入最为显著: AutoTune是一个自优化的典型实例[19], 它通过建立CPU和内存与Apache服务器两个配置参数之间的性能关系模型, 实现了Apache服务器自动调整内部资源管理策略以满足给定CPU和内存开销指标的能力; LEO(learning optimizer)是针对DB2 系列产品中提高数据库查询效率的技术, 它使用优化模型对查询操作进行优化, 并通过实际的查询结果不断地修正优化模型, 达到自优化的目的[20]; Jann等针对IBM pSeries Servers上内存、处理器等物理资源的动态重配置, 自动平衡多个系统实例的资源, 完成整个服务器系统的自适应[21]. 此外, 文献[22]介绍了自治计算思想在美国国家航空航天局ANTS项目中的具体实现与应用. 该应用中在采集到运行时刻信息后, 使用预设的算法对系统的多项性能指标进行预测, 最后自动地完成相应的调整. 虽然达到了自优化的目的, 但是上述的工作仍然存在一定的局限性, 它们一般关注自适应中的特定问题, 只能作用于特定的产品或工作场景, 当这些特定的因素改变时, 这些方法或技术便不再适用. 本文以SA为核心为自适应过程中可操作性、可信性等通用问题提供了一种有效解决方案, 具有更好的可复用性.

目前的一些研究者将模型驱动的思想应用到自适应研究中. 文献[23]提出了模型驱动的自治管理的方法, 采集应用系统的状态信息, 然后通过一些经典可靠性模型来分析并预测系统中各个部分的可靠性需求变化, 最后做出相应的调整以保证系统的可靠性. 文中也提到了该方法也可用于系统的自优化提高系统性能、降低成本等. 与本文的方法相比, 在自适应知识的表示方面, 该方法描述了有限的几种可靠性模型以指导自适应工作, 未提供通用性更强的

自适应知识表示机制. 在自适应操作方面, 该方法对于应用系统的调整仅限于一些参数的配置, 缺乏更有效的调整手段.

SA的研究对自适应方法与技术也给予了很大的重视. Richard Taylor[24]提出了一个基于SA的演化方法, 通过对SA中连接子的调整, 优化运行时刻系统的SA, 其中, 对SA的优化操作可以直接映射到使用指定编程框架Java-C2 开发的应用系统上; CMU的Rainbow项目将适应机制从应用系统中分离, 提高适应机制的复用程度的同时也扩大适用范围. 同时, Rainbow中使用体系结构风格作为系统的适应知识, 用于指导系统自适应活动的进行[25]; 国内南京大学提供了一个面向SA的自适应系统, 采用"事件—动作—规则"与"活跃不定式—策略"两种触发机制, 通过RSA完成应用系统的自适应工作[26]. 这些方法同样是使用SA在运行时刻指导系统进行适应活动, 与它们相比, 本文的方法主要有如下一些不同: 在自适应知识的表示方面, 本文采用了ECA、体系结构模式、体系结构风格 3 种表示方式, 较全面地考虑了自适应作用的范围, 其中后两类自适应知识可以图形化表示, 使自适应知识更利于使用和理解; 在自适应调整操作方面, 本文的方法不仅考虑了对应用系统的调整, 还考虑了通过对中间件环境的调整完成应用系统适应的操作方式, 例如本文的实例研究中提到的通过修改代理构件调整应用构件之间的交互, 以及通过添加截取器对服务进行自修复等, 这使得可进行的自适应调整更多样化, 功能也更强大.

5 结论

针对网构软件可适应性和自适应可操作性及适应结果可信性 3 个基本问题, 本文提出了一种基于体系结构的网构软件自适应方法, 即, 监测、分析、规划、实施等自适应活动均围绕软件体系结构展开. 方法的核心是具备反射性、动态性、推理性等新特性的网构软件体系结构模型, 以及相应的全生命周期体系结构建模工具和反射式中间件. 下一步工作主要包括完善目前的原型系统、引入更多的自适应验证方法与机制、通过数据挖掘等方式自动获取自适应知识等.

参考文献

1 杨芙清, 梅宏, 吕建, 等. 浅论软件技术发展, 电子学报, 2002, 30(12): 1901—1906
2 Mei H, Huang G, Zhao H Y, et al. A software architecture centric engineering approach for Internetware. Sci China Ser F-Inf Sci, 2006, 49 (6): 702—730 [DOI]
3 IBM Corp. Autonomic Computing: IBM's Perspective on the State of Information Technology. Oct 2001
4 Shaw M, Garlan D. Software Architecture: Perspectives on an Emerging Discipline. Englewood Cliffs: Prentice Hall, 1996
5 吕建, 马晓星, 陶先平, 等. 网构软件的研究与进展.中国科学, E 辑, 2006, 36(10): 1037—1080
6 Huang G, Mei H, Yang F Q. Runtime software architecture based on reflective middleware. Sci China Ser F-Inf Sci, 2004, 47(5): 555—576
7 Medvidovic N, Taylor R. A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng, 2000, 26(1): 70—93 [DOI]
8 杨杰. 基于软件体系结构的网构软件组装技术研究. 博士学位论文. 北京: 北京大学, 2007

9 梅宏, 陈锋, 冯耀东, 等. ABC: 基于软件体系结构、面向构件的软件开发方法. 软件学报, 2003, 14(4): 721—732
10 刘天成. 中间件服务自修复技术研究. 博士学位论文. 北京: 北京大学研究生院, 2006
11 Zhang N Y, Huang G, Lan L, et al. Pattern-based J2EE application deployment with cost analysis. In: International Conference on Software Engineering and Knowledge Engineering (SEKE), USA: Knowledge System Institute, 2007. 462—466
12 Zhu Y L, Huang G, Mei H. Quality attribute scenario based architectural modeling for self-adaptation supported by architecture-based reflective middleware. In: Proceedings of Asia Pacific Software Engineering Conference (APSEC). Los Alamitos: IEEE Computer Society, 2004. 2—9
13 兰灵. 基于中间件的网构软件自优化技术研究. 博士学位论文, 北京: 北京大学, 2007
14 Dudney B, Asbury S, Krozak J K, et al. J2EE antipatterns. London: Wiley Press, 2003
15 Marinescu F. EJB design patterns: advanced patterns. Processes, and Idioms. New York: John Wiley and Sons Inc, 2002
16 Xu J, Romanovsky A, Randell B. Coordinated exception handling in distributed object systems: from model to system implementation. In: Proceedings of the 18th International Conference on Distributed Computing Systems. Los Alamitos: IEEE Computer Sciety, 1998. 12—21 [DOI]
17 Yuan L, Dong J S, Sun J, et al. Generic fault tolerant software architecture reasoning and customization. IEEE Trans Reliab, 2006, 55(3): 421—435
18 Liu T C, Huang G, Fan G, et al. The coordinated recovery of data service and transaction service in J2EE. In: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC05). Los Alamitors, California: IEEE Computer Society, 2005. 485—490
19 Diao Y, Hellerstein J L, Parekh S, et al. Managing Web server performance with autotune agents. IBM Syst J, 2003, 42(1): 136—149
20 Markl V, Lohman G M, Raman V. LEO: an autonomic query optimizer for DB2. IBM Syst J, 2003, 42(1): 98—106
21 Jann J, Browning L A, Burugula R S. Dynamic reconfiguration: basic building blocks for autonomic computing on IBM pSeries servers. IBM Syst J, 2003, 42(1): 29—37
22 Dai M, Rouff Y S, Rash C A, et al. Modeling for NASA autonomous nano-technology swarm missions and model-driven autonomic computing. In: 21st International Conference on Advanced Information Networking and Applications (AINA 2007). Washington: IEEE Computer Society, 2007. 250—257
23 Dai Y S. Autonomic computing and reliability improvement. In: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05). New York: IEEE, 2005. 204—206
24 Oreizy P, Medvidovic N, Taylor R N. Architecture-based runtime software evolution. In: Proceedings of the International Conference on Software Engineering 1998 (ICSE'98). Kyoto: IEEE Computer Society, 1998. 177—186
25 Garlan D, Cheng S W, Huang A C, et al. Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Comput, 2004, 37(10): 46—54
26 俞春, 马骞, 马晓星, 等. 一种面向体系结构的软件系统自适应机制. 南京大学学报, 2006, 42(2): 120—130