软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2022,33(5):1849−1864 [doi: 10.13328/j.cnki.jos.006209]

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

基于调用链控制流分析的大型微服务系统性能建模与异常定位*

于庆洋[1,2], 白晓颖[3], 李明杰[1,2], 李奇原[4], 刘 涛[4], 刘泽胤[4], 裴 丹[1,2]

[1](清华大学 计算机科学与技术系, 北京 100084)

[2](北京信息科学与技术国家研究中心, 北京 100084)

[3](北京大数据先进技术研究院, 北京 100083)

[4](百度 商业平台研发部, 北京 100193)

通信作者: 白晓颖, E-mail: baixy@aibd.ac.cn

摘 要: 大型微服务系统中组件众多、依赖关系复杂, 由于故障传播的涟漪效应, 一个故障可能引起大规模服务异常, 快速识别异常并定位根因是服务质量保证的关键. 目前主要采用的调用链分析方法, 常常面临调用链结构复杂、实例数量庞大、存在大量小样本等问题, 因此提出基于调用链控制流分析, 将大量调用链结构聚合为少量方法调用模型; 并提出基于方法调用模型的执行时间分解模型及预测方法, 将实际值与预测值的相对误差超过设定阈值的待检测数据判定为异常. 采用百度凤巢广告业务系统某天超过 17 亿条调用链日志记录开展实验分析, 结果表明: 与数据驱动的调用序列分析方法相比, 提出的基于模型的方法可以大幅缩减调用链结构数量, 并有效分析和检测微服务性能异常及其根因.

关键词: 微服务系统; 性能异常检测; 根因分析; 调用链; 控制流分析

中图法分类号: TP311

中文引用格式: 于庆洋, 白晓颖, 李明杰, 李奇原, 刘涛, 刘泽胤, 裴丹. 基于调用链控制流分析的大型微服务系统性能建模与异常定位. 软件学报, 2022, 33(5): 1849–1864. http://www.jos.org.cn/1000-9825/6209.htm

英文引用格式: Yu QY, Bai XY, Li MJ, Li QY, Liu T, Liu ZY, Pei D. Performance Modeling and Anomaly Location of Large Microservice Systems Based on Trace Control Flow Analysis. Ruan Jian Xue Bao/Journal of Software, 2022, 33(5): 1849–1864 (in Chinese). http://www.jos.org.cn/1000-9825/6209.htm

Performance Modeling and Anomaly Location of Large Microservice Systems Based on Trace Control Flow Analysis

YU Qing-Yang[1,2], BAI Xiao-Ying[3], LI Ming-Jie[1,2], LI Qi-Yuan[4], LIU Tao[4], LIU Ze-Yin[4], PEI Dan[1,2]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](Beijing National Research Center for Information Science and Technology, Beijing 100084, China)

[3](Advanced Institute of Big Data, Beijing 100083, China)

[4](Department of Commercial Platform, Baidu Inc., Beijing 100193, China)

Abstract: In a large microservice system, there usually exist many services with complex dependencies among them. A failure in one component may propagate widely and cause large-scale service anomalies. To ensure system quality, it is critical to effectively identify abnormalities and locate root causes. Invocation-chain analysis is a commonly used method for service performance modeling and anomaly detection. Existing techniques are mostly data-driven, facing many challenges of big data analysis such as diversified chain structures, a vast number of instances, and imbalanced datasets that many structures have only a small number of samples. In counter to the problems, the study proposes a model-based approach which builds high-level abstractions of method invocation models based on control-flow

* 基金项目: 国家重点研发计划 (2019YFB1802504, 2019YFE0105500); 国家自然科学基金 (62072264)
收稿时间: 2020-03-30; 修改时间: 2020-06-11, 2020-09-21; 采用时间: 2020-11-18

analysis. The instances of various invocation-chain structures are clustered into various method invocation models, which can greatly reduce the size of chain structures. Performance models are built for the method invocation models, and thresholds are defined based on the predicted execution time derived from the performance model. Outliers in the trace logs are thus identified as candidates of anomalies. Experiments were exercised on real industry logs from Baidu PhoenixNest Ads system. A one-day log with over 1.7 billion records was selected. The experiment results show that, compared with pure data-driven sequence analysis methods, the proposed model-based approach can greatly reduce the size of invocation-chain structures while effectively analyzing and detecting microservice performance anomalies and root causes.

Key words: microservice system; performance anomaly detection; root cause analysis; invocation chain; control flow analysis

微服务架构采用功能相对单一、可独立开发部署的多服务组件的架构模式, 构建复杂的软件系统, 以提高系统的可扩展性和可维护性, 近年来得到越来越广泛地关注与应用[1,2]. 然而, 当系统功能复杂、微服务数量众多时, 服务之间常常存在着复杂的依赖关系; 当某一服务发生故障时, 可能产生涟漪效应, 大范围地影响系统运行质量, 导致系统性能下降甚至功能失效[3]. 因此, 在系统运行过程中, 如何及时、准确地检测到异常服务, 成为软件质量保障的一个关注热点.

调用链日志分析是一种常用的异常检测方法. 日志中通常记录了服务的调用与执行过程, 包括服务标识、调用关系、执行时间等信息, 通过信息提取、标注和分析, 为服务的功能和性能异常检测提供支持[4−6]. 近年来, 除了传统的统计方法, 机器学习、深度学习等新技术也被广泛地应用于性能分析和异常检测[7,8]. 现有调用链分析方法主要是针对调用链中的服务组件进行性能建模, 以识别性能异常的调用链及节点. 然而在大型微服务系统中, 同一服务组件内部的不同方法可能具有不同的执行特征, 仅从服务组件级难以准确分析性能特征, 因而检测不准确. 另一方面, 复杂的调用逻辑导致不同调用链结构非常多, 例如百度凤巢广告业务系统核心模块某天入库的日志记录共 17 亿多条, 有超过 200 万种方法级调用链结构. 在故障定位中, 调用链数量大、种类多对存储和分析带来诸多困难, 包括 (1) 需要存储大量的调用链信息, 影响执行效率; (2) 某些调用链结构因实例数太少, 难以有效建模; (3) 训练数据集难以覆盖所有调用链结构, 因而无法有效检测未覆盖的正常结构.

本文旨在通过分析服务方法间逻辑调用的特点, 一方面, 从更细粒度 (方法级) 进行性能建模和异常检测, 以提高检测的准确性; 另一方面, 基于控制流, 将大量调用链结构抽象为少量方法调用模型, 以提高数据存储和分析效率. 本文基于百度凤巢广告业务系统某天超过 17 亿条调用链日志记录开展实验分析. 结果表明: (1) 本文方法可以大幅缩减调用链结构数, 例如将 200 多万种调用链结构缩减为 6 000 多种方法调用模型; (2) 可以有效建模和检测各类方法性能, 实验中本文方法的性能异常检测 $F1$-score 值超过 0.95, 高于其他基线方法 0.25 以上; (3) 可以识别请求执行过程中引发性能异常的根因.

本文第 1 节介绍研究背景并分析相关工作, 第 2 节提出基于控制流的调用链构建方法并分析调用模式, 第 3 节提出基于控制流的性能异常定位方法, 第 4 节介绍实验及结果分析, 第 5 节总结并提出下一步研究方向.

1 背景及相关工作

微服务架构凭借其开发部署独立、功能扩展灵活、实例动态增减等优势[9], 在互联网领域得到广泛应用. 例如, Netflix 从单体架构演变到微服务架构, 以约 500 个微服务处理日均超过 20 亿应用请求[10]. 腾讯微信系统包含 3 000 多个微服务, 运行在 2 万多台虚拟机中[11]. 百度凤巢广告业务系统包含约 2 000 个微服务, 运行实例超过 2 万个, 每天处理百亿级应用请求. 大型微服务系统中微服务数量众多、实例动态增减、逻辑调用复杂, 当某一服务发生故障时, 其关联的服务也会受到影响, 从而导致大规模服务异常, 不易识别根因, 这对现有异常检测方法提出挑战.

1.1 现有异常检测方法

异常检测在大型分布式系统的运维方面发挥着重要作用, 而系统运行日志则被广泛应用于异常检测. 在基于系统日志的异常检测研究中, 使用的方法包括各类统计方法和机器学习方法等[7,8]. 文献 [12] 重点介绍并评估了 6 种常用日志异常检测方法, 包括 3 种监督学习方法 (逻辑回归、决策树、支持向量机) 和 3 种非监督学习方法 (日志聚类、主成分分析、不变量挖掘), 为研究和开发人员提供经验.

目前常用的异常检测日志主要包括关键性能指标 (KPI) 日志和调用链日志两大类. 基于 KPI 的异常检测研究, 通过检测系统中节点 (物理机、应用等) 的关键性能指标, 例如物理机的 CPU、内存、网络、磁盘 I/O 和应用的请求数、平均响应时间等, 来发现异常. 常用的方法包括滑动自回归 (ARIMA) 模型[13,14]、霍尔特-温特 (Holt-Winters) 模型[15,16]、时间序列分解 (TSD) 模型[17]、主成分分析 (PCA) 技术[18]、基于随机森林模型的异常检测 Opprentice[19]等. 由于 KPI 日志记录的是节点的平均性能指标, 没有从更细粒度分析, 例如不同请求对服务执行时间的影响不同, 因而分析和检测不够准确, 此外, 仅基于 KPI 日志进行分析, 难以识别不同服务间的关联, 因而难以有效分析微服务系统大规模异常下的根因.

由于 KPI 日志的局限性, 很多研究借助系统中服务组件的调用关系, 来更好地检测和定位系统异常. Sieve[20]首先通过自动过滤无关指标对系统中大量指标进行降维, 之后通过对网络连接信息的分析来构建服务组件调用有向图, 然后基于格兰杰因果关系检验 (Granger causality test), 分析不同组件中指标的关联, 识别其中的关键指标, 从而有效降低异常分析的复杂度. Microscope[21]通过捕获服务组件的定向连接信息来构建其依赖关系, 并且基于资源共享等因素构建非通信服务组件的依赖关系, 进而完成服务因果关系图的构建, 能够实时推断异常发生时的根因. 但这些研究主要着眼于服务组件间的调用关系, 并没有深入考虑各服务组件内部的不同方法, 分析粒度较粗, 且分析的指标依然是平均指标, 没有区分不同请求对服务性能的影响, 因而不够精准.

近年来, 随着请求跟踪 (tracing) 技术[4−6]的发展, 调用链日志成为分布式系统日志的重要组成部分. 调用链日志中蕴含着丰富的信息, 包括每次请求执行调用的各服务组件及其执行信息等, 为大型微服务等分布式系统的异常检测和根因定位提供支持. 随着深度学习的发展, 其在调用链日志异常检测研究中被广泛使用, 尤其是长短期记忆网络 (LSTM)[22]. Seer[23]利用深度学习方法分析调用链数据, 首先基于卷积神经网络 (CNN) 对数据降维并过滤掉不影响端到端性能的微服务组件, 再基于 LSTM 学习空间和时间模式, 以有效分析和预测服务质量 (QoS), 从而避免引起更大的故障. Multimodal LSTM[24]可以同时学习和检测调用链路径和服务性能两种模式, 在路径建模和检测方面, 学习基于不同前序执行序列的概率转移关系, 将转移概率较低的路径定义为异常; 在性能建模和检测方面, 学习并预测执行序列中各服务的执行时间, 计算实际值与预测值之差的平方值, 将处在高斯分布 95% 置信区间以外的值定义为异常. 此外, 文献 [25] 专注于调用链中服务的性能异常检测, 使用一种结合变分自编码器 (VAE) 和门控循环单元 (GRU) 的深度学习方法, 基于时序数据来学习服务的执行时间分布, 将概率低于设定阈值的值判定为异常. 深度学习方法在大规模运行日志实时分析中, 常常面临着严重的性能问题, 例如文献 [25] 中方法要为每个服务组件训练一个深度模型, 对于有几百上千个微服务的大型微服务系统来说, 开销太大. 文献 [26] 基于主成分分析技术识别引起微系统服务性能波动的关键方法, 但这种方法对数据要求较高, 需要每种调用链结构下的实例数达到一定规模, 才能有效进行性能异常检测, 而本文实验中某实例有 264 个性能异常, 其中 177 个异常所在调用链从未出现过, 因此这种方法难以应对大型微服务系统中复杂的调用模式.

综上所述, 目前大部分基于调用链的性能异常检测方法, 都是基于请求执行调用链对其中服务进行性能建模和异常检测. 在实际应用中, 方法的有效性严重受制于日志结构的复杂性、数据规模以及数据质量等. 以百度凤巢系统为例, 微服务数量众多, 有约 2 000 个微服务; 调用链复杂多样, 单日日志中有超过 200 万种调用链结构; 某些调用链结构下实例数很少, 难以有效建模. 以上原因导致现有性能异常检测方法难以奏效.

1.2 调用链研究粒度分析

目前, 基于调用链日志的研究粒度主要包括服务组件级[20−21,23−25]、接口级[3]和方法级[26]. 服务组件级日志记录了请求执行过程中调用的服务组件及其执行情况, 但没有区分服务组件内部的不同接口或方法的执行过程, 因此仅能分析和定位到服务粒度; 接口级日志记录了服务组件不同接口的调用与执行情况, 可以定位到服务的接口操作; 方法级日志粒度更细, 它记录了请求执行过程中相关组件内外部方法的调用与执行信息, 可以定位到具体的方法执行过程, 包括接口内部的方法调用等.

目前很多研究都仅着眼于服务组件级别, 没有明确区分服务组件内部的方法执行过程. 一方面, 可能受限于日志记录粒度, 由于会造成额外性能开销以及存在工程实现难度等原因, 仅记录了服务组件级日志; 另一方面, 可能受限于所研究系统的规模和复杂度, 一个微服务内可能只有一个简单的功能接口, 因而没有加以区分. 然而在面对

大型微服务系统时, 若不考虑服务组件内部的不同方法执行过程, 会导致分析粒度过大, 检测结果不准确, 主要表现在以下几个方面.

(1) 同一微服务中不同方法的执行时间可能有差距.

例如, 在本文研究的百度凤巢日志中有微服务 C 和 D, M_{C1} 是微服务 C 的一个方法, M_{D1} 和 M_{D2} 是微服务 D 的两个方法. 存在 $M_{C1}\rightarrow M_{D1}$ 和 $M_{C1}\rightarrow M_{D2}$ 两种调用过程, 其中 M_{D1} 和 M_{D2} 执行时间有明显差距, 平均值分别为 32.99 ms 和 5.94 ms.

(2) 同一微服务中不同方法调用相同方法时, 被调用方法的执行时间可能有差距.

例如, 在本文研究的百度凤巢日志中有微服务 E 和 F, M_{E1} 和 M_{E2} 是微服务 E 的两个方法, M_{F1} 是微服务 F 的一个方法, 存在 $M_{E1}\rightarrow M_{F1}$ 和 $M_{E2}\rightarrow M_{F1}$ 两种调用过程, 两者 M_{F1} 的执行时间差距较大, 平均值分别为 25.68 ms 和 98.38 ms.

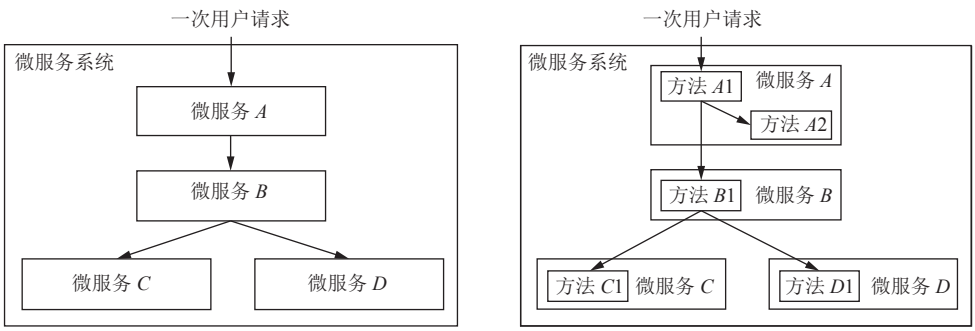
针对上述例子, 若从服务组件级调用链进行分析, 会将明显不同的方法调用链看成相同的, 从而影响分析和检测效果. 一般来说, 方法级调用链比服务组件级和接口级调用链粒度更细, 用于检测和定位更准确. 因此我们从更细粒度的方法级调用链日志开展分析和研究, 以达到更好的检测效果.

2 调用链构建与调用模式分析

2.1 调用链构建与表示

调用链描述一次请求执行过程中若干服务组件的执行和调用信息. 现有的一种调用链表示法是将一次请求执行中调用的服务或方法依据时间顺序构建为执行序列[24]. 然而执行序列难以描述分布式系统服务的复杂逻辑调用关系, 因此树结构[4,26]被用于描述调用链.

一个微服务系统由多个微服务组件构成, 一个微服务组件内部通常有一个至多个不同方法. 一次请求执行时某个方法可能调用多个 (组件内外部) 方法, 导致调用链产生多个分支, 图 1 展示了微服务系统中一次请求执行的不同粒度调用链示例.



(a) 服务组件级调用链　　(b) 方法级调用链

图 1　不同粒度调用链分析示例

调用链日志记录的结构类似, 以百度凤巢广告业务系统日志为例, 主要字段如表 1 所示.

为了更好地描述不同微服务间的逻辑调用关系, 本文以树结构来描述方法级调用链. 一次请求执行调用链的构建需要确定以下 3 个因素.

(1) 日志与请求的对应关系

TraceID 唯一标识一次请求执行的调用链, 该字段相同的日志属于同一调用链. 例如, 图 1(b) 中用户请求调用的各方法记录具有相同的 TraceID.

(2) 方法间的调用关系

ParentID 和 SpanID 可以确定调用关系:

1) ParentID 为空表示调用链起始方法. 例如, 图 1(b) 中方法 $A1$ 记录的 ParentID 为空;

2) 某方法记录的 ParentID 与其父方法记录的 SpanID 相同. 例如, 图 1(b) 中 $B1$ 记录的 ParentID 与 $A1$ 记录的 SpanID 相同, 表示 $A1$ 方法调用了 $B1$ 方法.

表 1 日志的主要字段

字段名称	描述
SpanID	唯一标识一次方法的执行日志
TraceID	唯一标识一次请求执行的调用链
MethodID	日志中方法的标识
AppID	方法所在应用的标识, 可以映射到微服务
ParentID	父方法的SpanID
Start	方法执行的起始时间点
End	方法执行的结束时间点
Cost	方法的执行时间

(3) 方法执行顺序

Start 和 End 可以确定方法间执行顺序. 例如, 图 1(b) 中 $C1$ 记录的 Start 值小于 $D1$ 记录中 Start 值, 表示 $C1$ 执行早于 $D1$. 本文假设分布式系统中, 不同机器之间已经进行了时间同步或是时间偏差不影响调用顺序分析.

本文将调用链映射为树结构, 发起请求的方法称为父方法, 映射为调用树中的父节点; 被调用方法称为子方法, 映射为树中的子节点; 多个子方法按照被调用顺序, 映射为树中的兄弟节点. 采用树的嵌套括号表示法来描述调用链结构. M_i 表示方法, M_i^n 表示方法 M_i 被重复调用了 n 次, n=1 时省略上标, $M_1(M_2)$ 表示方法 M_1 调用方法 M_2, $M_1(M_2,\ldots,M_n)$ 标识 M_1 的调用过程, 括号内为依据调用顺序排列的方法集合.

2.2 微服务调用模式

一次请求执行往往依赖一个或多个微服务中的方法协同完成, 因此理解微服务系统中方法的调用模式对其性能分析至关重要. 本文将微服务系统中方法调用模式归为 4 类.

(1) 无调用

方法的功能实现不依赖其他方法.

(2) 单分支调用模式

方法执行时固定调用某方法一次.

(3) 多分支调用模式

在微服务系统中, 一个方法的功能实现可能依赖多个方法协同完成, 在调用链中呈现为多分支的调用模式. 图 2 展示了以 M_0 为父方法的多分支调用模式, M_1,M_2,…,M_n 表示其调用的子方法, 依据调用顺序从左至右排列.

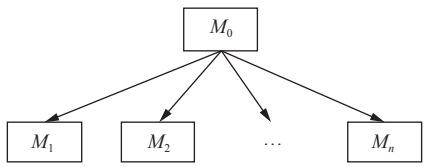


图 2 多分支调用模式

(4) 方法重复调用模式

在某些大型微服务系统中, 用户请求执行可能涉及大量且频繁的数据查询操作. 为避免单次请求因查询信息过多而超时, 将大量信息分批次查询是一种有效解决方案, 这在调用链中体现为方法的重复调用.

图 3 展示了以 M_0 为父方法、M_1 为子方法的重复调用模式. 例如, 凤巢广告业务系统中, M_0 负责某种物料信

息的查询, 一次用户请求可能涉及很多关键词, 系统分批次查询, 呈现为子方法 M_1 的大量重复调用, 在本文的实验数据日志中, n 最多达到数百次.

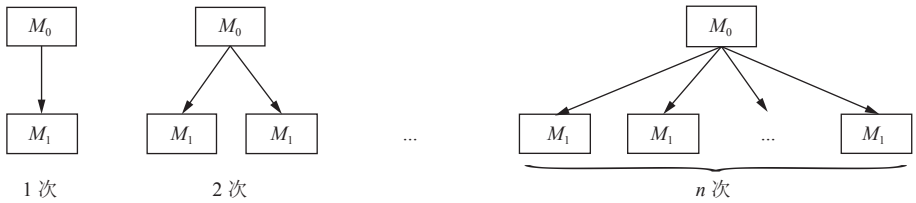


图 3 单种方法的重复调用模式

实际中, 常常存在多个方法多种组合的重复调用模式, 如图 4(a) 所示. M_0 负责多种物料信息的查询, 一次用户请求可能调用一种或几种查询方法, 这些方法调用没有固定的逻辑关系, 也无固定的执行次数, 如果将各种可能的调用序列展开, 可能对应大量不同的调用链结构, 如图 4(b)(c)(d). 在本文的实验数据中某方法对应的调用链结构达到 55 万多种, 多种方法的重复调用模式是导致大型微服务系统中调用链结构复杂多样的主要原因之一.

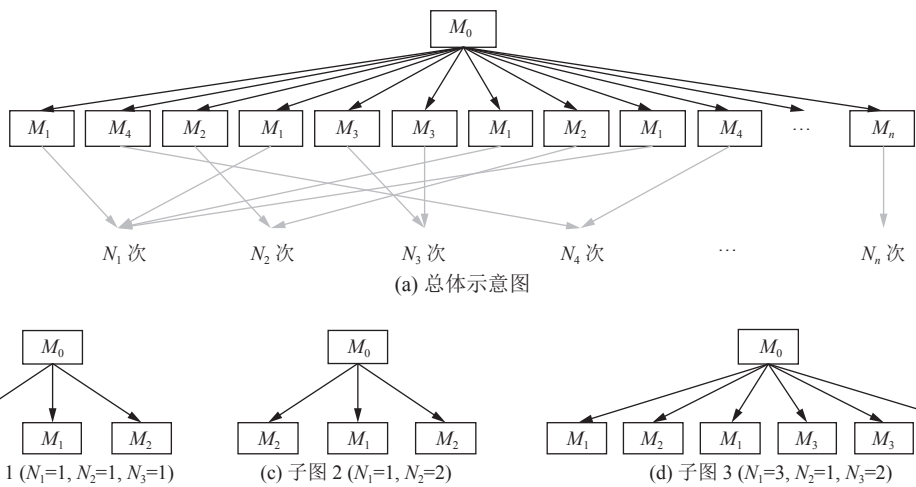


图 4 多种方法的重复调用模式

3 基于控制流的性能异常方法定位

微服务系统的调用链是不同调用模式的组合, 对每种调用链结构进行建模和检测势必耗费大量资源. 为缩减调用链的数量、提高分析效率, 本文提出一种基于方法控制流的调用链抽象方法, 将调用链实例按照其结构进行抽象和聚类, 每一类代表一种方法的调用模型, 从而将大量调用链结构抽象成少量方法调用模型, 在保留方法执行的业务场景和性能特征的同时, 可以有效降低建模和分析的复杂度.

3.1 方法调用模型抽象及调用模式识别

执行过程相同的调用链中方法的执行时间具有较高的一致性[27], 因此目前很多研究都是基于调用链结构对其中方法进行性能建模和检测, 然而面对微服务系统复杂的调用链结构难以奏效. 本文以调用链中的方法为单位进行方法调用模型抽象, 将具有相同前序调用路径和子方法的方法看作同一种方法调用模型, 每种子方法的调用次数则不必相同, 其中前序调用路径可以描述方法的业务场景, 子方法信息可以反映其实际执行情况. 对于方法 M, 其方法调用模型可以描述为三元组:

$$P_M = (C, N, \{M_1, M_2, \ldots, M_n\}) \tag{1}$$

表示方法 M 在应用场景 C 下有 N 种不同的子方法, 其中, C 为所在调用树从根节点到 M 的路径, $\{M_1, M_2, \ldots,$

$M_N\}$为 M 的不同子方法的集合, 调用 $M_i(i=1,2,\ldots,N)$ 有 $m_i>0$ 次.

方法调用模型抽象的示意图如图 5 所示.

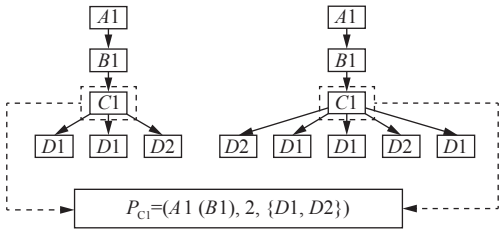


图 5 方法调用模型抽象示意图

图 5 中有两条结构不同的调用链, 以本文调用链表示法可依次描述为 $A1(B1(C1(D1^2,D2)))$ 和 $A1(B1(C1(D2,D1^2,D2,D1)))$. 以其中 $C1$ 方法为例, 虽然它们处在不同调用链中, 有不同的子方法执行序列, 但调用的子方法种类相同, 属于同一种方法调用模型, 其方法调用模型都可抽象为 $P_{C1}=(A1(B1), 2, \{D1,D2\})$. 通过方法调用模型抽象, 可以将大量不同调用链抽象为少量方法调用模型, 从而大大降低后续建模及检测的复杂度.

具有不同调用模式的方法往往具有不同的参数特征, 本文基于方法调用模型 P_M 的实例数据集识别其调用模式, 以便后续采用针对性的参数计算方法. 对调用模式的识别方法如下.

(1) 无调用. 此时 $N=0$.

(2) 单分支调用. 此时 $N=1$ 且各实例中方法调用次数均为 1.

(3) 多分支调用. 此时 $N>1$ 且各实例中各方法调用次数均为 1.

(4) 方法重复调用. 此时 $N\geqslant 1$ 且实例中存在调用次数大于 1 的方法.

3.2 同步调用模式下方法的执行时间分解

对于一个同步执行的方法, 其总执行时间包括不同部分的时间组合. 在微服务的方法调用中, 由于网络延迟等额外开销, 调用过程的开销相较子方法的执行时间不可忽略. 对于一个方法 M 及其调用的子方法 M_i ($i=1,2,\ldots,n$), 定义 $t(M)$ 为方法 M 的总执行时间, $t(M_i)$ 为子方法 M_i 的总执行时间, $t(M,M_i)$ 为方法 M 调用 M_i 引起的额外开销, 可能包括调用过程的网络延时、排队延时、M 调用 M_i 的代码段执行开销等, $t_0(M)$ 为 M 除调用开销外的基础执行时间, 有:

$$t(M)=t_0(M)+\sum_{i=1}^{n}t(M_i)+\sum_{i=1}^{n}t(M,M_i) \tag{2}$$

3.3 性能异常检测

3.3.1 整体思路

微服务中广泛使用日志记录执行信息. 若单独记录每次调用的额外开销, 一方面会引入大量的性能损耗, 另一方面由于不同节点的机器时钟难以完全同步[28], 使得记录的准确性难以保证, 因而直接测量每次调用的额外开销难以实现.

基于公式 (1) 方法调用模型, 本文定义方法 M 调用同一种方法 M_i 引起的额外开销服从同一分布 $D(M,Mi)$. 则公式 (2) 整理为公式 (3), 其中, $M_i^{(j)}$ 表示对方法 M_i 的第 j 次调用.

$$t(M)=t_0(M)+\sum_{i=1}^{N}\sum_{j=1}^{m_i}t(M_i^{(j)})+\sum_{i=1}^{N}\sum_{j=1}^{m_i}t(M,M_i^{(j)}) \tag{3}$$

在同步调用模式下, 父方法执行需等待子方法执行结束, 其执行时间与子方法执行时间相关. 特别是当某个子方法性能异常时, 父方法执行时间很可能异常, 因而分析和检测父方法的总执行时间难以有效识别根因. 据此本文对 (父) 方法总执行时间中除子方法执行外的其他开销进行建模和检测, 以 ξ 代表这个值, 基于公式 (3), 有:

$$\xi = t(M) - \sum_{i=1}^{N}\sum_{j=1}^{m_i} t(M_i^{(j)}) = t_0(M) + \sum_{i=1}^{N}\sum_{j=1}^{m_i} t(M, M_i^{(j)}) \tag{4}$$

本文以常数 $T(M,M_i)$ 近似 M 调用 M_i 的额外开销 $t(M,M_i)$, 以常数 $T_0(M)$ 近似除调用开销外的基础执行时间 $t_0(M)$, 则在特定方法调用模型 P_M 下由公式 (4) 可得公式 (5):

$$\xi = t(M) - \sum_{i=1}^{N}\sum_{j=1}^{m_i} t(M_i^{(j)}) = T_0(M) + \sum_{i=1}^{N} (m_i \times T(M, M_i)) \tag{5}$$

本文将实际执行时间 ξ 与基于公式 (5) 参数拟合的预测值 ξ'进行比较, 将相对误差大于阈值 ε 的方法判定为异常, 如公式 (6).

$$\frac{|\xi - \xi'|}{\xi'} > \varepsilon \tag{6}$$

3.3.2 方法重复调用模式参数计算

由于多种方法重复调用时存在多个 (3 个及以上) 参数, 本文实验发现基于最小二乘法同时拟合多个参数效果不理想, 因此本文选取仅重复调用一种方法的实例进行参数拟合. 由式 (5) 可得仅调用一种方法 M_i 时:

$$\xi = t(M) - \sum_{j=1}^{m_i} t(M_i^{(j)}) = T_0(M) + m_i \times T(M, M_i) \tag{7}$$

针对同一种方法调用模型的每个实例, ξ 的值可由已知数据 $t(M)$ 和 $t(M_i^{(j)})$ 求得, 以 b 表示 $T_0(M)$, c 表示 $T(M,M_i)$, ξ_k 表示该方法调用模型下第 k 个实例的 ξ 值, m_k 为第 k 个实例调用方法 M_i 的次数, 有:

$$\xi_k = b + m_k \times c \tag{8}$$

基于最小二乘法对 n 个实例数据进行拟合, 即求满足公式 (9) 函数取最小值时 b 和 c 的值:

$$f = \sum_{k=1}^{n} (b + m_k c - \xi_k)^2 \tag{9}$$

最小二乘法的求解过程, 本文不做赘述. 然而该模式下最小二乘法并非一直适用, 存在特殊场景, 即只存在特定次数的重复调用实例, 这一特殊场景将在后续其他模式参数计算中讨论.

基于上述思路, 可以求得单独调用每种方法时的 $T_0(M)$ 和 $T(M,M_i)$ 的值, 将 $T(M,M_i)$ 作为多种方法重复调用下的参数, 带入公式 (5), 则在涉及多种子方法的方法调用模型 P_M 下有:

$$T_0(M) = t(M) - \sum_{i=1}^{N}\sum_{j=1}^{m_i} t(M_i^{(j)}) - \sum_{i=1}^{N} (m_i \times T(M, M_i)) \tag{10}$$

以各实例的 $T_0(M)$ 平均值作为其预测值, 基于公式 (5) 计算在特定方法调用模型 P_M 下 ξ 的预测值

$$\xi' = T_0(M) + \sum_{i=1}^{N} (m_i \times T(M, M_i)) \tag{11}$$

由于方法重复调用模式下, 同一方法调用模型的实例中子方法调用次数不固定, 基于不同实例计算的标准差等参数不具备一致性. 本文依据经验把方法重复调用模式下 ε 初始值设置为 2. 在实际应用中, 运维人员可以根据实际情况进行动态调整, 若异常检测结果超过实际情况, 则相应提高 ε 取值, 否则相应减小 ε 取值, 以达到更好的效果.

3.3.3 其他模式参数计算

针对无调用、单分支调用、多分支调用以及方法重复调用模式的特殊场景 (即只存在特定次数的重复调用实例), 本文依据式 (4) 中:

$$\xi = t(M) - \sum_{i=1}^{N}\sum_{j=1}^{m_i} t(M_i^{(j)}) \tag{12}$$

计算相关实例中 ξ 的均值$\bar{\xi}$, 并以均值作为其预测值, 即在特定方法调用模型 P_M 和调用次数序列$\{m_1,m_2,\ldots,m_N\}$下, 有 ξ 的预测值:

$$\xi' = \bar{\xi} \tag{13}$$

在这些场景下, 同一方法调用模型的实例具有较高的一致性, 可以基于不同实例计算均值$\bar{\xi}$和标准差σ, 进而基于3σ异常检测算法来设置初始ε, 取$\varepsilon = 3\sigma/\bar{\xi}$. 在实际使用中, 同样可以基于效果动态调整阈值.

4 实验及结果分析

4.1 数据集介绍

随着微服务技术的发展, 对微服务系统的异常检测研究成为热点, 为了解决当前微服务研究缺少基准测试环境的问题, 文献 [29] 开发了名为 TrainTicket 的微服务基准测试系统. 它能模拟多种异常, 并记录相应的调用链日志, 为研究领域提供了一套优秀的微服务基准测试系统. 文献 [3] 中调研采集了 22 个工业领域中的典型故障案例, 并在 TrainTicket 系统中进行复现和研究. 虽然这套基准测试系统能够模拟多种异常, 但其系统复杂度 (约 40 个微服务[30]) 还远远达不到凤巢 (约 2 000 个微服务) 等大型微服务系统的程度, 也难以模拟其用户量及实际使用情况, 因此对大型线上微服务系统调用链日志的研究具有重要意义.

凤巢广告业务系统与其他大型微服务系统相比有其特殊性, 由于系统涉及大量且频繁的广告类信息查询操作, 为避免单次请求超时而将关键词分批次查询是有效手段, 这也导致请求执行的调用链结构更加复杂, 因此对其调用链的异常检测更具研究价值.

本文实验日志来自百度凤巢广告业务系统, 该系统主要基于 Java 语言开发, 保留了“方法”(Method) 的定义, 其日志的记录粒度为方法级别, 可以记录每次请求执行时各相关方法的调用与执行信息, 记录粒度非常精细. 目前系统每日产生数百亿条日志记录, 为降低性能损耗对日志记录进行采样是必要的. Dapper 开发团队研究发现, 即使很低的采样率 (1/1000) 依然能够提供充足的数据支持, 且系统性能损耗跟采样率成正比[5]. 因此凤巢对日志记录进行采样, 每日入库的日志记录总量仍然有数十亿至上百亿条. 本文选取系统核心业务模块某天 (0 时至 24 时) 共 17 亿多条调用链日志开展研究.

4.2 调用链构建及方法调用模型抽象

本文使用第 2.1 节的方法构建方法级调用链, 由 17 亿多条调用链日志构建出 200 多万种调用链结构. 调用链起始方法通常代表一类业务场景, 本文首先基于业务场景对调用链进行初始归类, 将起始方法相同的调用链放到一个集合进行统计分析, 其中结构种数最多的 10 个集合如表 2 所示. 由于可能涉及业务敏感信息, 本文隐藏了方法的实际标识.

表 2 不同结构数量前 10 的调用链集合统计表

起始方法标识	集合中不同结构数量	引起结构数量增多的主要调用模式
*A*1	553 049	多种方法重复调用
*A*2	428 288	多种方法重复调用
*A*3	302 744	多种方法重复调用
*A*4	155 010	多种方法重复调用
*A*5	146 705	多种方法重复调用
*A*6	69 381	多种方法重复调用+多分支调用
*A*7	59 280	多种方法重复调用
*A*8	45 383	多种方法重复调用
*A*9	33 758	多种方法重复调用
*A*10	31 350	多种方法重复调用+多分支调用

然后使用第 3.1 节的方法进行方法调用模型抽象, 将 200 多万种调用链结构抽象为 6 000 多种方法调用模型, 从而大大降低分析和检测的复杂度. 本文以结构种数最多的实例, 即表 2 中以 *A*1 为起始方法的调用链集合为例, 分析方法调用模型抽象的实验结果, 可以将该实例的 55 万多种不同结构建模为 22 种方法调用模型, 如表 3 所示, 其中前序调用为 0 表示调用链起始方法.

表 3 方法调用模型信息表

调用模式	方法标识	方法调用模型	实例数(个)
无调用	$C1$	$(A1(B), 0, \{\})$	25 373 608
	$C2$	$(A1(B), 0, \{\})$	46 262 902
	$C3$	$(A1(B), 0, \{\})$	7 244 179
	$C4$	$(A1(B), 0, \{\})$	2 780 820
	B	$(A1, 0, \{\})$	2456
	$A1$	$(0, 0, \{\})$	102 326
方法重复调用	B	$(A1, 1, \{C1\})$	9 710
		$(A1, 1, \{C2\})$	37 999
		$(A1, 1, \{C3\})$	379
		$(A1, 1, \{C4\})$	1718
		$(A1, 2, \{C1, C2\})$	50 099
		$(A1, 2, \{C1, C3\})$	2 893
		$(A1, 2, \{C1, C4\})$	25 271
		$(A1, 2, \{C2, C3\})$	3055
		$(A1, 2, \{C2, C4\})$	114 080
		$(A1, 2, \{C3, C4\})$	0
		$(A1, 3, \{C1, C2, C3\})$	54 134
		$(A1, 3, \{C1, C2, C4\})$	150 314
		$(A1, 3, \{C1, C3, C4\})$	4 762
		$(A1, 3, \{C2, C3, C4\})$	7 395
		$(A1, 4, \{C1, C2, C3, C4\})$	181 697
单分支调用	$A1$	$(0, 1, \{B\})$	645 962

表 3 以方法的调用模式 (详见本文第 2.2 节) 对方法进行归类, 罗列了各方法的所有方法调用模型, 并统计了各方法调用模型下的实例数. 其中 $P_B = (A1, 2, \{C3, C4\})$ 不存在实例数据.

考虑到业务空闲时间段用户请求数相对较少, 性能异常发生的概率较低, 而业务繁忙时间段更易检测和发现异常, 因此本文将日志中 15 时至 18 时共 3 小时的数据作为检测数据集, 其他时段数据作为建模数据集. 由于 3 小时的检测数据集包括数亿条日志, 对其全部标注十分困难. 考虑本文的研究重点是针对方法重复调用模式下父方法的性能异常检测以及各模式下的根因定位, 因此选取典型实例进行标注和实验分析.

4.3 方法重复调用模式

4.3.1 单种方法重复调用模式的实验结果

本文以表 2 中以 $A1$ 起始的调用链集合为研究对象进行实验分析. 由表 3 可知, 方法重复调用模式下仅调用 $C2$ 方法的实例数量明显多于仅调用 $C1$、$C3$、$C4$ 的实例数量, 且经过人工标注发现后 3 类实例中仅有个别异常, 检测结果不易展示检测效果. 因此以仅调用 $C2$ 方法的实例为例, 展示重复调用一种方法时的检测效果, 数据的标注及检测结果如图 6 所示, 此处取阈值 ε=2.

图 6 中共有 3 283 个数据点, 依据标注和检测结果分为 4 类, 统计信息如表 4 所示.

本文基于上述结果, 讨论如下.

(1) 图 6 中大部分数据的纵坐标值接近 0, 说明对大部分实例而言, ξ 的实际值与本文的预测值相对接近, 即本文的建模方法能反映实际情况.

(2) 由于实验数据没有经过清洗, 在建模和检测数据集中都可能存在少量性能异常值, 在图 6 中体现为数值远大于 0 的情况.

(3) 由于本文是对真实数据的标注, 为了反映实际检测结果, 所以没有调整正负样本比例, 因而正负样本比例差距很大, 为避免单一样本检测参数的片面性, 本文分别计算正负样本的检测结果, 包括精确率、召回率、$F1$-score. 表 4 的统计信息说明本文方法对单种方法重复调用实例的检测效果良好.

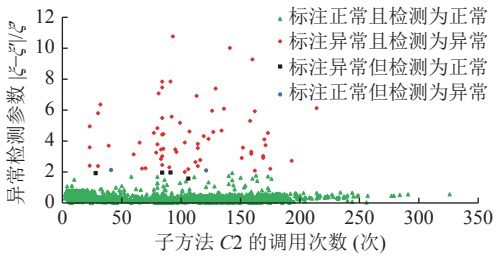


图 6　单种方法重复调用实例的检测结果图

表 4　单种方法重复调用实例检测结果统计表

标注分类	检测为正常	检测为异常	精确率 (%)	召回率 (%)	*F*1-score
正样本	3 210	2	99.88	99.94	0.999 1
负样本	4	67	97.10	94.37	0.957 1

4.3.2　多种方法重复调用模式的实验结果

在 3 小时检测数据集中, 方法重复调用的实例超过 10 万, 对所有实例进行标注成本太高. 因此本文从这些实例中随机选取 1 万个进行人工标注和实验, 结果如图 7 所示.

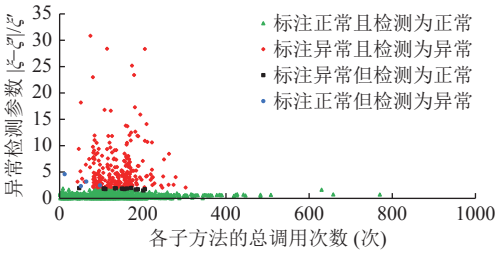


图 7　方法重复调用模式实例的检测结果图

图 7 检测结果的统计信息如表 5 所示.

表 5　多种方法重复调用实例检测结果统计表

标注分类	检测为正常	检测为异常	精确率 (%)	召回率 (%)	*F*1-score
正样本	9 732	4	99.85	99.96	0.999 0
负样本	15	249	98.42	94.32	0.963 2

目前基于调用链日志的异常检测研究有很多, 但真正适用于本文大型微服务场景的却很少. 例如, Multimodal LSTM[24]把部分重点放在了调用链结构 (或路径) 的异常检测, 然而大型微服务系统的复杂性导致其调用链结构复杂多样, 通过对日志的分析, 本文发现异常更多表现在方法性能上, 而极少出现不符合业务逻辑的错误调用, 且系统中存在大量的小样本结构, 因此不适合直接从结构上判断异常. 为了将本文方法与现有方法进行对比, 本文选取目前 3 种常用方案进行实验.

(1) 方案一

不考虑方法所在调用链, 仅基于每种方法实例的 3σ 检测: 将方法的执行时间看作正态分布, 计算建模数据集中该调用链集合中所有 B 方法实例的均值 μ 和标准差 σ, 对于一个待检测值 x_i, 将$|x_i-\mu| \geqslant 3\sigma$ 的值判定为异常.

(2) 方案二

以文献 [25] 的性能异常检测方法进行对比实验, 将建模粒度由原服务组件级调整为本文的方法级, 即基于该调用链集合中所有 B 方法实例的执行时间序列进行建模, 并用于检测.

(3) 方案三

基于具体调用链结构的 3σ 检测: 对调用链结构 (此处指子方法及子方法的调用次数) 相同的实例进行建模, 计算 B 方法执行时间的均值 μ 和标准差 σ; 对于一个待检测值 x_i, 从建模数据中寻找相同的调用链结构及对应参数, 将$|x_i-\mu| \geqslant 3\sigma$ 的值判定为异常.

3 种方案的实验结果如表 6 所示.

表 6　3 种方案的检测结果统计表

方案编号	标注分类	检测为正常	检测为异常	无法检测	精确率 (%)	召回率 (%)	$F1$-score
方案一	正样本	9 735	1	N/A	98.77	99.99	0.993 8
	负样本	121	143	N/A	99.31	54.17	0.701 0
方案二[25]	正样本	8 892	844	N/A	98.93	91.33	0.949 8
	负样本	96	168	N/A	16.60	63.64	0.263 3
方案三	正样本	4 836	74	4 826	99.20	49.67	0.662 0
	负样本	39	48	177	4.39	85.23	0.083 5

据表 6 所示, 本文讨论如下.

(1) 方案一虽然有比较高的 3 项指标, 但负样本召回率只有 54.17%, 与本文方法的 94.32% 相差很大; 且负样本 $F1$-score 值为 0.701 0, 比本文方法的 0.963 2 低很多. 这种方法没有考虑每次方法执行上的差异, 仅根据执行时间进行异常判断, 难以应对大型微服务系统复杂的调用过程. 例如, 本文实验数据中存在一组实例, 如表 7 所示.

表 7　方法重复调用模式的一组实例信息表

实例编号	所在调用链	B执行时间 (ms)	标注结果
1	$A1(B(C1^{41}))$	5 973	异常
2	$A1(B(C1^{160}))$	6 078	正常

我们易知, 上述实例中调用子方法 $C1$ 的次数会影响 B 的执行时间. 表 7 实例 1 中 B 的执行时间比实例 2 中短, 是异常值, 而执行时间更长的实例 2 却是正常值. 这类情况在大型微服务系统中非常普遍, 然而方案一只依据执行时间进行判断, 必然会出现错误, 因而检测结果不准确.

(2) 方案二中负样本的精确率、召回率和 $F1$-score 都比较低, 与本文方法差距很大. 文献 [25] 在服务性能建模和检测时同样没有考虑服务所在调用链, 仅基于服务的执行时间序列采用深度学习方法进行建模和检测, 由于调用子方法的次数并不是由时间序列决定的, 而是由实际用户请求引起的, 因而这类不考虑实际调用过程的建模和检测难以取得良好的效果.

(3) 方案三的检测结果中, “无法检测”表明建模数据集中没有该结构, 其数量超过一半 (5 003 个). 目前通常把这类未出现过的调用链结构数据作为异常数据处理, 本文沿用该思路计算精确率和召回率, 其各项指标均与本文方法存在差距. 据此推测, 对基于具体调用链结构的性能建模和检测方法来说, 由于建模数据集难以覆盖所有结构且某些结构的实例很少, 因此后续使用现有统计和机器学习方法等, 都难以取得良好的检测效果.

(4) 本文方法在建模前, 充分分析了方法调用链控制流, 将方法置于具体调用链中进行分析, 且为了解决方案三中的瓶颈, 进一步将方法调用链抽象成以方法为单位的方法调用模型, 针对同一种方法调用模型进行建模和检测, 保留了方法执行的业务场景特征和实际执行特征, 因而同一方法调用模型中的实例具有更接近的模式, 建模和检测效果更好.

4.4　其他模式

本文对方法重复调用模式和其他模式的参数计算方法不同, 前文验证了本文方法对重复调用模式建模和检测的有效性. 此处选取多分支调用的一个实例, 即表 2 中以 $A6$ 为起始方法的调用链集合, 来分析本文方法对其他模式的建模和检测效果. $A6$ 的多分支调用模式主要有 3 种, 如图 8 所示.

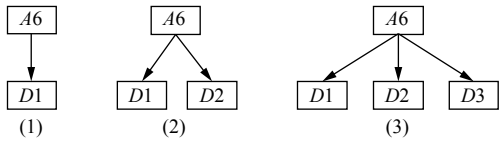


图 8 多分支调用实例的调用链结构

3 种方法调用模型可依次描述为:

(1) $P_{A6}=(0, 1, \{D1\})$

(2) $P_{A6}=(0, 2, \{D1, D2\})$

(3) $P_{A6}=(0, 3, \{D1, D2, D3\})$

此处基于 3σ 异常检测算法设置异常检测阈值 ε: 分别计算各方法调用模型实例中 $A6$ 除子方法执行时间外的其他开销 ξ 的均值 $\bar{\xi}$ 和标准差 σ, 取 $\varepsilon=3\sigma/\bar{\xi}$.

基于上述思路分别检测 3 种调用模式. 其中, 模式 (1) 共 20 725 个实例, 检测出 151 个异常实例, 异常占比 0.73%; 模式 (2) 共 21 826 个实例, 检测出 772 个异常实例, 异常占比 3.54%; 模式 (3) 共 4 852 个实例, 检测出 49 个异常实例, 占比 1.01%. 经人工分析, 虽然检测结果中异常占比有明显差别, 但检测结果基本符合实际情况.

4.5 异常根因定位

本文在检测方法性能时排除了子方法对父方法的影响, 因而对父子方法性能的检测结果是相对独立的, 检测结果即表明方法执行环节是否异常, 其中父方法异常说明其自身执行或调用子方法的网络开销异常, 从而更有效地识别根因. 大型微服务系统调用方法时可能产生排队延时, 本文将排队延时看作为网络开销的一部分. 本文以图 8 中模式 (3) 的一组实例进行分析, 如表 8 所示.

表 8 多分支调用模式一组实例的检测信息表

实例编号	$A6$执行时间 (ms)	$D1$执行时间 (ms)	$D2$执行时间 (ms)	$D3$执行时间 (ms)	其他开销 (ms)	$A6$检测结果
1	9 282	9 043	204	17	18	正常
2	595	50	101	5	439	异常

如表 8 所示, 实例 1 中 $A6$ 的执行时间达到了 9 282 ms, 而实例 2 中 $A6$ 的执行时间只有 595 ms, 但本文方法却将实例 1 检测为正常, 将实例 2 检测为异常. 这是因为在排除子方法执行时间后, 实例 1 中 $A6$ 的其他开销只有 18 ms, 而实例 2 中却达到了 439 ms. 容易发现实例 1 中子方法 $D1$ 的执行时间可能异常, 而本文对父子方法检测结果的独立性, 提高了根因分析的效率, 将根因定位在 $D1$ 或其后续调用过程. 而实例 2 的检测结果说明其 $A6$ 方法自身执行或调用子方法的网络异常.

然而, 若检测时不排除各子方法执行时间的影响, 仅对 $A6$ 方法总执行时间进行建模和检测, 则实例 1 极有可能被判定为异常, 而实例 2 则可能被判定为正常, 从而造成两个主要问题.

(1) 会将异常实例 2 判定为正常, 影响检测结果的准确性.

(2) 可能将实例 1 中 $A6$ 方法和其子方法 $D1$ 都判定为异常, 难以判定根因.

以上分析说明本文方法可以有效检测和定位请求执行过程中的异常方法. 本文旨在从方法级别检测和定位异常, 以便更好地进行后续分析. 但本文并没有提出对检测结果的后续分析方法, 通常基于一段时间内的检测结果作进一步聚合分析, 可以有效识别这段时间内性能异常的服务及其方法, 从而大大降低运维人员发现异常及定位根因的难度.

4.6 讨 论

4.6.1 数据集预处理及标注

即使系统正常运行时也可能会有少数请求因响应不及时等原因导致方法性能异常, 这是大型业务系统的常见问题. 本文方法在建模前不需要过滤建模数据集中的异常值, 具有一定的鲁棒性, 但应避免使用系统发生明显故障时间段的日志作建模数据集.

对大规模数据集的标注一直以来都是业界面临的难题之一. 本文在验证检测效果时采用人工方式对部分数据进行标注, 标注结果依赖于运维人员的经验和判断, 可作为依据用于实验分析, 但难以确保百分百准确, 尤其对于处在正常和异常临界点附近的值, 稍有偏差可能会对结果产生影响. 因此, 如何快速、准确标注大规模数据集依然是本文的努力方向之一.

由于本文方法在建模时, 并没有用标注数据, 是一种非监督的机器学习, 这大大节约了建模成本. 但在异常判定时, 我们难以通过对未标注数据集的学习来准确定义异常, 还是需要人工参与阈值设置. 但在这个阶段调整阈值的代价, 远远小于标注大规模数据集的代价.

4.6.2 拟合有效性分析

本文基于仅重复调用一种方法的实例拟合相关参数, 并将参数用于多种方法重复调用实例的异常检测. 本文研究中基本存在单独调用每种方法的实例, 若不存在, 则基于公式 (5) 用最小二乘法拟合多个参数是一种可行方案, 虽然拟合效果达不到两个参数的程度, 但在特定需求下可以应用.

为验证本文拟合方法的有效性, 以单种方法重复调用实例, 即表 3 中方法调用模型为 $P_B=(A1, 1, \{C1\})$ 的实例为例, 展示拟合效果如图 9 所示.

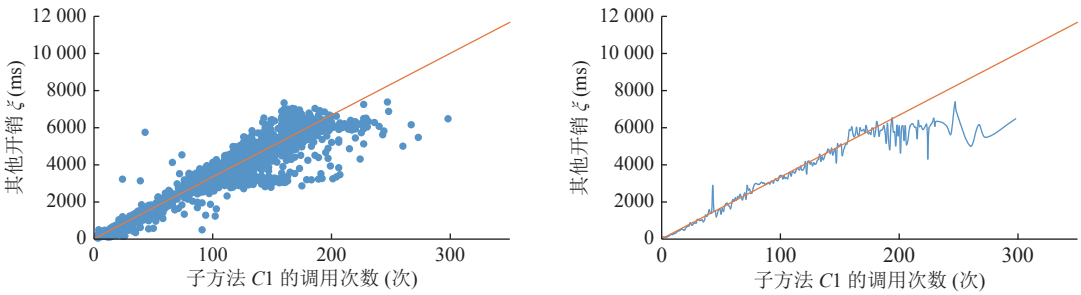


图 9 实例拟合效果图

图 9 中, 横坐标表示实例中子方法 $C1$ 的重复调用次数 (单位: 次), 纵坐标表示父方法 B 除子方法执行时间外的其他开销 ξ(单位: ms). 左图中蓝色点表示检测数据集中的实例数据; 右图中蓝色曲线为散点的平滑连接线, 各散点表示特定调用次数实例的 ξ 平均值, 橙色直线为本文方法对建模数据集实例拟合得到的预测直线.

如图 9 所示, 调用次数大约在 200 以内时, 本文方法拟合得到的直线基本能预测实际情况. 当调用次数增大到一定程度, 由于实例数较少, 不足以反映其平均情况, 所以在一定程度上发生偏离. 但这并不影响实际预测和检测效果, 本文实验部分已经验证了检测结果的准确性.

5 总结及下一步研究方向

大型微服务系统中存在方法重复调用和多分支调用等逻辑调用关系, 导致调用链结构非常多, 且某些结构的实例较少, 这对现有性能异常检测方法提出挑战. 本文从大型微服务系统的方法层面进行研究, 首先基于方法间调用关系, 以树结构构建方法调用链; 之后基于调用链控制流分析, 将大量调用链抽象为少量方法调用模型, 方法调用模型能保留方法执行的业务场景和性能情况; 然后提出同步调用模式下方法的执行时间分解模型, 针对各方法调用模型, 以除子方法执行时间外的其他开销分析和检测父方法性能情况, 从而有效检测异常并识别根因. 实验证明, 本文方法能够应对微服务系统中复杂的调用模式, 将大规模调用链结构抽象为少数方法调用模型, 大幅减少存储和分析的调用链结构数量, 从而提高异常检测效率, 并能有效检测和定位请求执行过程中服务方法的性能异常.

本文方法本质上是从软件功能分析的角度, 抽象刻画了调用链与软件功能相对应的“高层语义”. 未来这种模型驱动的方法可以和数据驱动的方法进一步有效结合, 提升服务性能异常的预测和检测能力. 本文工作主要是基于数据集的线下分析, 然而在实际线上系统中面临着许多严峻的挑战, 诸如如何从线上日志中快速有效地抽取并构建调用链、针对数量巨大的待检测实例如何保证检测效率、对异常检测结果如何快速有效地聚合分析以准确识别异常服务和方法等等, 这些也都是本文下一步工作的努力方向.

References:

[1] Di Francesco P, Malavolta I, Lago P. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: Proc. of the 2017 IEEE Int'l Conf. on Software Architecture (ICSA). Gothenburg: IEEE, 2017. 21–30. [doi: 10.1109/ICSA.2017.24]

[2] Neri D, Soldani J, Zimmermann O, Brogi A. Design principles, architectural smells and refactorings for microservices: A multivocal review. SICS Software-intensive Cyber-physical Systems, 2020, 35(1): 3–15. [doi: 10.1007/s00450-019-00407-8]

[3] Zhou X, Peng X, Xie T, Sun J, Ji C, Li WH, Ding D. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. on Software Engineering, 2021, 47(2): 243–260. [doi: 10.1109/TSE.2018.2887384]

[4] Fonseca R, Porter G, Katz RH, Shenker S, Stoica I. X-trace: A pervasive network tracing framework. In: Proc. of the 4th USENIX Symp. on Networked Systems Design & Implementation. Cambridge: USENIX Association, 2007. 271–284.

[5] Sigelman BH, Barroso LA, Burrows M, Stephenson P, Plakal M, Beaver D, Jaspan S, Shanbhag C. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report, 2010.

[6] Kaldor J, Mace J, Bejda M, Gao E, Kuropatwa W, O'Neill J, Ong KW, Schaller B, Shan PJ, Viscomi B, Venkataraman V, Veeraraghavan K, Song YJ. Canopy: An end-to-end performance tracing and analysis system. In: Proc. of the 26th Symp. on Operating Systems Principles. Shanghai: ACM, 2017. 34–50. [doi: 10.1145/3132747.3132749]

[7] Hodge V, Austin J. A survey of outlier detection methodologies. Artificial Intelligence Review, 2004, 22(2): 85–126. [doi: 10.1023/B:AIRE.0000045502.10941.a9]

[8] Chandola V, Banerjee A, Kumar V. Anomaly detection: A survey. ACM Computing Surveys, 2009, 41(3): 15. [doi: 10.1145/1541880.1541882]

[9] Richardson C. Microservices Patterns: With Examples in Java. Shelter Island: Manning, 2018.

[10] SmartBear. Why you can't talk about microservices without mentioning netflix. 2015. https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/

[11] Zhou H, Chen M, Lin Q, Wang Y, She XB, Liu SF, Gu R, Ooi BC, Yang JF. Overload control for scaling wechat microservices. In: Proc. of the ACM Symp. on Cloud Computing. Carlsbad: ACM, 2018. 149–161. [doi: 10.1145/3267809.3267823]

[12] He SL, Zhu JM, He PJ, Lyu MR. Experience report: System log analysis for anomaly detection. In: Proc. of the 27th IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Ottawa: IEEE, 2016. 207–218. [doi: 10.1109/ISSRE.2016.21]

[13] Pena EHM, de Assis MVO, Proença ML. Anomaly detection using forecasting methods ARIMA and HWDS. In: Proc. of the 32nd Int'l Conf. of the Chilean Computer Science Society (SCCC). Temuco: IEEE, 2013. 63–66. [doi: 10.1109/SCCC.2013.18]

[14] De Nadai M, van Someren M. Short-term anomaly detection in gas consumption through ARIMA and Artificial Neural Network forecast. In: Proc. of the 2015 IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS) Proc. Trento: IEEE, 2015. 250–255. [doi: 10.1109/EESMS.2015.7175886]

[15] Andrysiak T, Saganowski Ł, Maszewski M. Time series forecasting using Holt-Winters model applied to anomaly detection in network traffic. In: García HP, Alfonso-Cendón J, González LS, Quintián H, Corchado E, eds. Proc. of the Int'l Workshop on Soft Computing Models in Industrial and Environmental Applications Computational Intelligence in Security for Information Systems Conf. Int'l Conf. on EUropean Transnational Education. León: Springer, 2017. 567–576. [doi: 10.1007/978-3-319-67180-2_55]

[16] Ekberg J, Ylinen J, Loula P. Network behaviour anomaly detection using Holt-Winters algorithm. In: Proc. of the 2011 Int'l Conf. for Internet Technology and Secured Trans.. Abu Dhabi: IEEE, 2011. 627–631.

[17] Chen YY, Mahajan R, Sridharan B, Zhang ZL. A provider-side view of web search response time. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 243–254. [doi: 10.1145/2534169.2486035]

[18] Ringberg H, Soule A, Rexford J, Diot C. Sensitivity of PCA for traffic anomaly detection. ACM SIGMETRICS Performance Evaluation Review, 2007, 35(1): 109–120. [doi: 10.1145/1269899.1254895]

[19] Liu DP, Zhao YJ, Xu HW, Sun YQ, Pei D, Luo J, Jing XW, Feng M. Opprentice: Towards practical and automatic anomaly detection through machine learning. In: Proc. of the 2015 Internet Measurement Conf. Tokyo: Association for Computing Machinery, 2015. 211–224. [doi: 10.1145/2815675.2815679]

[20] Thalheim J, Rodrigues A, Akkus IE, Bhatotia P, Chen RC, Viswanath B, Jiao L, Fetzer C. Sieve: Actionable insights from monitored metrics in distributed systems. In: Proc. of the 18th ACM/IFIP/USENIX Middleware Conf. Las Vegas: Association for Computing Machinery, 2017. 14–27. [doi: 10.1145/3135974.3135977]

[21] Lin JJ, Chen PF, Zheng ZB. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In: Pahl C, Vukovic M, Yin J, Yu Q, eds. Proc. of the 16th Int'l Conf. on Service-oriented Computing. Lecture Notes in Computer Science, vol 11236. Hangzhou: Springer, 2018. 3–20. [doi: 10.1007/978-3-030-03596-9_1]

[22] Malhotra P, Vig L, Shroff G, Agarwal P. Long short term memory networks for anomaly detection in time series. In: Proc. of the 23rd European Symp. on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges, 2015. 89–94.

[23] Gan Y, Zhang YQ, Hu K, Cheng DL, He Y, Pancholi M, Delimitrou C. Seer: Leveraging big data to navigate the complexity of

performance debugging in cloud microservices. In: Proc. of the 24th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Providence: Association for Computing Machinery, 2019. 19–33. [doi: 10.1145/3297858.3304004]

[24] Nedelkoski S, Cardoso J, Kao O. Anomaly detection from system tracing data using multimodal deep learning. In: Proc. of the 12th IEEE Int'l Conf. on Cloud Computing (CLOUD). Milan: IEEE, 2019. 179–186. [doi: 10.1109/CLOUD.2019.00038]

[25] Nedelkoski S, Cardoso J, Kao O. Anomaly detection and classification using distributed tracing and deep learning. In: Proc. of the 19th IEEE/ACM Int'l Symp' on Cluster, Cloud and Grid Computing (CCGRID). Larnaca: IEEE, 2019. 241–250. [doi: 10.1109/CCGRID.2019.00038]

[26] Wang ZY, Wang T, Zhang WB, Chen NJ, Zuo C. Fault diagnosis for microservices with execution trace monitoring. Ruan Jian Xue Bao/Journal of Software, 2017, 28(6): 1435−1454 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5223.htm [doi: 10.13328/j.cnki.jos.005223]

[27] Sambasivan RR, Zheng AX, De Rosa M, Krevat E, Whitman S, Stroucken M, Wang W, Xu LH, Ganger GR. Diagnosing performance changes by comparing request flows. In: Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation. Boston: USENIX Association, 2011. 43–56.

[28] Kopetz H, Ochsenreiter W. Clock synchronization in distributed real-time systems. IEEE Trans. on Computers, 1987, C-36(8): 933–940. [doi: 10.1109/TC.1987.5009516]

[29] Zhou X, Peng X, Xie T, Sun J, Xu CJ, Ji C, Zhao WY. Benchmarking microservice systems for software engineering research. In: Proc. of the 40th Int'l Conf. on Software Engineering: Companion Proc. Gothenburg: Association for Computing Machinery, 2018. 323–324. [doi: 10.1145/3183440.3194991]

[30] Train Ticket: A benchmark microservice system. https://github.com/FudanSELab/train-ticket

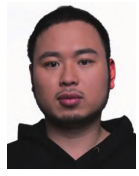
附中文参考文献:

[26] 王子勇, 王焘, 张文博, 陈宁江, 左春. 一种基于执行轨迹监测的微服务故障诊断方法. 软件学报, 2017, 28(6): 1435−1454. http://www.jos.org.cn/1000-9825/5223.htm [doi: 10.13328/j.cnki.jos.005223]

于庆洋(1991－), 男, 博士生, CCF 学生会员, 主要研究领域为软件工程, 智能运维.

刘涛(1984－), 男, 硕士, 主要研究领域为广告系统架构.

白晓颖(1973－), 女, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为计算机软件.



刘泽胤(1983－), 男, 硕士, 主要研究领域为广告系统架构.

李明杰(1995－), 男, 博士生, CCF 学生会员, 主要研究领域为软件工程, 智能运维.

裴丹(1973－), 男, 博士, 长聘副教授, 博士生导师, CCF 专业会员, 主要研究领域为基于机器学习的智能运维.

李奇原(1987－), 女, 硕士, 主要研究领域为软件工程, 分布式架构.