软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

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

基于调用链的微服务日志可视化分析及研究*

李文海[1,2], 彭 鑫[1,2+], 丁 丹[1,2], 向麒麟[1,2], 郭晓峰[1,2], 赵文耘[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: 李文海, E-mail: 16212010016@fudan.edu.cn

摘 要: 云计算时代,越来越多的企业开始采用微服务架构进行软件开发或者传统巨石应用改造.然而,微服务系统具有较高的复杂性和动态性,当微服务系统出现故障时,目前没有方法或者工具能够有效支持对故障根源的定位.本文首先定义了微服务日志的领域模型,在此基础上开发了一个原型工具 LogVisualization,可以实现微服务日志和调用链的关联展示.同时,LogVisualization 的可视化界面提供了五种不同的策略,用于支持用户对微服务故障根源的查找定位.本文将该原型工具应用于实际的微服务系统,通过与现有工具的实验对比,验证了该原型工具的有用性和有效性.

关键词: 微服务;调用链;日志;可视化;故障;根源定位

中图法分类号: TP311

中文引用格式: 李文海,彭鑫,丁丹,向麒麟,郭晓峰,赵文耘基于调用链的微服务日志可视化分析及研究.软件学报. http://www.jos.org.cn/1000-9825/ 0000.htm

英文引用格式: Li WH, Peng X, Ding D, Xiang QL, Guo XF, Zhao WY. Microservice log visualization and research based on trace. Ruan Jian Xue Bao/Journal of Software, 2016 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

Microservice log visualization and research based on trace

LI WEN-HAI[1,2], PENG XIN[1,2+], DING DAN[1,2], XIANG QI-LIN[1,2], GUO XIAO-FENG[1,2],

ZHAO WEN-YUN[1,2]

[1](School of Computer Science, Fudan University, Shanghai 200433, China)

[2](Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 200433, China)

Abstract: In the era of cloud computing, more and more enterprises are adopting microservice architecture for software development or traditional monolithic application transformation. However, microservice system has high complexity and dynamism. When microservice system fails, there is currently no method or tool that can effectively support the location of the root cause of failure. This paper first defines the domain model of microservice log. Based on this, a prototype tool named LogVisualization is developed. LogVisualization can realize the association display of microservice log and trace. At the same time, the visualization page of LogVisualization provides five different strategies to support the users locate the root cause of microservice failure. This paper applies the prototype tool to the actual microservice system, and verifies the usefulness and effectiveness of prototype tool by comparing with the existing tools.

Key words: microservice; trace; log; visualization; fault; location of root cause

* 基金项目: 国家自然科学基金(00000000, 00000000); 南京大学计算机软件新技术国家重点实验室开放课题(KFKT00000000)

Foundation item: National Natural Science Foundation of China (00000000, 00000000); State Key Laboratory for Novel Software Technology (Nanjing University)开放课题 (KFKT00000000)

收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00

CNKI 在线出版时间: 0000-00-00

微服务(Microservices)是一种软件架构风格,将一个大型复杂的软件应用分解成多个服务,每个服务都作为一个小而独立的系统进行实现和运行,并通过定义良好的网络接口提供对其内部逻辑和数据的访问.服务之间的相对独立性,使得软件应用具有更快的交付速度,更好的扩展性以及更强的自治性[1].微服务是软件服务设计,开发和交付的最新趋势,越来越多的企业选择采用微服务架构进行软件开发或者传统巨石应用的改造[2,3,4].例如,腾讯的微信系统由 2000 多个微服务组成,这些微服务运行在位于多个数据中心的 40,000 多台服务器上[5];Netflix 公司花了 7 年时间(2008.8 - 2016.1)完成从传统的巨石体系到微服务架构的迁移[6].

尽管微服务有许多好处,而且在工业界得到了广泛使用,但是这一领域依然面临不少的挑战,例如:如何合理地划分服务;如何实现有效的资源监控和管理;如何应对系统故障,实现自修复等等[7].其中,为了应对系统故障,实现自修复,首先需要能够对导致故障的根源进行有效定位.然而,在微服务系统里,可能导致故障的原因是多方面的,除了单个服务本身实现可能存在问题,更多地可能是因为服务的运行时环境,服务间的交互以及协作存在问题[8].微服务系统在提供服务时,处理单个请求可能会涉及到成百上千个服务,例如:亚马逊为了渲染一个页面,一般会进行 100-150 次的服务调用[9],任意一个服务异常都可能导致系统故障;同时,每个服务都会有一些环境配置,比如 CPU 和内存的配置,不当的配置可能导致服务在高负载的时候无法正常提供服务,而且每个服务一般都会部署多个实例,如果服务本身没有做好状态同步,那么服务多实例运行时也会导致故障;服务与服务之间的交互可能很复杂,例如不同服务需要跨节点通信以及服务之间存在异步调用,如果节点与节点之间网络通信存在问题或者异步调用的某些执行或者返回序列会导致问题,系统也会出现故障.所以,可能导致微服务系统故障的因素较多,要对故障的根源进行有效定位, 需要有相应的方法或者工具辅助.

微服务故障根源定位的方法支持方面,周翔等人[10]提出可以应用用增量调试[11]的方法.但是该方法无法处理服务本身实现上存在的问题,同时该方法需要有多个测试集群,以加快定位的速度.在工具支持方面,目前主要有三类工具.第一类是分布式系统的调用链追踪工具,比如 Zipkin[12],Jaeger[13],第二类是分布式系统日志的收集及可视化工具,其中以 ELK Stack[14]为代表,第三类是分布式系统执行的可视化及对比工具,即 ShiViz[15].然而,单纯使用 Zipkin 进行调用链追踪,只能知道调用链所涉及的服务, 而一个服务可能有多个实例,这些实例随机分散在集群的节点上,用户需要人工确定具体调用的服务实例,进而查看与该条调用链对应的日志;单纯使用 ELK Stack 进行日志的收集汇总,会导致日志信息的上下文缺失,即没有日志和服务的对应关系以及服务之间的调用关系;Shiviz 可以利用分布式系统的执行日志生成服务与服务之间的交互式通信图,同时提供了不同系统执行之间的对比功能,方便开发者从服务交互和通信的维度进行问题排查.目前没有任何工具可以较好地将日志和调用链关联展示,也没有相关文献研究将日志和调用链相关联对微服务故障根源定位所带来的帮助.而且,现有工具在服务多实例的问题排查方面,并没有提供支持.

为了解决上述问题,本文首先定义了微服务日志的领域模型,在此基础上实现了一个原型工具—LogVisualization,可以收集存储集群的日志、调用链数据以及节点和服务实例信息,并将调用链和相应的日志关联展示.同时该原型工具针对微服务故障根源定位提供了不同的策略支持,包括:单条调用链的日志查看,调用链之间的对比,服务异步调用分析,服务多实例分析以及调用链的分段,使用者可以通过可视化缩放以及灵活组合不同的策略来对故障的根源进行定位.最后,我们将该原型工具在 TrainTicket[8]这一微服务系统上进行了实际的实验,通过与 Zipkin 以及 ELK Stack 的对比,验证了该原型工具在微服务故障根源定位上的有用性和有效性.

本文组织结构如下:第一部分概述相关工具及研究现状.第二部分详细介绍本文构建的微服务日志领域模型,原型工具及实现细节.第三部分基于 TrainTicket 这一微服务系统进行实际实验和分析对比.第四部分针对原型工具实现以及实验过程进行讨论.第五部分对全文的研究工作进行总结,并展望未来工作.

1 相关工具及研究现状分析

微服务作为一个新兴的热门研究话题[16],目前已有的相关工作数量还是较少,而且现有研究工作主要关注微服务的设计[17],测试[18-21],部署[22-24]以及运行时调整[25]等方面,还没有相关文献研究基于调用链的微服务日志可视化及其对微服务故障根源定位的帮助.

微服务系统本身是一个分布式系统,分布式系统的问题定位方法主要包括调用链追踪,日志的分析以及可视化[26].根据 OpenTracing 的定义,一条调用链是 span 的有向非循环图,而所谓 span,代表的是该调用链当中服务的一次 RPC 调用[27].目前绝大多数分布式系统的调用链追踪工具都是参考 Google 的 Dapper[28] 实现的.Dapper 是 Google 生产环境下的分布式系统调用链追踪工具,其设计目标有三个:低消耗,对应用程序透明以及可扩展性.基于 Google 的 Dapper,Twitter 公司开发了一款分布式追踪系统—Zipkin.Zipkin 可以收集排查微服务架构中的延迟问题所需的时序数据,它管理这些数据的收集和查找.Zipkin 支持根据应用程序,调用链长度,注释或时间戳对所有调用链进行筛选或排序.选择调用链后,可以看到每一个 span 所占的时间百分比,从而可以识别问题应用程序. Zipkin 搭配 ELK Stack 等日志收集工具,虽然可以为每一条调用链附加上日志,但是具体实现方式上,是跳转到 ELK 等日志收集的可视化页面,同时将调用链标识作为过滤条件,自动筛选展示该调用链上的相应日志.该种实现方式需要在两个页面之间切换,而且每次只能查看单条调用链日志,面对可能导致微服务系统故障的诸多因素,并无法很好地辅助开发人员对根源进行定位.而受 Dapper 以及 Zipkin 的启发,Uber 公司开发了 Jaeger 这一分布式追踪系统.相比于 Zipkin,Jaeger 有着更好的客户端库支持,以便对应用程序进行插装,包括:Go, Java, Node, Python 以及 C++.同时,Jaeger 的内存占用更低,设计上更为现代化,更具可扩展性[29-30].

分布式系统的日志收集分析可视化方面,目前比较流行的系统有 Cloudera 公司提供的 Flume[31],Facebook 公司的 Scribe[32]以及 Elastic.co 公司的 ELK Stack[33].Flume 是一种分布式,高可靠且高可用的服务,用于有效地收集,聚合以及传输海量的日志数据.它的架构简单灵活,同时具有强大的容错能力,支持在日志系统中定制各类数据源,可以对数据进行简单处理,并写到各种数据接收方.Scribe 则是 Facebook 公司自己开发,同时在公司内部大量应用的开源日志收集系统,具有高可扩展性和鲁棒性,能够应对网络和节点故障.为了应对海量数据,Scribe 通常与 Hadoop 联合使用,Scribe 用于收集日志并推送到 HDFS 当中,Hadoop 则通过 MapReduce 作业定期进行处理.ELK Stack 是一套流行的一体化日志处理平台解决方案,提供日志收集、处理、存储、搜索、展示等全方位功能,主要由 ElasticSearch、LogStash 、Kibana 以及 Beats 所构成[34].ElasticSearch 是一个基于 Apache Lucene 的分布式搜索和分析引擎;Logstash 则是用 Ruby 编写的一个服务器端数据处理管道,支持多种数据源,同时可以使用 Grok、Mutat 等插件对数据进行过滤转换,然后将数据发送到指定目的地;Kibana 则是一个图形化的网页界面,允许用户使用图表对 Elasticsearch 当中存储的数据进行可视化;Beats 是 Elastic.co 公司后面开发的一款开源日志收集器,相比较 Logstash 而言,Beats 更为轻量级,可以将数百或者数千台机器中的数据发送到 Logstash 或者 Elasticsearch.然而这些分布式系统日志收集和展示工具,并没有特别考虑日志和调用链的关系,只是纯粹的日志收集可视化,没有提供日志和调用链关联展示的功能.

ShiViz 是一个可视化引擎,可以从微服务系统的执行日志生成服务与服务之间的交互式通信图,开发人员可以按需对图的某一部分执行展开、折叠或者隐藏操作.ShiViz 还支持两次系统执行之间的可视化对比,对于两次执行之间的差异会突出显示.然而,为了使用该工具,需要将收集的调用链信息转化为特定格式,同时该工具只能较好地支持对服务交互方面的问题进行排查,无法直接将每次执行和相应产生的日志关联展示,而且在服务多实例问题排查方面帮助有限.

2 方法及实现

本文首先定义微服务日志的领域模型,在此基础上实现原型工具—LogVisualization,用于收集所需的日志信息、调用链数据以及集群节点和服务实例的信息并可视化展示,同时提供不同的策略让用户查找导致系统故障的根源.

2.1 微服务日志领域模型

微服务日志领域模型如图 1 所示.该模型从内容上可以分为三个部分:日志,调用链,服务与集群信息.

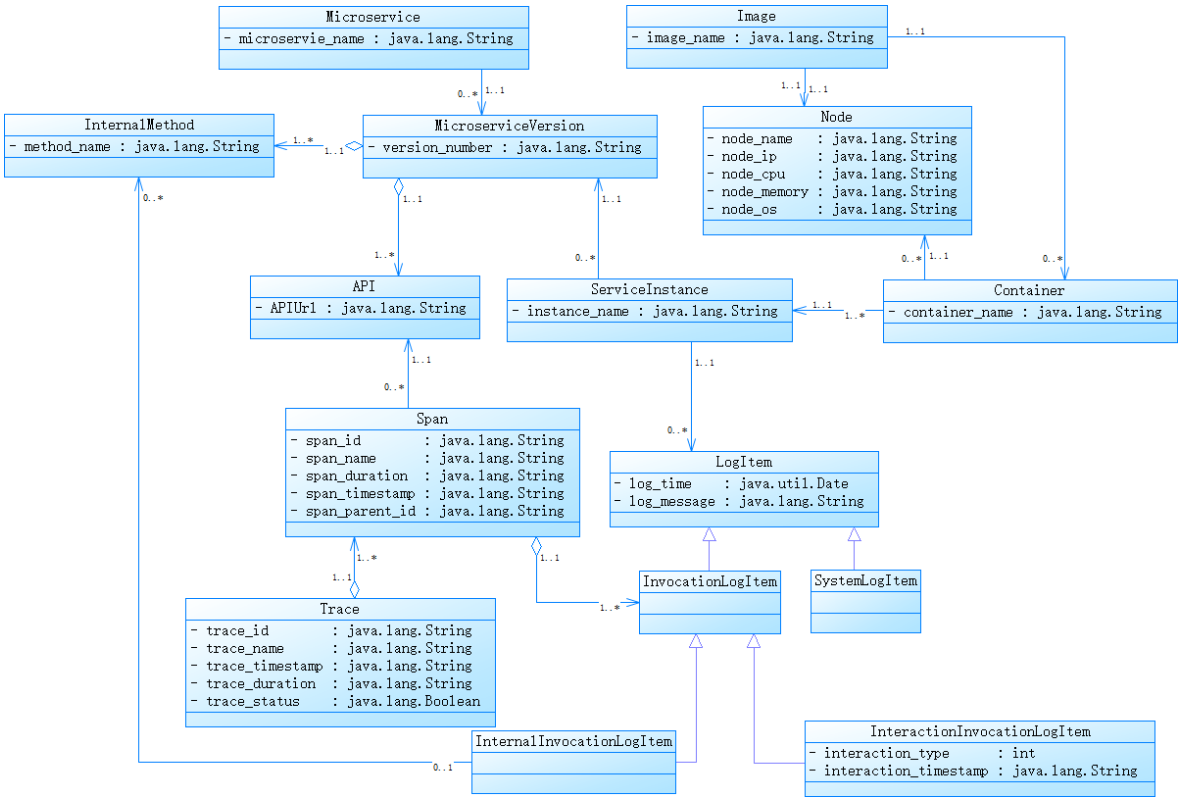

Fig.1 Log Domain of Microservice

图 1 微服务日志领域模型

首先,微服务系统运行时所产生的每一条日志,都称为日志项(LogItem),每个日志项包含两个关键属性:日志时间戳以及日志内容.日志项分为两大类,一类是单个微服务的系统日志(SystemLogItem),比如服务启动时所产生的日志项;另一类是服务与服务之间通信所产生的调用日志(InvocationLogItem).调用日志又可以细分为交互调用日志(InteractionInvocationLogItem)和内部调用日志(InternalInvocationLogItem),交互调用日志指的是服务接收请求以及返回请求结果所产生的日志项,内部调用日志则是指服务在处理请求时,服务内部方法所产生的日志项.

微服务系统每次处理请求时的完整服务调用序列,构成一条调用链(Trace),每条调用链有唯一标识,时间戳等属性,其中状态属性标识调用链的对错.调用链的基本组成单位是 span(Span),span 表示的是单个服务与服务之间的调用,每个 span 也都有自己的唯一标识,会记录服务调用的时间戳以及调用时长,同时 span 通过 span_parent_id 属性明确彼此之间的顺序关系,没有 span_parent_id 属性的 span 称为根 span.服务之间的调用是通过服务提供的 API 实现的,所以 span、调用日志以及 API(API)这三者可以相关联.同时,在微服务系统当中,每个服务(Microservice)都可能有多个版本(MicroserviceVersion),不同版本的微服务所提供的 API 以及实际内部方法(InternalMethod)实现可能存在差异.每个服务都可能有多个服务实例(ServiceInstance),每个服务实例可能包含一到多个容器(Container),这些容器运行在集群的工作节点(Node)上,负责加载指定的镜像(Image)并启动.同时,服务实例在进行业务处理时会产生日志,所以每个日志项都可以和服务实例相关联.

2.2 原型工具体系结构

基于定义的微服务日志领域模型,本文实现了一个可以将日志与调用链关联展示的原型工具 LogVisualization,其体系结构如图 2 所示.

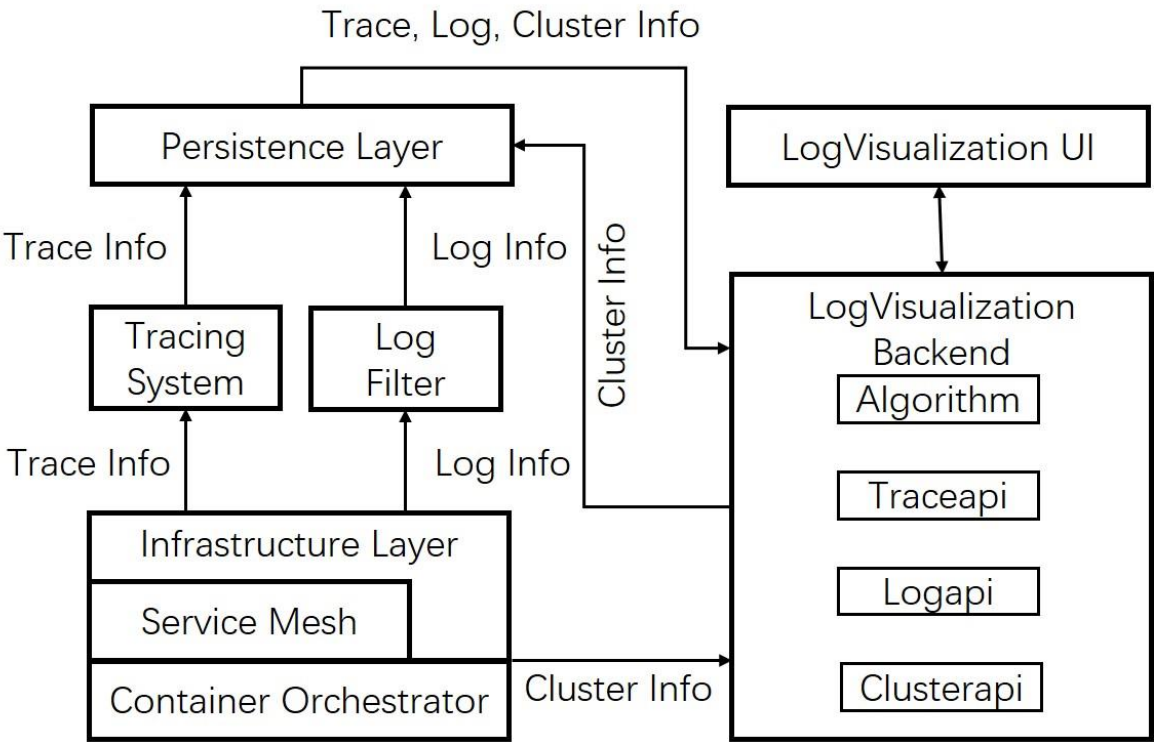

Fig.2 Architecture of Prototype Tool

图 2 原型工具体系结构

容器编排工具和服务网格共同构成基础设施层,由基础设施层管理集群以及微服务系统的部署运行.目前,运行微服务应用的最好形式就是容器[35],因为容器可以封装应用程序的业务代码以及所有必需的运行时环境和设置,同时容器可以实现不同应用程序之间的隔离.借助于容器编排工具,可以实现微服务应用的自动部署,扩容和管理.服务网格是一个比较新的概念,针对每一个服务实例,服务网格都会并行部署一个边车进程,从而控制服务与服务之间的网络通信,实现微服务系统当中请求的可靠传递[36].借助于基础设施层,我们可以实现日志和调用链数据的收集和发送.调用链数据会被发送到调用链追踪工具当中,而日志则会发送到过滤层,一方面去除无用的日志,另一方面对日志进行处理,提取日志当中的关键属性.同时,原型工具后台会定期调用基础设施层的 API 获取集群信息,主要包括集群节点的信息以及运行在集群当中的服务实例信息.这些信息都会被发送到持久化层存储.

后台还提供了三类 API 用于支持前台界面的可视化分析,包括:调用链 API,用于将调用链分类,统计分析调用链所涉及服务的信息等;日志 API,用于获取某条调用链或者某个服务实例的所有日志;算法 API,为分析调用链当中的异步调用以及对调用链分段提供算法支持.

2.3 原型工具实现

原型工具的具体实现如图 3 所示.

在基础设施层的具体实现中,本文使用了 Kubernates[37]进行容器的编排管理.作为一个开源容器集群管理项目,Kubernetes 提供了容器化应用部署、规划、更新、维护的一系列机制,保证云平台中的容器始终能够按照用户的期望状态运行.Kubernetes 中所有的容器均在 Pod 中运行,一个 Pod 可以承载一个或者多个相关的容器,Pod 一旦被创建,Kuberetes 就会持续监控 Pod 以及 Pod 所在节点（Node）的健康状况.

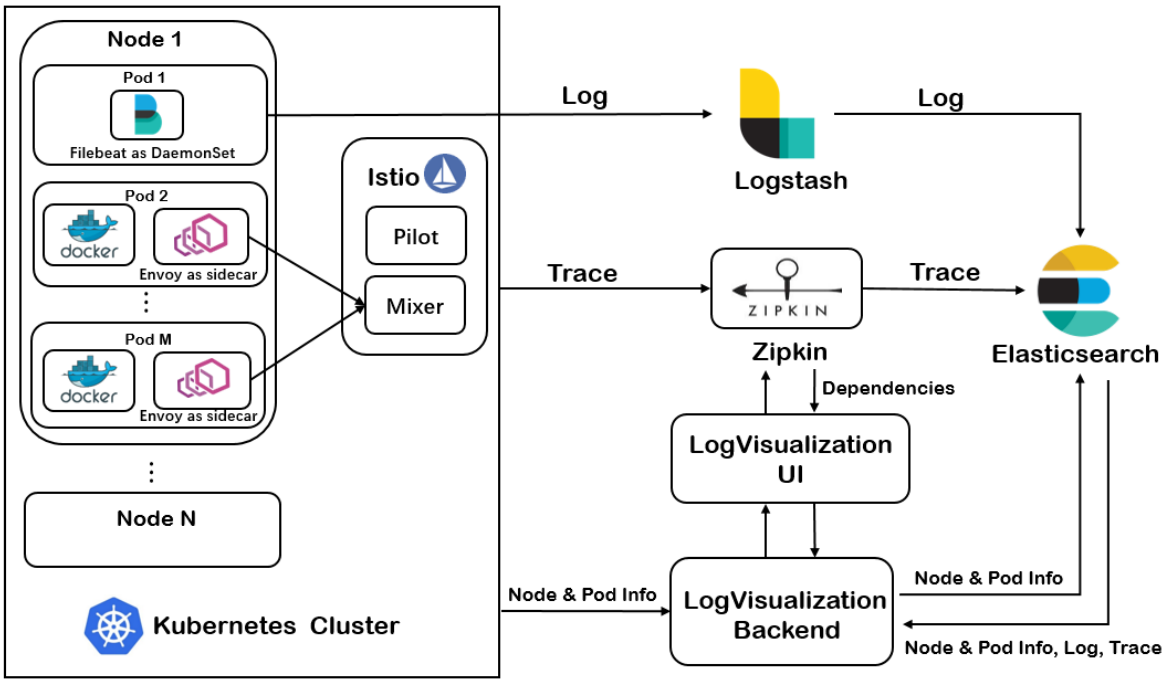

Fig.3 Implementation of LogVisualization

图 3 LogVisualization 实现

在此基础之上,本文利用 Istio[38]建立已部署服务的网络、对微服务间的通信调用进行监控拦截.Istio 在每一个微服务 Pod 中启动一个单独的进程 Envoy,从 Istio 控制平台的 Pilot 模块拉取对应的指令和规则、修改容器的 iptables 来代理应用程序的所有出入流量、并定期向控制平台的 Mixer 模块发送监控数据.这种不改动原应用程序的实现、通过额外的容器来扩展和增强主容器的方法被叫做边车（Sidecar）模式.通过边车,Istio 可以实现调用链信息的自动生成和传递,随后使用 Zipkin 进行调用链信息的收集,并将数据持久化存储到 Elasticsearch 中.

为了收集系统日志,集群的每个节点都会以 DaemonSet 的方式运行 Filebeat.Filebeat 属于 Beats,能够从指定目录读取应用程序的日志、对日志进行简单的过滤和处理（例如异常日志的多行拼接）后再发送给 Logstash.每条日志在 Logstash 中经过定制的 grok 插件,被分为请求、响应、内部方法和异常信息四种类型,同时提取出每个类别对应的关键字段.最后,Logstash 同样会将日志存储到 Elasticsearch 中.

核心部分 LogVisualization 以前后台分离的形式实现.后台接收前台发送的请求,从 Elasticsearch 查询相应信息、组装数据并返回给前台；另外,后台与 Kubernetes API Server 交互、定期获取集群中所有节点与 Pod 的相关信息并存储到 Elasticsearch 中.前台在 Zipkin 界面的基础上进行了修改,与后台进行交互,实现调用链与日志的关联可视化.

2.4 策略

2.4.1 概念解释

定义 1:请求类型(RequestType)对应于微服务系统当中的业务请求,比如 TrainTicket 包括登录,查询车次,订票,退票,查询订单等业务请求,这些业务请求都对应于一种请求类型

定义 2:调用链类型(TraceType)是对属于同一请求类型的调用链的细分.一种请求类型,在实际生产环境中,会有成千上万条调用链与之对应,因为用户可能多次发起该类型的请求.以调用链长度和调用链所经过的服务种类为依据,可以将属于同一请求类型的调用链分类

定义 3:服务依赖图(ServiceDependency)表示的是微服务系统在实际运行时,服务与服务之间的依赖关系.服务依赖图是动态生成的,是对所用调用链当中服务依赖关系的汇总.服务依赖图的节点是微服务系统运行期间所有被调用的服务,服务与服务之间如果有边相连,代表服务与服务之间有依赖关系

2.4.2 策略支持

微服务系统在实际运行时,会产生大量的日志和调用链数据.为了更高效地分析查找导致微服务系统故障的根源,需要能够按照一定的条件对数据进行过滤,缩小分析的数据量.LogVisualization 的可视化界面允许用户以天为单位,指定分析日志和调用链数据的时间段,可视化界面会以树的结构显示指定时间段内的所有请求类型,以及每个请求类型下存在的调用链类型.同时,每个调用链类型下会显示属于该调用链类型的所有调用链标识,每条调用链标识下会显示该条调用链经过的所有服务.每个请求类型和调用链类型都包含一些统计信息,比如正确以及错误的调用链数量.每条调用链也具有统计信息,比如异常,错误以及正常的日志项数量.

为了更加直观地展示请求类型和调用链类型的统计信息,可视化界面会通过饼状图的方式显示所有请求类型的错误率,点击某一请求类型后会自动展示该请求类型下所有调用链类型的错误率.同时,该饼状图和树形结构是联动的.可视化界面还会展示指定时间段内的服务依赖图,同时会根据服务的错误率对服务的颜色进行渲染,错误率越高,则渲染的颜色越深.其中,服务的错误率是指在指定时间段内,经过该服务的所有调用链当中,错误的调用链所占的比率.可视化界面如图 4(a)所示.

可视化界面提供了五种不同的策略,让用户对导致系统故障的根源进行查找.

策略 1:单条调用链日志查看.用户点击单条调用链以后,可视化界面会在服务依赖图中高亮显示该条调用链所经过的服务,同时展示该条调用链对应的日志链.日志链默认按照服务调用的 URI 排序,同一个 URI 调用产生的日志会按照时间排序,对应于请求接收、请求处理以及请求返回的顺序,如图 4(b)所示.按照这种方式组织日志,可以方便用户以服务的 API 为单位进行故障根源查找.同时,用户也可以选择按照时间对日志链中的所有日志项排序,方便查看此次请求处理的整个过程.用户针对单条调用链的所有日志项可以应用一些过滤器进行过滤,比如关键字筛选.通过过滤,某些情况下用户可以对微服务系统当中的数据溯源,从而更好地定位问题根源.用户点击依赖图中调用链经过的服务,会显示该服务在该条调用链上产生的日志以及相应的实例信息和节点信息.

策略 2:不同调用链的对比.同一个请求类型下的不同调用链具有可对比性,因为同一请求类型下的所有调用链具有相同的业务逻辑,正常情况下,调用链之间不会存在太大差异.如果一条调用链长度明显短于其它调用链,而且该条调用链所经过的服务包含于其它调用链所经过的服务,那么该条调用链的最后一个服务可能就是问题所在.在可视化界面当中,用户选择两条调用链以后,两条调用链所经过的服务会以三种颜色进行高亮:共同经过的服务对应一种颜色,各自经过的服务对应不同的颜色,如图 4(b)所示.同时,点击共同经过的服务,会显示服务在不同调用链上产生的日志.日志所包含的信息,比如输入输出数据,可以指引用户思考调用链差异产生的原因,从而辅助用户对问题根源进行定位.

策略 3:服务异步调用分析.服务与服务之间的通信方式可能很复杂,比如服务与服务之间是异步调用,如果没有处理好异步调用的返回顺序,可能会导致微服务系统故障.可视化界面支持对调用链当中的异步调用进行分析.当用户查看某一调用链类型时,可视化界面会调用后台接口(算法 1)分析该调用链类型下的所有调用链,以确定该调用链类型是否存在异步调用.如果存在异步调用,前台界面会将异步调用涉及的服务以边框加粗的方式高亮显示.同时,在所有调用链中实际存在的服务序列以及不同序列对应的错误率,会以柱状图的形式显示,用户点击特定服务序列,可以查看该服务序列对应的所有调用链标识,通过调用链标识可以查看具体的日志链.

策略 4:服务多实例分析.微服务系统当中,每一个服务一般都是多实例运行的,以应对高负载高并发的场景.然而,如果无法保证不同服务实例的状态一致性,或者不同实例对应于服务的不同版本,都可能会导致问题.可视化界面中,如果用户查看某一请求类型或者调用链类型,会以柱状图的方式展示该类型下所有调用链涉及的微服务以及微服务的错误率.点击某一服务以后,会显示该服务被调用的所有服务实例以及相应的错误率.如果存在某个服务实例错误率远高于或者远低于其它实例,那导致系统故障的根源很可能就是服务多实例,没有做好

状态同步或者某一版本的服务存在问题.

策略 5:调用链分段.实际生产环境中的微服务系统,在处理单个请求时,可能会涉及到成百上千个微服务,产生的调用链因为长度较长,不适合进行完整的分析.需要找到一种方式,对调用链进行分段,从而允许用户以段为单位进行调用链分析.可视化界面的实现方式是,首先需要用户输入一个整数值,该值用于根据服务之间的调用关系对服务依赖图分块,调用链中处于相同分块的微服务即形成一段.由于处于相同分块的微服务调用是比较频繁的,所以块内部服务调用出错的概率理论上低于块与块之间服务调用出错的概率.通过这种方式,可以指引用户重点查看不同块之间服务调用情况,辅助用户查找故障根源.

Fig.4 (a)Visualization Page

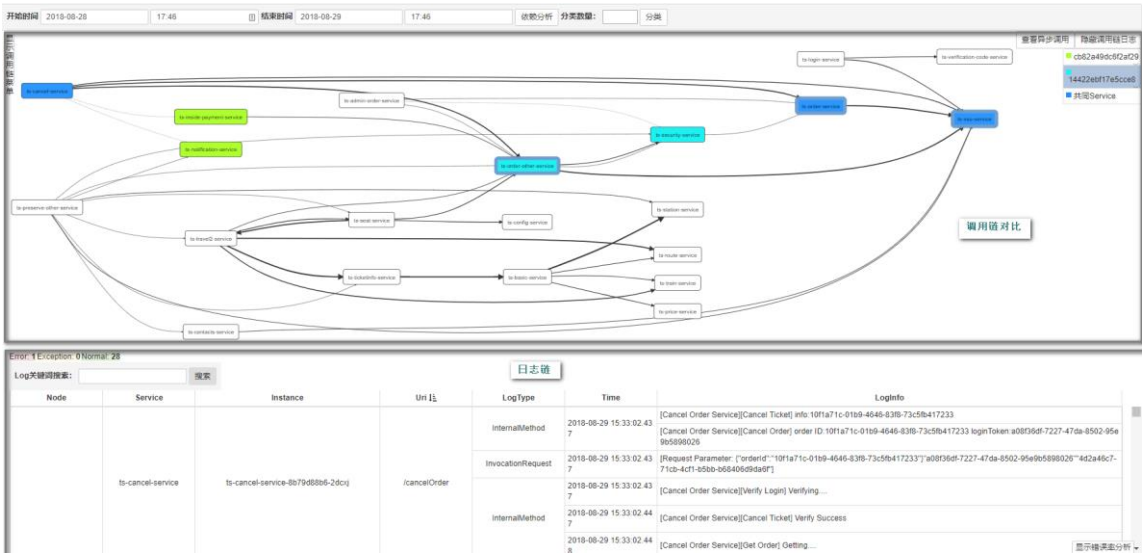

Fig.4 (b)Strategy Support

图 4: (a) 可视化界面 (b)策略支持

2.5 算法

2.5.1 算法 1:服务异步调用分析算法

为了对微服务系统中的服务异步调用进行分析,首先给出如下定义.微服务集合$S = \{s_1, \dots, s_n\}$,调用链$L = (s_i, \dots, s_m | s_i \in S,\ 1 \le i, m \le n)$,调用链集合$C = \{l_i, \dots, l_p | 1 \le i, p \le A_{m-1}^{m-1}, \forall c_k \in C, \forall l_i, l_j \in c_k : |l_i| = |l_k|, l_i$和$l_j$中$\forall s_q$,其数量相等}.对于上述定义,有如下性质:

1. 对于任意服务都可以出现在调用链任意位置.
2. 调用链有序且服务可重复.
3. 同一调用链集合中,任意两个调用链中任意一个服务的调用次数相同.
4. 任意调用链都是以同步调用开始

算法 1:**服务异步调用分析算法**

输入:C[m][n] //调用链集合二位数组,每一行为一个调用链.
输出: AsynServices //异步调用服务集合

```
Begin
/* 以第一个调用链 C[1][n]为基础,找出二维数组中元素相同的列,即找出同步调用的服务*/
01: for i ← 1 to m-1
02:     for j ← 0 to n-1
03:         do if C[i][j] ≠ C[1][j]
04:             then Count[j]++
05:     end for
06: end for
/*对于输入中不相同的列,将其列号记录到异步调用下标集合 asynIndics 中,相同的列,向其对应的服务记录到同步调用服务集合 syncServices 中*/
07: for i ← 0 to n-1
08:     do if Count[i] = 0
09:         then syncServices.add(C[1][i])
10:     else
11:         asynIndics.add(i)
12: end for
/*同步调用集合 syncServices 中的服务也可能在异步调用中出现,所以现在判断,同步调用的服务是否在输入中异步调用范围中出现,若出现,则为异步调用,反之则为同步调用.*/
13: for i ← 0 to m-1
14:     for j in asynIndics
15:         do if syncServices.contains(C[i][j])
16:             then syncServices.remove(C[i][j])
17:     end for
18: end for
/*最后将该调用链的服务集合中去掉同步调用集合,剩下的就为异步调用服务*/
19: Services ← Set(C[1][n])
20: AsynServices ← Services.remove(syncService)
End
```

算法分析的服务调用链基于日志产生,按该服务被调用的时间进行排序.异步调用的起始服务后的调用都会被归结为异步调用,比如以下微服务调用:

$$s_1 \to s_2 \to \begin{Bmatrix} s_3 \to s_2 \\ s_5 \to s_6 \to s_7 \end{Bmatrix} \to s_8$$

s_1和s_2为起始的同步调用,s_3和s_5为异步调用的起始服务,s_4, s_6, s_7为异步调用中调用的服务,最后调用s_8服务.最终形成的调用链为$s_1 \to s_2 \to (\text{异步调用}) \to s_8$.对于异步调用的部分,保持每个异步链的相对序列不变,然后组合.对于上述调用可得调用链:

$$s_1 \to s_2 \to s_3 \to s_5 \to s_2 \to s_6 \to s_7 \to s_8$$
$$s_1 \to s_2 \to s_3 \to s_5 \to s_6 \to s_2 \to s_7 \to s_8$$
...

最终在异步调用部分出现的服务,都会被归类到异步调用范围,上述调用链中异步调用范围为:

$$\{s_2, s_3, s_5, s_6, s_7\}$$

2.5.2 算法 2:微服务依赖图分块算法

谱聚类算法[39]从图论中演化而来,后来在聚类中得到了广泛应用.它的主要思想是把所有的数据看做空间中的点,这些点之间使用边连接.距离较远的两个点之间的边权重值较低,而距离较近的两个点之间的边权重值较高,通过对所有数据点组成的图进行切图,让切图后不同的子图间边权重和尽可能的低,而子图内的边权重和尽可能的高,从而达到聚类的目的.

在微服务系统中,服务与服务之间的调用形成了调用链,不同的调用链之间形成了服务调用网络.虽然服务与服务之间的调用是有向的,但是当我们在分析服务与服务之间的调用关系时,可以忽略其调用方向,因为我们只关注服务与服务之间的紧密程度.我们将调用链形成的调用网络映射为无向图 $G(V,E)$,V 表示无向图中点的集合($v_1,v_2,...v_n$),即调用网络中的微服务的集合,E 表示无向图中边的集合($e_1,e_2,\ldots,e_n$),即调用网络中微服务之间的调用的集合.我们定义权重 w_{ij} 为点 v_i 和 v_j 之间的权重,即服务 i 与服务 j 间之间的调用次数,所以 $w_{ij}=w_{ji}$.基于以上定义,利用谱聚类算法,本文实现了微服务依赖图分块算法.

算法 2:微服务依赖图分块算法
输入:微服务调用网络映射的邻接矩阵 W,聚类数量 k **输出:**分类点集$A_i,\ldots,A_k, A_i = \{j \mid y_i \in C_i\}$
Begin 01: 计算归一化拉普拉斯矩阵L_{sym} 02: 计算前 k 个L_{sym}的特征向量 $u_1,\ldots,u_k$ 03: 令$U \in \mathbb{R}^{n\times k}$,$U$包含列向量 $u_1,\ldots,u_k$ 04: 矩阵$T \in \mathbb{R}^{n\times k}$通过将行归一化为范数 1,即赋值$t_{ij} = u_{ij}/(\sum_k u_{ik}^2)^{1/2}$ 05: for i = 1,…, n, 令向量$y_i \in \mathbb{R}^k$为矩阵 T 的第 i 行 06: 将点集$(y_i)_{i=1,\ldots,n}$使用 k-means 算法聚类到分类$C_1,\ldots,C_k$ End

3 实验

3.1 实验准备

本文实验使用的 TrainTicket 是一个开源的模拟 12306 火车票业务的系统,共有 41 个业务微服务,包括注册登录、订票、退票、行车路线查询等主要流程.之所以选择这个测试系统,主要是由于各大开源软件社区中能够找到的用于实验的微服务系统数量非常有限且规模普遍很小,TrainTicket 是目前可以使用的规模相对较大的一个,除此之外,它还包括消息中间件服务、分布式缓存服务和数据库服务等基础服务,更加接近现实中的微服务生产环境.

本文对 TrainTicket 进行了改造、通过 Spring AOP 拦截控制台日志、添加与调用链相关的 traceId 等信息,通过这种方法将日志与调用链相关联.随后,分别向改造后的系统中注入了四种不同类型的故障,故障的类型和重现过程如表 1 所示.

Table 1 Table of Experiment Fault

表 1 实验故障列表

故障序号	故障类型	故障重现过程
故障一	有异常抛出的简单错误	在用户退票过程中注入的一个故障,当退票用户为 VIP 用户时某个验证身份的参数传错导致退票失败.
故障二	无异常抛出的逻辑错误	计算车票价格的模块由于逻辑错误导致用户会在页面上看到某条火车线路上二等座价格比一等座还高出很多,由于是单纯的业务逻辑错误,整个过程不会打印异常日志.
故障三	多实例状态不一致	当管理员锁定了两个车站后,所有关于这两个车站的车票都将无法退票.但由于锁定状态仅存储在实例内部,多个实例之间没有做状态同步会出现锁定车站后仍然可以退票的情况.
故障四	异步调用顺序不一致	系统退票过程中退票服务会异步发送两个请求给退款服务和订单服务,而这两个服务都会修改数据库中订单的状态,只有退款服务首先完成对订单状态的修改才能够成功退票,否则退票失败.

3.2 实验过程与结果

实验参与者为两个对测试系统业务流程有一定了解的软件开发人员.实验首先要求参与者根据故障描述分别重现上述四个故障,再交替使用两种工具对故障根源进行排查.对比实验采用查看 Zipkin 收集的调用链信息与 Kibana 存储的日志信息相结合的故障排查方法,故障和参与者使用工具的对照如表 2 所示.

Table 2 Tool-Fault Table of Experiment

表 2 实验工具-故障对应表

参与者	故障一	故障二	故障三	故障四
参与者 A	Zipkin + Kibana	LogVisualization	Zipkin + Kibana	LogVisualization
参与者 B	LogVisualization	Zipkin + Kibana	LogVisualization	Zipkin + Kibana

在进行故障排查时,要求参与者记录定位到故障服务所需的时间,使用本文构建的工具时需要记录运用到了哪几种类型的故障排查策略以及策略使用的先后顺序,最终得到的实验结果如表 3 所示.

从表 3 的数据可以总结出,除了故障二,使用本文构建的工具 LogVisualization 定位故障的时间比结合使用 Zipkin 和 Kibana 要短很多,那是因为在故障二中,使用 Zipkin + Kibana 的参与者并没有能够找到故障的根源.除此之外,我们可以看到参与者在实际故障排查过程中的策略选择和顺序与前文总结是相符的、工具提供的几种策略对于特定故障的排查是有用且高效的.

本文在参与者完成全部四个故障的实验后,围绕工具的使用感受、两种工具的对比等方面对他们进行了简短的访谈.两位参与者均表示 LogVisualization 学习起来比较容易、上手快,图标和颜色的高亮能够帮助他们快速定位出错的服务、精确到异常的日志.与 Zipkin 和 Kibana 相结合的方法比较,LogVisualization 提供了对于整个系统运行状态的总体概览,可以直观地看到各个业务流程、服务乃至实例的错误率和比重,可以引导他们逐步深入直至找到故障根源.将调用链和日志相结合、显示在同一界面避免了反复切换 Zipkin 和 Kibana、查找对应错误日志的麻烦,更加准确和高效.同时,工具提供的异步调用检测、不同调用链的对比分析等功能为故障排查提供了新的思路,也能够更好地监控服务之间的调用情况.

Table 3 Experiment Result

表 3 实验结果

故障	工具	是否定位到故障服务？（是/否）	定位的总时间（分钟）	使用策略及先后顺序（1->2->3）	能够判断/猜测出故障的根本原因(是/否)
故障一	LogVisualization	是	10	1->2->3	是
	Zipkin + Kibana	是	18	-	是
故障二	LogVisualization	是	20	1->2	是
	Zipkin + Kibana	是	16	-	否
故障三	LogVisualization	是	10	1->2->3	是
	Zipkin + Kibana	是	16	-	是
故障四	LogVisualization	是	4	1->2->4	是
	Zipkin + Kibana	是	12	-	是

最后,参与者也认为本文构建的工具目前还有一些可以改进之处,例如应当在右上角添加异步调用的标识、方便工具使用者明确其意义.另外,如果可以借鉴 Kibana 中日志的查找和过滤功能,也对日志内容中的字段进行提取和分类,可以更好地实现数据的查找和溯源.

4 讨论

本文实现的原型工具依赖于业务微服务系统在输出日志时,往日志当中注入相应的调用链信息(例如,针对 Java 应用可以利用 AOP 方式增强日志信息),对微服务系统有一定的侵入性.一种解决方法是改造 Istio,监控微服务之间的网络通信,自动往服务调用产生的日志当中注入调用链信息.同时,用于测试的 TrainTicket 微服务系统所包含的服务数量较少,无法产生较长的调用链,所以调用链分段的策略无法在实验当中被验证.因为参与实验的人员需要对实验微服务系统有一定了解,限于客观条件,参与实验的人员数量较少.

5 结论与展望

本文实现了一个原型工具 LogVisualization,可以将微服务系统的日志和调用链关联展示.工具的可视化界面提供了五种不同的策略支持:单条调用链日志查看,不同调用链对比,服务异步调用分析,服务多实例分析以及调用链分段.用户可以灵活组合不同的策略,对导致微服务系统故障的根源定位.同时,通过在实际微服务系统当中的应用以及与 zipkin 和 ELK Stack 的实验对比,证明了原型工具的有用性和有效性.

由于目前的原型工具无法自动往日志信息当中注入相应的调用链信息,对业务微服务系统有一定的侵入性,将来的一个工作方向是改造 Istio,通过拦截服务与服务之间的网络通信,自动往日志信息中注入调用链信息.同时,将来会考虑将调用链中的延迟信息,集群的资源使用情况等也以合适的方式展示在可视化界面中,更好地支持用户对导致微服务系统故障的根源定位.

References:

[1] J. Lewis and M. Fowler, “Microservices,” 25 Mar. 2014; martinfowler.com/articles/microservices.html

[2] O. Zimmermann, “Microservices Tenets: Agile Approach to Service Development and Deployment,” Computer Science—Research and Development, 2017, 32(3-4):301–310.

[3] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. 2017. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In 2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017. 21–30. DOI:https://doi.org/10.1109/ICSA.2017.24.

[4] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In 36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016. 57–66. DOI:https://doi.org/10.1109/ICDCS.2016.11.

[5] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam,Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017. 253–262. DOI:https://doi.org/10.1109/ICSE-SEIP.2017.32.

[6] https://gotocon.com/dl/goto-amsterdam-2016/slides/RuslanMeshenberg_MicroservicesAtNetflixScaleFirstPrinciplesTradeoffsLessonsLearned.pdf.

[7] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis and Stefan Tilkov. Microservices: The Journey So Far and Challenges Ahead. IEEE Software, 2018, 35(3):24-35. DOI: https://doi.org/10.1109/MS.2018.2141039

[8] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao.2018. Poster: Benchmarking Microservice Systems for Software Engineering Research. In Proc. International Conference on Software Engineering: Companion Proceeedings (ICSE’18). 323–324.

[9] http://www.thefabricnet.com/application-delivery-service-challenges-in-microservices-based-applications/

[10] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. 2018. Delta debugging microservice systems. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). ACM, New York, NY, USA, 802-807. DOI: https://doi.org/10.1145/3238147.3240730.

[11] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. IEEE Trans. Software Eng. 2002, 28(2):183–200.

[12] Zipkin. https://zipkin.io/

[13] Jaeger. https://www.jaegertracing.io/

[14] ELK Stack. https://www.elastic.co/elk-stack

[15] Shiviz. https://bestchai.bitbucket.io/shiviz/

[16] C. Pahl and P. Jamshidi, “Microservices: A Systematic Mapping Study,” Proc. 6th Int’l Conf. Cloud Computing and Services Science(CLOSER 16), 2016, 137–146.

[17] Sara Hassan and Rami Bahsoon. 2016. Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap. In IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016. 813–818.

[18] Andr´e de Camargo, Ivan Luiz Salvadori, Ronaldo dos Santos Mello, and Frank Siqueira. 2016. An architecture to automate performance tests on microservices. In Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services, iiWAS 2016, Singapore, November 28-30, 2016. 422–429

[19] Robert Heinrich, Andr´e van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. 2017. Performance Engineering for Microservices: Research Challenges and Directions. In Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L’Aquila, Italy, April 22-26, 2017. 223–226

[20] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In 36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016. 57–66.

[21] Gerald Schermann, Dominik Sch¨oni, Philipp Leitner, and Harald C. Gall. 2016. Bifrost: Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies. In Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016. 12.

[22] Wilhelm Hasselbring. 2016. Microservices for Scalability: Keynote Talk Abstract. In Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016. 133–134

[23] Sander Klock, Jan Martijn E. M. van der Werf, Jan Pieter Guelen, and Slinger Jansen. 2017. Workload-Based Clustering of Coherent Feature Sets in Microservice Architectures. In 2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017. 11–20

[24] Philipp Leitner, J¨urgen Cito, and Emanuel St¨ockli. 2016. Modelling and managing deployment costs of microservice-based cloud applications. In Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC 2016, Shanghai, China, December 6-9, 2016. 165–174

[25] Sara Hassan and Rami Bahsoon. 2016. Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap. In IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016. 813–818

[26] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D.Ernst. 2016. Debugging distributed systems. Commun. ACM, 2016, 59(8):32–37.

[27] http://opentracing.io/documentation/

[28] https://storage.googleapis.com/pub-tools-public-publication-data/pdf/36356.pdf

[29] https://www.jaegertracing.io/docs/

[30] https://sematext.com/blog/jaeger-vs-zipkin-opentracing-distributed-tracers/

[31] https://flume.apache.org/

[32] https://github.com/facebookarchive/scribe/wiki

[33] Liu Kai. Architecture analysis and application of massive data log system. Journal of Changchun University of Technology, 2016, 37(6):581-586 . DOI: https://doi.org/10.15923/j.cnki.cn22-1382/t.2016.6.13

[34] Chen JJ, Liu HH. Distributed ELK log analysis system based on Kubernetes. Electronic Technology & Software Engineering, 2016(15):211-212

[35] https://techbeacon.com/3-reasons-why-you-should-always-run-microservices-apps-containers

[36] https://blog.buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/

[37] Kubernetes. https://kubernetes.io/

[38] Istio. https://istio.io/

[39] von Luxburg, Ulrike. A tutorial on spectral clustering. Statistics & Computing, 2007, 17(4):395-416. DOI: https://doi.org/10.1007/s11222-007-9033-z

附中文参考文献:

[33] 刘锴. 海量数据日志系统架构分析与应用[J]. 长春工业大学学报:自然科学版, 2016, 37(6):581-586

[34] 陈建娟, 刘行行. 基于 Kubernetes 的分布式 ELK 日志分析系统[J]. 电子技术与软件工程, 2016(15):211-212.