计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.210100149

一种面向异常传播的微服务故障诊断方法

王 焘[1,2] 张树东[3] 李 安[1] 邵亚茹[3] 张文博[1,2]
1 中国科学院软件研究所 北京 100190
2 中国科学院软件研究所计算机科学国家重点实验室 北京 100190
3 首都师范大学信息工程学院 北京 100048
(wangtao@iscas.ac.cn)

摘 要 微服务软件架构将大型复杂应用软件拆分成多个可独立部署的相互之间通过轻量级通信机制协作的微服务,从而实现了应用软件的敏捷开发和持续交付。然而,应用软件的微服务数量众多,调用关系复杂,当某个微服务出现故障时会引发与之交互的微服务也出现异常,从而大幅增加了软件应用出现故障的可能性。面对众多异常微服务,考虑到异常的传播性,如何高效、准确地定位引发异常的故障微服务,成为亟待解决的问题。针对该问题,文中提出一种面向异常传播的微服务故障诊断方法。首先,监测微服务度量信息与微服务之间的调用行为;然后,基于回归分析构建度量与 API 调用之间的回归模型以检测异常微服务;同时,构建微服务依赖图以刻画微服务间的异常传播;最后,基于服务依赖图以及异常服务集合得到故障传播子图,并基于 PageRank 算法找出最有可能引发异常的根因,即故障微服务。实验结果表明,该方法能够有效检测异常服务,准确诊断故障微服务,同时具有较低的开销。

关键词:故障诊断;微服务;服务调用;度量关联;异常传播
中图法分类号 TP311

Anomaly Propagation Based Fault Diagnosis for Microservices

WANG Tao[1,2],ZHANG Shu-dong[3],LI An[1],SHAO Ya-ru[3] and ZHANG Wen-bo[1,2]
1 Institute of Software,Chinese Academy of Sciences,Beijing 100190,China
2 State Key Laboratory of Computer Sciences,Institute of Software,Chinese Academy of Sciences,Beijing 100190,China
3 Information Engineering College,Capital Normal University,Beijing 100048,China

Abstract Microservice architectures separate a large-scale complex application into multiple independent microservices. These microservices with various technology stacks communicate with lightweight protocols to implement agile development and continuous delivery. Since the application using a microservice architecture has a large number of microservices communicating with each other,the faulty microservice should cause other microservices interacting with the faulty one to appear anomalies. How to detect anomalous microservices and locate the root cause microservice has become one of the keys of ensuring the reliability of a microservice based application. To address the above issue,this paper proposes an anomaly propagation-based fault diagnosis approach for microservices by considering the propagation of faults. First,we monitor the interactions between microservices to construct a service dependency graph for characterizing anomaly propagation. Second,we construct a regression model between metrics and API calls to detect anomalous services. Third,we get the fault propagation subgraph by combining the service dependency graph and the detected abnormal service. Finally,we calculate the anomaly degree of microservices with a PageRank algorithm to locate the most likely root cause of the fault. The experimental results show that our approach can locate faulty microservices with low overhead.

Keywords Fault diagnosis,Microservices,Service invocation,Metric correlation,Anomaly propagation

1 引言

随着互联网应用的蓬勃发展,传统软件架构已难以满足灵活多变的业务需求。微服务架构将单个应用拆分成若干个可独立部署的微服务,相互间通过轻量级通信协议协同。不同的微服务围绕自己独立的业务进行开发,可采用多样化的

到稿日期:2021-01-19 返修日期:2021-05-11
基金项目:国家重点研发计划(2017YFB1400804);国家自然科学基金(61872344);北京市自然科学基金(4182070);中国科学院青年创新促进会人才专项(2018144)
This work was supported by the National Key Research and Development Project(2017YFB1400804),National Natural Science Foundation of China(61872344),Natural Science Foundation of Beijing(4182070) and Youth Innovation Promotion Association of the Chinese Academy of Sciences(2018144).
通信作者:张树东(zsd@cnu.edu.cn)

编程语言，选取不同的后台存储技术。这种特性便于软件的敏捷开发与持续交付，为传统软件架构中面临的问题提出了良好的解决方案[1]。目前，越来越多的互联网企业采用微服务架构开发、部署分布式应用软件，如腾讯的微信系统将3000多种微服务运行在20000多台物理主机上[2]。然而，微服务数量繁多，如Netflix在线服务系统运行500多种微服务，6000多个微服务实例，每天处理20多亿个API请求。同时，微服务间交互复杂，如亚马逊调用100多个微服务的接口来构建一个页面，增加了发生故障的可能性以及定位故障原因的难度[3-4]。相较于传统软件架构，微服务架构在运行监测及故障诊断等方面面临着巨大挑战。首先，在服务调用监测方面，微服务采用多样化的技术栈，而目前的服务调用监测方法主要面向特定操作系统或编程语言，通用性较差。其次，在异常检测方面，交互式应用通常会调用多个微服务以处理用户请求，仅监测某个微服务只能获得有限的信息，难以刻画应用的整体运行状态。尤其是在故障诊断方面，由于异常的传播性，某个微服务发生故障可能导致与其直接或间接相关的服务也出现异常，上述方法都难以确定故障的根本原因。因此，快速检测微服务异常，有效刻画异常的传播，准确定位故障根因，已成为保障微服务应用可靠性的关键之一。

针对上述问题，本文提出一种面向异常传播的微服务故障诊断方法。首先，监测微服务度量信息与微服务之间的调用行为；然后，基于回归分析构建度量与API调用之间的回归模型以检测异常微服务；同时，构建微服务依赖图以刻画微服务间的异常传播；最后，基于服务依赖图以及异常服务集合得到故障传播子图，并基于PageRank算法找出最有可能引发异常的根因，即故障微服务。该方法旨在及时检测异常微服务，准确定位引发微服务异常的故障微服务及服务接口，从而提高系统管理员排查故障原因的效率。综上所述，本文的创新点主要体现在以下3个方面。

(1)在监测方面：本文使用服务代理方式获取微服务调用行为，无须领域知识及代码侵入，具有较好的通用性和可扩展性。

(2)在异常检测方面：本文提出基于Lasso回归的服务建模方法，能够自动化构建运行状态模型，并解决传统方法因度量共线性所带来的建模不准确的问题。

(3)在故障诊断方面：本文建模多服务调用与多度量的相关性，引入PageRank算法，考虑微服务间故障传播的相互影响，动态评估微服异常程度以准确定位问题原因。

2　相关工作

近年来，随着Kubernetes，SpringCloud等微服务管理基础框架的广泛应用，微服务技术在软件开发、部署、管理等方面的优势日益明显，微服务架构被众多互联网企业所采用。国内外学者针对微服务的运行监测、异常检测与故障诊断等技术开展了大量的研究，在我们此前的综述[5]中对此作了分析与探讨，当前分布式软件系统故障诊断方法主要包括基于执行轨迹分析、基于性能度量分析、基于服务关联分析等3类方法。

基于执行轨迹分析的方法面向交互式应用软件，将监测代码注入到目标软件中，分析请求的执行流程。CloudDiag[6]将监测代码注入到软件代码的特定位置，还原请求处理的执行流程，记录函数处理请求的时间，使用快速矩阵恢复算法查找延迟增加的函数调用。然而，云计算平台屏蔽了应用软件内部的执行状态，同时应用软件的巨大规模使得故障诊断越发困难。我们此前的工作[7]针对执行逻辑和性能表现异常进行故障诊断。构建方法调用树以刻画服务请求处理的执行轨迹，计算实时请求与正常轨迹间的树编辑距离来评估实时请求的异常程度，并使用主成分分析定位引发性能异常波动的方法调用。该方法可以有效监测服务请求轨迹，实现了面向方法调用的细粒度故障诊断。但是，其仍然需要了解应用的技术体系，在关键位置加入监测点，如果不能准确、全面地设定监测位置，则会导致较高的错报率。

基于性能度量分析的方法调用操作系统或应用软件提供的接口搜集监测数据，分析系统度量的变化情况。CloudPD[8]是面向公有云的轻量级自动化的故障诊断框架，监测虚拟机和物理服务器的多种度量，事先将故障发生所表现的度量变化定义为签名，通过在线签名匹配来检测频繁发生的故障。该方法事先标记已知故障所对应的表现，在待检测故障已知时，具有较高的准确性和及时性，但难以应对未知故障所引起的异常。Chen等[9]提出一种基于对抗分析的贝叶斯网络训练方法，其通过无监督学习构建贝叶斯网络模型，并通过检测度量值与模型的偏差来检测性能度量异常。Cherkasova等[10]使用线性回归估算应用的CPU使用，自变量为各类事务的CPU时间，因变量为系统的总CPU时间，通过检查事务CPU时间统计分布的变化来分析性能异常。该方法仅能检测到事务处理的CPU时间异常，无法应对其他物理资源度量，如内存、磁盘、网络。我们此前的工作[11]提出一种基于在线聚类的用户请求类型的学习及分类方法，建立了相应的请求与资源的关联模型[11]；在特定请求空间下，基于局部异常因子自动化诊断异常物理资源[12]的方法，进而根据物理资源的波动情况动态调整监测周期，以减少网络传输的监测数据量，从而降低监测所造成的目标系统性能衰减[13]。

基于服务关联分析的方法通过分析服务间关联性的变化来检测应用软件运行状态的异常。Roots[14]提供了可扩展的PaaS云监测服务接口，当应用软件调用云服务时，Roots拦截并记录事件，然后将其与特定的请求相关联，计算应用以及云服务的调用延迟。CauseInfer[15]监测服务的TCP延迟作为SLO度量，根据服务间的TCP连接建立无向连接图，进而分析服务依赖的方向；同时，收集服务运行时在应用、进程和操作系统等方面的度量数据，构建度量间依赖关系。与之类似，Lin等[16]抓取服务间网络数据包以分析服务调用关系，当服务调用延迟违背SLO，则检测为性能异常，通过分析服务依赖关系推断性能异常的潜在原因。Sieve[17]首先观察度量信号变化，自动过滤不重要的度量以减少度量数量，与Mariani等[18]的工作类似，其采用格兰杰因果检验发现度量间的因果关系以构建有向图，然后通过检测微服务间的依赖性变化来定位故障根因。与之类似，Nie等[19]提出一种基于专家反馈的迭代式因果图学习方法，使用FP-Growth算法从历史数据中挖掘关联规则，基于Pearson系数判断关联方向以生成因

果图，而后专家对故障诊断结果进行标记，使用随机森林算法对因果图进行迭代更新。

我们此前的工作[20]采用典型相关分析方法挖掘依赖服务的度量相关性，进而通过检测度量相关性的变化来检测度量异常。进一步，我们提出了一种基于关联挖掘的服务配置错误检测方法，发现频繁改变配置项，计算配置项的名称、取值和类型的相似性，生成配置项对的关联系数，再根据过滤规则确定关联配置项对集合，进而检测配置项间的不一致所产生的配置故障[21]。以上方法面向度量或配置的特定故障类型，挖掘服务对的相关性，因未能形成全局调用关系图而难以准确定位故障原因。

3 面向异常传播的故障诊断方法

3.1 方法概述

本文提出一种基于异常传播的微服务故障诊断方法，在定位问题根本原因时，分析服务间的依赖关系以及交互行为，研究故障传播模型，分析故障发生的可疑路径，评估路径上各服务出现故障的概率，据此定位故障服务。该方法的技术路线如图1所示，主要包括服务运行监测、运行状态建模、异常服务检测和故障服务诊断4个阶段。

(1)服务运行监测：运行时有效搜集监测数据是分析服务运行状态及诊断服务故障的基础。本文通过服务代理获取服务度量和服务调用两个方面的监测数据。当微服务启动时，使用“边车”(sidecar)模式自动为其加载服务代理以监测其运行状态。服务度量信息包括物理资源使用(如CPU利用率)及服务性能表现(如服务吞吐量)等；服务调用信息包括请求发送方微服务信息、被调用微服务信息及请求处理信息等。

(2)运行状态建模：根据历史监测数据构建服务的运行状态模型，包括服务度量模型及服务调用模型。服务度量模型使用Lasso回归刻画服务在正常运行状态下服务度量间的关联关系，以检测出现异常的微服务。服务调用模型构建并更新服务调用拓扑图刻画服务间的影响，以分析引发异常的根因微服务。

(3)异常服务检测：在系统运行时，实时监测服务度量值，使用构建的服务度量模型预测服务度量值，计算模型残差变化以量化预测值与监测值的偏差，当以上偏差超出阈值，则判定为偏离模型，检测该微服务出现异常。

(4)故障服务诊断：如果某个微服务被越多的异常微服务调用，或者其调用越多的异常微服务，则该微服务的异常可能性就越高。PageRank算法能够很好地反映故障传播现象，从而达到量化故障程度的效果。因此，本文将检测的异常微服务集合对应到服务调用拓扑图以形成异常传播子图，然后基于PageRank算法评估各微服务的异常程度并进行排序，从而确定可能引发异常的根因，即故障微服务。

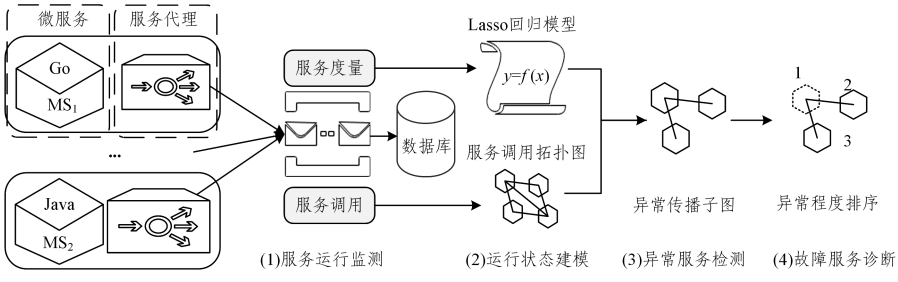

图1 方法概况

Fig. 1 Approach overview

3.2 微服务运行监测

服务运行监测是异常检测及故障诊断的基础，其搜集微服务在运行时的物理资源和服务性能等度量数据，获取微服务间的调用信息，进而建模微服务依赖关系。

3.2.1 服务度量监测

监测度量可以直观反映单个微服务的运行状态，微服务出现故障通常表现为相应度量的异常变化。例如，代码陷入死循环或线程进入循环等待会占用较多CPU时间，表现为监测到的CPU占用率过高；代码发生内存泄漏表现为监测到的内存使用在一段时间内不断升高等。本文实时监测物理资源与服务性能这两类微服务度量数据。物理资源度量监测通过调用操作系统所提供的接口获取监测数据，主要包括CPU时间、内存使用量、磁盘读写频率、网络流量等。服务性能度量监测通过接收、转发微服务请求或响应请求时解析数据内容获取监测数据，主要包括每秒接收请求数量、请求处理延迟、请求处理吞吐量、请求\响应数据包大小等。

3.2.2 服务调用监测

微服务之间松散耦合，各微服务具有独立的技术栈，如多样化的编程语言，从而简化了微服务的更新升级，但也给跨语言的服务调用监测带来了困难。本文采用基于服务代理(Proxy)的调用监测方法，无须考虑被监测服务的开发语言与内部执行逻辑，可实现透明的服务调用监测。服务代理即轻量级网络代理，与微服务部署在一起，转发微服务发送的请求和返回的响应，同时监测微服务调用及响应信息，以构建服务拓扑结构图。为了应对微服务跨部门和跨公司的服务代理通信问题，在实现方面，本文采用服务网格Istio[1)]的服务代理，其底层调用Kubernetes[2)]的服务发现机制，较好地实现了跨域的微服务通信。

服务代理监测到的请求调用信息表示为：

$$Ni=\{rqstUID, svcUID, calleeList, svcInfo\}$$

1) Istio: Connect, secure, control, and observe services. https://istio.io

2) Kubernetes. https://kubernetes.io/

其中，*rqstUID* 为请求调用的标识符，由服务调用的发起者生成，通过服务代理转发到被调用者，以监测服务间的调用关系；*svcUID* 为服务标识符；*calleeList* 为服务调用列表；*svcInfo* 为服务调用的相关信息。

$$svcInfo=\{src, dest, startTime, endTime, dur, proxy, collector\}$$

其中，*src* 为发起请求调用的微服务；*dest* 为接收请求的微服务；*startTime* 和 *endTime* 分别为发起和接收请求调用的时间；*dur* 为从接收请求到返回响应的处理延迟；*proxy* 为微服务的服务代理；*collector* 为监测数据搜集器，汇聚服务调用的监测数据以构建服务调用拓扑。

3.2.3　服务依赖图的构建

服务依赖用于表达微服务间的因果关系，服务依赖定义为：给定两个服务实例 X 和 Y，如果实例 X 的状态变化可以影响实例 Y 的状态，并且实例 Y 对实例 X 没有影响，则认为实例 Y 对实例 X 具有依赖关系，表示为 $X->Y$，或者在图中可以表示为实例 X 是实例 Y 的父节点，即 $X\in P(Y)$。服务依赖用以描述服务发生故障时，故障服务对调用它的其他服务的影响。可以通过监测服务调用行为，刻画服务调用拓扑以反映服务依赖关系，从而构建有向无环图，图中的节点表示服务实例，节点之间的有向边表示服务实例间的依赖关系。

服务代理将监测到的微服务调用信息转发给数据收集器 *collector* 以汇总监测信息，并构建服务依赖图。服务依赖图主要用来反映故障发生时异常在服务间的传播情况，以定位故障根因服务。每一次服务调用，服务代理都会生成监测数据记录，本文根据这些信息构建服务依赖图。根据 *rqstUID* 对监测记录分类，其中成功调用的请求成对出现，而单个偶然出现的记录则可能表示实例不存在或者调用失败，此类记录将被丢弃。而后，根据记录中的 *src* 和 *dest* 字段确定服务请求调用的发送方和接收方，以及服务依赖的方向，并更新到依赖图中。

服务依赖图构建的具体方法如算法 1 所示。服务调用列表 T 存储服务代理监测到的服务调用记录，列表中每一个项表示一条调用记录；初始化服务依赖图 G 以及用于存储监测数据的二元数组 $A[0\cdots n][0\cdots n]$，其中 $V(G)$ 表示图 G 的节点集合，$E(G)$ 表示图 G 的边集合，n 为微服务数量，将节点集和边集都初始化为空（第 1－4 行）；遍历监测记录 T，从中提取出发出和接收请求的微服务，将其编号作为数组下标，并将监测数据存储在数组 A 中（第 5－8 行）；遍历二元数组 A，如果在发出和接收请求的微服务监测数据中未检索到请求记录，则删除该调用信息（第 9－17 行）；根据二元数组 A 构建服务依赖图，将节点和边信息存储到图 G 中（第 18－26 行）；返回构建的服务依赖图 G（第 27 行）。

算法 1　服务依赖图的构建

输入：服务调用列表 T

输出：服务依赖图 G

执行过程：

```
1. Initialize Graph G;
2. V(G)←null;
3. E(G)←null;
4. Initialize array A[0…n][0…n];
5. for all Ti in T do:
6.      get source,destination fromTi;
7.      A[source][destination]←Ti;
8. end for
9. for i← 0 to n do:
10.   for j←i+1 to n do;
11.        if A[i][j] is not null and A[j][i] is null do:
12.        A[i][j]← null;
13.     else if A[i][j] is null and A[j][i] is not null
14. do:
15.        A[j][i]← null;
16.     end if
17.   end for
18. end for
19. for i←0 to n do:
20.     for j←i+1 to n do;
21.     if A[i][j] is not null do:
22.          add i,j into V(G);
23.          add(i,j) into E(G);
24.     end if
25.   end for
26. end for
27. return G.
```

3.3　微服务异常检测

3.3.1　微服务调用资源建模

服务处理请求会占用系统物理资源，监测到的相应物理资源的度量值会发生变化，如调用的服务执行计算密集型任务，就会占用较多 CPU 资源，那么此时监测到的 CPU 占用率就会增加。服务接口调用次数与度量值之间存在着相关性，并且由于不同服务接口提供不同的服务，调用不同服务接口所占用的资源也不同。

本文使用回归模型对服务接口调用和物理资源度量的关联进行建模，以反映服务调用与物理资源使用间的关系。回归模型的因变量是度量值，自变量是服务接口的调用次数。某些度量之间会存在相关性，如服务请求频率与网络带宽的变化，这是由于微服务之间每发起一次服务请求必然会占用一定的网络带宽，回归模型需要能够应对共线性问题。因此，本文采用 Lasso 回归模型[22]，与传统多元线性回归方法相比，Lasso 回归加入了惩罚函数以减少模型中参数的数量，将一些影响较小参数的系数减小为零，从而降低共线性问题所带来的影响。

微服务调用建模如图 2 所示，微服务 ms_r 具有 q 个调用接口 $si_1, si_2, \cdots, si_q$，p 个微服务 $ms_1, ms_2, \cdots, ms_p$，调用微服务 ms_r 的服务接口。$X_t^i=(x_{t1}^i, x_{t2}^i, \cdots, x_{tq}^i)^{\mathrm{T}}$ 表示在时刻 t，微服务 ms_i 对微服务 ms_r 的 q 个服务接口的调用次数，其中元素 x_{t1}^i 表示时刻 t 微服务 ms_i 调用微服务 ms_r 的服务接口 t_1 的次数，对其做标准化处理，作为 Lasso 回归模型的解释变量。Y_t 表示微服务 ms_i 的物理资源度量 m 在时刻 t 的监测值，作为模型的响应变量。那么，微服务 ms_i 物理资源使用的回归模型可表示为：

$$Y_t=\sum_{i=1}^{p}\sum_{j=1}^{q}\omega_j^i x_{tj}^i+\alpha$$

其中，p 为对该服务发起调用的服务的个数，q 为该微服务的服务接口数量，x_{tj}^i 为在时刻 t 微服务 ms_i 调用服务接口 j 的次数，ω_j^i 为 x_{tj}^i 的回归系数，α 为随机误差。

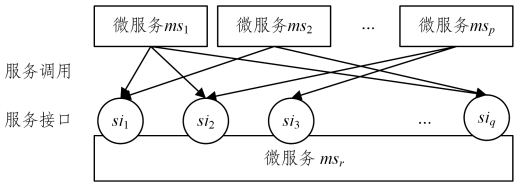

图 2 微服务调用示例

Fig. 2 Example of calling microservices

在约束条件 $\sum_{i=1}^{p}\sum_{j=1}^{q}|\omega_j^i|\leqslant c$ 下，通过坐标下降法极小化误差项 $\sum_{t=1}^{T}(Y_t-\sum_{i=1}^{p}\sum_{j=1}^{q}\omega_j^i x_{tj}^i-\alpha)^2$，其中，$T$ 是用于回归建模数据的时间长度，参数 c 通过广义交叉验证法来选择：

$$CGV(c)=\frac{1}{N}\frac{RSS(c)}{[1-p(c)/N]^2}$$

其中，$RSS(c)$ 表示残差平方和，$RSS=\sum_{i=1}^{N}(Y_i-Y_i(c))^2$；$p(c)$ 为有效回归系数的个数；N 为所监测度量的数量。

3.3.2 微服务异常评估

3.3.1 节提出了微服务调用资源模型以刻画微服务接口调用与微服务物理资源度量的关联性。基于统计学原理，对于正常运行的应用软件，这种关联性长期稳定存在[13]。例如，对于多层架构的 Web 应用，前端 Web 服务器 HTTP 请求的激增通常会引起后端数据库服务器 SQL 请求的激增，那么 HTTP 请求和 SQL 请求之间，以及请求数量与服务器物理资源使用量之间显然存在着关联性。当微服务出现异常时，服务接口调用与物理资源度量的关联性将会出现明显波动[12]。那么，将不同微服务的多种服务接口调用次数代入模型，可以得到物理度量的预测值，当其与实际监测到的度量值存在较大偏差时，则检测微服务出现了资源使用异常。实际值偏离程度使用残差来表现：

$$R_i(t)=Y_i(t)-\overline{Y_i(t)}$$

其中，$Y_i(t)$ 为微服务第 i 个度量在时刻 t 的监测值，$\overline{Y_i(t)}$ 为微服务第 i 个度量在时刻 t 的预测值。

通过监测残差变化来检测异常，当残差超出阈值则检测为异常发生。由于不同微服务的残差波动程度具有差异性，本文基于历史监测数据动态调整残差的阈值。异常评估需要阈值来确定残差是否正常，而该阈值只有在已知故障残差分布的情况下才能合理设置。但在实践当中，差异化的服务特性决定了残差值的多样性，本文使用其历史值的统计信息来动态获得阈值。与服务正常运行状态不同，系统故障在实际生产环境中并不常见，观察到的残差大多是正常噪声，根据我们此前工作的分析[24]，残差值符合高斯分布。因此，本文根据统计学中普遍采用的 3δ 定律，为每个微服务实例设置阈值为 $\mu+3\sigma$，其中，μ 为该服务实例历史残差的平均值，而 σ 为历史残差的标准差。

3.4 微服务故障诊断

在云计算环境下，应用通常由大量的微服务构成，微服务间存在着复杂的交互行为，单个微服务发生故障可能会导致调用其的微服务或其调用的微服务表现异常。例如，后端数据访问服务的磁盘读写率飙升，可能是由于前端某些服务频繁读写后端数据所造成的。再如，前端应用服务吞吐量锐减，可能是后端数据访问服务出现故障而难以获取数据所致。因此，当某些微服务出现故障，往往会在很短的时间内造成与其交互（调用其或其调用）的众多微服务表现出异常。因此，本节在上节检测到异常微服务的基础上，分析故障发生的请求处理路径即故障传播途径，根据与异常服务的交互频率评估路径上各微服务出现故障的概率，从而诊断可疑故障微服务。

本文根据检测的异常服务集合以及构建的服务依赖图生成微服务异常传播图。在服务依赖图中保留检测为异常的微服务实例节点以及这些节点间的有向边，生成仅包含异常节点的最大连接子图。异常传播图反映了微服务故障对与其具有调用关系的微服务所造成的影响。本文采用图中心度算法评估故障传播图中单个节点对其他节点的影响程度，得分越高的微服务越有可能是引发异常的根本原因。图中心度算法[23]是社交网络分析学中用于衡量图中节点重要程度的算法。其中，度中心性方法只考虑到一个节点对与其距离为 1 的其他节点的影响，无法考虑异常在图中的传播。特征向量中心性方法是度中心性方法的扩展，节点的重要程度也与该节点的相邻节点的重要程度有关。因此，本文采用基于特征向量中心性方法改进的 PageRank 算法[24]来评估节点的重要程度。PageRank 算法的核心思想是，如果一个节点被很多其他节点链接，其重要程度相应地会较高；如果一个重要程度很高的节点链接到另外某个节点，该节点的重要程度会相应提高。

本文使用 PageRank 算法评估微服务的异常程度。在微服务的调用拓扑图中，如果某个微服务被越多的微服务调用，则该微服务的重要程度越高，发生故障时对整个异常传播图的影响也越大。同时，被该微服务调用的微服务，重要程度也相应提高。具体步骤如下：

(1)将异常度量的值归一化到[0,1]区间内，对于异常度量来说，异常得分是归一化之后的预测残差的绝对值。

(2)异常微服务各度量的异常程度之和为微服务 s 的异常程度值的初值：

$$AD(s)=[mad_1,mad_2,\cdots,mad_n]^N$$

其中，mad_i 为度量 i 的异常程度值，n 为度量数量。

(3)微服务 s_i 的异常程度值迭代计算为：

$$AD^k(s_i)=\frac{1-q}{N}+q\times\sum_{s_j\in I(s_i)}^{I(s_j)}\frac{AD^{k-1(s_j)}}{O(s_j)}$$

其满足约束 $|AD^k-AD^{k-1}|<\delta$，表示节点的异常程度值 AD^k 趋于稳定，则迭代结束。其中，$AD^k(s_i)$ 为第 k 次迭代后服务 s_i 的异常分数，AD^k 和 AD^{k-1} 分别为第 k 次迭代和第 $k-1$ 次迭代后微服务各度量异常分数构成的列向量，δ 为迭代停止的阈值。

PageRank 算法中的迭代过程在不断扩大各服务实例异常程度的相对差异，最终会达到稳定的收敛状态。根据上述算法，对微服务的异常程度排序，得分越高表示该微服务在异常传播图中影响越大，则该微服务越有可能是引发此次异常传播的故障微服务。

4　实验评价

4.1　实验环境

实验部署环境如图 3 所示，将典型微服务应用 Sock-Shop[1] 和 Social-Network[2] 作为实验对象，使用 Docker[3] 容器部署微服务，Kubernetes 管理容器集群，Pod 副本最大数量设置为 3，Istio 作为服务代理与每个容器共同部署在相同 Pod 中，Apache JMeter[4] 模拟用户发起对应用的请求，错误注入器将故障注入应用系统，故障诊断器定位故障。表 1 列出了应用部署的软件和硬件信息。

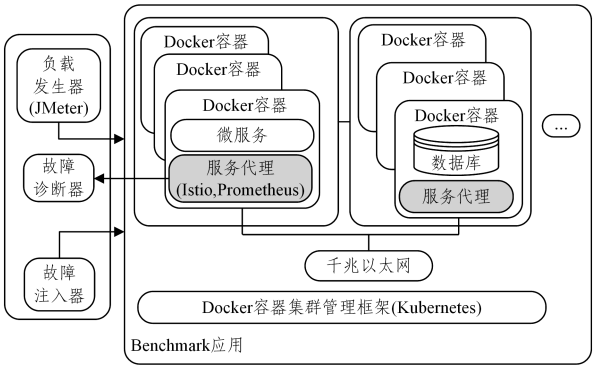

图 3　实验环境

Fig. 3　Experimental environment

表 1　软硬件配置

Table 1　Software/hardware configurations

系统名称	物理信息	软件信息
Benchmark 应用	Intel Xeon E-5-2620 CPU/16 GB RAM/500 GB Disk/1 000 Mbps NET/4 Hosts	Kubernetes1.13/Istio1.1/Prometheus
故障诊断系统	Intel Core i5-2300/8 GB RAM/200 GB Disk/1 000 Mbps NET	apache-jmeter-3.1
负载生成器		
故障注入器		

本文使用开源负载发生器 JMeter 发送 HTTP 请求以模拟并发用户的访问。为了模拟真实应用场景，生成的负载符合电子商务基准测试 TPC-W 规范[1]。TPC-W 定义了 3 种负载模式，包括浏览、购物和订购。客户在一个会话中连续请求访问网站，两个相邻请求的间隔时间设置为 2 s。实验采用 3 种负载模式，随机改变负载类型。

4.2　实例研究：Sock-Shop

4.2.1　实验对象

Sock-Shop 是基于微服务框架 Spring Boot、使用 Go kit 及 Node.js 开发的在线销售的典型小型规模微服务应用，包括 13 个微服务。在 Sock-Shop 中，Front-end 接收用户请求，调用其他微服务，聚合结果通过前端页面显示给用户，使用 NodeJS 开发；Order Service 实现订单生成、查询、更新等，使用 Java 和 .NET 开发；Payment Service 提供在线支付，使用 Go 开发；User Service 管理用户信息，使用 Go 开发；Catalogue Service 实现产品分类和目录管理，使用 Go 开发；Cart Service 管理购物车，包括添加、删除、修改购物车等，使用 Java 编写；Shipping 提供送货服务，接收来自 Order 的订单信息并生成发货任务，使用 Java 开发；Queue-Master Service 读取发货任务并模拟发货过程，使用 Java 编写；不同核心服务采用单独的数据库进行存储。Sock-Shop 中不同的服务采用多种语言编写，整个应用分为多个功能单一，可独立开发、部署、扩展的微服务，是典型的微服务应用，因此，本文采用 Sock-Shop 作为实验对象。

4.2.2　故障注入

根据美国联邦标准(1037C)，故障分为随机故障和系统故障[5]。随机故障(如 CPU 处理故障、网络拥塞)是具体出现时间无法确定的导致设备单元无法正确执行的意外情况；系统故障(如配置故障)是在特定条件下出现的设备规格中的错误。本文注入这两类典型的故障，以 3 个典型的故障为例，在实验中对其诊断，以验证方法的有效性。故障(1)和故障(2)属于随机故障，本文在第 100 s 触发故障，在第 200 s 结束它们；故障(3)属于系统故障，本文在实验过程中触发故障并持续到实验结束。每个实验持续 5 min(300 s)，使用 Apache Jmeter 3.1 生成并发为 300 的工作负载。我们将故障注入的功能实现为服务形式，并在 GitHub 开源，通过 Kubernetes 在目标微服务启动时加载故障注入服务[6]，并在目标微服务运行时进行故障注入操作。从 Sock-Shop 的 13 个微服务中随机选取 7 个微服务，每次实验向其中一个微服务注入 3 种故障之一，每种故障实验 50 次，具体注入故障如下。

故障(1)：网络数据包丢失[25]。本文调用 Linux 流量控制工具 TC(Traffic Control)[7] 造成 25% 的网络数据包丢失。每个实验中，模拟负载 300 s，在第 100 s 注入故障，故障持续 100 s，然后停止故障。

故障(2)：响应时间延迟[16]。本文对目标微服务进行代码注入，当该微服务接收请求时，调用实现的休眠服务，在休眠期间不处理用户请求，休眠周期随机设置在[0.01，5.00] s 以评价方法对响应延迟的敏感性，故障持续存在。

故障(3)：数据库配置故障[26]。在第 100 s 将 MongoDB 的访问权限更改为只读，使微服务在处理请求时因不能往数据库写入并保存数据而无法正常返回请求响应，故障持续存在。

4.2.3　故障诊断

本文首先量化计算各微服务实例的异常程度，而后根据

1) Sock Shop: A Microservices Demo Application. https://microservices-demo.github.io/

2) SocialNetwork. https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork

3) Docker. https://www.docker.com/

4) Apache JMeter. http://jmeter.apache.org/download_jmeter.cgi

5) Fault. https://en.wikipedia.org/wiki/Fault_(technology)

6) Fault Injector. https://github.com/iscas-microservice-team/microFaultInjection

7) Traffic Control. http://man7.org/linux/man-pages/man8/tc.8.html

异常程度从高到低进行 Top-k 排序，依次排查最可疑即异常程度最高的 k 个微服务实例。k 越小，则需要排查的可疑微服务实例数量越少，排查效率就越高，本文实验取 Top-1 和 Top-3 两种情况进行考查。微服务故障诊断的实验结果如表 2 所列，实验分析发现，故障诊断准确率与该微服务被其他微服务调用和调用其他微服务的频率（即服务调用拓扑图节点的出度和入度）、微服务对异常资源的敏感程度（如服务是网络密集型）以及 k 的取值等相关。Front-end 服务没有被其他服务调用，该微服务的异常表现较为显著；当将 k 值设为 3 时，即 3 次检测准确率最多会有 9% 的提升；当 k 大于 3 时，准确率的提升较小。对于故障(1)，我们观察到，Shipping 服务和 Payment 服务出现较多漏报现象，这是由于它们不是网络密集型服务，对网络资源占用不敏感；对于故障(2)，我们观察到，当在 Shipping 服务注入随机休眠的代码时，并未检测到该故障，分析发现这是由于该服务未对其他服务发起调用请求，使得其他服务无故障表现；对于故障(3)，我们观察到，MongoDB 数据访问服务以及调用其 Cart，User，Order 等微服务都出现异常，通过故障传播分析准确定位到了数据访问服务故障，因而考虑故障传播因素具有必要性。

表 2 故障检测的准确率

Table 2 Accuracy of fault detection

（单位：%）

微服务	front-end	catalogue	carts	order	user	shipping	payment
Top 1	100	91	82	93	87	77	72
Top 3	100	96	90	95	91	84	81

4.2.4 监测开销

本文使用服务代理，通过转发微服务的调用及响应以监测微服务行为，并且通过调用操作系统接口监测物理资源使用情况。在微服务启动时，会自动生成并加载服务代理，在请求数据包的头部信息中对请求和响应进行标记，然后将监测数据汇总并分析，从而延长服务响应时间。本文对比分析了服务代理启动前后服务的平均启动时间，在不开启和开启服务代理的情况，对 Front-end Service，User Service，Order Service，Catalogue Service 等微服务各调用 1000 次并返回，求出服务响应时间。如表 3 所列，在开启服务代理前，服务响应时间分别为 1.41 s，1.39 s，1.14 s，1.50 s，1.13 s，1.42 s，1.15 s；在开启服务代理后，服务响应时间分别为 1.48 s，1.45 s，1.24 s，1.54 s，1.21 s，1.47 s，1.24 s。服务响应时间增加了约 3.25%～8.49%，这是由于每次服务调用只需要收集很少的信息，对服务响应时间影响较小。

表 3 故障检测响应时间开销

Table 3 Overhead of fault detection in response time

（单位：s）

微服务	front-end	catalogue	carts	order	user	shipping	payment
未启动检测	1.41	1.39	1.14	1.50	1.13	1.42	1.15
启动检测	1.48	1.45	1.24	1.54	1.21	1.47	1.24

4.3 实例研究：Social-Network

4.3.1 实验对象

为了验证本文方法对于较大规模实际应用的适用性，本节采用微服务应用标准测试集 DeathStarBench[1)][27] 中的 Social-Network 应用作为实验对象，这是中等规模的典型微服务架构应用。该应用包含 36 个微服务，微服务间采用 RPC 进行通信，由 C，C++，Java，node.js，Python，Scala，PHP，Javascript，Go 等 9 种编程语言开发。Social-Network 采用端到端的服务实现了广播模式的社交网络，用户通过 HTTP 协议发送请求，首先请求到达负载均衡器 Nginx，而后使用 php-fpm 模块组合展示后端广告、搜索等微服务的响应报文，消息传输采用 Thrift RPCs 实现。用户可以给其他用户分享文本、媒体、链接、标签，同时广播给关注其动态的用户，可以进行阅读、收藏、回复、转发等操作。该应用同时包括机器学习插件，用以搜索、推荐、统计。后端采用 Memcached 进行缓存，MongoDB 持久化存储用户状态和行为数据。

4.3.2 故障注入

如表 4 所列，本文根据文献[28]研究得出的分布式软件典型故障，采用文献[29]提出的故障注入常用方法。在 Social-Network 中，随机选择微服务，每种故障注入操作分别对应于 3 个故障，每个故障重复注入 3 次，这样共计搜集 108 个故障实例来验证方法的有效性。每次实验持续 10 min，其中，0～5 min 未注入错误，应用处于正常运行状态；第 6 min 注入故障，应用处于故障运行状态。每次实验，负载的并发数量以 150 为均值，以 50 为标准差随机变化，搜集运行过程中的监测数据。

表 4 故障注入列表

Table 4 List of injected faults

故障类别	故障注入操作
物理资源竞争	CPU 消耗，在请求处理代码中注入额外的指数计算操作，分别占用 30%，60% 和 90% 的 CPU 周期
	网络拥塞，使用 Linux TC 命令引起 15%，45% 和 75% 的网络数据包丢失
	内存泄漏，在每次请求处理时，随机分配额外 10 KB～1 MB 的对象，并引用静态变量
	IO 干扰，在相同主机的多个 Docker 容器中调用 StressLinux 工具执行写磁盘操
数据库	随机删除表格索引，以减慢记录查找的操作时间
	挂起部署数据库实例的 Docker 容器，引起访问数据库服务的操作失效
	每个连接锁定一个表，并尝试锁定其他的表。当一个连接对其他休眠表执行“INSERT DELAYED”操作，将会出现死锁
	截断一个表导致表插入的速度变慢
服务及应用	删除异常处理代码，造成触发异常时不能够正确处理
	挂起某些部署微服务的 Docker 容器，造成请求失败
	设置错误的微服务访问路径，造成服务访问失效
	设定错误的用户认证策略，造成应用出现非法访问

4.3.3 故障诊断

本文采用准确率、召回率和准确值来评价方法的有效性。为了验证本文方法的先进性，将本文方法与 Roots[14] 和 Sieve[17] 进行对比实验，结果如表 5 所列。

表 5 方法对比实验结果

Table 5 Comparison of experimental results

方法	准确率/%	召回率/%	响应时间/s
本文方法	86	91	1.47
Sieve	72	78	1.21
Roots	62	83	2.14

1) DeathStarBench. https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork

在准确率和召回率方面，Roots 关注于微服务性能变化，虽然实验表明大多数故障会表现出微服务性能的改变，如响应时间、响应信息，但部分故障不能立即表现出性能变化，如资源竞争未达到临界值而显著影响性能，因此召回率为 83%。同时，本文实验中模拟负载的并发量随机波动，从而导致将性能变化错误检测为故障，因此准确率为 62%。Sieve 需要先验知识以区别正常和异常的监测数据，由于相对于正常时的训练数据，故障时的训练数据较少；同时，该方法仅考虑到了度量间的两两相关性，但实验表明度量变化是受负载类型变化影响的，因此，Sieve 难以达到有较好的效果，其准确率为 72%，召回率为 78%。

在性能开销方面，Roots 由于采用从物理资源到服务执行轨迹的全栈监测，处理开销较大，请求处理的响应时间为 2.14 s；Sieve 仅调用操作系统接口采集物理资源信息，处理开销较小，请求处理的响应时间为 1.21 s；本文方法采集物理资源信息，而在应用层仅采集服务交互信息，因此处理开销介于 Sieve 和 Roots 之间，请求处理的响应时间为 1.47 s。

结束语　本文针对当前微服务监测及故障诊断所面临的挑战，提出了一种面向异常传播的微服务故障诊断方法并进行了实验验证。首先，基于服务度量构建 Lasso 回归模型以刻画服务的运行状态，检测异常度量及服务。同时，基于代理机制实现微服务调用的透明监测，构建服务调用拓扑，以分析微服务间的异常传播，进而提出了一种基于 PageRank 的故障服务定位方法，以量化各微服务的异常影响，定位引发异常的故障服务。最后，基于典型微服务应用 Sock-Shop 和 Social-Network，注入典型故障，验证所提方法的有效性及开销。本文旨在检测异常并定位故障，但不能及时采取措施保障系统的可用性。进一步的工作将引入集群管理框架（如 Kubernetes）和微服务框架（如 SpringCloud）的容错机制，包括负载均衡、动态伸缩、熔断等，实现高效的故障处理，以保障微服务系统的性能与可靠性。

参 考 文 献

[1] THÖNES J. Microservices [J]. IEEE Software, 2015, 32(1): 116-127.

[2] ZHOU X, PENG X, XIE T, et al. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study[J]. IEEE Transactions on Software Engineering, 2021, 41(2): 243-260.

[3] RAJAGOPALAN S, JAMJOOM H. App-Bisect: Autonomous Healing for Microservice-Based Apps[C]// Usenix Conference on Hot Topics in Cloud Computing. USENIX Association, 2015: 1-14.

[4] HEORHIADI V, RAJAGOPALAN S, JAMJOOM H, et al. Gremlin: Systematic Resilience Testing of Microservices[C]// IEEE 36th International Conference on Distributed Computing Systems (ICDCS). Nara: IEEE Press, 2016: 57-66.

[5] WANG T, ZHANG W B, XU J W, et al. A Survey of Fault Detection for Distributed Software Systems with Statistical Monitoring in Cloud Computing[J]. Chinese Journal of Computers, 2017, 40(2): 397-413.

[6] MI H B, WANG H M, ZHOU Y F, et al. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2013, 24(6): 1245-1255.

[7] WANG Z Y, WANG T, ZHANG W B, et al. Fault Diagnosis for Microservices with Execution Trace Monitoring[J]. Journal of Software, 2017, 28(6): 1435-1454.

[8] SHARMA B, JAYACHANDRAN P, VERMA A, et al. CloudPD: Problem determination and diagnosis in shared dynamic clouds[C]// IEEE/IFIP International Conference on Dependable Systems & Networks. Budapest: IEEE Computer Society Press, 2013: 1-12.

[9] CHEN W X, XU H W, LI Z, et al. Unsupervised Anomaly Detection for Intricate KPIs via Adversarial Training of VAE [C]// IEEE Conference on Computer Communications. Paris: IEEE Press, 2019: 2641-9874.

[10] CHERKASOVA L, OZONAT K, MI N, et al. Automated anomaly detection and performance modeling of enterprise applications[J]. ACM Transactions on Computer Systems, 2009, 27(3): 1-32.

[11] WANG T, WEI J, ZHANG W B, et al. Workload-Aware Anomaly Detection for Web Applications[J]. Journal of Systems and Software, 2014, 89(3): 19-32.

[12] WANG T, ZHANG W B, YE C Y, et al. FD4C: Automatic Fault Diagnosis Framework for Web Applications in Cloud Computing [J]. IEEE Transactions on Systems, Man and Cybernetics: Systems, 2016, 46(1): 61-75.

[13] WANG T, ZHANG W B, XU J W, et al. Adaptive Monitoring Based Fault Detection for Cloud Computing Systems[J]. Chinese Journal of Computers, 2018, 41(6): 1332-1345.

[14] JAYATHILAKA H, KRINTZ C, WOLSKI R: Performance monitoring and root cause analysis for cloud-hosted web applications[C]// Proceedings of the 26th International Conference on World Wide Web. New York, 2017: 469-478.

[15] CHEN P, QI Y, ZHENG P, et al. Causeinfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems[C]// IEEE Conference on Computer Communications. Toronto: IEEE Press, 2014: 1887-1895.

[16] LIN J, CHEN P, ZHENG Z. Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments[C]// International Conference on Service-Oriented Computing. Cham: Springer, 2018: 3-20.

[17] THALHEIM J, RODRIGUES A, AKKUS I E, et al. Sieve: actionable insights from monitored metrics in distributed systems [C]// Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference. New York: Association for Computing Machinery, 2017: 14-27.

[18] MARIANI L, MONNI C, PEZZE M, et al. Localizing Faults in Cloud Systems[C]// IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). Västerås: IEEE Press, 2018: 262-273.

[19] NIE X, ZHAO Y, SUI K, et al. Mining causality graph for automatic web-based service diagnosis[C]// IEEE 35th International Performance Computing and Communications Conference (IPC-

CC). Las Vegas: IEEE Press, 2016: 1-8.
[20] WANG T, WEI J, QIN F, et al. Detecting Performance Anomaly with Correlation Analysis for Internetware[J]. Science China Information Sciences, 2013, 56(8): 082104(15).
[21] WANG T, CHEN W, LI J, et al. Association Mining Based Consistent Service Configuration[J]. Journal of Computer Research and Development, 2020, 57(1): 188-201.
[22] HASTIE T, TIBSHIRANI R, FRIEDMAN J. The Elements of Statistical Learning[M]. New York: Springer, 2009: 1-745.
[23] Cambridge University. Network Science [EB/OL]. https://www.sci.unich.it/~francesc/teaching/network/.
[24] LANGVILLE A N, MEYER C D, HENDLER J. Google's PageRank and Beyond: The Science of Search Engine Rankings [M]. Princeton University Press, 2011.
[25] JIANG M, MUNAWAR M A, REIDEMEISTER T, et al. Efficient fault detection and diagnosis in complex applications with information theoretic monitoring[J]. IEEE Transactions on Dependable and Secure Computing, 2011, 8(4): 510-522.
[26] DEAN D J, NGUYEN H, WANG P, et al. A. Sailer and A. Kochut, PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds[J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(6): 1742-1755.
[27] GAN Y, ZHANG Y Q, CHENG D L, et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems[C]// Proceeding of 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). NewYork: Association for Computing Machinery, 2019: 3-18.
[28] ZHOU J, CHEN Z, WANG J, et al. A Data Set for User Request Trace-Oriented Monitoring and its Applications [J]. IEEE Transactions on Services Computing, 2018, 11(4): 699-712.
[29] PHAM C, WANG L, TAK B C, et al. Failure Diagnosis for Distributed Systems using Targeted Fault Injection [J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 28(2): 503-516.

WANG Tao, born in 1982, Ph.D, associate professor, master supervisor, is a senior member of China Computer Federation. His main research interests include fault diagnosis, software reliability, and microservices.

ZHANG Shu-dong, born in 1969, Ph.D, professor, Ph.D supervisor, is a senior member of China Computer Federation. His main research interests include distributed computing and microservices.