软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(11):3461−3480 [doi: 10.13328/j.cnki.jos.006031]

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

场景驱动且自底向上的单体系统微服务拆分方法*

丁 丹[1,2], 彭 鑫[1,2], 郭晓峰[1,2], 张 健[1,2], 吴毅坚[1,2]

[1](复旦大学 计算机科学技术学院,上海 201203)

[2](上海市数据科学重点实验室(复旦大学),上海 201203)

通讯作者: 彭鑫, E-mail: pengxin@fudan.edu.cn

摘 要: 作为云原生应用的一种典型形态,微服务架构已经在各种企业应用系统中被广泛使用.在企业实践中,许多微服务都是在单体架构的遗留系统基础上通过微服务拆分和改造形成的,其中的拆分决策(特别是数据库拆分)对于微服务系统的质量有着很大的影响.目前,单体系统的微服务拆分决策主要依赖于人的主观经验,整个过程成本高、耗时长、结果不确定性很高.针对这一问题,提出一种场景驱动、自底向上的单体系统微服务拆分方法.该方法以场景驱动的方式,通过动态分析获取单体遗留系统运行时的方法调用和数据库操作信息,然后基于数据表之间的关联分析生成数据库拆分方案,接着再自底向上进行搜索,产生相应的代码模块拆分方案.基于这种方法,实现了一个原型工具 MSDecomposer,将拆分过程可视化,并支持多种维度的反馈调整策略.基于多个开源软件系统进行了案例研究,研究结果表明,该方法能够显著加快微服务拆分决策的速度,减轻开发人员的决策负担,得到的拆分结果是合理的.

关键词: 单体系统;微服务;场景驱动;自底向上;拆分

中图法分类号: TP311

中文引用格式: 丁丹,彭鑫,郭晓峰,张健,吴毅坚.场景驱动且自底向上的单体系统微服务拆分方法.软件学报,2020,31(11):3461−3480. http://www.jos.org.cn/1000-9825/6031.htm

英文引用格式: Ding D, Peng X, Guo XF, Zhang J, Wu YJ. Scenario-driven and bottom-up microservice decomposition method for monolithic systems. Ruan Jian Xue Bao/Journal of Software, 2020,31(11):3461−3480 (in Chinese). http://www.jos.org.cn/1000-9825/6031.htm

Scenario-driven and Bottom-up Microservice Decomposition Method for Monolithic Systems

DING Dan[1,2], PENG Xin[1,2], GUO Xiao-Feng[1,2], ZHANG Jian[1,2], WU Yi-Jian[1,2]

[1](School of Computer Science, Fudan University, Shanghai 201203, China)

[2](Shanghai Key Laboratory of Data Science (Fudan University), Shanghai 201203, China)

Abstract: As a typical form of cloud-native application, microservice architecture has been widely used in various enterprise applications. In enterprise practice, many microservices are formed by decomposing and transforming the legacy system of monolithic architecture. The decomposition decision, especially database decomposition, has a great impact on the quality of the microservice system. At present, the microservice decomposition decision mainly depends on the human subjective experience. The whole process is costly, time-consuming, and uncertain. To solve this problem, this study proposes a scenario-driven, bottom-up microservice decomposition method for monolithic system. This method uses scenario-driven method to obtain the method call and database operation information of monolithic system by dynamic analysis, and to generate database decomposition scheme based on analyzing the association among data tables, and then it searches from the bottom up to generate the corresponding code module decomposition scheme. Based on this method, this study implements a prototype tool MSDecomposer, which visualizes the decomposing process and supports feedback adjustment strategies of multiple dimensions. This study conducts case studies based on several open-source software systems, and the results show

• 基金项目: 国家重点研发计划(2018YFB1004803)
Foundation item: National Key Research and Development Program of China (2018YFB1004803)
收稿时间: 2019-12-02; 修改时间: 2020-02-07; 采用时间: 2020-03-09; jos 在线出版时间: 2020-04-21

that the method proposed in this study can significantly speed up the decision-making of microservice decomposition, reduce the decision-making burden of developers and the final result is reasonable.

Key words: monolithic system; microservice; scenario-driven; bottom-up; decomposition

传统软件系统大多采用所谓的单体架构,即所有代码被统一构建和部署,数据集中化管理.随着需求的变更和软件的演化,单体系统的规模往往会不断变大、复杂度越来越高、使用的技术逐渐过时,导致维护成本大幅提升、新特性的交付周期延长.在过去几年里,大部分云计算基础设施都具备了弹性、自恢复和可伸缩的能力,虚拟化和容器技术发展迅速,这些因素共同促进了微服务(microservices),一种区别于单体架构的软件架构风格被广泛接受和应用.

微服务将一个大型、复杂的软件系统分解成一组独立运行、以轻量级通信机制进行交互的相对较小的服务[1].这些服务围绕业务功能进行构建,可以独立开发、测试、部署和更新,以此带来的好处包括系统架构的解耦、单个服务的交付周期缩短、技术选型更灵活、可扩展性更好、复用性更高等[1].Neal Ford 于 2018 年 12 月发布的"微服务成熟度状态"报告[2]显示,在调查涉及的 866 个受访者中,超过 50%的人表示,他们的组织中有超过一半的新项目使用了微服务,近 70%的人使用了容器作为微服务的部署方式,86%的人认为他们的微服务项目已经取得了部分成功.这很大程度上说明微服务已经成为一种趋势,并成功运用于生产实践中.

在企业实践中,许多微服务系统都是在单体架构的遗留系统基础上,通过微服务拆分和改造形成的.微服务相对于单体架构有诸多优势,但将已有的单体系统拆分成微服务并不是一件容易的事情,Netflix 公司花费了 7 年时间才完成从单体到微服务的架构迁移[3].同时,不正确、不合理的微服务架构反而会加大系统调试和故障定位的难度[4,5].Eric Evans 提出的"领域驱动设计(domain driven design)"[6]方法,由于其所强调的限界上下文(bounded context)的概念能够自然地映射成微服务,被广泛用作微服务设计的指导原则.但这种方法缺乏正式的建模语言和工具支持[7],要求架构师对业务领域有深入的了解,很难在短时间内得出合适的拆分方案.此外,实际的单体系统往往规模庞大、年代久远、实现与文档偏差大,导致原设计文档不可信,加大了人工分析的难度.

针对以上问题,本文提出一种场景驱动、自底向上的单体系统微服务拆分方法.该方法基于测试驱动的理念,通过监控典型应用场景测试用例的执行,动态分析获取单体系统运行时的方法调用和数据库操作信息,生成系统的数据访问轨迹图.根据数据访问轨迹图对底层的数据表图进行加权和聚合,基于数据表之间的关联分析生成数据库拆分方案,接着再自底向上进行搜索,产生相应的代码模块拆分方案.为了验证方法的可行性,本文在使用已有工具 Kieker[8,9]获取系统运行时数据的基础之上,开发了一个微服务拆分辅助工具 MSDecomposer,并在多个开源软件系统上进行了经验研究.实验结果表明,这种方法得到的拆分结果合理,且能够显著加快微服务拆分决策的速度,减轻开发人员的决策负担.

1 背景及相关工作

传统软件系统的模块化重构是软件工程研究领域的一个重要分支,早在 1972 年,Parnas 就提出了对系统进行模块化解构的标准[10].随后,围绕着信息隐藏、高内聚低耦合等原则,依赖静态结构分析、性能数据分析、仓库挖掘等手段,诞生了一系列软件模块化方法和工具[11−14].然而,微服务的模块化和重构作为微服务实践的一大挑战[15],目前仍然缺少实用的方法和工具支持[16],相关研究主要集中在经验总结和方法论上.

Francesco 等人[17]对 18 位在工业界真实参与了微服务化迁移改造的工程师进行了访谈,将迁移过程分成了逆向工程、架构转移和前向工程这 3 个子过程,分别总结了这 3 个子过程在具体实施时遇到的主要问题和挑战.Taibi 等人[18]则将微服务化改造过程分成 3 种类型,包括现有功能的迁移、重新开发以及通过"绞杀模式"实现新功能,并总结提出了一个迁移过程框架.Jonas 等人[19]在文献数据库中进行关键词搜索,筛选出 10 篇与微服务拆分相关的论文,总结其中涉及的重构方法,并将方法分成了静态代码分析、元数据(架构描述、UML 图等)导向、工作流数据(通信数据、性能检测数据等)导向以及动态微服务拆分(依据运行时环境、资源消耗等信息)这 4 种类别,对开发设计人员在重构方法的选择上给予一定的决策指导.

在以"领域驱动设计"为指导原则的微服务拆分方面,前人已做出不少探索,但在设计之初,需要投入较多精

力进行业务分析和领域建模.Rademacher 等人[20]针对领域驱动设计中出现的细节缺失问题,提出一种模型驱动的开发方法,通过引入中间模型,明确了微服务接口和部署细节.其中提到的 AjiL[21]工具可以用于微服务系统的可视化建模以及中间模型的生成.Levcovitz 等人[22]通过人工识别子系统、对数据库表进行分类,从而根据代码静态依赖图自底向上进行微服务划分.Chen 等人[23]分析了业务需求、绘制数据流图,将具有相同输出数据的操作和对应数据划分为一个微服务.

单体系统的微服务化拆分也可以借鉴传统软件模块化的思路和方法,已有不少这方面的研究.Service Cutter[24]是一个用于服务拆分的可视化工具,输入是用户自定义的一系列接口操作和用例,根据操作间的耦合程度对接口进行聚类划分.但是 Service Cutter 要求用户编写所有输入文件,时间成本很高,并不适用于大型系统的改造.Abdullah 等人[25]则使用系统访问日志和无监督的机器学习方法将系统自动分解为具有相似性能和资源要求的 URL 组,将每组 URL 映射为一个微服务.这种方法将请求文档的大小和日志中的 URI 请求响应时间作为机器学习的输入特征,缺点在于只将性能相似度作为度量标准,没有考虑设计上的内聚性或耦合性,拆分结果不利于服务的修改和扩展.Mazlami 等人[26]通过分析代码和变更历史,根据代码的逻辑、语义等方面的关联,生成类之间的耦合关系图,最后对图进行聚类、得到微服务拆分方案.而 Jin 等人[27]认为,程序的很多行为并没有显式反映在源码中,且代码级别的关联并不完全等同于功能的内聚.他们通过监控系统、动态收集代码的执行路径,实现了一种面向功能的微服务抽取方法.以上两种方法分别从静态和动态两个角度对系统进行分析,再寻找恰当的聚类方法对系统进行以类为最小移动单位的拆分.但是传统软件模块化方法并不能实现对单体系统的数据库拆分,而没有进行数据库拆分的微服务拆分是不彻底的.随着系统的演化和数据量的增长,数据库操作将成为整个系统的性能瓶颈.在数据拆分之后,一个类的不同方法可能会由于所操作的数据不同而被划分到不同的微服务中,所以不可避免地会涉及到对类的拆分,这就要求在拆分粒度上达到比类更细化的方法级别.

单体系统的微服务拆分存在不同于单体系统模块化的地方.Taibi 等人[28]就指出,从单体到微服务,应该是去识别可以从单体中隔离出来的独立业务流程,而不仅仅是对不同 Web Service[29]的特征提取;微服务之间的访问,包括私有数据和共享库,都必须被小心地分析.前文提到的工作,一部分缺乏对独立业务流程的识别,另一部分没有将数据作为重要的拆分对象、给予充分的关注.本文在这两点上做了尝试和结合,以现有的代码和数据模式为出发点,将业务流程、方法调用链和数据表这三者进行关联,使得业务的独立性、代码的内聚耦合性、数据间的关联度都成为最终的拆分依据.同时,本文所提方法的自动化程度和推荐方案的完整度更高,减少了人工输入的数据量和人工分析时间.

2 微服务拆分方法

现实中的大部分应用系统都是以数据为中心、围绕数据库进行构建的,因此,系统对于数据的访问方式很大程度上体现了业务逻辑.本文所提出的微服务拆分方法将数据关联作为微服务拆分的主要依据,通过分析不同业务场景下的数据访问规律,生成数据拆分方案,进而生成代码模块的拆分方案.方法的输入为一套覆盖了单体系统绝大部分业务场景的测试用例,每个用例标记一个权重,表示该用例对应业务场景的相对重要程度.方法的输出为系统的微服务拆分方案,包括每个微服务应该包含的数据表和代码.

方法的整体流程如图 1 所示,可分为“生成数据访问轨迹图”、“生成微服务拆分方案”、“反馈调整”这 3 个主要部分,整个过程是迭代可控的.

- 第 1 部分由“标记测试用例权重”“在单体系统上配置监控工具”“运行测试用例、收集日志”和“构建数据访问轨迹图”这 4 个步骤组成,主要是利用已有的监控工具 Kieker 对单体系统在不同业务场景下的方法调用和数据访问轨迹进行监控和记录.
- 第 2 部分对应图 1 中的第 5 步~第 9 步,即通过分析数据访问轨迹生成数据表之间的关联权重图,利用聚类算法生成数据表的拆分方案,再根据数据访问轨迹图自底向上进行搜索,生成代码拆分方案.
- 图 1 中表示为虚线的附加步骤代表方法中的第 3 部分,通过人工的反馈对结果做出微调,使得最终的拆分方案更符合实际需求.

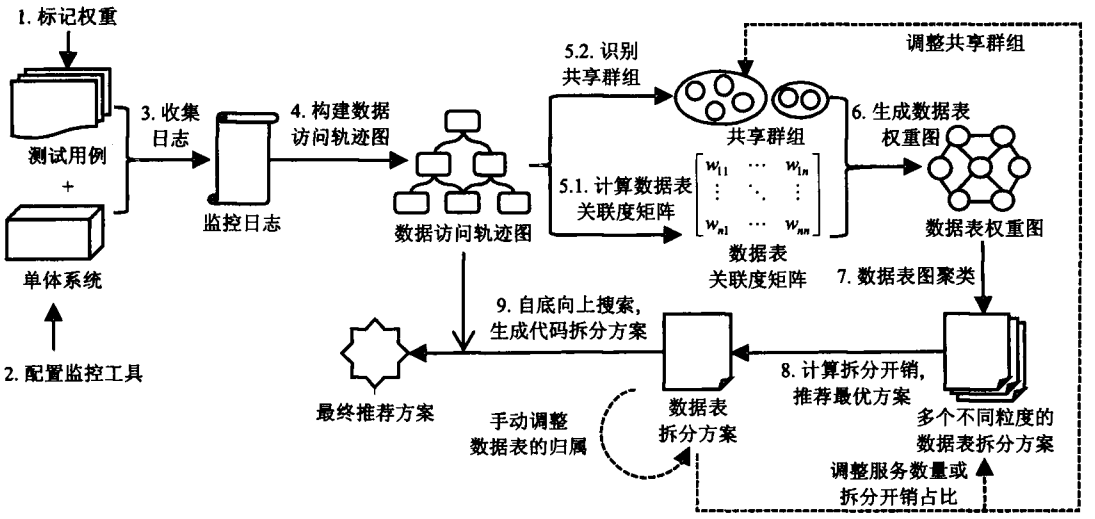


Fig.1 Microservice decomposition scheme generation process

图 1 微服务拆分方案生成流程

2.1 生成数据访问轨迹图

本文使用场景级别的测试用例,即每个用例(use case)对应一个用户使用级别的场景(scenario).例如,一个管理员发通告的场景,包含了查询办公室列表、根据办公室查询用户信息、保存通告内容、刷新页面这 4 个请求.每个请求在服务器端产生一条方法调用链(trace),由处理这个请求所调用的一系列方法(method)组成,其中有些方法会执行 SQL 语句(SQL)、对数据表(table)中的某些字段进行增删改查.图 2 的模型描述了上述概念和它们之间的关系.收集所有测试用例执行的方法调用链,并将他们与对应的 SQL 语句、数据表相关联,就形成了一张数据访问轨迹图.

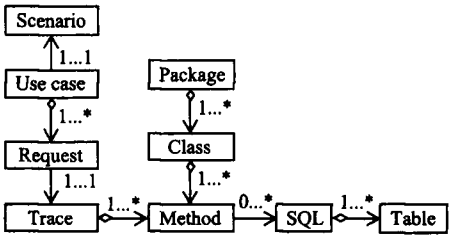


Fig.2 Model of data access trajectory graph

图 2 数据访问轨迹图模型

2.1.1 输入准备

本文通过在原单体系统运行测试用例的方法收集数据、构建数据访问轨迹图,故需要准备以下输入信息.

(1) 测试用例:用户使用场景级别的测试用例,一个场景包含若干个界面输入或点击操作.用例数量没有限制,但至少覆盖所有主要业务流程和功能.测试既可以通过运行自动化测试用例的方式,也可以通过手动执行的方式实现;

(2) 用例权重:每个用户使用场景级别的测试用例对应一个用例权重,权重大小根据对应场景的重要度进行设置,默认为 1.对于一些比较重要或者使用频率很高的业务场景,例如火车站订票系统的订票场景、购物网站的下单流程,可以适当增加权重.

绝大多数情况下,获取以上信息所花费的时间要少于深入分析系统、构建领域模型的时间,且一个对系统比较了解的开发人员、测试人员甚至系统管理人员就可以给出相对完整的信息,无需有架构设计的相关背景.

2.1.2 数据访问轨迹图生成

输入测试用例和对应权重,通过监控系统的代码执行路径,生成如图 3 所示的数据访问轨迹图.

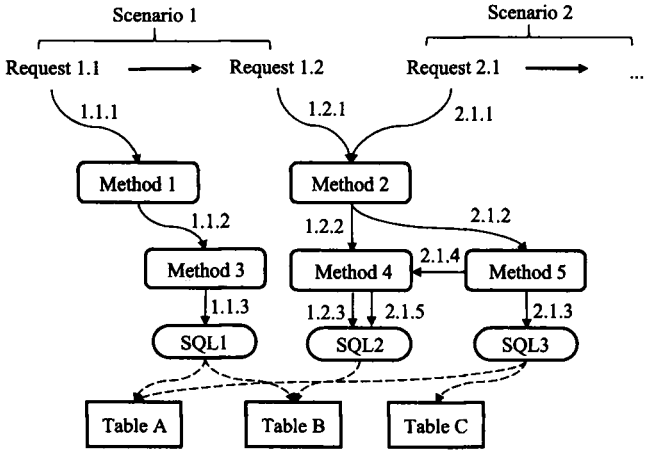


Fig.3 Data access trajectory graph

图 3 数据访问轨迹图

其中每个场景(scenario)由若干个请求(request)串联而成,每个请求对应一条方法调用链,调用链的起点为服务端接收请求的第一个方法(一般情况下是控制层(controller)方法).调用链可能存在分岔情况,例如请求 2.1 (request 2.1)对应的调用链中,方法 5(method 5)会先执行 SQL3,对表 A(table A)和表 C(table C)进行操作,然后调用方法 4(method 4)、执行 SQL2,对表 B(table B)进行操作,所以请求 2.1 对应的调用链可以表示如下.

$$\text{Trace 2.1: Method 2} \rightarrow \text{Method 5}\begin{cases}\rightarrow \text{SQL3} \rightarrow \text{Table A, Table C}\\ \rightarrow \text{Method 4} \rightarrow \text{SQL2} \rightarrow \text{Table B}\end{cases}.$$

每条调用链都有唯一的 ID 标识,即使同一场景下产生的两条调用链对应的代码执行路径完全相同(即所经过的方法和 SQL 语句以及执行顺序完全一样),这两条调用链的 ID 也是不同的.调用关系用单向箭头表示,箭头指向被调用的方法或 SQL 语句,箭头上的标号也是唯一的,表示本次调用所属的调用链 ID 和调用序号.

2.2 生成微服务拆分方案

根据上一步生成的数据访问轨迹图,可以结合微服务化拆分有关的多个考量维度,生成一张无向加权的数据表图,然后通过对数据表进行聚类划分,得到数据表的拆分方案,最后在数据访问轨迹图中自底向上进行搜索、获取与数据表拆分结果相符合的代码拆分方案.

2.2.1 考量维度

(1) 数据关联度:出现在同一个 SQL 语句、请求或者场景中的数据之间具有更加密切的关联关系,这种关联与相应 SQL 语句、请求或者场景的执行频率正相关,表示这些数据经常被一起操作.为了信息隐藏和减少数据耦合,这些数据表更倾向于划分到同一个微服务中.执行频率相同的情况下关联度的强弱:同 SQL 语句>同请求>同场景.

(2) 数据共享度:被多个不同的场景、请求或 SQL 语句操作的数据,共享程度比较大.以系统的日志表为例,每当服务端进行一项数据操作后,都会向日志中插入一条新的记录,这种情况下,日志和所有业务数据都有一定的关联度,即日志表被所有业务数据表所共享.但是从设计上来看,日志表不应该划分给任何一个业务服务,依据单一职责的原则,应该单独将其作为一个微服务提取出来.与关联度相反,对于共享度的影响:场景>请求>SQL 语句.可以理解为,被多个场景共同访问的数据和代码更有可能是一个独立的业务功能.

(3) 拆分开销:由于本文关注的是已有单体系统的拆分,不仅要考虑微服务划分的合理性,也应该根据实

际的拆分代价决定拆分粒度,如果原代码的耦合度过高,可以考虑先以大粒度拆分服务、再逐步细化.一般情况下的开销:SQL 拆分>方法拆分>方法移动(即类的拆分).

2.2.2 生成数据表权重图

从数据关联度和共享度这两个维度出发,将数据访问轨迹图转化成如图 4 所示的数据表图,图中每个顶点表示数据库中的一张表,根据以下方法向图中添加边,并计算权重.

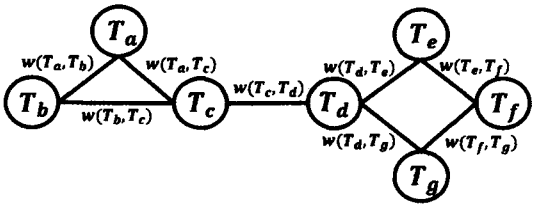


Fig.4 Database table graph

图 4 数据表图

(1) 基础定义

a) 定义数据访问轨迹图中共包含 $N_{scenario}$ 个场景、N_{Trace} 条方法调用链、N_{SQL} 条 SQL 语句和 n 张表.

b) 第 i 个场景 $Scenario_i$ 的权重定义为 $W_{Scenarioi}$,由设计人员根据业务场景的重要程度,于每个测试用例执行之前进行设置,即为"输入准备"步骤中的"用例权重".

c) 第 i 条方法调用链 $Trace_i$ 的权重定义为 W_{Tracei},等于这条调用链所属的场景的权重.

d) 第 i 条 SQL 语句 SQL_i 的权重定义为 W_{SQL_i},等于每个场景的权重与其执行 SQL_i 次数的乘积的累加和.

$$W_{SQL_i}=\sum_{k=1}^{N_{Scenario}} W_{Scenario_k}\cdot T_{Scenario_k(SQL_i)} \tag{1}$$

其中,$T_{Scenario_k(SQL_i)}$ 表示第 k 个场景执行 SQL_i 的次数.

(2) 计算数据表关联度矩阵

对于两个数据表 T_a,T_b,从场景、方法调用链和 SQL 语句这 3 个级别分别计算它们之间的关联度.

a) 定义 $Scenario(T_i)$为操作了数据表 T_i 的场景的集合,T_a,T_b 两个数据表关于场景级别的关联度表示为 $C_{Scenario}(T_a,T_b)$,等于同时操作 T_a,T_b 这两张表的场景的累加权重与操作其中任意一张表的场景的累加权重之比.

$$C_{Scenario}(T_a,T_b)=\frac{\sum W_{Scenario\in Scenario(T_a)\cap Scenario(T_b)}}{\sum W_{Scenario\in Scenario(T_a)\cup Scenario(T_b)}} \tag{2}$$

b) 定义 $Trace(T_i)$为操作了数据表 T_i 的方法调用链集合,T_a,T_b 两个数据表关于调用链级别的关联度表示为 $C_{Trace}(T_a,T_b)$,等于同时操作 T_a,T_b 这两张表的调用链的累加权重与操作其中任意一张表的调用链的累加权重之比.

$$C_{Trace}(T_a,T_b)=\frac{\sum W_{Trace\in Trace(T_a)\cap Trace(T_b)}}{\sum W_{Trace\in Trace(T_a)\cup Trace(T_b)}} \tag{3}$$

c) 定义 $SQL(T_i)$为操作了数据表 T_i 的 SQL 语句集合,T_a,T_b 两个数据表关于 SQL 级别的关联度表示为 $C_{SQL}(T_a,T_b)$,等于同时操作 T_a,T_b 这两张表的 SQL 语句的累加权重与操作其中任意一张表的 SQL 语句的累加权重之比.

$$C_{SQL}(T_a,T_b)=\frac{\sum W_{SQL\in SQL(T_a)\cap SQL(T_b)}}{\sum W_{SQL\in SQL(T_a)\cup SQL(T_b)}} \tag{4}$$

将 3 个级别的关联度按下式进行加权累加,得到表 T_a,T_b 的总关联度 $C_{Total}(T_a,T_b)$.

$$C_{Total}(T_a,T_b)=\alpha_1\cdot C_{SQL}(T_a,T_b)+\alpha_2\cdot C_{Trace}(T_a,T_b)+\alpha_3\cdot C_{Scenario}(T_a,T_b) \tag{5}$$

其中,α_1=0.6,α_2=0.3α_3=0.1.

(3) 识别共享群组

共享度为单个数据表的特性,不能直接反映在边的权重上.本文首先识别出共享度较高的数据表,再根据这些表之间的依赖程度进行分类,最终得到若干个共享群组.每一个共享群组包含若干张共享表,且这些表之间存在较强的依赖关系,倾向于共同提取出来作为一个微服务.

对于第 i 张表 T_i,定义其场景、调用链和 SQL 这 3 个级别的共享度如下.

a) 场景级别的共享度 $S_{Scenario}(T_i)$,等于操作表 T_i 的场景数量占总场景数量的比例.

$$S_{Scenario}(T_i)=\frac{|Scenario(T_i)|}{N_{Scenario}} \tag{6}$$

b) 调用链级别的共享度 $S_{Trace}(T_i)$,等于操作表 T_i 的调用链数量占总调用链数量的比例.

$$S_{Trace}(T_i)=\frac{|Trace(T_i)|}{N_{Trace}} \tag{7}$$

c) SQL 级别的共享度 $S_{SQL}(T_i)$,等于操作表 T_i 的 SQL 语句数量占总 SQL 语句数量的比例.

$$S_{SQL}(T_i)=\frac{|SQL(T_i)|}{N_{SQL}} \tag{8}$$

3 个级别的共享度按下式进行加权累加,得到表 T_i 的总共享度 $S_{Total}(T_i)$.

$$S_{Total}(T_i)=\beta_1\cdot S_{SQL}(T_i)+\beta_2\cdot S_{Trace}(T_i)+\beta_3\cdot S_{Scenario}(T_i) \tag{9}$$

其中,β_1=0.2,β_2=0.8,β_3=1.

计算每张表的共享度并从高到低排序,根据一定策略截取前 x 张表作为共享表.实际上,x 的值和表总数 n 有一定关系,n 越大,x 也应该越大.假设共享表数量占总表数量的比例为 y,通过定义一些特殊点(见表 1)获取 x 和 y 的拟合关系曲线,如图 5 所示.

Table 1 Special points of total table number and shared table proportion

表 1 表总数与共享表占比特殊点

表的总数 n	10	20	30	50	100	150 以上
共享表占比 y	0.3	0.25	0.2	0.15	0.1	0.08

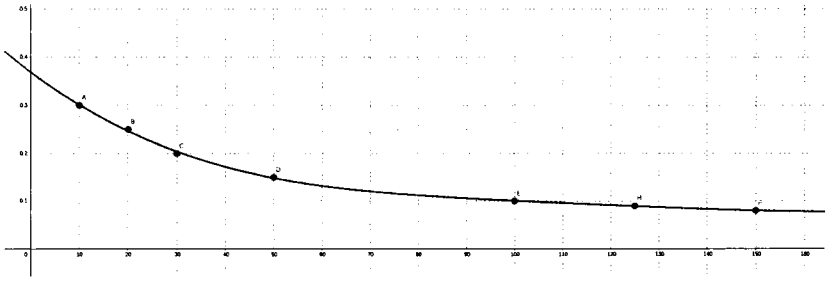


Fig.5 Fitting curve of total table number-shared table ratio

图 5 表总数-共享表占比拟合曲线

最终共享表的数量即为表总数与共享表占比之积,再取整.

$$x=\lfloor n\cdot y\rfloor \tag{10}$$

截取一定数量共享度高的数据表后,还需要对这些表之间的依赖关系进行分析.共享表 T_a 对共享表 T_b 的依赖度同样分为 SQL、调用链和场景这 3 个级别.

a) T_a 对 T_b 的 SQL 依赖度定义为 $D_{SQL}(T_a\rightarrow T_b)$,等于同时操作 T_a,T_b 这两张表的 SQL 语句数量占操作 T_a 的 SQL 语句数量的比例.

$$D_{SQL}(T_a\rightarrow T_b)=\frac{|SQL(T_a)\cap SQL(T_b)|}{|SQL(T_a)|} \tag{11}$$

b) T_a对 T_b的调用链依赖度定义为 $D_{Trace}(T_a \to T_b)$,等于同时操作 T_a,T_b这两张表的调用链数量占操作 T_a的调用链数量的比例.

$$D_{Trace}(T_a \to T_b) = \frac{|Trace(T_a) \cap Trace(T_b)|}{|Trace(T_a)|} \tag{12}$$

c) T_a对 T_b的场景依赖度定义为 $D_{Scenario}(T_a \to T_b)$,等于同时操作 T_a,T_b这两张表的场景数量占操作 T_a的场景数量的比例.

$$D_{Scenario}(T_a \to T_b) = \frac{|Scenario(T_a) \cap Scenario(T_b)|}{|Scenario(T_a)|} \tag{13}$$

T_a对 T_b的依赖度可以看作是 T_a对 T_b的"数据依附"关系,依赖度越高,这种依附关系越明显,T_a越倾向于和 T_b在一起.但依赖度是双向的,如果 T_a对 T_b的依赖度很高,而 T_b对 T_a的依赖度很低,可以认为 T_b和 T_a有一种"主从"关系,而 T_b与其他多个表之间也可能存在这种"主从"关系.比较典型的例子如图 6 所示,在实验系统 JeeSite[30]中一共有 22 张数据表,sys_log 表负责记录每一次数据操作的日志,对其他数据表(如图中的 sys_user, sys_office 等)的操作都会同步插入一条 sys_log 数据.所以这些数据表会呈现出对 sys_log 表的强依赖,但对 sys_log 表的操作却是独立于这些表的.

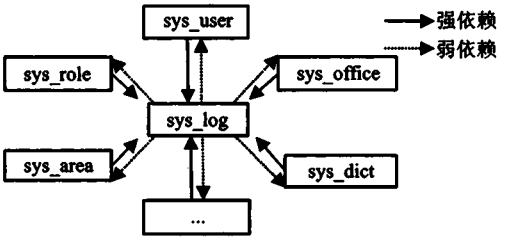


Fig.6 Database table dependency graph around sys_log table

图 6 以 sys_log 表为中心的数据表依赖图

为了避免将以 sys_log 表为中心的多个不同子领域的表都划分到一起,只有当满足以下 4 个条件之一时,T_a和 T_b才会被归入同一个共享群组.

- 条件 1:$D_{SQL}(T_a \to T_b)+D_{SQL}(T_b \to T_a)>\gamma_1$.
- 条件 2:$D_{Trace}(T_a \to T_b)+D_{Trace}(T_b \to T_a)>\gamma_2$.
- 条件 3:$D_{Scenario}(T_a \to T_b)+D_{Scenario}(T_b \to T_a)>\gamma_3$.
- 条件 4:$D_{SQL}(T_a \to T_b)+D_{SQL}(T_b \to T_a)+D_{Trace}(T_a \to T_b)+D_{Trace}(T_b \to T_a)>\gamma_4$ 并且

$$D_{Scenario}(T_a \to T_b)+D_{Scenario}(T_b \to T_a)>\gamma_5.$$

其中,γ_1=1.4,γ_2=1.6,γ_3=2,γ_4=2.8,γ_5=1.

根据以上条件,识别出的 x 张共享表被分成 r 个共享群组 $G'_1,...,G'_r$.最后,考虑一种特殊情况,即对于一张非共享表 T_c,如果 T_c无论是在 SQL 级别、调用链级别还是场景级别,只跟一个共享群组 G_d中的表有关联,那么我们倾向于把 T_c和 G_d中的表划分到一起.所以,需要遍历所有非共享表,将只跟一个共享群组有关联的表加入该群组中,得到最终的共享群组 $G_1,...,G_r$.

(1) 数据表权重矩阵生成

根据公式(5)计算 n 张数据表两两之间的关联度,得到 $n \times n$ 的对称矩阵 M,矩阵的第 i 行第 j 列元素 m_{ij}等于 $Table_i$与 $Table_j$之间的总关联度 $C_{Total}(T_i,T_j)$.显然,$m_{ij}=m_{ji}$.矩阵 M 对角线上元素值为 1.

遍历每个共享群组,针对第 p 个群组 G_p中的共享表 T_q,对矩阵 M 的第 q 行和第 q 列元素做如下调整.

$$m_{qi}=m_{iq}=\begin{cases}\max(m_{iq},\delta_1), & T_i \in G_p \\ \delta_2 \cdot m_{iq}, & T_i \notin G_p\end{cases}, \quad i=1,...,n \tag{14}$$

其中,δ_1=0.9,δ_2=0.2.

公式(14)的意义在于,对于连接同一个共享群组中两张表的边,保持一个比较大的权重,增加它们被划分到同一个微服务的概率;而与共享表相连的其他边的权重削减为原来的 0.2 倍,降低该边相连的两张表被划分到同一个微服务的可能性.经过调整的矩阵 M 将会作为数据表图的邻接矩阵成为下一步图聚类算法的输入.

2.2.3 数据表图聚类

图的聚类方法有很多种[31−33],本文经过实验对比后,选择了 Girvan-Newman 算法(下文称 G-N 算法)[34−36]用于数据表图的聚类,被聚到一起的表最后会被划分到同一个微服务中.作为一种经典的社区发现算法,G-N 算法认为:社区网络的重要特性是连接两个社区的边会有更高的权重,如果将这些边找出来并删除,剩下的网络就被很自然地划分为多个社区.为此,算法提出使用边介数(betweenness)作为衡量一条边是否应该从图中删除的标准.一条边的边介数定义为网络中经过这条边的所有最短路径的数量.G-N 算法不断重复"计算边介数"与"删除边介数最高的边"这两个步骤,直到图中所有边都被删除.每次删边后算法都会重新计算社区结构,最终会生成一棵社区结构树(如图 7 所示),树的每一层对应一种对原网络的划分(partition),自顶部向下,社区数量会越来越多,划分粒度越来越细,最后,每个顶点都是一个独立的社区.这种连续的、粒度逐步细化的树形结构正好适应于微服务拆分过程,第 2.3 节所描述的"反馈调整"也是基于 G-N 算法的这种特性.

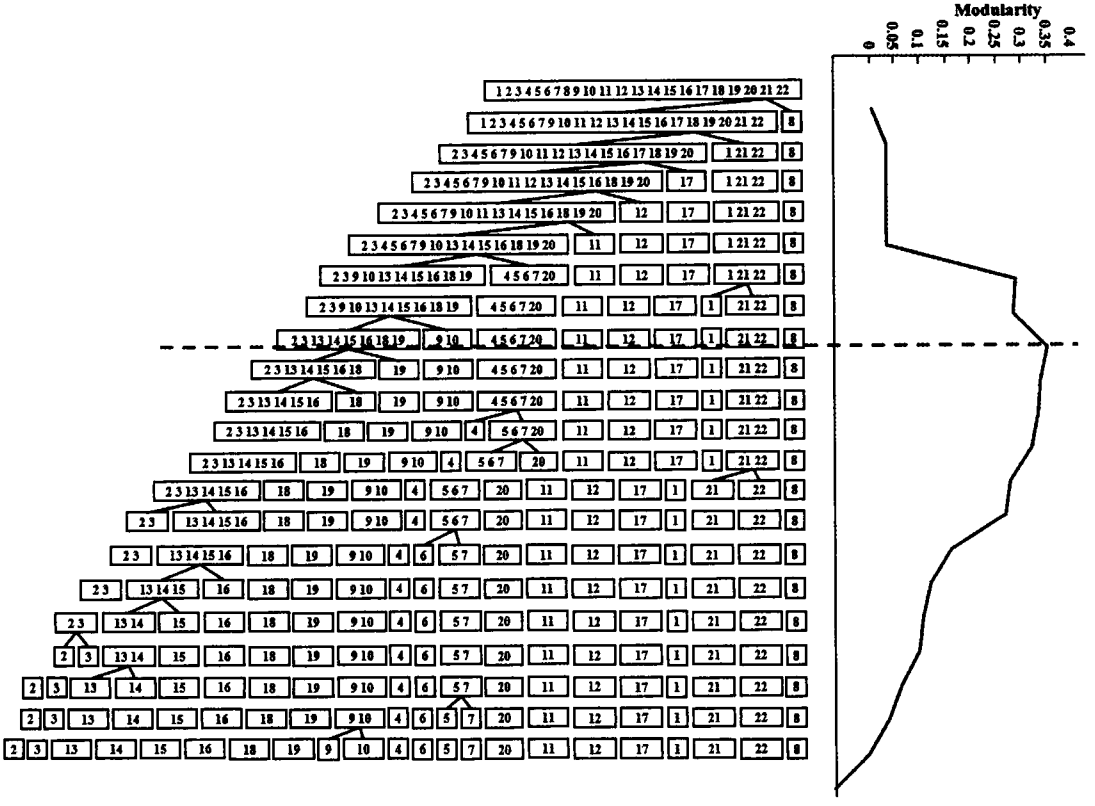


Fig.7 Partition process of database tables

图 7 数据表划分过程

为了从多种不同粒度的拆分方案中选择一个最合适的推荐方案,G-N 算法引入模块度(modularity)[36]作为衡量标准,其定义如式(15)所示.

$$Modularity = \sum_{i=1}^{c}(e_{ii} - a_i^2) \tag{15}$$

公式(15)中,e_{ii} 表示社区 i 内所有边的权重占整个网络所有边权重的比例,a_i 表示与社区 i 内顶点相连的所有边的权重占整个网络所有顶点所连边的权重的比例.模块度的取值范围为[−0.5,1),值越大,说明网络具有越

强的聚类特性.遍历社区结构树的每一层,计算对应划分方案 P_x 的模块度,作为推荐方案的一个指标参数 $Score_{Modularity}(P_x)$.整个数据表图的聚类划分过程如图 7 和算法 1 所示(图 7 展示了实验系统 JeeSite 中 22 张数据表的真实聚类过程对应的社区结构树和模块度计算结果,每个数字代表一张数据表,一个方框中的所有表对应于一个服务,其中的虚线标明了模块度最高的拆分方案).

算法 1. 数据表图聚类算法.

输入:数据表图的邻接矩阵 *M*.

输出:数据表图的社区结构树和每一层对应的模块度.

```
1:  function Cluster(M)
2:      根据邻接矩阵 M 构建图 G,G.v 为图的点集,G.e 为图的边集
3:      maxModularity  //记录最大模块度
4:      maxBetweenness  //记录最大边介数
5:      Map<Modularity,Partition> partitionMap  //保存模块度和对应的划分结果
6:      while G.e≠∅
7:          for edge in G.e
8:              计算 edge 的边介数 edge.betweenness
9:              maxBetweenness=Max{maxBetweenness,edge.betweenness}
10:         end for
11:         for edge in G.e
12:             if edge.betweenness==maxBetweenness
13:                 remove edge from G.e  //删除边介数等于最大边介数的边
14:             end if
15:         end for
16:         获取图 G 当前的划分结果 P_current
17:         计算 P_current 的模块度 Score_Modularity(P_current)
18:         maxModularity=Max{maxModularity,Score_Modularity(P_current)}
19:         partitionMap.put(curModularity,curPartition)
20:     end while
21: return partitionMap
```

2.2.4 计算拆分开销

给定一个数据表划分方案 P_i,可以依据原单体系统的静态代码结构,从 SQL 语句、方法、类这 3 个层面自底向上进行拆分开销的计算.如果一条 SQL 语句操作的若干个数据表归属于两个及以上不同的微服务,则该条 SQL 语句需要进行拆分.一条 SQL 的拆分代价定义为 μ_1,需要拆分的 SQL 总数定义为 $V_{SQL}(P_i)$.同理,如果一个方法包含的 SQL 语句需要拆分或者该方法包含了两条属于不同微服务的 SQL 语句,那么它也需要进行拆分,对应的拆分代价定义为 μ_2,需要拆分的方法总数定义为 $V_{Method}(P_i)$.最后,除非一个类中所有方法都属于同一个微服务,否则需要进行类的拆分,拆分代价定义为 μ_3,需要拆分的类的总数定义为 $V_{Class}(P_i)$.拆分总开销 $Cost(P_i)$通过式(16)计算.

$$Cost(P_i)=\mu_1\cdot V_{SQL}(P_i)+\mu_2\cdot V_{Method}(P_i)+\mu_3\cdot V_{Class}(P_i) \tag{16}$$

其中,$\mu_1=1,\mu_2=0.5,\mu_3=0.1$.

假设数据表图划分过程共产生了 K 种不同粒度的划分方案,给每个方案的拆分开销按式(17)打分.

$$Score_{Cost(P_i)}=\frac{\mathrm{Max}\{Cost(P_i),i=1,...,K\}-Cost(P_i)}{\mathrm{Max}\{Cost(P_i),i=1,...,K\}-\mathrm{Min}\{Cost(P_i),i=1,...,K\}} \tag{17}$$

由于最终的推荐方案倾向于减小拆分开销,所以将所有方案拆分开销的最大值与最小值之差作为分母,将

最大值与当前方案的拆分开销之差作为分子,开销越大,分子越小,分数越低.

2.2.5 确定拆分方案

最终的拆分决策需要结合模块度和拆分开销两个方面,依据公式(18)计算数据表拆分方案 P_i 的总分 $Score_{Total}(P_i)$,选择总分最高的一个方案作为数据表的拆分推荐方案.

$$Score_{Total}(P_i)=\omega_1\cdot Score_{Modularity}(P_i)+\omega_2\cdot Score_{Cost}(P_i) \tag{18}$$

其中,$\omega_1+\omega_2=1$.

确定了数据表的拆分方案后,对于每个服务中的数据表,在第 2.1 节生成的数据访问轨迹图中,顺着调用关系搜索操作这些表的 SQL 语句、方法和类,即可得到和数据拆分对应的代码拆分方案.最终的拆分方案包含了每个服务对应的数据表、SQL 语句、方法和类,以及需要拆分的 SQL 语句、方法和类.

2.3 反馈调整

不同单体系统间的巨大差异性,决定了最终的拆分方案只是作为实际拆分时的重要参考,而不能完全取代人工分析.方案中可能存在一些不合理或者不现实的地方,为了使推荐方案更贴近用户的实际需求,在确定最终拆分方案的整个过程中,用户可以对一些参数和中间结果做出调整,反馈行为将作用在拆分方案生成的不同时期.具体有以下 4 种反馈调整方式.

(1) 调整共享群组:由于共享群组的识别是在数据表图进行聚类划分前完成的,所以在运行聚类算法之前,可以先由用户确认提取出的共享群组是否合适.这样做是因为共享表的数量是由拟合曲线推算出来的估计值,现实中的两个不同的系统即使数据表的数量相同,应该被提取出的共享表数量也不一定相同.所以,通过人工确认的方法可以删除掉那些明显不适合单独提取出来的表,避免生成权重矩阵时错误地削减了相应边的权重.除此之外,用户可以根据自己对于业务领域和数据模型的了解,手动添加或者修改共享群组,从而起到间接优化权重矩阵的作用.

(2) 调整模块度与拆分开销的比例:实际拆分中,往往由于时间限制或代码质量的原因(比如原代码的耦合度过高),系统拆分人员想要减少 SQL 语句和方法的拆分数量,达到先将整个系统拆开再逐个细化的目标.公式(18)中的两个参数 ω_1 和 ω_2 分别代表了模块度与拆分开销在计算总分时的占比,可以调整这两个参数的比例,将拆分开销控制在用户可以接受的范围之内.

(3) 调整服务数量:由于时间或者资源的限制,拆分人员可能为了赶工期而减少服务数量,也可能由于单台部署机器的存储资源有限而选择细化数据表的拆分粒度.如第 2.2.3 节所述,G-N 算法天然地生成了一系列粒度不同的数据表划分方案,可以从中选择一个最符合用户期望的粒度.

(4) 手动调整数据表的归属:在个别情况下,无论如何调整参数,最终结果也无法达到用户的预期.那么在给出最终方案后,用户可以手动调整个别数据表所归属的服务.这种情况下,同样可以给出相应的代码拆分方案,作为实际拆分的指导.

3 工具实现

本文基于第 2 节的微服务拆分方法实现了一个基于 Spring Boot[37]框架的原型工具 MSDecomposer,它包含“动态监控”、“数据处理”和“可视化”这 3 个模块.其中,“动态监控”模块基于已有的开源工具 Kieker 实现了对系统运行时数据的收集功能;“数据处理”模块读取系统运行时数据,构建数据访问轨迹图;“可视化”模块便于工具使用者进行反馈调整,同时对微服务拆分方案进行展示.

本节最后给出了工具的性能测试结果,并对结果进行分析.

3.1 动态监控

Kieker[8,9]是一个开源的应用性能监控工具,通过对目标系统进行动态代理,获取代码执行路径.监控信息被写入日志文件中,其中每一条记录表示一次方法调用,包含了被调用方法的签名、调用链 ID、调用时间、调用顺序和堆栈深度,利用这些信息,可以还原系统运行期间所有的方法调用链.

为了记录场景信息,本文对 Kieker 进行了改造,新增图 8 所示的页面.在每个测试用例执行前,输入对应的场景名(scenario name)和权重(scenario weight),整个用例执行期间生成的每条日志都会带有这两项信息.

Scenario Name	Start Time	End Time	Scenario Frequency
注册	2019/8/20 下午3:22:28	2019/8/20 下午3:22:32	1.0
登录	2019/8/20 下午3:22:38	2019/8/20 下午3:22:40	1.0
查看商品	2019/8/20 下午3:23:02	2019/8/20 下午3:23:03	1.0
添加购物车	2019/8/20 下午3:23:25	2019/8/20 下午3:23:27	1.0
下订单	2019/8/20 下午3:23:44	2019/8/20 下午3:23:45	1.5

Fig.8 Scenario input page after modifying Kieker

图 8 Kieker 改造后的场景输入界面

为了将方法调用链和操作的数据相关联,每次方法调用返回后,在 Kieker 中拦截 JDBC 执行的 SQL 语句,使用语法解析工具 JSqlParser[38]获取 SQL 语句操作的数据表名,再将拦截的 SQL 语句和对应的数据表名连同调用链 ID 等属性一同打印到日志文件中.

3.2 数据处理与分析

Kieker 监控运行时系统,并将监控日志写入日志文件中.MSDecomposer 读取日志文件,根据日志中每条记录的调用链 ID、调用实体等信息,构建数据访问轨迹图.同时,MSDecomposer 从日志文件中解析出项目的静态结构(例如包(package)、类(class)和方法(method)).为了方便后续的查看和搜索,MSDecomposer 将数据访问轨迹和项目的静态结构一起存储在图数据库 Neo4j[39]中.

图 9 展示了实验系统 Spring JPetStore[40]中“修改个人信息”这个测试场景在 Neo4j 中关联的子图,图的左半边是场景涉及的类和包的结构;右半边对应于图 2 的数据访问轨迹图,从上到下依次为场景涉及的方法、SQL 语句和数据表.不同的节点类型使用不同的颜色表示,节点之间的连线表示包含或调用关系,记录着场景名、权重等关键信息.由于在进行数据存储时创建了两个虚拟节点 Root 和 Entry,分别作为包结构的根节点和方法调用链的入口节点,所以这两个节点会出现在所有场景的关联子图中.

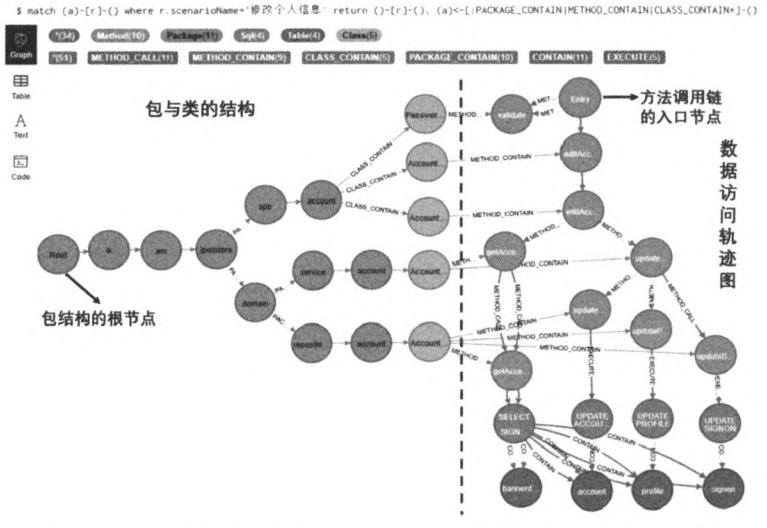


Fig.9 Data access trajectory and static structure of ‘modify personal information’ scenario

图 9 “修改个人信息”场景的数据访问轨迹和静态结构

3.3 用户反馈与调整

MSDecomposer 提供了可视化界面,便于用户的反馈调整和拆分方案的展示.如图 10 所示的“拆分方案页面”主要用于接收用户反馈,生成数据表的拆分方案.页面分为左、右两栏:左栏展示数据表拆分的推荐方案以及需要拆分的 SQL 语句、方法和类的数量;右栏从上到下 3 个组件分别用于调整拆分开销占比、服务数量和共享群组.每次用户调整参数后,页面左栏会刷新,显示重新计算后的推荐方案.在经过多次调整、确定数据表拆分方案后可以进入“手动调整页面”(如图 11 所示).

Fig.10 Interface of MSDecomposer——Split proposal page

图 10 MSDecomposer 界面—拆分方案页面

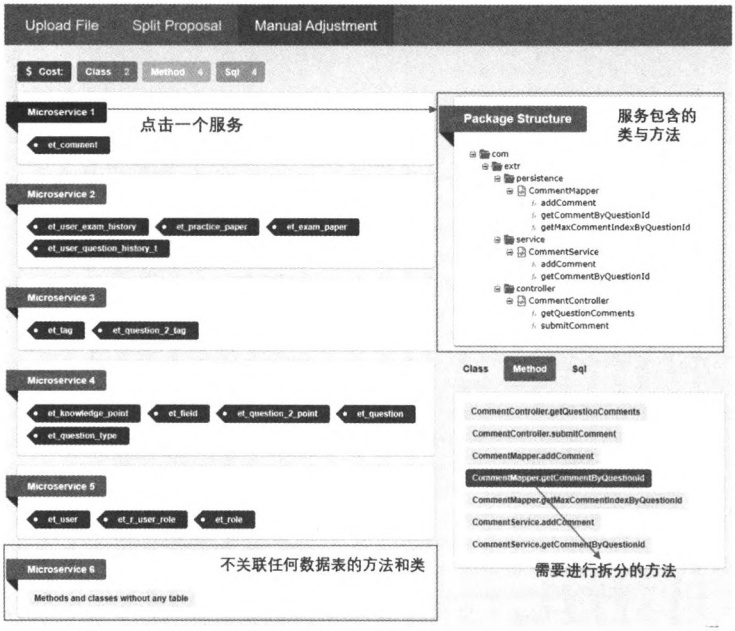


Fig.11 Interface of MSDecomposer——Manual adjustment page

图 11 MSDecomposer 界面—手动调整页面

在"手动调整页面",用户可以拖动标签、改变任意数据表的服务归属.点击其中一个微服务标题,页面右上角部分会显示与该服务包含的数据表相关联的代码结构,粒度精确到方法级别,且与原单体系统的包结构相对应.页面右下角显示该服务包含的所有 SQL 语句、方法和类,其中,箭头标识出的表示需要进行拆分.另外,存在一些可以监控到、但没有与任何数据表相关联的方法和类(比如某些读取缓存、直接返回的场景),由于没有实际执行 SQL 语句,无法依据数据关联将它们划分到任何一个服务中,所以 MSDecomposer 将这些代码收集到一个单独的微服务中,设计人员或实施拆分的人员可以根据实际情况自行决定这些代码的服务归属.

3.4 算法性能测试

在完成监控工具配置、场景整理和数据收集后,实际使用工具的时间很短,其中耗时最长的部分在于使用 G-N 算法对数据表图进行聚类.由于 G-N 算法在边介数的计算上时间复杂度很高,所以这种算法一般用来处理几百个节点的中小规模网络.在拥有 2.5GHz CPU,8GB RAM 的计算机上单独运行 G-N 算法,随机生成含有 20 个~100 个顶点的数据表图,以 44%的比例生成点与点之间的边,边的权重为大于 0 小于 1 的随机值.其中,44%的比例值是根据实际数据表图的边数与对应的全连接图的边数之比计算出的平均值.相同顶点数量的实验重复 10 次,求得平均边数与平均耗时,得到性能测试结果见表 2.

Table 2 Performance test results of G-N algorithm

表 2 G-N 算法性能测试结果

顶点数量	平均边数	平均耗时(ms)
20	82	588.4
30	193	1 853.2
40	343	5 050.7
50	542	8 332.8
60	779	15 142.4
70	1 055	30 102.2
80	1 383	59 033.2
90	1 759	119 139.6
100	2 198	201 506

可以看出,100 个顶点以内的数据表图,G-N 算法可以在 4min 内得出结果.我们认为,这样的性能对于普通的中小型项目是完全可以接受的.对于一些数据库规模很大、有成百上千张数据表的项目,人工分析其代码结构和数据关联的难度也会相应增大,据我们所知,可能需要花费几周甚至几个月的时间.相比之下,几个小时或者几天的算法运行时间并不会让人觉得不可忍受.当然,在这种情况下,也可以使用其他算法复杂度更低的聚类算法进行替换,例如谱聚类算法[31].实际上,MSDecomposer 已经内置了谱聚类和其他几种算法的实现,但是这些算法由于并不会像 G-N 算法一样生成社区结构树,也就无法实现基于社区结构树的反馈调整策略.

4 案例研究

本文选取了 4 个开源软件系统进行案例研究,研究系统分属电商、ERP(企业资源计划)、线上测试和社交博客等个不同的领域类型,项目规模在开源软件系统中相对较大.下面首先以其中一个系统 Spring JPetStore 为例,直观地给出本文所提出的微服务拆分方法的输入输出和中间结果,通过与文献[27]中的方法结果和适用范围进行比较,验证本文方法可以得出正确合理、可解释、且更加精准的拆分方案.案例研究的第二部分挑选了 5 位具有微服务研究背景和开发经验的硕士研究生,对 4 个实验系统进行了人工分析并给出微服务拆分方案.通过对比,总结本文提出的方法有哪些优缺点,且其是否符合微服务开发者的经验直觉、是否能够真正地加速拆分决策过程.最后,讨论部分指出了本方法的几点局限性.

4.1 Spring JPetStore案例分析

Spring JPetStore[40]是一个基于 Spring[41],Spring MVC 和 Mybatis[42]框架实现的小型宠物商店系统,包含了用户认证、商品展示、下订单等一系列电商网站必备的功能模块.该系统最早是 Sun 公司开发的一个项目示例,用于展示如何使用 Mybatis 框架.由于其结构简单、功能完善,常被用作实验和教学.本文以这个经典的电子商

务平台系统作为案例,阐述微服务拆分方法的分析过程.

首先,通过人工分析 Spring JPetStore 的主要功能和流程,我们得出了如表 3 所示的 8 个主要测试场景;然后,人工为每个场景分配一定的权重,权重的取值范围为[1,2].考虑到"登录"和"下订单"是比较重要的场景,而"浏览商品"出现频率最高,所以适当地增加了这 3 个场景的权重,其他场景权重均为最小值 1.表 3 同时给出了每个场景中进行了数据操作的调用链类型和对应的入口方法名,其中重复出现的调用链类型不再标记入口方法名.所谓的调用链类型是指:如果两条调用链执行的方法、SQL 语句以及执行顺序完全相同,那么这两条调用链的调用链类型相同.不同的场景可能会包含相同的调用链类型,比如"浏览商品"和"添加购物车"都有 T4 和 T5 这两种调用链类型.需要说明的是,如果两条调用链的入口方法名相同,即访问的后台接口相同,对应的调用链类型不一定相同.访问同一个后台接口,传入参数的不同,可能导致调用链执行的方法或执行顺序不完全相同(例如后台逻辑存在 if-else 分支,根据传入参数的值做不同的操作),但这种情况在 Spring JPetStore 项目中并没有出现.

Table 3 Test scenarios list of Spring JPetStore

表 3 Spring JPetStore 测试场景列表

测试场景	权重	调用链类型编号(调用链入口方法名)
注册	1	T1(newAccount)
登录	1.5	T2(loadUserByUsername)
搜索商品	1	T3(searchProducts)
浏览商品	2	T4(viewCategory),T5(viewProduct),T6(viewItem)
添加购物车	1	T4,T5,T7(addItemToCart)
下订单	1.5	T8(newOrder),T9(viewOrder)
查看订单	1	T10(listOrders),T9
修改个人信息	1	T11(editAccount)

为了直观地展示调用链与数据表的关联,表 4 列出了表 3 中所有调用链类型执行的 SQL 语句编号及每条 SQL 语句操作的数据表(重复出现的 SQL 语句不再标记数据表).同样可以看到,不同的调用链类型可能会执行相同的 SQL 语句,虽然筛选查询条件(即 WHERE 子句后的条件)的参数传入值会不同,但是它们对应于同一个数据访问层方法、操作的数据表是一样的.

Table 4 Trace type list of Spring JPetStore

表 4 Spring JPetStore 调用链类型列表

调用链类型	执行的 SQL 语句编号(操作的数据表名)
T1	SQL1(account),SQL2(profile),SQL3(signon)
T2	SQL4(account,profile,signon,bannerdata),SQL5 (product)
T3	SQL6(product)
T4	SQL7(category),SQL5
T5	SQL8(item,product),SQL9(product)
T6	SQL10(product,item,inventory)
T7	SQL11(inventory),SQL10
T8	SQL12(sequence),SQL13(sequence),SQL14(inventory),SQL15(orders),SQL16(orderstatus),SQL17(lineitem)
T9	SQL18(orders,orderstatus),SQL19(lineitem),SQL10,SQL11
T10	SQL20(orders,orderstatus)
T11	SQL21(account),SQL22(profile),SQL23(signon),SQL4,SQL5

使用 MSDecomposer 对 Spring JPetStore 进行监控和分析,从监控到的 12 张表中提取出了一个共享群组,包含 category(宠物大类,例如猫、狗),product(具体种类,例如斗牛犬、吉娃娃),item(更加详细的体型/年龄/性别划分,例如成年/未成年、雌性/雄性)和 inventory(每个 item 对应的库存数量)这 4 张和宠物相关的数据表.不对该共享群组做任何修改,同时也不对微服务数量和拆分代价所占比例(默认为 0,不考虑拆分代价)做任何定制化调整,即跳过"反馈调整"的步骤,直接生成最终拆分方案.方案共产生 4 个微服务,原系统中的数据表被划分到了 3 个微服务中 MS1~MS13,而 MS4 包含了所有与数据表无关的方法和类.

- MS1 包含 4 张与用户相关的表,分别是 account(账户表,记录用户的姓名、邮件、住址等信息),profile(用户偏好表,记录用户喜欢的宠物类型),signon(专门记录用户名和密码)和 bannerdata(存储用户偏好的宠

物种类对应的头像图片).

- MS2 包含了共享群组中的 4 张与宠物商品有关的表:category、product、item 和 inventory.
- MS3 包含了 4 张与订单有关的表,分别是 orders(记录订单详细信息,包括下单用户、下单时间、地址等信息),orderstatus(记录订单状态),sequence(产生订单的全局唯一 ID)和 lineitem(订单中的商品项,每一项包含一个 item 和对应的购买数量).
- MS4 不包含任何数据表和 SQL 语句,只有两个类及类中包含的方法.其中,PasswordEqualsValidator 类实现了 Spring 框架的 Validator 接口,用于校验前端输入的新密码和重复密码是否相等.由于在校验后直接返回结果,所以该类不会和数据库中的数据产生关联.第二个类是 Cart(购物车),由于数据库中没有购物车这张表,所以该类始终保存在内存中,用户添加或删除购物车中商品的行为都只是对内存中 Cart 实例的操作,所以同样不会和任何数据表有关联.

对于 MS4,从设计角度上来说,可以选择将其中的 PasswordEqualsValidator 类和 MS1 合并、将 Cart 类和 MS3 合并,最终,Spring JPetStore 被分为用户、商品和订单这 3 个微服务.从类和接口的分配结果来看,和文献[27]中的实验结果是一致的,这一定程度上验证了本文所提方法的正确性.

另外,即使拆分代价所占比例为 0,计算出的拆分方案也不需要拆分任何的 SQL 语句或者方法.原因是在生成数据表图时,已经适当地增加了出现在同一条 SQL 的数据表之间的权重,所以拆分结果一定程度上倾向于少拆 SQL 语句.

Spring JPetStore 是一个比较理想的系统,它的结构简单、实现规范,得出的微服务拆分方案并不需要拆分任何的 SQL 语句、方法或者类.本文共实验了 4 个系统(如表 5 所示),其中,Exam++(模拟驾考系统)和 JeeSite(ERP 系统)这两个系统在同样没有任何反馈调整的情况下,计算出的拆分方案需要进行一定数量 SQL 语句、方法和类的拆分.这种情况在现实中很常见,比如,由于原始设计或者实现的原因,一条 SQL 语句合并查询了多张数据表,如果这些数据表最终需要拆分到不同的微服务中,那么这条 SQL 语句以及执行这条 SQL 语句的方法都需要进行拆分,然后将本地方法调用转化成远程调用.文献[27]中的方法无法识别这种情况,因为它最小的拆分粒度为类,即一个类和它所有的方法最终都只能归属于同一个微服务.另外,同样是从数据开始拆分,和 Levcovitz 等人[22]通过人工识别子系统对数据表进行分类相比,MSDecomposer 实现了对数据的半自动化拆分,节省了人工分析的时间.而和已有工具 Service Cutter[24]相比,MSDecomposer 的输入准备只有粗粒度的场景级别的测试用例,Service Cutter 需要使用者以固定格式编写所有的系统操作和用例描述,显然更加复杂、费时.

Table 5 Decomposition result of experimental systems

表 5 实验系统拆分结果

项目名称	数据表数量	微服务数量	需要拆分的类数量	需要拆分的方法数量	需要拆分的 SQL 数量
Spring JPetStore	12	3	0	0	0
zb-blog[43]	15	3	0	0	0
Exam++[44]	15	5	2	4	4
JeeSite[30]	22	9	10	27	33

4.2 对比实验

本文挑选了 5 位具有微服务研究背景和开发经验的硕士研究生,让他们分别对本文的 4 个实验系统进行人工分析,并给出数据表和代码的拆分方案.针对每个实验系统,参与者得到的信息包括系统的 Git 地址、数据库地址以及网页访问地址.他们可以访问系统启动后的前台和管理员界面,了解系统的主要功能和流程,也可以看到系统的数据模型和源码.

如表 6 所示,在分析每个系统的过程中,实验参与者会记录从开始到得出完整的数据表拆分方案、类划分以及方法划分方案的时间.其中,类和方法的划分方案是指要明确每个类或每个方法归属于哪一个或哪几个服务,允许实验参与者通过分析一部分代码后进行总体的估算.考虑到参与者不一定都是按照"先分析数据表、再分析代码"这样的原则进行拆分,所以数据和代码拆分的时间可能会有重合的部分,还需要再记录总的分析时

间,这个总时间小于等于数据表、类和方法拆分时间的总和.最终的平均时间统计使用截尾平均数,即去掉一个最长时间和一个最短时间后的3组数据的平均值,这样做的目的是降低极端值对结果的影响.

Table 6 Time records of manual decomposition (min)

表6 人工拆分时间记录 (分)

项目名称	实验参与者	数据表拆分时间	类划分时间	方法划分时间	总分析时间
Spring JPetStore	A	41	19	33	93
	B	85	120	250	455
	C	42	6	13	61
	D	45	20	20	65
	E	90	195	645	930
	截尾平均	57.3	53	101	204.3
zb-blog	A	49	38	55	142
	B	150	100	300	550
	C	68	14	22	104
	D	57	35	35	92
	E	110	220	755	1 085
	截尾平均	78.3	57.7	130	265.3
Exam++	A	44	34	43	121
	B	60	150	240	450
	C	72	20	48	140
	D	65	20	20	85
	E	245	430	1410	2 085
	截尾平均	65.7	68	110.3	237
JeeSite	A	67	52	82	201
	B	120	80	240	440
	C	91	53	103	247
	D	80	40	40	120
	E	190	320	1095	1 605
	截尾平均	97	61.7	141.7	296

为了与人工拆分时间相对比,表7给出了使用MSDecomposer进行拆分的时间记录.其中,Kieker配置时间是指在实验系统中配置改造后的Kieker监控工具,并成功启动实验系统所经历的时间;场景整理时间是指分析系统功能,并得到所有场景级别的用例和对应权重的时间;数据收集时间是指运行所有测试用例、获取监控日志的时间;工具使用时间就是使用MSDecomposer从数据处理、运行拆分算法到生成最终拆分方案的时间.由于这些操作是顺序发生的,所以总耗时就是前面4项操作耗时的总和.

Table 7 Time records of MSDecomposer Tool (min)

表7 MSDecomposer工具拆分时间记录 (分)

项目名称	Kieker配置	场景整理	数据收集	工具使用	总耗时
Spring JPetStore	40	30	20	3	93
zb-blog	50	40	30	5	125
Exam++	30	60	30	5	125
JeeSite	60	70	40	8	178

对比表6和表7的数据可知,人工拆分的平均总耗时都是大于工具拆分的总耗时的.这表明MSDecomposer的确能够加速整个拆分决策的过程.工具加速的主要环节在于给定数据表拆分方案的前提下,自底向上搜索获取类和方法的拆分方案.同时,工具存在着很大的优化空间,其中,Kieker的配置步骤根据项目结构、使用技术的不同而不同,初始时需要耗费一定时间摸索配置,如果之后遇到类似的框架技术可以运用之前的经验减少配置时间.除此之外,目前的数据收集是手动输入场景信息、点击按钮来区分不同场景的测试用例,以后可以考虑自动化实现——导入场景信息与测试用例的映射文件、测试用例执行前后在回调函数内部调用相应接口自动实现场景切换.

值得注意的是,5位实验参与者对于同一系统经过人工分析得出的数据划分方案并不是完全相同的,差异主要集中在拆分粒度上.以实验系统JeeSite为例,5位实验参与者给出的数据表拆分方案对应的服务数量最少

的是 4 个,最多的有 19 个.其中,4 个服务的方案简单地将系统按照原本的项目模块分成了“系统设置”、“内容管理”、“办公自动化”和“代码生成”这 4 个部分,而 19 个服务的方案则对这 4 个部分做了更进一步的细化,比如将“办公自动化”这个部分细化成了“请假”、“通告”和“审核”这 3 个服务.目前,在微服务拆分的粒度方面并没有权威的方法论指导,需要视系统的实际需求和使用场景进行权衡.

随后,本文向实验参与者展示了 MSDecomposer 的使用方法和拆分结果.首先,5 位参与者都认可工具以场景级别的测试用例作为数据拆分依据具有一定的合理性,且都认同相比于领域模型和其他设计模型,我们能够更为轻松地获得这些测试用例.对 MSDecomposer 的拆分方案评分(满分 5 分),5 位参与者分别给出了 5 分、4 分、3.5 分、3 分和 3 分.令我们感到高兴的是,所有参与者都对 MSDecomposer 给出的拆分方案持认可的态度,他们认为:生成的方案符合人的直觉,对帮助他们了解系统、得到最终的方案有很大的帮助.另外,参与者们都反映 MSDecomposer 工具简单易学,能够迭代反馈,相比人工分析更加细致和高效.

4.3 讨 论

上述案例研究验证了本文提出的单体系统微服务拆分方法在实际系统的拆分中可以很好地运行,在更短的时间内生成理想的微服务拆分方案.然而,本文的实验和方法还存在以下几点不足.

- 第一,参与对比实验需要具有软件工程和微服务领域的研究背景,本文寻找到的符合要求的实验参与者人数较少,与实际从事微服务开发的从业人员相比,在实践经验等方面存在一定的差距,导致实验结果缺乏足够的代表性.
- 第二,可以用于案例研究的开源单体系统很少,且数据库规模普遍较小.实际生产系统可能会有成百上千张数据表,本文提出的方法能否在这样的系统上得出比较好的结果仍未可知,但至少提供了一种新的思路.
- 最后,场景的划分对实验结果有一定的影响.由于本文的假设前提是同一场景中涉及的数据和代码倾向于划分到同一个微服务中,所以划分场景、识别每个场景中包含的用户操作对于得出正确的拆分方案十分重要.这要求测试用例的设计者对系统的功能和业务流程有一定程度的了解,如果对场景的识别不够准确或不够全面,都可能对最终的结果产生负面的影响.

5 总 结

微服务系统相对于单体系统有着可扩展性强、复用性高等诸多优点,但从单体到微服务的拆分过程目前主要依赖于人工分析,成本高、耗时长.本文提出了一种场景驱动、自底向上的单体系统微服务拆分方法,并在此基础上实现了原型工具 MSDecomposer.拆分方法的输入为一组用户场景级别的测试用例和权重,通过分析系统运行时监控日志、构建数据访问轨迹图和数据表图,再对数据表图进行聚类得到合适的数据拆分方案,最后从数据表出发,自底向上搜索得到代码模块的拆分方案.最终的拆分方案给出了从数据表、SQL 语句、方法到类的完整拆分建议,且中间过程中允许用户进行多个维度的反馈调整.实验验证了方法的可行性与拆分结果的正确性,并且从实验参与者的反馈情况来看,这种半自动化的方法产生的结果的确可以作为微服务拆分的重要参考,尤其是在系统逻辑比较复杂、数据表很多的情况下,将极大地减轻开发人员的决策负担.

由于目前的方法对于数据的拆分粒度限制在表的级别,而原单体系统由于数据库设计的问题可能存在一张表含有几十甚至几百个字段,在进行微服务拆分时就可能涉及到拆表的操作,所以我们未来的一个工作方向就是将数据拆分粒度细化到字段级别,从而得到更加准确和耦合度更低的微服务拆分方案.

References:

[1] Lewis J, Fowler M. Microservices: A definition of this new architectural term. 2014. https://martinfowler.com/articles/microservices.html

[2] Ford N. The state of microservices maturity. 2018. https://www.oreilly.com/programming/free/the-state-of-microservices-maturity.csp

[3] Meshenberg R. Microservices at netflix scale: First principles, tradeoffs, lessons learned. 2016. https://gotocon.com/dl/goto-amsterdam-2016/slides/RuslanMeshenberg_MicroservicesAtNetflixScaleFirstPrinciplesTradeoffsLessonsLearned.pdf

[4] Zhou X, Peng X, Xie T, *et al*. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. on Software Engineering (Early Access), 2018. [doi: 10.1109/TSE.2018.2887384]

[5] Zhou X, Peng X, Xie T, *et al*. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: Proc. of the ESEC/SIGSOFT FSE. 2019. 683–694.

[6] Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Upper Saddle River: Pearson Education, 2003.

[7] Rademacher F, Sachweh S, Zündorf A. Towards a UML profile for domain-driven design of microservice architectures. In: Proc. of the Int'l Conf. on Software Engineering & Formal Methods. Springer-Verlag, 2017. 230–245.

[8] Kieker. 2019. http://kieker-monitoring.net/

[9] Hoorn AV, Waller J, Hasselbring W. Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proc. of the Int'l Conf. on Performance Engineering (ICPE). 2012. 247–248.

[10] Parnas DL. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 1972,15(12): 1053–1058.

[11] Fowler M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[12] Chatterjee M, Das SK, Turgut D. WCA: A weighted clustering algorithm for mobile ad hoc networks. Cluster Computing, 2002, 5(2):193–204.

[13] Andritsos P, Tzerpos V. Information-theoretic software clustering. IEEE Trans. on Software Engineering, 2005,31(2):150–165.

[14] Lin Y, Peng X, Cai YF, *et al*. Interactive and guided architectural refactoring with search-based recommendation. In: Proc. of the ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. 2016. 535–546.

[15] Jamshidi P, Pahl C, Mendonça NC, *et al*. Microservices: The journey so far and challenges ahead. IEEE Software, 2018,35(3): 24–35.

[16] Pahl C, Jamshidi P. Microservices: A systematic mapping study. In: Proc. of the Int'l Conf. on Cloud Computing and Services Science. 2016. 137–146.

[17] Francesco PD, Lago P, Malavolta I. Migrating towards microservice architectures: An industrial survey. In: Proc. of the IEEE Int'l Conf. on Software Architecture (ICSA). IEEE, 2018. 29–39.

[18] Taibi D, Lenarduzzi V, Pahl C. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing, 2017,4(5):22–32.

[19] Fritzsch J, Bogner J, Zimmermann A, Wagner S. From monolith to microservices: A classification of refactoring approaches. In: Proc. of the Int'l Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. Springer-Verlag, 2018.

[20] Rademacher F, Sorgalla J, Sachweh S. Challenges of domain-driven microservice design: A model-driven perspective. IEEE Software, 2018,35(3):36–43.

[21] AjiL. 2019. https://github.com/SeelabFhdo/AjiL

[22] Levcovitz A, Terra R, Valente MT. Towards a technique for extracting microservices from monolithic enterprise systems. arXiv:1605.03175, 2016.

[23] Chen R, Li SS, Li Z. From monolith to microservices: A dataflow-driven approach. In: Proc. of the IEEE Asia-Pacific Software Engineering Conf. (APSEC). 2017. 466–475.

[24] Gysel M, Kölbener L, Giersche W, *et al*. Service cutter: A systematic approach to service decomposition. In: Proc. of the European Conf. on Service-Oriented and Cloud Computing (ESOCC). 2016. 185–200.

[25] Abdullah M, Iqbal W, Erradi A. Unsupervised learning approach for Web application auto-decomposition into microservices. Journal of Systems and Software, 2019,151:243–257.

[26] Mazlami G, Cito J, Leitner P. Extraction of microservices from monolithic software architectures. In: Proc. of the IEEE Int'l Conf. on Web Services (ICWS). 2017. 524–531.

[27] Jin WX, Liu T, Zheng QH, *et al*. Functionality-oriented microservice extraction based on execution trace clustering. In: Proc. of the IEEE Int'l Conf. on Web Services (ICWS). 2018. 211–218.

[28] Taibi D, Lenarduzzi V. On the definition of microservice bad smells. IEEE Software, 2018,35(3):56–62.

[29] Web service. 2019. https://www.w3.org/TR/ws-arch/

[30] JeeSite. 2019. https://github.com/thinkgem/jeesite/

[31] Luxburg U. A tutorial on spectral clustering. Statistics & Computing, 2007,17(4):395–416.

[32] Pons P, Latapy M. Computing communities in large networks using random walks. Journal of Graph Algorithms and Applications, 2006,10(2):191–218.

[33] Traag VA. Faster unfolding of communities: Speeding up the Louvain algorithm. CoRR abs/1503.01322, 2015.

[34] Girvan M, Newman MEJ. Community structure in social and biological networks. National Academy of Sciences of the USA (PNAS), 2001,99(12):7821–7826.

[35] Newman MEJ. Fast algorithm for detecting community structure in networks. arXiv:cond-mat/0309508, 2003.

[36] Newman MEJ, Girvan M. Finding and evaluating community structure in networks. arXiv:cond-mat/0308217, 2004.

[37] Spring boot. 2019. https://spring.io/projects/spring-boot/

[38] JSqlParser. 2019. http://jsqlparser.sourceforge.net/

[39] Neo4j. 2019. https://neo4j.com/

[40] Spring-Jpetstore. 2019. https://github.com/making/spring-jpetstore

[41] Spring. 2019. https://spring.io/

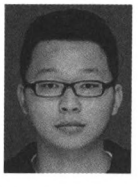
[42] Mybatis. 2019. https://blog.mybatis.org/

[43] zb-Blog. 2019. https://gitee.com/skyblue7080/zb-blog

[44] Exam++. 2019. https://gitee.com/ocelot/examxx/

丁丹(1994－),女,硕士,主要研究领域为微服务拆分.

彭鑫(1979－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为代码大数据,智能化软件开发,软件维护与演化,移动计算与云计算.



郭晓峰(1996－),男,硕士生,主要研究领域为微服务拆分,分布式链路数据质量问题发现与修复.

张健(1984－),男,硕士生,主要研究领域为微服务,自然语言处理.

吴毅坚(1979－),男,博士,副教授,CCF 专业会员,主要研究领域为软件工程,软件维护与演化.