软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(9):2729−2743 [doi: 10.13328/j.cnki.jos.006075]

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

基于日志挖掘的微服务测试集缩减技术*

陈立哲, 吴 际, 杨海燕, 张 奎

(北京航空航天大学 计算机学院,北京 100191)

通讯作者: 吴际, E-mail: wuji@buaa.edu.cn

摘 要: 微服务系统每轮迭代过程中都需要进行回归测试,大量重复测试会造成资源浪费,可通过减少测试用例集的规模来降低成本,以提高测试效率.现有测试用例集缩减技术主要依赖系统规约和架构描述作为输入,对于具有服务自治、调用关系不确定等特点的微服务系统实用性受限.并且,现有测试用例集缩减技术很少考虑使用场景,测试用例集很难反映用户关切.提出了一种基于 API 网关层日志挖掘的测试用例集缩减技术,从 API 网关层日志中挖掘出能够反映服务使用场景的频繁调用路径,建立频繁路径与测试用例的关联关系,进而构建搜索图,并基于启发式搜索生成测试用例缩减集.描述了该技术的完整过程,并基于一个集成办公微服务系统进行了实验.实验结果表明:该技术能够缩减测试用例规模 40%以上,且缺陷发现能力降幅不超过 10%.

关键词: 微服务;软件测试;测试用例集缩减;日志挖掘;API 网关层

中图法分类号: TP311

中文引用格式: 陈立哲,吴际,杨海燕,张奎.基于日志挖掘的微服务测试集缩减技术.软件学报,2021,32(9):2729−2743. http://www.jos.org.cn/1000-9825/6075.htm

英文引用格式: Chen LZ, Wu J, Yang HY, Zhang K. Microservice test suite minimization technology based on logs mining. Ruan Jian Xue Bao/Journal of Software, 2021,32(9):2729−2743 (in Chinese). http://www.jos.org.cn/1000-9825/6075.htm

Microservice Test Suite Minimization Technology Based on Logs Mining

CHEN Li-Zhe, WU Ji, YANG Hai-Yan, ZHANG Kui

(School of Computer Science and Engineering, Beihang University, Beijing 100191, China)

Abstract: In each iteration of microservice system, regression testing should be executed. A large number of repeat testing will cause waste of resources. Therefore, it is necessary to minimize the test suite to reduce costs and to improve testing efficiency. Current test suite minimization technologies mainly rely on system specification and architecture description as input, which is limited to the practicability of microservice system with the characteristics of service autonomy and uncertain call relationship. Moreover, current test suite minimization technologies rarely take the usage scenarios into consideration, and the test suite is difficult to reflect user's concerns. This study proposes a test suite minimization technology based on API gateway access logs mining. This technology mines frequent paths from API gateway access logs which reflects the dynamic operation of microservice system. The relationship between frequent paths and test cases is established to construct search graph. Then, origin test suite is minimized with heuristic search of the graph. This paper explains the whole process of the technology. The experiments based on an integrated OA microservice system show that the scale of the test suite is reduced by more than 40%, and its defect detection ability is reduced by no more than 10%.

Key words: microservice; software testing; test suite minimization; logs mining; API gateway layer

单块架构(monolithic)由于模块化问题而难以管理、扩展和维护[1],很难有效应对 Web 应用系统复杂化问题.很多大型互联网企业像 Netflix,Amazon 和 Spotify 等,采用了微服务架构来重构它们的业务系统.

* 基金项目: 国防科技创新特权
Foundation item: Privilege Projects of National Defense Science and Technology Innovation
收稿时间: 2020-01-10; 修改时间: 2020-02-18, 2020-03-28, 2020-04-26; 采用时间: 2020-05-07

微服务架构(microservices)[2−5]是一种高度模块化的架构模式,它提倡将应用系统根据业务种类和关系划分成一组服务,每个服务能够被独立部署到生产环境、运行在独立的进程中,服务间采用轻量级通信机制交换信息.与单块架构相比,微服务架构具有业务内聚、易扩展、高容错、平滑升级等特点,更能适应 Web 应用系统的业务快速扩展和迭代的现实场景.

微服务系统通过不断迭代来扩展功能.每轮迭代都会改变服务本身及其依赖关系,可能引入新的故障,需要通过回归测试来保证系统质量[6].回归测试最直接的策略就是重复执行原有全部测试用例,但这种策略会造成不必要的资源浪费.为此,很多关于回归测试技术的研究提出:可以针对特定目标(例如保持覆盖性、降低成本、关注变更等)进行有选择性的测试,提高有限资源的使用效率[7].其中,测试用例集选择、排序和缩减是 3 种主要的回归测试技术[6].

测试用例集缩减是一个 NP 完全问题,旨在获得原测试用例集的一个子集,该子集能够保持原测试用例集的覆盖能力[6].其典型技术有启发式方法、基于模型的方法和测试数据聚类的方法等,这些技术大都需要完整一致的系统规约和架构描述作为输入才能实施.而在工程实践中,测试人员很难获得微服务系统的规约与架构信息[8],这涉及管理和技术两个层面的因素:在管理层面,各个服务独立开发运维、高度自治,难以统一维护所有服务及其版本演化过程的文档;在技术层面,受延迟绑定、请求转发等机制的影响,特别是智能化网关的引入,使得服务之间调用关系变得不可预测,难以获得确定的服务调用关系描述.这导致在微服务系统的回归测试中难以应用现有技术.另一方面,现有测试用例集缩减技术大都关注软件自身实现的正确性,很少考虑用户使用场景,这可能造成测试场景和使用场景发生偏离,测试用例集不能聚焦用户关注的服务,导致测试效率降低.例如:为特定用户群体定制的迭代版本只依赖部分服务,强调对所有服务的覆盖可能会造成不必要的测试.

值得注意的是:在微服务系统持续交付过程中,可以采集到大量的 API 网关层日志.API 网关层是用于集中分发服务请求的关键组件,其日志记录了微服务系统运行过程中的每一次 API 请求,内容包括调用者 IP、被请求的 URL 和状态码等信息,主要用于监控服务运行状况.借鉴 Web 日志挖掘技术[9],能够从 API 网关层日志中生成反映用户使用场景的服务调用路径信息,进而关联到测试用例中,作为缩减测试用例集的依据.

本文提出了一种基于 API 网关层日志挖掘的微服务回归测试用例集缩减技术,主要包括日志挖掘、测试用例关联和缩减这 3 个环节.首先,在日志挖掘环节,从 API 网关层日志中提取数据集,进而设计频繁模式挖掘算法,从数据集中挖掘用户使用频繁的服务及其调用关系,即服务调用频繁路径;其次,在测试用例关联环节,将服务调用频繁路径匹配到对应测试用例上,以赋予测试用例关于用户使用场景的信息——支持度;最后,在缩减环节,基于支持度对测试用例进行启发式搜索,找到符合覆盖准则的测试用例缩减集.为验证方法的有效性,基于实际项目数据对该方法进行了实验和分析,结果表明:该方法既易于实现,也能提高测试效率.

本文第 1 节概要介绍微服务回归测试和 Web 日志挖掘的相关研究.第 2 节详细阐述本文所提测试用例集缩减技术的各个环节.第 3 节简要叙述实验验证及分析.第 4 节进行总结.

1 相关研究

1.1 微服务回归测试

对于微服务系统,回归测试在其每一轮迭代都有发生,单元测试、服务测试和端到端测试都需要进行回归以保证系统质量.微服务单元测试主要用于发现服务内部代码实现存在的缺陷,可以通过单元测试工具完成,例如 xUnit[10]、Mockito[11]等,其回归测试技术与单块架构软件基本一致.服务测试和端到端测试关注服务之间的连通性,涉及多个服务及其之间的调用关系.由于服务自治、动态绑定、访问限制带来的挑战[12],服务测试与端到端测试的回归测试技术与单块架构软件差别较大,是研究和实践关注的重点[13].

在测试用例选择方面,主要是基于系统迭代时发生的变化来筛选相关联的测试用例[6].由于传统黑盒测试技术也适用于单个服务回归测试,大部分研究工作都关注服务之间组合关系的变化,包括接口变化[14,15]、业务流程变化[14−17]和动态绑定关系变化[15,16]等方面.其中,文献[14,16]采用路径分析方法,该方法比较迭代前后服务之间的调用路径变化,进而选择包含这些路径变化的测试用例;文献[17]提出一种图遍历技术,该方法与路径分

析方法类似,但更贴近代码层面,需要代码生成的控制流图作为输入;文献[15]分析服务接口契约的依赖和冲突关系变化来筛选测试用例.值得注意的是:很多研究都将服务之间的调用关系抽象为有向图,并将测试用例抽象为图上的路径.

在测试用例排序方面,基于覆盖性的排序研究占多数.文献[18]提出了一种基于服务贡献度的排序技术,该技术将服务及其调用关系类比程序控制结构并为其分配贡献度,进而计算测试用例的整体贡献度,贡献度高的测试用例排序靠前.文献[19]提出了一种基于多因素度量的测试用例排序技术,通过计算工作流分支(WBs)、XPath 重写图分支(XRGBs)和 WSDL 元素(WEs)等因素的综合度量,结合所需覆盖准则对测试用例排序.在基于多因素排序的技术中,贪心算法被较多采用[20],爬山算法和遗传算法也偶有采用[21].考虑到微服务架构与关系图的相似性,文献[22]提出了一种基于图的微服务分析和测试框架,该框架通过微服务系统的需求规约(BDD)生成关系图,遍历关系图得到所有服务调用路径集合,进而从目标服务是否迭代、调用关系优先级和追溯的规约层级等方面来对调用路径对应的测试用例排序.

在测试用例缩减方面,除了直接应用单块架构软件的启发式方法、基于模型的方法和测试数据聚类等方法外,文献[23]提出了一种基于多目标优化的方法,使得测试用例集的覆盖性、可靠性和执行开销等目标综合最优.文献[24]提出了一种基于 Petri 网的方法,该方法从微服务系统业务规则的 Petri 网模型制定缩减规则,进而结合输入参数组合方式缩减测试用例集合.从近几年检索到的论文数量上看,测试用例集缩减方面的研究明显少于测试用例选择和排序.

1.2 Web日志挖掘

Web 日志记录了用户访问和资源使用信息,是 Web 系统知识挖掘的一种主要数据来源[9].Web 日志蕴含了用户访问和系统运行的知识,常被用于性能分析、流量分析、服务改进、用户行为建模和商业智能化等目的.Web 日志挖掘过程通常包括数据预处理、模式发现和模式应用这 3 个步骤.其中:数据预处理阶段对原始日志数据进行清理整合,转换为格式化数据集;模式发现阶段则根据不同目标,采用相应的数据挖掘技术,从格式化数据集中生成感兴趣的模式集合,例如统计分析、路径分析、关联规则挖掘、序列模式挖掘和聚类等等;模式应用阶段则是对挖掘到的模式集合进行分析和应用,例如基于关联模式向用户推荐其感兴趣的商品[25]等等.

在针对 Web 日志的各种数据挖掘技术中,频繁模式挖掘技术是最早被研究、也是应用最为广泛的技术之一[26],它能从 Web 日志中挖掘出用户频繁访问模式和系统资源之间的频繁关联模式,通常被作为发现频繁路径、关联规则和序列模式等目标的支撑技术.频繁模式挖掘问题可描述如下:给定集合 $D=\{T_1,T_2,T_3,\ldots,T_n\}$,每个事务 T_i 都是项集合的子集,当确定一个频繁阈值 $s(0<s<n)$时,找出所有项的组合模式 p,使得 p 在 D 中的出现频次超过 s.其中,p 的出现频次也称为支持度 c,频繁阈值 s 也称为最小支持度.文献[26]综述了频繁模式挖掘的各类算法及常用的应用场景,常见的算法有 Aprori,FP-growth 等等.在软件测试方面的应用侧重于代码缺陷检测,但尚未发现有研究将其应用于微服务系统的回归测试.

微服务系统的 API 网关层日志记录了其运行过程中的每一次 API 请求.在工程实践中,开发运维人员主要关注故障记录和调试信息,而占据日志数量绝大部分的请求成功记录通常被忽略.事实上,这些正常数据蕴含了一段时间内用户对不同业务场景的关注程度和服务之间的动态关联等知识,同样具有很高的价值.因此,本文基于频繁模式挖掘技术,对微服务系统 API 网关层日志进行挖掘,从而获得用户频繁关注的服务调用路径,用于测试用例集缩减.

2 方法描述

本文提出的技术主要解决在微服务系统工程实践中难以获得系统规约和架构描述时如何缩减测试用例的问题,其输入主要有 API 网关层日志和原测试用例集,输出为测试用例缩减集,过程主要包括 API 网关层日志挖掘、测试用例匹配和测试用例缩减这 3 个环节,如图 1 所示.下面详细描述各个环节.

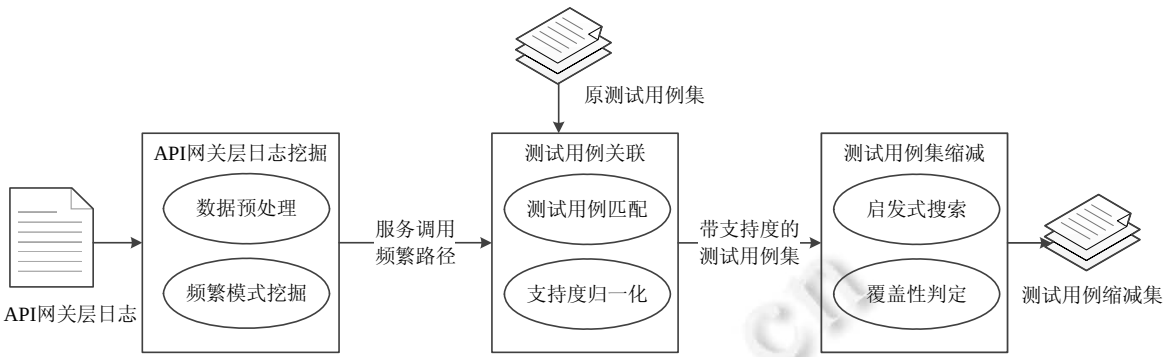


Fig.1 Sketch map of the method process

图 1 方法过程示意图

2.1 API网关层日志挖掘

该环节从 API 网关层日志中挖掘出服务调用频繁路径,主要包括数据预处理和频繁模式挖掘两项活动.其中:数据预处理从日志数据中构造用户请求链并生成待挖掘的事务集,频繁模式挖掘则基于挖掘算法从事务集中抽取服务调用频繁路径.

2.1.1 数据预处理

数据预处理是从原始数据中获得挖掘所需格式化数据的重要过程,一般占整个数据挖掘工作量的 60%[9].微服务系统中的 API 网关日志,其形式可能随系统运维要求、记录机制等各有不同,但一般都包含了请求时间、请求者地址、服务名称、负载均衡器分配的服务实例地址、状态码、响应时间、http 代理等信息.本方法即针对包含上述信息的 API 网关日志进行预处理,包括数据清洗、用户请求识别、用户请求链生成和事务集生成等 4 个步骤.

(1) 数据清洗

数据清洗即是从 API 网关日志中剔除故障请求记录和系统信息(例如自检),仅保留请求成功记录,即响应状态码为"2*xx*"的请求记录,并从中筛选出上述所需字段信息,形成结构化数据集.

(2) 用户请求识别

API 网关日志中的请求包括来自系统外部的用户请求以及服务之间的调用请求,本步骤即是将用户请求记录识别出来,作为构造用户请求调用链的起点.可以采用以下规则识别出用户请求记录:

规则 1. 若某请求记录的请求者地址不在微服务系统已注册服务列表中,则该请求是用户请求.

微服务系统中的服务通常需要在服务注册中心进行注册和发布,例如 zookeeper 或者 eureka,如果能从中获得已注册微服务列表,就可以准确判定哪些请求是用户请求.该规则属于直接判定规则.该规则需要服务注册中心可用且已注册服务列表实时更新,否则判定会失效.

规则 2. 若某请求记录的请求方代理为浏览器程序,则该请求是用户请求.

用户发起的请求通常需要通过各种浏览器程序,例如 Chrome,Firefox 等等,如果 http 代理信息中包含这些浏览器程序信息,则该记录往往是用户请求记录.该规则属于间接判定规则,对于非人工的外部请求无法识别,同时还要维护用户使用的浏览器程序列表,以增加判定的准确性.

规则 3. 解析出所有服务名称及其对应实例地址的列表,若某请求记录的请求者地址不在列表中,则该请求是用户请求.

对于给定的 API 网关日志,需要关注的是被请求服务的情况,没有被请求的服务,可以认为不属于信息挖掘对象.该规则也属于间接判定规则,无需获取或维护额外信息,仅靠 API 网关日志信息即可进行判定;但对于日志增量的场景,则每次都需要重新扫描全部日志.

应当根据实际应用来采用合适的判定规则.当然,还可以根据额外记录的日志信息来增加判定规则,例如增

加请求触发页面信息,如果该信息不为空,说明该请求是通过前端页面触发,可判定该请求为用户请求.

(3) 用户请求链生成

用户请求链生成以用户请求为起点,构建与之关联的服务调用关系,包括聚集与用户请求相关联的服务间请求列表和从该关联请求列表生成调用链等活动,流程图如图 2 所示.

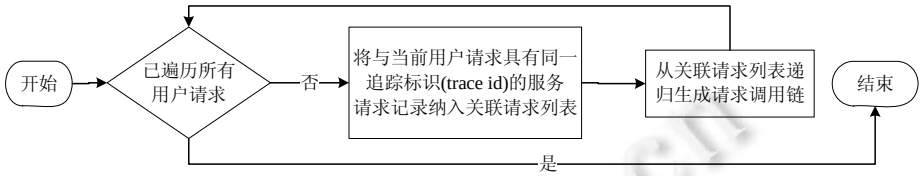


Fig.2 Flow chart of user request chain generation

图 2 用户请求链生成流程图

其中,关联请求列表的形成基于微服务架构的另一重要组件“服务调用链监控”来实现.服务调用链监控主要通过“埋点”技术,在微服务系统运行过程中采集、分析和展示服务调用情况,为故障诊断、性能优化等提供支撑,例如,大众点评的 CAT,Twitter 的 Open Zipkin 以及 Naver Pinpoint 等.关联同一用户请求的服务请求共享同一个追踪标识(trace id),故通过判定追踪标识的一致性即可形成用户请求对应的关联请求列表.

请求调用链的树形结构特征则使得其生成需要采用递归算法来实现,表 1 展示了一种基于深度优先的递归算法.该算法遍历关联请求列表中的每条日志(第 1 行),如果日志数据不在当前请求链中,则基于该日志构造请求链上的新节点(第 2 行~第 4 行),并以该节点作为根节点、以剩余日志数组递归执行算法(第 5 行),直至关联请求列表遍历完成退出递归.

Table 1 Request chains generation algorithm

表 1 基于深度优先的调用链生成算法

算法. *generateRequestChain*(*logs*,*chain*).

参数:*logs*,关联请求列表(输入);*chain*,请求链(输出).

```
1  for each (log,index)∈logs do
2    if not chain.descendants_contain(log) then
3      if chain.log.service=log.service then
4        node←new ChainNode(log)
5        generateRequestChain(log.slice_to_end(index+1),node)
6        chain.append_child(node)
7      end if
8    end if
9  end for
```

(4) 事务集生成

由于用户请求链的数据结构为树形结构,生成待挖掘事务集的过程就是生成从根节点到所有叶子节点路径序列的过程.算法也是基于深度优先搜索树结构,伪代码见表 2.

Table 2 Transaction set generation algorithm

表 2 基于深度优先的事务集生成算法

算法. *generateTransactions*(*node*,*stack*,*D*).

参数:*node*,请求链节点(输入);*stack*,辅助栈(输入);*D*,事务集(输出).

```
1  stack.append(node.id)
2  if node.children=∅ then
3    D.append(stack.reverse(·))
4  else then
5    for each n∈node.children do
6      generateTransactions(n,stack,D)
7    end for
8  end if
9  stack.pop(·)
```

该算法在搜索过程中通过一个栈结构来生成从根节点到叶子节点的路径(第 1 行),若栈顶节点没有后继节点,则倒排栈元素构成一条事务(第 2 行、第 3 行);若有后继节点,则递归处理(第 4 行~第 6 行).当前节点处理完后弹栈(第 9 行).

2.1.2 频繁模式挖掘

频繁模式挖掘算法的类型多种多样,例如基于候选集拼接的算法、基于树的算法和基于递归后缀的算法,分别适用于不同的挖掘场景[26],应用时可能需要根据实际情况进行调整和重构.对于微服务系统的频繁调用路径挖掘场景,挖掘算法需要考虑以下两个特点.

(1) 从数据集来看,每个服务在调用路径上的位置不能交换,即项集是有序的,属于序列模式挖掘[26].同时,由于调用路径上的相邻服务之间必须存在调用关系,则一个调用路径的子路径必须是连通的,例如“$a\to b\to c$”的子路径可以是“$a\to b$”或者“$b\to c$”,但不能是“$a\to c$”.可以计算出,长度为 n 的调用路径存在 $n(n+1)/2$ 个子路径;

(2) 从挖掘目标来看,最终输出的频繁路径应当包含尽可能多的服务.例如,“$a\to b$”和“$a\to b\to c$”如果都是频繁的,那么只需要输出“$a\to b\to c$”作为频繁路径.即:对于输出的频繁路径,不存在任何一条包含它路径也是频繁的,属于最大频繁模式挖掘[24].

基于上述特点,提出如下定义:$S=\{S_1,S_2,S_3,\ldots,S_m\}$为服务集合,序列 $T_i=\langle S_1,S_2,S_3,\ldots,S_k\rangle(0<k\leqslant m)$为一条服务调用路径,$k$ 为调用路径的长度,$D=\{T_1,T_2,T_3,\ldots,T_n\}$为待挖掘服务调用路径集合,$T$ 为所有 T_i 的子路径组成的集合,对于 $t\in T$,$c(t)$表示 t 在 D 中的出现频次,即 t 的支持度,m 为频繁阈值(最小支持度),则:

定义 1. 频繁路径集 $F=\{t|c(t)>m,t\in D\}$,F_k 是长度为 k 频繁路径构成的集合,即 $F_k=\{t|len(t)=k,t\in F\}$.

推论 1. 如果一条路径 t 不是频繁的,则包含 t 的路径也不是频繁的.

定义 2. 最大频繁路径集 $P=\{p|\nexists q\in P\wedge q\supset p,p\in F\}$.

推论 2. 如果一条路径 t 是最大频繁路径,则 t 的所有子路径都不是最大频繁路径.

本文所描述的频繁路径挖掘算法是从 D 挖掘最大频繁路径集 P 的算法,参照 Apriori 算法[26]框架设计,伪代码见表 3.

Table 3 Frequent calling paths mining algorithm

表 3 频繁路径挖掘算法

算法. *mineFrequentPaths*(*D*,*m*,*P*).

参数:*D*,事务集(输入);*m*,最小支持度(输入);*P*,最大频繁路径集(输出).

```
1  F←new Frequent(m)
2  F1←F.construct_one_frequents(D)
3  k←1
4  while Fk≠∅ do
5     Fk+1←F.construct_candidate_frequents(D,Fk,F1)
6     for each p∈Fk+1 do
7        c←D.count(p)
8        if c<m then
9           Fk+1.remove(p)
10       end if
11    end for
12    k++
13 end while
14 k−−
15 while k>0 do
16    for each p∈Fk do
17       if not P.subpath(p) then
18          P.add(p)
19       end if
20    end for
21    k−−
22 end while
```

首先,可以根据定义 1 直接计算 F_1(第 2 行);其次,在推理 1 的基础上,通过得到 F_k 和 F_1 的笛卡尔积,生成长

度为 k+1 路径的候选集(第 5 行),进而根据定义 1,从候选集中筛选出 F_{k+1}(第 6 行~第 9 行);最后,根据定义 2 和推理 2,从长度最长的频繁路径集到 F_1 依次确定最大频繁路径集(第 14 行~第 22 行).

2.2 测试用例关联

该环节主要建立服务调用频繁路径与测试用例的对应关系,并将支持度信息赋值给相应测试用例,主要包括测试用例匹配和支持度归一化这两项活动.其中:测试用例匹配从频繁路径集中搜索与测试用例关联的服务调用路径;支持度归一化将支持度转化为 0 到 1 区间内的数值,为后续启发式搜索提供输入.

2.2.1 测试用例匹配

测试用例匹配主要是建立频繁路径与测试用例之间的对应关系,由于单元测试不涉及服务调用关系,不在本文讨论范围,本文主要关注服务测试和端到端测试.

(1) 服务测试匹配

服务测试关注服务之间的调用关系,其测试用例可以直接转化为一条服务调用路径(或称为测试路径).匹配过程中,从最长的频繁路径开始搜索,对每条频繁路径与测试路径进行匹配,可能出现以下 4 种结果.

① 精确匹配到一条频繁路径;

② 测试路径是一条或多条频繁路径的子路径;

③ 有一条或多条频繁路径是该测试路径的子路径;

④ 其他情况.

对于匹配结果①,直接将精确匹配的频繁路径关联到测试用例上;对于匹配结果②,说明测试路径是频繁的,但不是最大频繁的,故将匹配到的所有频繁路径中支持度最大的频繁路径关联到测试用例上;对于匹配结果③,虽然该测试路径本身不是频繁的,但其包含最大频繁路径,故将匹配到所有频繁路径中长度最长的频繁路径关联到测试用例上;对于匹配结果④,直接判定为不频繁,不关联任何频繁路径.

(2) 端到端测试匹配

端到端测试的测试用例可能对应一组服务调用路径(或称为测试路径集合),需要对测试路径集合中的每一条测试路径进行匹配.匹配结果的处理方式与服务测试匹配类似,但将频繁路径对应到整个测试用例上时,需选择长度最长的频繁路径进行关联.

2.2.2 支持度归一化

考虑到启发式搜索需要关注测试用例之间的差异性,需统一支持度和差异值的取值范围,故频繁路径的支持度不能直接赋予与之关联的测试用例,需要进行归一化处理.同时,长度更长的频繁路径对系统的覆盖程度更高,应优先于长度短的频繁路径,为此,本文将[0,1)区间划分为 $k_{\max}$ 段($k_{\max}$ 为 k 的最大值),将 F_k 中频繁路径 p 的支持度换算到半开区间$[(k-1)/k_{\max},k/k_{\max})$上,换算公式为

$$c_u=\frac{k-1+c/n}{k_{\max}} \tag{1}$$

其中,c 为 p 的频次,n 为事务集 D 的规模,c_u 为归一化后的支持度.

归一化之后,即可将支持度赋予关联的测试用例.对于未匹配到频繁路径的测试用例,则支持度赋值为 0.

2.3 测试用例集缩减

该环节基于支持度对原测试用例集搜索其子集,进而输出符合覆盖准则要求的测试用例缩减集,主要包括启发式搜索和覆盖性判定这两项活动.其中:启发式搜索方法基于图搜索方法提出,覆盖性判定则确认搜索到的测试用例子集满足覆盖要求.

2.3.1 启发式搜索

启发式搜索的目的在于找到一组支持度较大且相互间差异也较大的测试用例集合:支持度较大说明测试用例集符合用户关切,差异较大是为了使测试用例集尽快收敛于覆盖准则要求.注意到支持度属于测试用例自身属性,差异性属于测试用例两两关系的属性,故可以构建一个图来建模这两项参数,图的节点表示测试用例,

图的边表示测试用例之间的差异性.则测试用例搜索可以转化为对图的遍历问题.

定义 3. 测试用例搜索图 $G=(N,E)$,其中:N 为节点集合,表征每个测试用例,每个节点包含 2 个属性,即测试用例的标识和支持度;E 为边的集合,表征测试用例之间的关系,每条边有 1 个属性,即边两端所表征测试用例的差异性量化值.

本文使用杰卡德距离[25]量化测试用例的差异性,具体做法为:获得测试用例 a 和 b 覆盖的服务,分别形成集合 S_a 和 S_b,计算两个集合的杰卡德距离 d_j,即:

$$d_j(S_a,S_b)=\frac{|S_a\cup S_b|-|S_a\cap S_b|}{|S_a\cup S_b|} \tag{2}$$

基于定义 3 即可设计从测试用例集生成搜索图的算法,伪代码见表 4.该算法遍历所有测试用例,为测试用例创建一个新节点添加到图中(第 1 行~第 3 行),支持度属性为公式(1)中的 c_u,并构造该节点与已有节点的边也添加到图中(第 4 行~第 11 行),边的属性通过公式(2)计算获得.

Table 4 Searching graph generation algorithm

表 4 搜索图生成算法

算法. *generateGraph*(*T*,*G*).

参数:*T*,附带支持度信息的测试用例集(输入);*G*,搜索图(输出).

```
1  for each t∈T do
2    nn←Graph.create_node(t)
3    G.nodes.append(nn)
4    for each n∈G.nodes do
5      if n≠nn then
6        t_n←n.get_testcase(·)
7        d←d_j(t_n,t)
8        e←Graph.create_edge(n,m,d)
9        G.edges.append(e)
10     end if
11   end for
12 end for
```

由于搜索测试用例的目标是使得测试用例的支持度较大且相互间差异也较大,故本文采用一种启发式算法——贪心算法,来遍历搜索图,算法伪代码见表 5.

Table 5 Searching graph traversal algorithm

表 5 搜索图遍历算法

算法. *searchGraph*(*G*,*f*,T_r).

参数:*G*,搜索图(输入);*f*,测试准则判定函数(输入);T_r,测试用例缩减集(输出).

```
1  n_s←G.find_max_c_u(·)
2  while not G=∅ and not f(T_r) do
3    v←0, n_e←null
4    for each n∈G.nodes do
5      if n≠n_s then
6        e←G.get_edge(n_s,n)
7        vt←n.c_u+e.d_j
8        if v<vt then
9          v←vt, n_e←n
10       end if
11     end if
12   end for
13   T_r.append(n_s)
14   G.remove_node_with_edges(n_s)
15   n_s←n_e
16 end while
```

该算法从图中支持度最大的节点开始遍历(第 1 行、第 2 行),计算当前节点到每一个相连节点的转移值,该值为相连节点的支持度与边上属性的和,进而查找转移值最大的第 1 个相连节点(第 4 行~第 12 行),然后从图

中移除当前节点及其边,并将新找到的节点作为当前节点继续遍历过程(第 13 行~第 15 行).该算法的终止条件主要有两个:一个是图为空,另一个则是已经搜索到的测试用例集满足覆盖性判定要求(第 4 行).

2.3.2 覆盖性判定

为了确保启发式搜索获得的测试用例集是“覆盖安全”[12,28]的,需要判定其是否满足特定的覆盖准则,例如服务覆盖、调用关系覆盖、特定调用路径覆盖和功能覆盖等等.实现过程中,需将选定的覆盖准则定义为函数,并将该函数的指针传入搜索图生成算法中作为搜索过程终止的条件.本文在实验中使用调用关系覆盖进行覆盖性判定,即,搜索获得的测试用例集所包含服务调用关系必须覆盖原测试用例集所包含的服务调用关系.设 R_{T_r} 表示测试用例缩减集中的服务调用关系集合,R_T 表示原测试用例集中的服务调用关系集合,则该覆盖准则可定义为 $f(T_r)=R_T\subseteq R_{T_r}$.

3 实验验证

为验证技术的有效性,使用某集成办公系统开发过程的数据进行了实验.

3.1 系统介绍

该集成办公系统基于微服务架构设计开发,为多部门、多密级和跨地域的机构人员提供服务.系统提供的功能包括综合信息展示、公文流转、流程审批、计划管理、机构人员管理、合同管理、经费管理和物资管理等.由于业务类型和数据安全等问题,该系统由多个团队联合开发,开发周期经历了 7 个轮次的迭代,根据测试过程数据统计了每个迭代轮次的服务数量(个)、API 网关日志数量(条)以及服务测试和端到端测试的测试用例数量(个)、发现的缺陷数量(个)和执行时间(时)情况(测试执行方式为手工测试与自动化测试工具相结合的方式),见表 6.

Table 6 Statistical data of an integrated business system

表 6 某综合业务系统开发过程统计数据

迭代轮次	服务数量	API 网关日志记录数量	服务测试			端到端测试		
			测试用例数	缺陷数	执行时间(时)	测试用例数	缺陷数	执行时间(时)
1	34	46 394	437	225	40.41	143	65	16.52
2	53	91 219	609	214	50.12	215	98	40.12
3	62	148 851	652	92	56.39	228	52	42.20
4	69	234 317	683	63	57.72	232	23	42.85
5	72	419 351	711	31	58.81	232	11	42.73
6	101	912 653	1 132	115	88.69	271	27	48.25
7	105	1 035 519	1 165	47	93.21	275	13	48.91

从表 6 可以看出:在早期的 3 次迭代中,服务和测试用例的数量增长较快,属于系统构建阶段;在后两次迭代中,服务和测试用例的数量增长较慢,属于改进阶段.随着测试用例数量的增加,执行测试所需的人天数也随之增加,发现的缺陷数却在减少,第 4 轮、第 5 轮的缺陷数远低于前 3 轮.一方面,这是系统不断改进的结果;另一方面,系统本身增量不大,但要执行的测试用例比以前的迭代轮要多,这就造成了一定程度的浪费.到第 6 轮时,由于增加了多密级、跨地域等特性的支撑性服务,测试用例数量又进一步增加,第 7 轮服务基本稳定,但测试执行过程依然花费了比之前所有轮次更多的资源,类似于前 5 轮的变化.与此同时,API 网关层日志的数量迅速增加,记录了系统在各种业务场景下的大量用户操作信息,具备实施本文提出方法的输入条件.

3.2 实验设计

为了验证本文提出的技术是否能够有效缩减测试用例集,并且分析 API 网关层日志规模对缩减效果的影响,基于项目数据采用后验方式进行实验验证,即:在已知测试用例执行结果的情况下,统计缩减后测试用例集数量、发现缺陷数量和执行时间,计算评价指标,进而与原始数据进行对比分析.设计如下两个实验.

(1) 实验 1:测试用例缩减效果实验.

依据文献[6],实验采用测试用例集缩减率 P_e(公式(3))和缺陷发现能力影响率 P_i(公式(4))这两个指标分析

基于本文技术缩减后测试用例集的有效性:

$$P_e=\left(1-\frac{\text{缩减后测试用例集数量}}{\text{原测试用例集数量}}\right)\times 100\% \tag{3}$$

$$P_i=\left(1-\frac{\text{缩减后测试用例集发现缺陷数量}}{\text{原测试用例集发现缺陷数量}}\right)\times 100\% \tag{4}$$

从公式(3)可知:P_e越大,测试用例缩减得越多,节省测试资源越多;从公式(4)可知:P_i越小,缩减后测试用例集的缺陷发现能力越接近原测试用例集.

为了验证用户使用信息能否在测试用例集缩减中起到积极作用,实验中将本文所提技术与一种经典的随机搜索[6]进行对比,即随机地从原测试用例集中构造测试用例缩减集,并保证满足测试覆盖准则.伪代码见表 7,其搜索过程结束条件就是测试覆盖准则满足(第 1 行),当不满足覆盖准则时,从原测试用例集中随机选择一个测试用例并添加到测试用例缩减集中(第 2 行、第 3 行).在本实验中,随机搜索采用的覆盖准则与本文所提技术保持一致,即要求缩减后测试用例集覆盖原测试用例集的服务调用关系.

Table 7 Random searching algorithm

表 7 随机搜索算法

算法. *RandomSearch*(T,f,T_r).

参数:T,原测试用例集(输入);f,测试准则判定函数(输入);T_r,测试用例缩减集(输出).

```
1  while not f(T_r) do
2      t←T.random_select(·)
3      T_r.append(t)
4  end while
```

实验步骤如下.

S1. 挖掘第 k($1\leqslant k\leqslant 6$)轮的 API 网关层日志记录,获得频繁路径集,标记为 FP_k;

S2. 将第 k+1 轮的测试用例分为沿用测试用例集和新增测试用例集,即完全沿用自第 k 轮的测试用例,在该系统中与第 k 轮测试用例完全相同,新增测试用例则为第 k+1 轮根据系统增量新生成的测试用例;

S3. 基于 S1 中获得的频繁路径集对 S2 中的沿用测试用例集进行缩减,并将缩减后的测试用例集与 S2 中的新增测试用例集合并,作为第 k+1 轮缩减后的测试用例集 T_1;

S4. 采用随机搜索技术对 S2 中的沿用测试用例集进行缩减,并将缩减后的测试用例集与 S2 中的新增测试用例集合并,作为第 k+1 轮用于对比的测试用例集 T_2;

S5. 对照测试记录统计 T_1 和 T_2 的缺陷发现数量,计算各自的 P_e 和 P_i,然后进行对比分析;

S6. 统计缩减后测试用例执行时间,计算测试用例缩减后节省的时间.同时,统计每轮迭代生成 T_1 过程的时间开销,进而验证测试用例缩减带来的时间收益;

(2) 实验 2:API 网关层日志规模对缩减效果的影响实验.

S7. 挖掘第 7 轮的 API 网关层日志记录,获得频繁路径集,标记为 FP_7;

S8. 基于 FP_k($1\leqslant k\leqslant 7$)对第 7 轮测试用例集进行缩减,分别得到 7 个缩减后的测试用例集;

S9. 计算对比缩减后测试用例集的 P_e 和 P_i,分析 API 网关层日志规模对缩减效果的影响.

3.3 实验结果与分析

(1) 实验 1 结果与分析

根据实验 1 的过程,挖掘出第 1 轮~第 6 轮的 API 网关层日志,并将结果分别用于第 2 轮~第 7 轮测试用例集的缩减,然后统计缩减后的测试用例数、发现的缺陷数以及工作量,并计算 P_e 和 P_i.其中,测试用例集 T_1 相关数据见表 8,测试用例集 T_2 相关数据见表 9.同时,统计每轮迭代生成 T_1 过程的时间开销,计算环境为英特尔酷睿 i7 处理器、32GB 内存和 CentOS7.6 操作系统,相关数据见表 10.

Table 8 Statistical data of T_1 in the experiment 1

表 8 实验 1 中 T_1 相关数据

迭代轮次	服务测试					端到端测试				
	测试用例数	缺陷数	P_e(%)	P_i(%)	执行时间(时)	测试用例数	缺陷数	P_e(%)	P_i(%)	执行时间(时)
2	385	199	36.78	7.00	32.26	115	93	46.51	5.10	21.60
3	373	89	42.79	3.26	32.75	80	49	64.91	5.76	16.23
4	418	63	38.79	0.00	34.09	79	23	65.94	0.00	15.87
5	439	31	38.25	0.00	35.52	78	11	66.37	0.00	15.12
6	726	114	35.86	0.86	56.79	65	27	76.01	0.00	12.75
7	632	47	45.75	0.00	51.03	63	13	77.09	0.00	12.13

Table 9 Statistical data of T_2 in the experiment 1

表 9 实验 1 中 T_2 相关数据

迭代轮次	服务测试					端到端测试				
	测试用例数	缺陷数	P_e(%)	P_i(%)	执行时间(时)	测试用例数	缺陷数	P_e(%)	P_i(%)	执行时间(时)
2	341	151	44.00	29.44	29.31	132	66	38.60	32.65	24.32
3	294	59	54.91	35.87	26.93	94	34	58.77	34.61	19.78
4	336	37	50.80	41.26	28.72	92	11	60.34	52.17	19.13
5	312	14	56.12	54.84	27.97	94	3	59.48	72.72	20.12
6	765	76	32.42	33.91	58.46	103	19	61.99	29.63	22.33
7	683	29	41.37	38.29	53.25	86	6	68.73	53.85	16.43

Table 10 Time expenses in T_1 generation and the time saved using test suite minimization

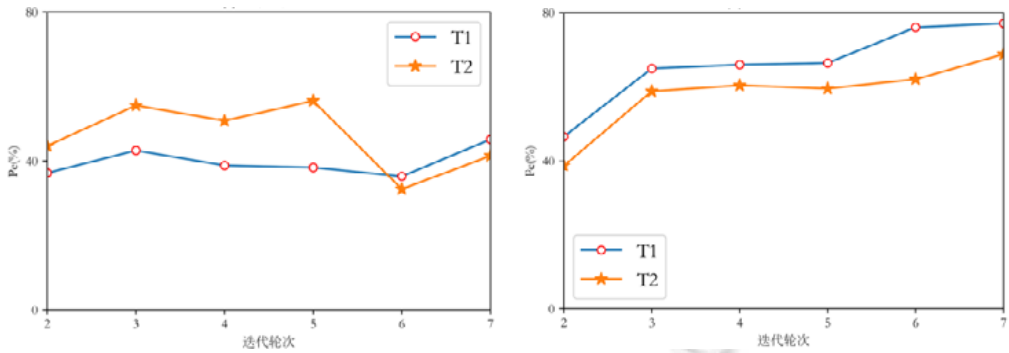
表 10 T_1 生成过程的时间开销和测试用例缩减后节省的时间

迭代轮次	T_1 生成过程的时间开销			测试用例缩减后节省的时间			
	日志挖掘环节(时)	服务测试缩减环节(时)	端到端测试缩减环节(时)	服务测试(时)	百分比(%)	端到端测试(时)	百分比(%)
2	0.011 2	0.001 1	0.000 9	17.86	35.63	15.80	39.38
3	0.033 5	0.001 6	0.001 3	23.64	41.92	22.42	53.12
4	0.092 7	0.001 6	0.001 3	23.63	40.93	23.72	55.35
5	0.176 1	0.001 6	0.001 3	23.29	39.60	22.61	52.91
6	0.385 4	0.002 1	0.001 3	31.90	35.96	25.92	53.72
7	0.919 6	0.002 1	0.001 3	42.18	45.25	32.48	66.40

从表 8 和表 9 中的数据可以看出:两种测试用例缩减技术在满足覆盖准则的条件下,都能缩减测试用例集的规模,对其缺陷发现能力的影响各有差异.为了直观比较两种方法缩减测试用例的效果,分别将 P_e 和 P_i 绘制成折线图,其中,P_e 对比折线图如图 3 所示,P_i 对比折线图如图 4 所示.

从图 3 中可以看出:本文提出的测试用例缩减技术与基于随机搜索的测试用例缩减技术都能大幅减少测试用例的规模,进而减少测试执行所需的工作量,并且缩减能力大致相当,约 40%左右.同时也表明了在微服务系统实际使用过程中,各个服务调用频率确实存在差异.从图 4 可知:基于随机搜索技术缩减的测试用例集合 T_2 的缺陷发现能力同样产生了明显下降,降幅达到至少 30%以上;特别是在新增测试用例较少时,其缺陷发现能力受影响越大.这说明:由于服务之间关联的动态性,微服务架构的服务测试不能简单参照单块架构的测试覆盖准则,否则会造成不少缺陷被遗漏的情况.与之相对的,基于本文提出技术缩减的测试用例集合 T_1 的缺陷发现能力并未受到太大影响,降幅不超过 10%;尤其是在新增测试用例较少时,基本能维持原有测试用例集的缺陷发现能力.这主要是因为 T_1 在满足测试覆盖准则的同时,一并考虑了系统实际运行过程的用户使用信息,从而保留了频次高的服务调用路径对应的测试用例,这些路径类似于高速公路中“枢纽”,请求频次多、用户使用场景覆盖较为全面,相应地,测试用例对整体测试用例集缺陷发现能力的贡献更为突出.

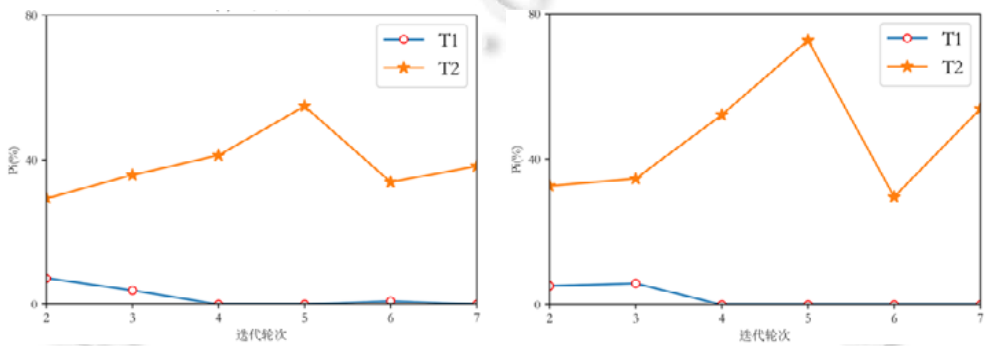
通过表 10 数据可知:本次实验中 T_1 生成过程的时间开销不超过 1 小时,远远小于测试用例缩减后节省的时间,说明本文所提技术在实验中确实起到了节省测试时间的作用.同时,本文所提技术的时间开销主要集中在日志挖掘环节,对于近百万级规模的日志挖掘时间相对较长,这是由于未做增量式挖掘而造成每次迭代都要完整扫描整个日志数据的缘故,是下一步需要改进的地方.



(a) 服务测试 (b) 端到端测试

Fig.3 P_e comparison line chart of two test suites

图 3 两种测试用例集的 P_e 对比折线图



(a) 服务测试 (b) 端到端测试

Fig.4 P_i comparison line chart of two test suites

图 4 两种测试用例集的 P_i 对比折线图

另外,由于第 2 轮、第 3 轮迭代中 T_1 和 T_2 都出现了遗漏的缺陷,为了进一步分析本文所提技术的应用效果,我们对第 2 轮、第 3 轮迭代中 T_1 和 T_2 遗漏的缺陷和已发现的缺陷进行了梳理,根据案例过程文档定义的缺陷等级(致命、严重、一般、建议)和这些缺陷对应测试用例的支持度(见公式(1))平均值进行了统计对比,见表 11 和表 12.

从表 11 和表 12 的缺陷等级分布对比可看出:T_1 发现了全部“严重”等级以上的缺陷,遗漏的缺陷主要是“建议”等级的缺陷;而 T_2 在各个等级的缺陷上均有遗漏.这进一步说明了本文所提技术在保留测试用例发现缺陷的能力上要好于仅考虑测试覆盖准则的随机搜索技术.同时,T_1 遗漏缺陷对应的测试用例支持度平均值为 0,表明这些缺陷分布在日志中未出现或出现频次小于频繁阈值的业务上,具有明显的规律性,可以从已移除的测试用例中增选支持度为 0 的测试用例,对缺陷发现能力的“缺损”进行弥补.

Table 11 Statistical data of defects found and missed in the second and third rounds by T_1

表 11 第 2 轮、第 3 轮 T_1 发现和遗漏的缺陷情况统计

迭代轮次	统计项	服务测试					端到端测试				
		致命	严重	一般	建议	支持度平均值	致命	严重	一般	建议	支持度平均值
2	已发现	2	64	124	9	0.652 1	1	27	41	24	0.645 3
	遗漏	0	0	2	13	0.000 0	0	0	0	5	0.000 0
3	已发现	0	31	46	12	0.632 4	0	18	19	12	0.623 6
	遗漏	0	0	0	3	0.000 0	0	0	0	3	0.000 0

Table 12 Statistical data of defects found and missed in the second and third rounds by T_2

表 12 第 2 轮、第 3 轮 T_2 发现和遗漏的缺陷情况统计

迭代轮次	统计项	服务测试					端到端测试				
		致命	严重	一般	建议	支持度平均值	致命	严重	一般	建议	支持度平均值
2	已发现	1	45	89	16	0.583 7	1	18	27	20	0.610 8
	遗漏	1	19	37	6	0.660 8	0	9	14	9	0.615 6
3	已发现	0	21	29	9	0.551 2	0	13	10	11	0.593 6
	遗漏	0	10	17	6	0.720 1	0	5	9	4	0.576 3

(2) 实验 2 结果与分析

根据实验 2 的过程,挖掘出第 7 轮的 API 网关层日志,结合实验 1 中对前 6 轮日志的挖掘结果,分别用于第 7 轮测试用例集的缩减,然后统计缩减后的测试用例数量、发现缺陷数量,进而计算 P_e 和 P_i,结果见表 13.

Table 13 Statistical data in the experiment 2

表 13 实验 2 的相关数据

日志来源的迭代轮次	日志数量	服务测试				端到端测试			
		测试用例数	缺陷数	P_e(%)	P_i(%)	测试用例数	缺陷数	P_e(%)	P_i(%)
1	46 394	487	34	58.20	27.65	132	9	52.00	30.77
2	91 219	434	35	62.74	25.53	84	11	69.45	15.38
3	148 851	446	39	61.71	17.02	83	11	69.81	15.38
4	234 317	439	39	62.31	17.02	78	11	71.64	15.38
5	419 351	414	43	64.46	8.51	69	11	74.53	15.38
6	912 653	632	47	45.75	0.00	63	13	77.09	0.00
7	1 035 519	589	47	49.44	0.00	60	13	78.18	0.00

从表 13 中的数据可以看出:日志规模对测试用例集的规模缩减程度和缺陷发现能力都有影响,日志规模越大、覆盖面越全,测试用例集缩减效果就越好.这是由于从 API 网关层日志中挖掘到的频繁路径更丰富,匹配到的测试用例集更多,对原先支持度为 0 的测试用例也被赋予了支持度,为启发式搜索提供的信息量更大,有利于搜索到更优的测试用例集.

3.4 实验结论

综合上述两个实验结果,可得出以下结论.

(1) 本文提出的测试用例缩减技术能够缩减测试用例规模 40%以上,且从缺陷等级上看,其缺陷发现能力要好于单纯考虑测试覆盖准则的随机搜索技术;

(2) 本文提出的测试用例缩减技术对原测试用例集缺陷发现能力的降幅不超过 10%,且遗漏的缺陷主要分布在日志中未出现或很少出现的业务上;

(3) 本文提出的测试用例缩减技术受 API 网关层日志规模的影响,日志规模越大、覆盖面越全,测试用例集缩减效果就越好.

需要说明的是:受项目素材限制,上述结论是否适用于不同领域、各种规模的微服务系统,还需要进一步开展工程实证.

4 总结与下一步工作

本文阐述了一种基于 API 网关层日志挖掘的微服务系统测试用例集缩减技术,详细描述了该技术的框架和每个步骤环节,并通过实验对其有效性进行了验证.该技术不要求微服务系统规约和架构描述作为输入信息,而是从 API 网关层日志挖掘服务及其调用关系,并作为测试用例缩减的依据,适用于微服务系统工程实践.该技术挖掘出的频繁路径信息真实反映了服务动态运行情况,使测试用例集侧重用户使用频次高的业务功能,避免测试场景偏离使用场景,一定程度上提高了测试效率.该技术包含了覆盖准则的判定要求,是“覆盖安全”的方法.该技术各环节均可使用程序实现,无需手工干预,能够自动化缩减测试用例集.

从技术落地的角度考虑,下一步将开展如下 3 个方面的工作.

(1) 本文提出的技术主要关注对原有测试用例集的缩减,未考虑系统变化对测试用例选择的影响,故缩减后的测试用例集不能发现未实现服务调用关系中的缺陷,实验中是通过直接沿用新增测试用例弥补这一覆盖性.我们将研究日志变化与微服务系统变化之间的关系,进而将日志变化作为参考因素,对新增的测试用例进行选择;

(2) 本文提及的日志挖掘技术是非增量式的,每一轮迭代都要重复扫描整个日志内容,这在实际应用过程中会造成一定的计算资源开销.我们将结合增量式日志挖掘方法改进本文所提技术,并探讨模式挖掘的可复用工作模式,一次挖掘、多处使用,进一步提升技术性能;

(3) 本文所提技术在本质上是通过挖掘日志信息获得用户使用模式,进而将其作为测试用例缩减的参考因素.事实上,该技术并不依赖于 API 网关层,对于其他架构特征,例如服务网格等,只要能获得服务调用日志数据,便可应用本技术.下一步将尝试在不同领域、不同架构模式的系统中应用本技术,采用更多的项目素材进行工程实证研究.

References:

[1] Fan CY, Ma SP. Migrating monolithic mobile application to microservice architecture: An experiment report. In: Proc. of the 2017 IEEE Int'l Conf. on AI & Mobile Services (AIMS). 2017. 109−112.

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

[3] Newman S. Building Microservices: Designing Fine-grained Systems. O'Reilly Media, 2015. 27−41.

[4] Mauro T. Adopting microservices at Netflix: Lessons for architectural design. 2015. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

[5] Larrucea X, Santamaria I, Colomo-Palacios R, Ebert C. Microservices. IEEE Software, 2018,35(3):96−100.

[6] Yoo S, Harman M. Regression testing minimization, selection and prioritization: A survey. Software Testing, Verification and Reliability, 2012,22(2):67−120.

[7] Pietrantuono R. On the testing resource allocation problem: Research trends and perspectives. Journal of Systems and Software, 2020. 161−175.

[8] Canfora G, Di Penta M. Service-oriented architectures testing: A survey. In: Proc. of the Software Engineering. Springer-Verlag, 2009. 78−105.

[9] Lu Z, *et al*. Web log mining. In: Proc. of the Web Intelligence. Berlin, Heidelberg: Springer-Verlag, 2003. 173−194.

[10] Meszaros G. xUnit: Test Patterns Refactoring Test Code. Boston: Addison-Wesley, 2007. 21−33.

[11] Kaczanowski T. Practical Unit Testing with JUnit and Mockito. 2013. 1−25.

[12] Peuster M, Dröge C, Boos C, Karl H. Joint testing and profiling of microservice-based network services using TTCN-3. ICT Express, 2019,2(5):150−153.

[13] Qiu D, Li BX, Ji SH, Leung H. Regression testing of Web service: A systematic mapping study. ACM Computing Surveys, 2014, 47(2):1−46.

[14] Li ZJ, Tan HF, Liu HH, Zhu J, Mitsumori NM. Business-process-driven gray-box SOA testing. IBM System, 2008,47(3):457−472.

[15] Khan TA, Heckel R. On model-based regression testing of Web-services using dependency analysis of visual contracts. In: Proc. of the 14th Int'l Conf. on Fundamental Approaches to Software Engineering: Part of the Joint European Conf. on Theory and Practice of Software (FASE 2011/ETAPS 2011). 2011. 341−355.

[16] Li BX, Qiu D, Leung H, Wang D. Automatic test case selection for regression testing of composite service based on extensible BPEL flow graph. Journal of Systems and Software, 2012,(85):1300−1324.

[17] Liu HH, Li ZJ, Zhu J, Tan HF. Business process regression testing. In: Proc. of the 5th Int'l Conf. on Service-oriented Computing (ICSOC 2007). 2007. 157−168.

[18] Chen L, Wang ZY, Xu L, Lu HM, Xu BW. Test case prioritization for Web service regression testing. In: Proc. of the 5th IEEE Int'l Symp. on Service Oriented System Engineering (SOSE 2010). 2010. 173−178.

[19] Mei LJ, Chan WK, Tse TH, Merkel RG. Tag-based techniques for black-box test case prioritization for service testing. In: Proc. of the 9th Int'l Conf. on Quality Software (QSIC 2009). 2009. 21−30.

[20] Azizi M, Do H. Graphite: A greedy graph-based technique for regression test case prioritization. In: Proc. of the IEEE Int'l Symp. on Software Reliability Engineering Workshops (ISSREW). 2018. 245−251.

[21] Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. IEEE Trans. on Software Engineering, 2007,33(4): 225−237.

[22] Ma SP, Fan CY, Chuang Y, Liu IH, Lan CW. Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Generation Computer Systems, 2019,100:724−735.

[23] Bozkurt M. Cost-aware Pareto optimal test suite minimisation for service-centric systems. In: Proc. of the 15th Annual Conf. on Genetic and Evolutionary Computation (GECCO 2013). 2013. 1429−1436.

[24] Dong WL. Test case reduction technique for BPEL-based testing. In: Proc. of the 2008 Int'l Symp. on Electronic Commerce and Security (ISECS 2008). 2008. 814−817.

[25] Mobasher B, Dai H, Luo T, Nakagawa M. Effective personalization based on association rule discovery from Web usage data. In: Proc. of the 3rd Int'l Workshop on Web Information and Data. 2001. 9−23.

[26] Aggarwal CC, Han J. Frequent Pattern Mining. Springer Int'l Publishing Switzerland, 2014. 19−36.

[27] Hemmati H, Arcuri A, Briand LC. Achieving scalable model based testing through test case diversity. ACM Trans. on Software Engineering and Methodology (TOSEM), 2013,22(1):601−642.

[28] Coviello C, Romano S, Scanniello G, Marchetto A, Corazza A, Antoniol G. Adequate vs. inadequate test suite reduction approaches. Information and Software Technology, 2020,119(1):1−19.

陈立哲(1985－),男,工程师,主要研究领域为软件测试,数据挖掘,机器学习.

杨海燕(1974－),女,讲师,主要研究领域为软件测试,软件安全性与可靠性.

吴际(1974－),男,博士,副教授,博士生导师,CCF 专业会员,主要研究领域为软件测试,软件建模,软件安全性与可靠性.

张奎(1985－),男,讲师,主要研究领域为基于模型的软件设计与验证,软件成本度量,自然语言处理.