软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(7):3167−3187 [doi: 10.13328/j.cnki.jos.006523]

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

面向 Java 微服务系统的透明请求追踪及采样方法*

黄梓程, 陈鹏飞, 余广坝, 陈泓仰

(中山大学 计算机学院, 广东 广州 510006)

通信作者: 陈鹏飞, E-mail: chenpf7@mail.sysu.edu.cn

摘　要: 微服务因其敏捷的开发方式、快速的部署方式, 逐渐成为以云为基础的软件系统的主流架构方式之一. 但是, 微服务系统结构复杂, 动辄上百个服务实例, 而且服务之间的调用关系异常复杂, 当微服务系统中出现异常时, 难以定位故障根因. 为了解决这个问题, 端到端请求追踪 (trace) 成为微服务系统监控的标配. 然而现有的分布式请求追踪实现方式对应用程序具有侵入性, 严重依赖于开发者对请求追踪的经验, 无法在运行时控制追踪功能的开启和关闭. 这些不足不仅会增加开发者的负担, 而且限制了分布式请求追踪技术的实际应用. 设计并实现对程序开发者透明的请求追踪系统 Trace++, 能够自动生成追踪代码, 利用动态代码插桩技术将追踪代码注入到运行中的应用程序. Trace++对程序低侵入, 对开发者透明, 能够灵活控制追踪功能的开启和关闭. 此外, Trace++的自适应采样方法有效减少了请求追踪产生的开销. 在微服务系统 TrainTicket 上的实验结果证明, Trace++能够准确发现服务依赖关系. 在开启请求追踪时, 性能开销接近于源代码插桩, 在关闭请求追踪时无性能开销. 此外, Trace++的自适应采样方法在采样到具有代表性样本的同时减少了 89.4% 的追踪数据.

关键词: 请求追踪; 动态插桩; 采样; 微服务; 云计算

中图法分类号: TP311

中文引用格式: 黄梓程, 陈鹏飞, 余广坝, 陈泓仰. 面向Java微服务系统的透明请求追踪及采样方法. 软件学报, 2023, 34(7): 3167–3187. http://www.jos.org.cn/1000-9825/6523.htm

英文引用格式: Huang ZC, Chen PF, Yu GB, Chen HY. Transparent Request Tracing and Sampling Method for Java-based Microservice System. Ruan Jian Xue Bao/Journal of Software, 2023, 34(7): 3167–3187 (in Chinese). http://www.jos.org.cn/1000-9825/6523.htm

Transparent Request Tracing and Sampling Method for Java-based Microservice System

HUANG Zi-Cheng, CHEN Peng-Fei, YU Guang-Ba, CHEN Hong-Yang

(School of Computer Science and Engineering, SUN Yat-Sen University, Guangzhou 510006, China)

Abstract: Microservice is becoming the mainstream architecture of the cloud-based software systems because of its agile development and rapid deployment. However, the structure of a microservice system is complex, it often has hundred of service instances. Moreover, the call relationship between services is extremely complex. When an anomaly occurs in the microservice system, it is difficult to locate the root causes of the anomaly. The end-to-end request tracing method becomes the standard configuration of a microservice system to solve this problem. However, current methods of distributed request tracing are intrusive to applications and heavily rely on the developers' expertise in request tracing. Besides, it is unable to start or stop the tracing functionality at runtime. These defects not only increase the burden of developers but also restrict the adoption of distributed request tracing technique in practice. This study designs and implements a transparent request tracing system named Trace++, which can generate tracing code automatically and inject the generated code into the running application by using dynamic code instrumentation technology. Trace++ is low intrusive to programs, transparent to developers, and can start or stop the tracing functionality flexibly. In addition, the adaptive sampling method of Trace++ effectively reduces the cost

* 基金项目: 广东省重点领域研究计划 (2020B010165002); 国家自然科学基金青年项目 (61802448); 广东省自然科学基金面上项目 (2019A1515012229); 广州市基础与应用基础研究项目 (202002030328)
收稿时间: 2021-02-08; 修改时间: 2021-08-03; 采用时间: 2021-10-20; jos 在线出版时间: 2022-11-30
CNKI 网络首发时间: 2022-12-01

of request tracing. The results of the experiments conducted on TrainTicket, a microservice system, show that Trace++ can discover the dependencies between services accurately and its performance cost is close to the source code instrumentation method when it starts request tracing. When the request tracing functionality is stopped, Trace++ incurs no performance cost. Moreover, the adaptive sampling method can preserve the representative trace data while 89.4% of trace data are reduced.

Key words: request tracing; dynamic instrumentation; sampling; microservice; cloud computing

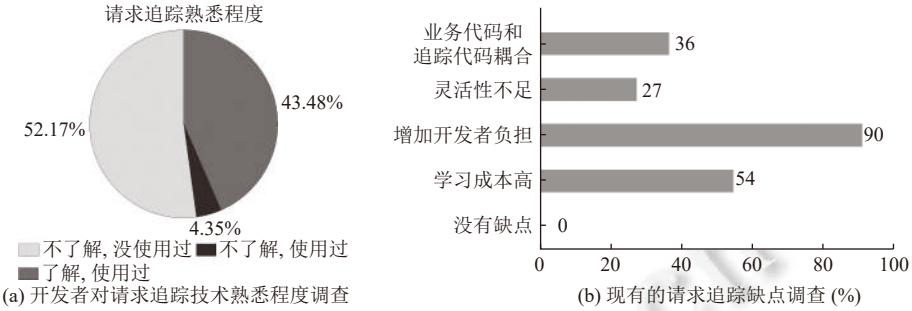
微服务架构因其敏捷开发、连续部署、快速发布等优点, 满足了 IT 系统日渐频繁的变更需求, 成为当下主流的软件架构. 在微服务架构下, 应用程序被拆分为一组松耦合、功能最小化的服务, 服务与服务之间通过轻量级的通信机制进行沟通[1]. 微服务系统运行在分布式的网络环境中, 客户端发起的一次请求操作, 需要经过多个模块、多个中间件、多台机器. 服务的细粒度化虽然使微服务的开发和部署具有强大的灵活性, 但也使服务之间的依赖关系变得复杂, 对运维人员理解系统行为造成了极大障碍, 特别是在系统出现异常时, 由于下游服务的故障会导致上游服务也失效, 形成级联效应[2], 这对定位异常根因带来了严峻的挑战[3].

请求追踪是指在请求进入分布式系统到请求完成并返回响应的这段时间内, 将系统中因处理请求而产生的因果相关的事件相关联, 生成请求在系统中执行路径的过程[4]. 在微服务系统这种分布式的环境下, 对请求进行追踪, 记录请求的执行路径和处理时间, 从而提供详细的微服务系统运行时视图, 有助于运维人员了解服务间的交互, 对问题诊断提供重要的参考信息. 端到端请求追踪在学术界和工业界具有很高的关注度[4,5], 这是因为发现和解决分布式系统中的性能问题一直以来都非常具有挑战性. 传统的监控手段如日志和指标只能监控单个进程或是单台机器的状态, 在分布式的场景中远远不够, 需要能够关联起不同组件的监控方法, 提供组件间依赖以及协同的视图[6,7]. 请求追踪技术被广泛地用于分布式系统的负载建模[8]、异常检测[3,9]、性能监控和诊断[10−14]、资源使用归因[15,16]等领域. 对于微服务系统来说, 请求追踪已经成为监控的标准配备. 针对微服务系统的请求追踪场景, 已存在多种开箱即用的工具, 例如 Zipkin、Jaeger, 使用这些工具实现请求追踪需要对程序源码进行修改. 这种方式有 4 点不足: 第一, 需要获得程序的源代码. 这在一些情况下不能满足, 在构建微服务系统的时候使用第三方组件是一种很常见的做法, 这些被复用的组件不一定提供源代码. 第二, 需要理解业务代码和追踪代码. 实现请求追踪需要在程序中设置追踪点, 当程序运行到追踪点时, 追踪点上的追踪代码记录下执行轨迹. 在哪个位置埋点和在埋点处插入什么代码, 都是开发者需要明确和解决的问题, 这无疑会增加开发者的负担. 第三, 缺少对追踪功能的灵活控制, 现有的请求追踪功能启动后无法关闭, 不利于在生产环境中使用. 第四, 后期升级和维护追踪代码成本高, 现有的设计方案造成业务代码和追踪代码的紧密耦合, 后续更新追踪逻辑时需要同时更新原有业务代码. 另外, 现有的请求追踪系统使用固定的采样率, 采样时对正常追踪数据和异常追踪数据一视同仁, 会导致数量稀少但对运维监控有重要价值的异常追踪数据丢失.

相比人工修改源程序实现请求追踪的方法, 自动化、透明的插桩方法能够克服上述不足. 自动化插桩方法通过二进制重写或者依赖函数库打桩来在程序中添加追踪点和追踪代码[5], 像 Aspect-oriented programming[17]、hotpatching 等技术被用于实现代码动态插桩[18,19], 从而避免了对源码的修改, 是一种对程序开发者透明的方法, 适用范围广, 通用性较好. 同时, 自动化的插桩方法将应用程序视为黑盒, 可以在不了解程序代码的情况下构建出执行轨迹[20−23], 减少了开发者实现请求追踪的学习成本. 基于自动化插桩的请求追踪方法比起基于源程序插桩的请求追踪方法具有众多优势. 但另一方面, 由于自动化插桩的请求追踪方法使用统计推理的方式, 利用从各个组件收集的数据来构建请求追踪路径[20,22,24−26], 在准确性和完整性上比侵入式的源代码插桩方法差. 非透明的源代码插桩方法对每一个请求产生一个唯一的标识符, 并且让标识符伴随着请求访问不同的组件传递, 以此记录下请求追踪路径. 现有的请求追踪方法在代码插桩的透明性和准确性上做权衡, 没有一种方法能够兼具自动化插桩和源代码插桩的优点.

虽然请求追踪已成为微服务监控技术体系下的“一等公民”, 但是对请求追踪使用情况的问卷调查表明开发者对请求追踪的了解和使用程度处于比较初级的阶段, 图 1 展示了部分的调查结果. 在调查中, 23 位被调查者包括工业界的开发者和学术界做微服务相关研究的研究人员, 有 12 位被调查者表示对请求追踪并不了解, 11 位使用过请求追踪工具的被调查者中, 有 4 位被调查者是直接在源代码中插桩追踪代码, 有 5 位被调查者使用了具有追踪功能的组件. 在回答“微服务系统现有请求追踪方式的缺点”这个问题时, 被调查者最多次指出的缺点是增加了

开发者的负担, 要兼顾业务代码和追踪代码, 其次是学习成本高, 要了解请求追踪的概念和模型. 值得一提的是, 23 位被调查者都认为需要在程序运行时控制请求追踪功能的开启和关闭, 达到降低性能开销的目的.



(a) 开发者对请求追踪技术熟悉程度调查 (b) 现有的请求追踪缺点调查 (%)

图 1 请求追踪调查

针对现有追踪方法的不足, 以及为了满足开发者对请求追踪的实际需求, 本文设计了基于动态代码插桩的自动化请求追踪工具 Trace++, 能够赋予 Java 微服务应用程序请求追踪能力. Trace++区别于现有请求追踪工具的地方在于: 利用动态代码插桩的方式实现了与源代码插桩实现的请求追踪同等的准确性, 也具有自动化插桩方法的低侵入性, 对开发者透明, 不需要开发者有任何请求追踪的知识和经验, 能够在程序运行时开启和关闭请求追踪功能. 同时考虑到大量的请求追踪数据产生的存储开销问题, Trace++实现了自适应的采样方法, 根据追踪数据的延时分布来计算采样率, 在保证带有异常延时的追踪数据不被丢失的情况下, 压缩追踪数据, 减少存储空间.

实现 Trace++需要解决以下挑战.

(1) 使用动态代码插桩实现低侵入的请求追踪, 在程序运行时注入追踪代码, 以及移除追踪代码.

(2) 微服务提倡使用轻量级的通信机制, 本文选取 http、gRPC 这两种目前使用广泛的通信协议实现追踪.

(3) 对开发者透明, 不需要开发者有请求追踪和微服务应用程序架构的相关知识.

(4) 容器是微服务应用程序的运行载体, Trace++需要具备对容器中应用程序请求追踪的能力.

(5) 通过自适应采样降低追踪数据的存储开销, 带有异常时延信息的追踪数据在采样中不能被丢失.

本文在开源的微服务系统 TrainTicket[27]进行测试, 使用 Trace++对 TrainTicket 进行请求追踪赋能, 使用混沌工程[28]的方法在 TrainTicket 中注入故障, 收集追踪数据并对追踪数据进行采样. 实验结果表明, Trace++能有效地获取请求的执行路径, 追踪功能开启时对请求响应时间的影响接近于源代码插桩方法, 关闭追踪功能时对应用程序的性能无影响, 自适应采样在保留有代表性的追踪数据的同时减少了 89.4% 的追踪数据.

Trace++具有以下创新点.

(1) 自动化透明请求追踪. 不需要开发者对请求追踪技术有任何了解, 让开发者专注于业务逻辑.

(2) 动态代码插桩实现低侵入性. 程序运行时动态插桩追踪代码, 达到与源码插桩同等的准确性和完整性.

(3) 灵活的追踪功能控制. 可以随时开启、关闭追踪功能, 不影响程序的正常执行, 关闭追踪时无额外开销.

(4) 使用自适应采样算法减少请求追踪的开销. 根据追踪数据的延迟分布动态调整采样率, 确保异常延迟数据不丢失.

本文第 1 节介绍背景并分析比较请求追踪和自适应采样的相关工作. 第 2 节介绍 Trace++的原理和设计. 第 3 节介绍 Trace++的具体实现. 第 4 节是案例学习. 第 5 节介绍实验设计和实验结果. 第 6 节总结本文, 并给出下一步的工作.

1 背景及相关工作

1.1 请求追踪技术背景

当前针对微服务请求追踪已有多种方式, 例如 Dapper[14], OpenTracing, SkyWalkin 和 OpenTelemetry 等. 虽然

不同请求追踪方式在追踪数据的格式上不完全一致, 但它们实现请求追踪的基本思想都是将请求与一个全局唯一的标识符关联起来, 当请求跨越不同的进程时, 让标识符跟随请求进行传播, 以此关联起追踪过程中产生的数据.

一条完整的请求追踪路径通常使用 trace、span、spancontext 等数据模型来构建完成. 其中, trace 表示一次请求追踪过程, 代表一次请求的完整执行路径. 使用唯一的 trace_id 标识. Span 是构建 trace 的基本单元, 表示系统中具有开始时间和执行时长的逻辑运行单元, 使用唯一的 span_id 标识. Spancontext 包含 span 的基本信息, 随着请求传播, 依靠它来建立不同服务 span 间的关联关系. 图 2 展示了跨服务传播 spancontext 的过程. OpenTracing 将 spancontext 放置在请求中, 传播到调用的服务 B, 服务 B 将服务 A 的 span_id 作为 parent_id, 构建服务间的调用关系.

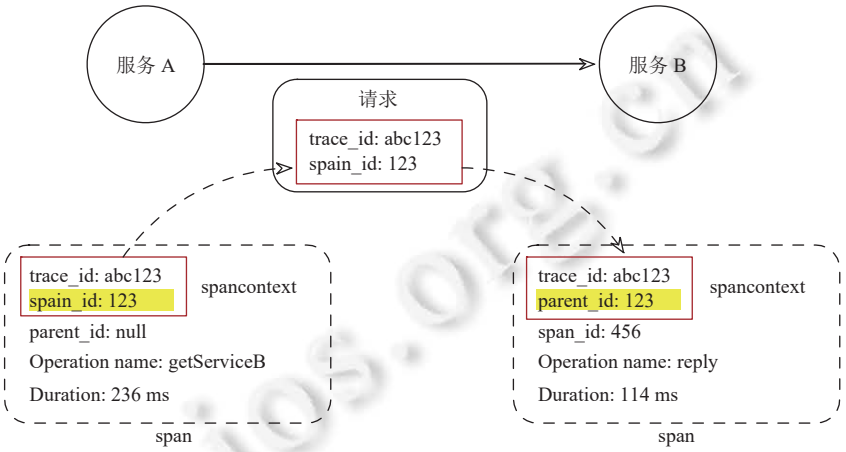


图 2 跨服务传播 spancontext

请求追踪具有一套固定的操作流程, 应用开发者据此在应用程序中插桩代码. 从而实现当服务接收到请求时, 执行相应的处理逻辑并生成 span, 记录服务调用开始时间戳. 如果需要调用下游服务, 则把 span 的 spancontext 伴随请求传递到下游服务, 在下游服务中提取 spancontext, 处理请求并使用 spancontext 生成关联 span, 如果需要继续调用下游服务, 则重复以上过程. 当服务调用返回时, 关闭 span 使得 span 记录下调用结束的时间戳. 以图 2 中的服务调用关系为例, 图 3 展示了其请求追踪的流程.

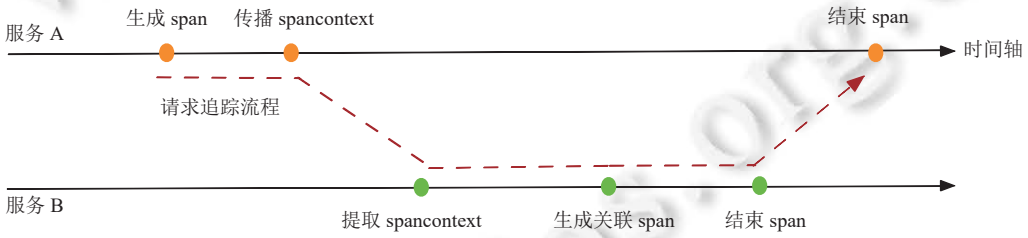


图 3 请求追踪流程

1.2 请求追踪技术相关工作

请求追踪技术在分布式系统的故障诊断、性能调优、系统理解等运维场景下应用广泛, 在学术界和工业界出现了大量的请求追踪研究工作和请求追踪系统. 根据实现方式可以将现有工作分为两类, 基于源代码插桩的请求追踪方法和基于非源代码插桩的请求追踪方法.

1.2.1 源代码插桩实现的请求追踪

现代软件在架构上按照功能进行分层. 源代码插桩的方法侵入软件堆栈的不同层次, 在代码中添加追踪点, 追踪点上记录系统的状态, 用特定的标识符串联起不同追踪点产生的信息, 构建不同事件粒度的追踪路径. 源代码插桩可分为应用层插桩、中间件层插桩、系统层插桩.

应用层插桩直接修改分布式软件系统的源代码. StarDust[16]在应用代码中插桩代码来生成请求的活动记录, 活动记录包含时间戳, 以及 CPU、缓存、网络、磁盘等资源的使用信息, 可以用来审计资源使用情况和请求延迟. StarDust 使用 breadcrumb 信息关联起与某一请求关联的所有活动记录, breadcrumb 保存在线程私有数据中, 并在请求跨越机器时伴随 RPC 调用传递. Canopy[29]提供了一系列支持不同执行模型的 API 供开发者在源码中使用, 以实现请求追踪. 这种方式不需要开发者手工构建和报告追踪数据的组成元素, 降低了在系统中引入请求追踪的门槛, 避免插桩时可能出现的错误. MilliScope[30]通过 Web 服务来生成请求的活动记录和请求 ID, 用请求 ID 串联起在不同 Web 服务生成的活动记录, 请求 ID 以 URI 参数或者 SQL comment 的形式在不同服务间传递.

应用层插桩的实现需要结合应用程序的特点, 通常是不可迁移的. 相比之下, 中间件层插桩的方法更为通用, 只要在插桩过的中间件上构建应用程序, 就可以实现请求追踪. 但是, 从追踪的粒度上来说, 应用层插桩具有函数级的追踪粒度, 而中间件层插桩的方法捕获不到应用程序内部的执行状况, 追踪粒度只能达到服务级. Pinpoint[31]支持对 J2EE 中间件插桩, 从而使得运行在 J2EE 平台上的应用程序无需修改就可以获得请求追踪的能力, 具有模块级别追踪粒度. Dapper[14]将对应用层透明作为请求追踪的设计目标之一, 并且为了对应用开发者也实现尽可能的透明, Dapper 将追踪点设置在 RPC 框架中, 使得能够对应用程序的 RPC 进行追踪. Reynolds 等人[12]对 Mace[32]做修改, 使得用 Mace 构建出来的分布式系统能够追踪事务的开始和结束、信息的发送和接收.

系统层插桩对系统调用进行拦截和篡改来实现对运行在操作系统上的分布式应用程序进行追踪. Cuong 等人[33]实现了一个包含网络 IO 操作函数的共享库, 并用 LD_PRELOAD 机制实现对 Libc 中的 send、recv 等网络 IO 操作相关的系统调用做替换, 使得能够在调用这些系统调用时记录下追踪信息. vPath[23]对操作系统内核和虚拟机监控器进行修改, 监视线程在 TCP 连接上的数据发送和接收的系统调用, 从而追踪节点内部和节点之间的因果. BorderPatrol[21]同样也使用 LD_PRELOAD 机制对 Libc 中的一些函数进行替换, 实现对连接创建操作 (open, socket, pipe 等)、描述符修改操作 (close, dup, fcntl)、信息读取和写入操作进行追踪. 类似地使用 LD_PRELOAD 进行系统调用拦截和替换的追踪系统还包括 WAP5[22].

源代码插桩的方式能够构建出完整、准确的请求执行路径, 并且可以有选择地对系统的状态进行记录, 得到不同粒度的追踪视图. 但这种方法对应用程序, 或者中间件, 或者操作系统具有侵入性, 使用时需要满足一些条件, 比如需要源码、要求程序使用动态链接, 对开发者不能实现完全的透明.

1.2.2 非源代码插桩实现的请求追踪

源代码插桩实现请求追踪要求程序开发者熟悉链路追踪的方式和相关的代码库. 为了减轻开发者在开发程序时的负担, 使请求追踪更加灵活, 学术界和工业界也出现了利用非源代码插桩实现请求追求的方式. 非源代码插桩实现请求追求的方式主要可分为两类: 基于统计推断的请求追踪方法和基于特定编程语言特性的自动插桩请求追踪方法.

基于统计推断的请求追踪方法将分布式系统看作是黑盒, 先收集系统在运行时产生的预定义的数据, 对数据进行分析从而发现数据之间的因果关系, 构建请求执行路径. Project 5[20]提出两种算法来推断因果关联路径, 其中一种算法使用来自 RPC 消息中的时间信息推断调用间因果关系, 另一种算法使用信号处理的方法, 根据相似时间信号匹配来找到同一节点接收和发送的消息. WAP5[22]是 Project 5 的改进版, 能够更好地适用于时钟不同步和具有延迟波动大的分布式环境中. Anandkumar 等人[24]假设请求在访问分布式组件时遵循一个已知的半马尔科夫过程模型, 使用最大似然法则来匹配单个请求的执行路径与请求产生的足迹信息. Sengupta 等人[25]提出一种方法, 将日志和请求的先验模型作为输入, 为正在执行的请求产生它的动态执行概要. Wang 等人[26]的工作中使用无监督学习算法来对大量的日志数据进行分析, 提取出能够将调用者和被调用者关联起来的关键属性, 主要思路是利用数据中的时序信息和提取具有统计意义的模式. 基于统计推断的请求追踪方法避免了对分布式系统源码的修改, 不要求对系统内部的构造和实现有了解. 但是, 使用这种方法需要有大量的数据进行分析, 而且推断出来的结果可能不准确, 特别是在一些复杂的请求执行模式下, 如并发请求, 发现正确的因果关系是一个巨大的挑战.

基于特定编程语言特性的自动插桩请求追踪方法利用一些编程语言的特性, 在程序启动或运行时将请求追踪代码自动地插桩到程序中. 这种方式最具有代表性的实现是基于 JavaAgent premain 实现的 SkyWalking Java

Agent 和 OpenTelemetry JAR. 它们预先在常见的框架或代码库中插入请求追踪的代码, 然后在 Java 虚拟机启动时, 执行 main 函数之前, 将原有框架或代码库替换成插桩后的框架或代码库. 这种方式无需开发者了解请求追踪的前置知识, 只需要开发者在程序启动时输入相应的参数就可以在程序中自动插桩请求追踪的代码. SkyWalking Java Agent 和 OpenTelemetry JAR 只能支持对已进行插桩过的框架或代码库进行替换, 维护和更新这些框架和代码库需要巨大的人力成本. 本文提出的 Trace++与 SkyWalking Java Agent 和 OpenTelemetry JAR 相比, 无需预先对框架和代码库进行插桩, 而是直接在程序运行时将请求追踪代码插桩到程序中, 这降低了维护和更新请求追踪框架的成本. 图 4 分别展示 SkyWalking Java Agent 和 Trace++请求追踪的实现方式. Trace++还能够在程序运行时动态地开启和关闭请求追踪功能, SkyWalking Java Agent 和 OpenTelemetry JAR 都仅能插入请求追踪代码, 但无法动态地移除请求追踪代码以关闭请求追踪. 在容器化应用程序的请求追踪实现上, SkyWalking 要求开发者在构建应用镜像时使用定制的基础镜像, 已有的容器化应用如果要实现请求追踪则需要对应用镜像进行重新构建. 而 Trace++则是在宿主机上直接对容器内运行的应用程序进行插桩, 对应用不做任何前置要求. 相比之下, Trace++具有更强的灵活性.

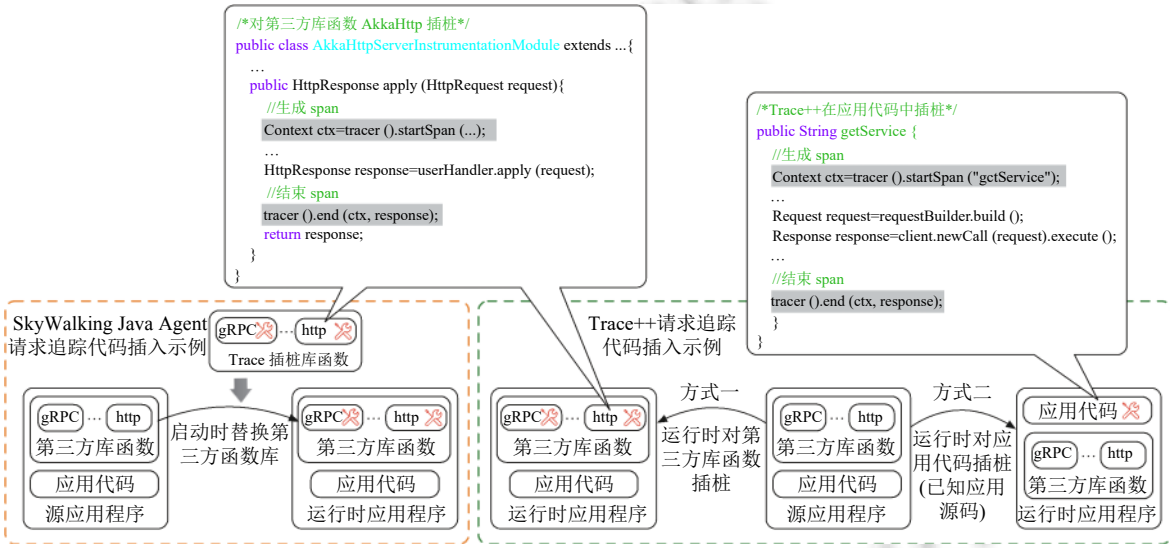


图 4 SkyWalking Java Agent 与 Trace++的请求追踪实现方式

1.3 请求追踪采样相关工作

请求追踪的数据能构建服务依赖, 在定位根因时带来很大的帮助, 但是全量采集请求追踪数据会产生海量的数据, 既不利于后期的数据分析, 也会造成巨大的存储开销. 例如, 微信中包含运行在超过 20000 台服务器上的 3000 多个服务, 每天的请求总量通常为 10^{10}–10^{11} 次, 每天产生数百 TB 的请求追踪数据[34]. 但是, 在请求追踪数据中大部分都是相似和冗余的, 没有必要把全量请求追踪数据存储下来. 因此为了降低存储成本的开销, 学术界和工业界提出了对请求追踪数据进行采样的设想: 减少存储相似且冗余的请求追踪数据.

Zipkin (https://zipkin.io/) 和 Jaeger (https://www.jaegertracing.io/) 等请求追踪系统采用随机采样的方案, 采样率低于 1%. 虽然随机采样的方式简单快捷, 可以极大地降低存储成本, 但随机采样可能会导致部分有价值的请求追踪数据丢失. JCallGraph[35]仅对响应成功的请求进行采样, 保留所有请求失败的 trace. 虽然 JCallGraph 能够在一定程度上克服随机采样的缺点, 但这种方式还是可能会丢失请求成功但响应延迟很高的异常追踪. Bauer 等人[36]将追踪采样引入一致性检查, 他们不停对请求追踪进行采样, 直到没有发现有新的一致性信息的追踪. 分层聚类抽样方法[37]根据请求追踪的种类来进行偏差采样, 从而最大限度地提高请求追踪数据的多样性, 但它也会丢失响应延迟异常的请求追踪数据. Sifter[38]利用请求追踪数据对目标系统的正常行为进行建模, 从而将与模型不匹配的请求

追踪数据视为异常数据并提高这些数据的采样概率. 由于 Sifter 只关注请求追踪数据的结构, 因此具有异常时间特征的请求追踪数据会被忽略. 本文提出的采样方法能够根据追踪数据的延迟分布动态调整采样率, 确保具有异常延迟的追踪数据不丢失.

2 Trace++设计原理

Trace++的设计目标为对应用程序低侵入, 对开发者完全透明, 具有灵活的追踪功能控制, 可以在没有程序源码的情况下, 对运行中的应用程序进行请求追踪赋能, 不需要开发者具备请求追踪或者应用程序的知识, 同时具有自适应采样能力.

2.1 Trace++系统架构

图 5 展示 Trace++的总体设计. Trace++包含控制器、存储中心和可视化 UI 等组件, 实现追踪功能赋能, 追踪数据收集、采样、查询和展示的功能. 控制器对微服务系统中的每一个服务实例植入追踪代理, 通过代理控制追踪功能的启动与关闭. 追踪代理负责对服务实例进行动态代码插桩, 自动将请求追踪的代码插桩到程序中, 收集追踪数据, 发送到存储中心. 在存储中心, Trace++使用自适应的采样算法减少追踪数据的存储开销, 丢弃相似的追踪数据, 保留有代表性的追踪数据. 可视化组件提供了查询追踪数据的前端界面, 将追踪数据以调用链的形式展现出来.

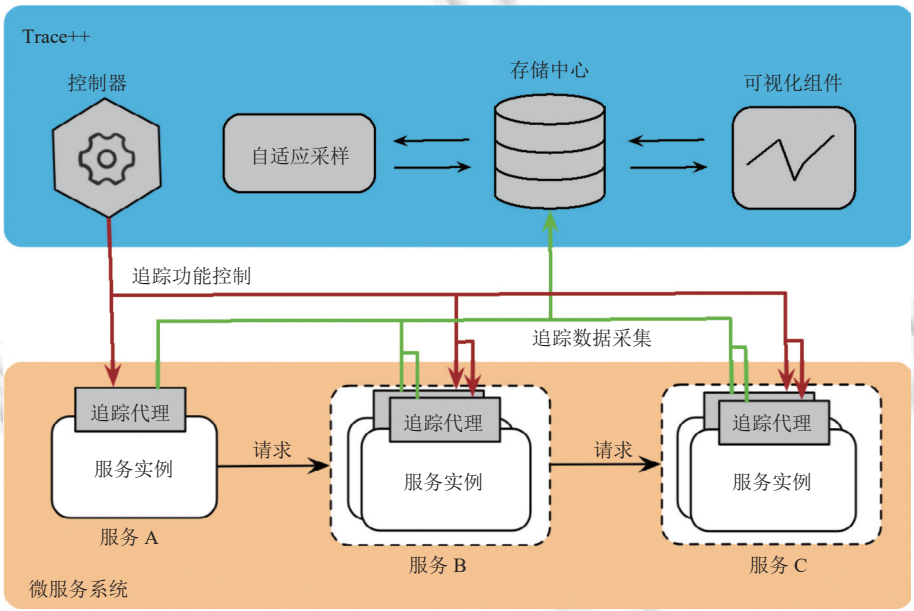


图 5 Trace++系统架构

2.2 透明请求追踪

在请求追踪自动化以及对用户透明的要求下, Trace++使用动态代码插桩的方式在运行时修改程序的代码. 为了实现动态代码插桩, Trace++对每个服务实例植入追踪代理, 由代理实施对应用程序字节码修改, 实现 OpenTracing 的请求追踪能力, 收集产生的追踪数据. 控制器管理追踪代理, 包括将代理植入服务中, 给代理下发追踪开启或关闭的命令, 告知代理插桩的位置和内容. 后文图 6 展示 Trace++的透明的请求追踪方案设计.

2.2.1 控制器

控制器用来实现追踪代理植入和追踪代理控制. 在微服务系统中, 为了提高可用性, 一个服务具有多个服务实例, 控制器需要对每一个服务实例都植入追踪代理. 植入追踪代理利用了 Java 语言的 javaagent 机制, 这种机制常用来对 Java 程序进行增强, 提供一种能够在程序运行时修改 Java 字节码的方法. 利用 javaagent 机制, Trace++将

外部的追踪代理植入到程序中, 追踪代理将等待控制器的指令, 根据指令来实施动态代码插桩. 控制器与追踪代理通过 Unix socket 进行通信, 通信的两方事先规定好了交换数据的格式和意义. 要使追踪代理实施正确的动态代码插桩操作, 控制器需要告知追踪代理插桩的位置以及插桩的内容. Trace++如果要达到源代码插桩一样的插桩效果 (主要是插桩的位置一致), 需要有关于源代码的知识. 但另一方面, Trace++要实现对用户透明的特性, 不能依赖于用户对源码具有了解. 应用程序在实现网络功能时会用到第三方库, 如 Okhttp、HttpURLConnection、RestTemplate 等, 应用代码调用这些库来处理请求和响应的发送、接收. 对这些库做代码插桩同样能实现请求追踪, 从而不需要了解应用程序源码.

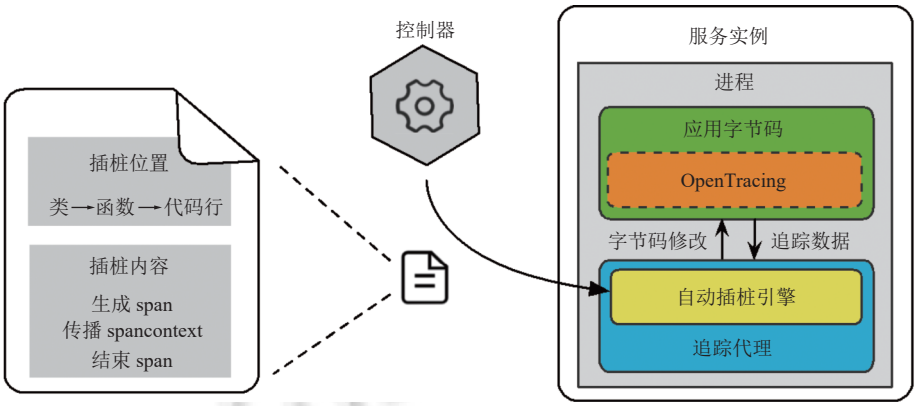


图 6 Trace++中透明的请求追踪方案设计

第三方库对请求的处理包含两个阶段: 第 1 个阶段是请求的发送, 第 2 个阶段是请求响应的接收. 按照 OpenTracing 的追踪模型, 控制器将插桩位置定位在请求发送之前和响应接收之后. 在请求发送之前, 需要插桩代码生成 span, 将 spancontext 附加到请求. 在响应接收之后, 需要插桩关闭 span 的代码.

2.2.2 追踪代理

追踪代理以 javaagent 的形式在运行时被植入到服务中, 并打开一个端口以监听来自控制器的指令. 当收到控制器的指令时, 追踪代理修改程序字节码以实现动态代码插桩. JVM (Java virtual machine) 运行程序时需要加载类文件, 类文件中的内容是程序的字节码. JVM 提供字节码变换机制来实现对已加载类文件的修改, 追踪代理使用这种技术实现动态代码插桩.

追踪代理的自动插桩引擎主体部分是一个类文件变换器, 自动插桩引擎根据控制器发送的插桩位置确定要修改的类, 类文件变换器获取该类的字节码. 然后, 根据控制器指定的位置在类的字节码中添加插桩代码. 追踪代理对网络库中实现请求处理的类进行动态插桩, 由于调用者和被调用者在一次请求处理的过程具有不同的操作, 调用者的操作包括请求发送和响应接收, 被调用者的操作包括请求接收和响应发送, 对这两种角色的插桩操作不同. 对于调用者, 追踪代理在请求发送之前的位置插入生成 span 的代码和将 spancontext 附加到请求上的代码, 在响应接收之后的位置插入关闭 span 的代码, 从而在 span 中记录一次完整的服务调用过程. 对于被调用者, 需要用附加在请求中的 spancontext 生成关联 span, 所以追踪代理在请求接收之后的位置插入从请求中提取 spancontext 的代码, 并用它生成 span, 作为调用者 span 的子 span. 在被调用者发送响应之后的位置, 追踪代理需要插入关闭 span 的代码, 从而在 span 中记录被调用者的请求处理过程. 后文图 7 展示调用方和被调用方的插桩示意图.

不同于现有的请求追踪工具, Trace++能够动态控制请求追踪功能的开启和关闭, 切换在程序运行时完成, 不需要修改或者重启程序. 追踪代理利用字节码修改机制动态插桩追踪代码, 反过来也可以利用这种机制将修改过的字节码恢复为原来的字节码, 从而关闭追踪功能.

2.2.3 容器场景

容器是运行微服务的载体, 为应用程序提供了与宿主机相隔离的运行环境. Trace++目标是动态赋能 Java 微

服务系统请求追踪, 需要对运行在容器中的应用程序进行动态插桩. 容器是一个独立的运行环境, 追踪代理的植入和对追踪代理下发命令都需要在容器内进行. Trace++提供插桩工具包, 只要将工具包分发到容器内部, 再利用容器提供的接口就能在外部调用工具包实施动态插桩. 工具包存放于宿主机上, 容器将宿主机上工具包所在的目录动态挂载到容器内部. 这种实现方式带来的好处是主机上存在的多个容器共享一个工具包, 避免在容器内部创建工具包副本带来的存储开销. 控制器调用物理机上的容器管理守护进程提供的接口, 发起对挂载在容器中的工具包的调用, 对容器中的服务植入追踪代理, 使用追踪代理进行动态代码插桩, 图 8 展示了这个过程.

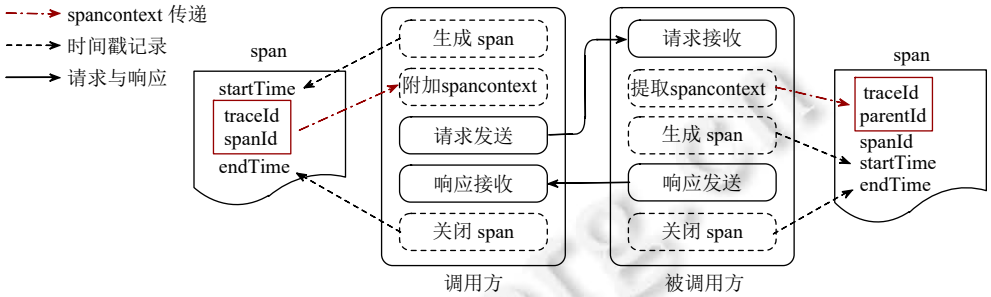


图 7 插桩示例. 虚线框表示插桩的内容

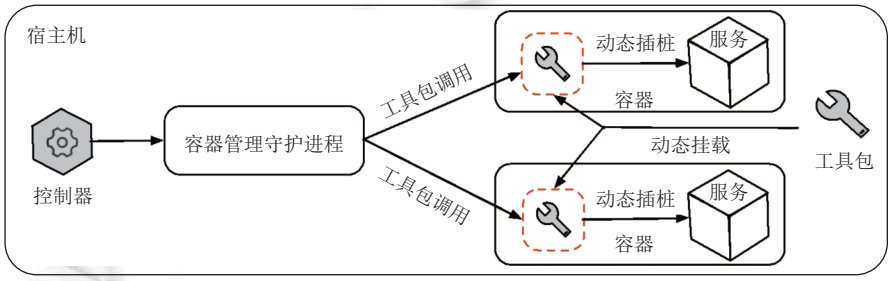


图 8 容器内动态插桩

2.3 自适应采样

自适应采样的目标在于保留追踪数据中具有代表性的样本, 去除冗余样本, 减少追踪数据的数量. 追踪数据中记录的请求返回码和时延信息都能够反映微服务系统的健康状况. 对于请求返回码异常 (对 http 请求, 返回码大于等于 400 则判断为异常, 对 gRPC 请求返回码不是 0 则判断为异常) 的追踪数据, Trace++认为这些异常请求都是需要被关注的, 会直接将这类追踪数据保留. 而对于请求返回码正常的追踪数据, 无法直接判断这类追踪数据是否具有代表性, Trace++会根据时延信息对追踪数据进行采样. 下面主要介绍 Trace++如何根据时延信息对请求返回码正常的追踪数据进行采样.

2.3.1 数据稀疏性

微服务系统处于正常运行的状态时, 请求的时延波动较小, 分布在一个狭窄的取值范围内. 当微服务系统出现故障时, 请求会出现极高或者极低的时延. 微服务系统在大部分时间中正常运行, 具有正常时延水平的追踪数据占了绝大多数. 相比之下, 偏离正常时延水平的追踪数据数量显得十分稀少. 数据的取值范围很大, 但大部分数据的取值都集中在一个狭窄的区间内. 文献 [39] 将这种数据分布形式称为数据稀疏性. 对具有稀疏性的数据进行采样, 可以利用数据的分布特性来对处在不同取值区间的数据样本设定不同的采样偏好, 如果一个取值区间内数据分布密集, 只需要较低的采样率就可以得到有代表性的样本. 如果取值区间内数据分布稀疏, 就需要较高的采样率来减少有代表性的样本被漏掉的可能性.

2.3.2 代表性采样

对于请求返回码正常 (例如小于 400 的 http 请求) 的追踪数据, Trace++使用请求的平均时延作为样本的代表性

评估指标, 如果样本的平均时延接近接近总体的平均时延, 说明不同时延水平的数据都有被采样到, 样本的代表性好. 在复杂的微服务系统中每一类请求都有特定的时延模式, 访问微服务系统不同 API 的请求产生的时延不同. 考虑到如果对不同种类的请求混合采样, 不同时延模式的请求之间会存在一定的干扰. 例如, 用户登录请求的延时为十几毫秒, 用户文件下载请求的延时可能高达数百毫秒, 那么 100 ms 的用户登陆的异常请求可能会被用户文件下载的正常请求所掩盖. 因此, 根据时延的分布对追踪数据进行采样时, 需要先将追踪数据按照请求的种类进行分类.

由于 Trace++产生的追踪数据记录了请求对应的 URI (uniform resource identifier) 和请求经过的每个服务对应的 span, Trace++根据 URI 和 span 的特征信息对追踪数据做分类. 本文将具有相同 URI 和相同的 span 种类且每个种类有相同数量的追踪数据判定为同一类追踪数据. 对于某一类追踪数据, 在考虑样本代表性的同时, 还得考虑采样率, 样本代表性越好, 需要的采样率越高, 采样带来的开销降低越小. Trace++允许设定样本时延均值和总体时延均值之间的差距, 在确保差距的情况下使采样的样本数最小.

霍夫丁不等式[40]可以用来求解满足样本时延均值与总体时延均值差值的条件下所需要的样本数量, 给定均值误差 ε, 数据的取值范围 $[a,b]$ 置信度 δ, 总体的数据量 N, 根据霍夫丁不等式可计算出需要的样本数量为:

$$n=\frac{1}{G+\frac{1}{N}} \tag{1}$$

$$G=\frac{2\varepsilon}{(b-a)^2\ln\frac{2}{1-\delta}} \tag{2}$$

其中, 置信度 δ 表示在 M 次独立采样过程中有 $M\cdot\delta$ 次采样能够满足样本均值与总体均值的误差小于等于 ε. 由于追踪数据的时延分布是稀疏的, 异常请求的时延可能非常高, 导致 $(b-a)^2$ 的值很大, 从而使 G 的值接近于 0, 得到的样本数量 n 接近于 N, 采样带来的增益小. 如果把原有的数据按照时延的取值范围分在不同的桶中, 让密集分布的数据和稀疏分布的数据处于不同的桶, 对这些桶分别进行采样, 就能够降低样本数量. 对于数据稀疏分布的桶, 由于 $(b-a)^2$ 比较大, 样本数仍然接近于 N, 而对于数据密集分布的桶, $(b-a)^2$ 的值比较小, 计算出的样本数 n 会远小于 N 所以最后汇总的样本数会有效减少.

假设将数据划分为 k 个桶, 每个桶收纳数据的时延范围为 $[a_i,b_i]$, 桶中的数据总量为 N_i $(1\leqslant i\leqslant k)$, 如果每个桶采样的均值误差都为 ε, 置信度都为 δ, 所有桶汇总后的样本与总体的均值误差也为 ε, 汇总的样本数为:

$$n=n_1+\ldots+n_k \tag{3}$$

$$n_i=\left\lceil\frac{1}{\frac{2\varepsilon}{(b_i-a_i)^2\ln\frac{2}{1-\delta}}+\frac{1}{N_i}}\right\rceil,\ 1\leqslant i\leqslant k \tag{4}$$

但是, 每个桶的均值误差并不一定要统一设置为 ε, 一些桶的均值误差可以设置得比较大, 例如对于数据分布很稀疏的桶, $(b_i-a_i)^2$ 的值极大, 减小 ε 对 n_i 的取值影响不大. 反过来, 对于数据分布很密集的桶, 由于 $(b_i-a_i)^2$ 的值较小, 增大 ε 能够使 n_i 的值变小. 用 ε_i 表示每个桶的均值误差, 只需要确保总的误差均值等于 ε:

$$\frac{N_1\cdot\varepsilon_1+\ldots+N_k\cdot\varepsilon_k}{N}=\varepsilon \tag{5}$$

前面的分析建立在数据已经分好桶的情况下, 可以发现分桶的结果对采样率的影响较大, 好的分桶方式把稀疏数据聚集在一起, 与密集数据分开, 从而能够从密集数据的采样中得到更多效益. 桶的数量 k 以及桶的时延范围 $[a_i,b_i]$ 都需要根据数据的分布特性进行设置, 才能达到较好的采样效果.

追踪数据的代表性采样问题可以形式化为: 给定大小为 N 的有序数据集 $X=\{x_1,\ldots,x_N\}$, 均值误差为 ε_0, 置信度为 δ, 找到 X 的一组划分 $P=\{[a_i,b_i]|1\leqslant i\leqslant k\}$, 令 N_i 表示 $[a_i,b_i]$ 中的数据量, 以及找到每个划分 $[a_i,b_i]$ 对应的均值误差 ε_i, 使得:

$$\frac{N_1\cdot\varepsilon_1+\ldots+N_k\cdot\varepsilon_k}{N}\leqslant\varepsilon_0 \tag{6}$$

$$\min\left(\sum_{i=1}^{k}\left\lceil\frac{1}{\frac{2\varepsilon_i}{(b_i-a_i)^2\ln\frac{2}{1-\delta}}+\frac{1}{N_i}}\right\rceil\right) \tag{7}$$

使用动态规划算法可以找到上述问题的最优解, 使用最少的样本数达到满足条件的均值误差, 但是需要$O(n^4)$的时间复杂度[39]. 另外一种启发式算法虽然找到的解不一定最优, 但是可以将时间复杂度降低到$O(n)$, 并且保证采样的效果不会差于随机采样[39]. 启发式算法简化了桶的均值误差设置, 统一设置为ε_0, 公式 (6) 的条件得到满足, 然后使用贪心算法进行数据分桶. 按照从小到大的顺序访问数据, 每拿到一个数据, 将数据加入到当前的桶中(开始有一个空桶), 使用公式 (4) 计算出数据加入后桶的采样样本数, 如果样本数比数据加入前的样本数大, 则创建一个新桶, 并把数据从当前桶移到新桶中, 将新桶作为当前桶, 算法 1 给出了具体过程.

算法 1. 启发式的数据分桶算法.

输入: 从小到大排列的数据集$X=\{x_1,\ldots,x_N\}$, 均值误差ε_0, 置信度δ;
输出: 将数据划分的桶集合$B=\{b_1,\ldots,b_k\}$.

1. $B_{\text{cur}}\leftarrow\{x_1\}$
2. $B\leftarrow\{B_{\text{cur}}\}$
3. $i\leftarrow 2$
4. **WHILE** $i\leqslant N$
5. $\quad n_{\text{before}}\leftarrow$根据式 (4) 计算$B_{\text{cur}}$采样的样本数
6. $\quad$将x_i添加到B_{cur}中
7. $\quad n_{\text{after}}\leftarrow$根据公式 (4) 计算$B_{\text{cur}}$采样的样本数
8. $\quad$**IF** $n_{\text{after}}>n_{\text{before}}$
9. $\quad\quad$从B_{cur}中移除x_i
10. $\quad\quad B_{\text{cur}}\leftarrow\{x_i\}$
11. $\quad$将B_{cur}添加到B中
12. $\quad$**END IF**
13. **END WHILE**

算法 1 的输出提供了数据的划分, 使用公式 (4) 算出每一个桶的采样样本数, 然后桶内进行随机采样得到对应数量的样本, 将每个桶的样本汇总得到总体的代表性样本.

3 Trace++ 系统实现

本文实现了 Trace++原型系统, 功能涵盖追踪数据的生成、收集、存储、采样和可视化. Trace++实现了对 http 和 gRPC 请求的自动化追踪, 让开发者可以在无请求追踪的前置知识和不理解微服务系统内部实现的情况下, 准确地捕获微服务系统的服务调用关系.

3.1 追踪数据生成

完成一次请求涉及到客户端和服务端的操作, Trace++使用动态代码插桩技术干预客户端和服务端的请求处理过程, 生成追踪数据. 控制器和追踪代理的设计提供了灵活可控的插桩能力. 控制器以规则脚本[41]的形式为追踪代理提供插桩位置和插桩内容, 如图 9 所示. 针对客户端和服务端, 控制器生成不同的规则脚本.

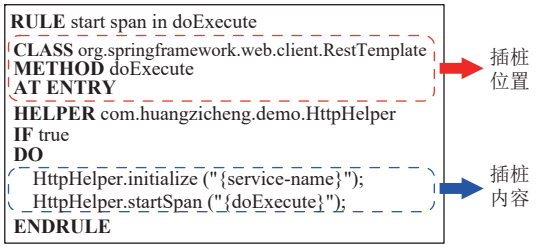


图 9 规则脚本

3.1.1 客户端动态代码插桩

Http 客户端承担请求发送和响应接收的功能, Trace++控制器为 http 客户端生成描述以下 3 种插桩场景的规则脚本: 请求发送前生成 span, 请求发送时附加 spancontext, 接收响应后关闭 span. 以 RestTemplate 这种 http 客户端为例, RestTemplate 提供众多发送请求的函数, 但是并不需要针对每一个函数做代码插桩, 发送请求的核心功能由 doExecute 函数实现, 发送请求的函数都会调用 doExecute 函数, 所以控制器只需对 doExecute 函数做插桩, 就能覆盖到 RestTemplate 的所有请求场景, 如 GET、POST、PUT 等. 生成 span 和附加 spancontext 的操作可能出现在不同的函数内, 为了让 span 可以跨函数访问, span 被存放在线程的私有化存储中, 这样做同时还可以确保不同请求产生的 span 不会交错在一起, 并发的不同请求在不同的线程中处理, 一个线程中的请求生成的 span 访问不到其他线程中的请求生成的 span, 不会建立错误的关联关系. Http 请求头部支持自定义字段, spancontext 以自定义字段的方式随着请求从客户端传递到服务端.

gRPC 客户端以函数调用的形式访问远端服务, 客户端将调用参数通过网络发送给服务端, 服务端接收参数并运行对应的处理逻辑, 得到的结果通过网络返回给客户端. 为了在 gRPC 客户端发送调用参数时生成 span 和传递 spancontext, Trace++对 gRPC 客户端添加拦截器. 拦截器是网络通信中常用的一种技术, 目的在于对请求和响应进行预处理或后处理. gRPC 客户端拦截器负责在发送请求时生成 span, 将 spancontext 添加到请求中, 在响应返回后关闭 span. gRPC 将客户端与服务端的连接抽象为 channel, 拦截器是 channel 对象的一个字段. 客户端的阻塞或者非阻塞请求都会经过 channel 对象的 newCall 函数进行处理, 因此 Trace++将插桩点设置在 newCall 函数中. 当 newCall 函数被调用时, 将会触发插桩代码, 插桩代码使用反射机制设置 channel 对象的拦截器字段. 虽然每次发送 gRPC 请求都会调用 newCall, 但是 Trace++确保插桩代码只在第 1 次调用 newCall 时被触发.

3.1.2 服务端动态代码插桩

Http 服务端的工作模式是接收请求, 返回响应. 对于服务端的动态代码插桩, 控制器生成描述以下两种场景的规则脚本: 在接收到请求时提取 spancontext 并生成 span, 在发送响应后关闭 span. 以使用 springboot 框架实现的 Web 服务端为例, 所有的请求都要经过 DispatcherServlet 类的 doDispatch 函数处理, 来将请求分发给具体的处理函数, 并且返回响应也在这个函数里面. Trace++将插桩点设置在 doDispatch 函数的入口和出口, 在函数入口插桩代码提取 spancontext 和生成 span, 在函数出口插桩代码关闭 span.

类似于对 gRPC 客户端的修改, Trace++对 gRPC 服务端添加拦截器. 拦截器在接收到 gRPC 请求时将 spancontext 提取出来, 利用该 spancontext 生成 span. 另外, 拦截器还在服务端发送响应后将 span 关闭. Trace++选择回调函数 onHalfClose 作为插桩代码的触发函数, 在 onHalfClose 函数的出口设置插桩点. 因为服务端接收请求必然会调用 onHalfClose, 所以确保了插桩的代码会在接收请求时被触发. 与客户端在 channel 对象中添加拦截器的做法不同, 服务端需要对 server 对象添加拦截器. server 对象是 gRPC 服务端的抽象, 对它进行修改需要先获取 server 对象的引用, 然后使用反射机制在 server 对象中添加拦截器. 但是, 因为插桩点在 onHalfClose 函数内部, 在这个函数内部没有任何关于 server 对象的引用, 因此插桩代码修改不了 server 对象. Java 的对象保存在堆内存中, 使用 JVMTI (JVM tool interface) 可以在堆中查找并获取对象. 因此, onHalfClose 函数中的插桩代码被触发时, 首先在堆内存中搜索, 得到 server 对象, 然后对 server 对象添加拦截器. Trace++确保插桩代码只被触发一次, server 对象查找和修改操作只有一次性开销.

3.1.3 容器目录动态挂载

为了对运行在容器中的应用进行透明请求追踪, 现有的方法是将请求追踪的功能预置在定制的容器基础镜像中, 在此基础镜像上构建应用程序, 这也是 SkyWalking 所采用的方案. 虽然这种方案实现了透明请求追踪, 但是只适用构建阶段的应用程序, 无法用于已经启动的容器应用.

Trace++支持对运行中的容器应用实现透明请求追踪. 对于运行在容器内的服务实例, 插桩需要在容器内部进行. 在自动化插桩操作前, Trace++将控制器和追踪代理, 以及辅助脚本、Java 库打包为工具包, 利用动态目录挂载技术将工具包分发到运行在同一台宿主机上的所有容器内部, 在宿主机上通过容器管理守护进程提供的接口调用工具包. 动态目录挂载避免了工具包分发时多次复制给容器和宿主机带来的存储开销, 让容器能共享宿主机上的工具包. 实现动态目录挂载的关键是突破容器的资源隔离机制, 容器有独立的文件系统命名空间, 正常情况下容器和宿主机不共享文件. Trace++使用 setns 系统调用侵入容器的文件系统命名空间, 这样可以获得较高权限来操作容器内部文件系统. Trace++在这个命名空间内创建块设备文件, 创建的块设备文件与宿主机上的块设备文件具有一致的设备 ID, Linux 上的块设备文件是硬盘分区的抽象, 等于把宿主机上的硬盘分区复制到了容器之中. 因为块设备文件代表的是硬盘的某个分区, 挂载后才可以对其中的文件系统进行访问. 处在容器的文件系统命名空间中使得 Trace++具有较高的权限, 可以实施目录挂载操作, Trace++将块设备文件挂载到容器内目录使宿主机上的文件在容器内部可见.

3.2 追踪数据采集、存储和可视化

请求追踪过程中产生的 span 在结束后将被追踪代理统一收集, 追踪代理每隔 1 000 ms 将 span 批量发送到追踪数据存储中心. 追踪数据存储中心对接收到的 span 进行验证并建立索引, 存储到 Cassandra 或者 Elasticsearch 中. Trace++的可视化组件接收用户的查询请求, 从数据存储中心获取指定数据, 将追踪数据可视化为服务拓扑图和甘特图, 展示服务间的依赖关系和服务时延等信息. 存储组件和可视化组件分别使用分布式请求追踪系统 Jaeger 的 Collector 组件和 Query 组件作为实现.

3.3 追踪数据采样

属于同一类请求的追踪数据有两个特征: 它们的 URI 具有相同前缀, 它们对应的追踪数据具有相同的 span 种类并且每个种类所对应数量也相同. Trace++ 利用 URI 和 span 的特征, 使用前缀树算法对追踪数据进行分类. 由于在微服务系统中可能会存在一些异步调用, 异步调用会导致 span 的顺序存在随机性, 无法保证同类的请求每次都是同样的 span 顺序, 导致判断错误, 因此本文在判断是否是同一类请求时并没有考虑 span 之间的顺序.

URI 是一串字符, 可以被分割为许多片段, 前缀树的每个节点表示 URI 的一个片段, 从前缀树的根节点到叶子节点的一条路径可以表示完整的 URI, 同时, URI 所对应的追踪数据保留在叶子节点中. 由于前缀树的特点, 相邻叶子节点满足具有相同前缀的特征, 如果相邻叶子节点包含的追踪数据具有相同的 span 种类且每个种类有相同的数量, 就认为这些追踪数据是由同一类请求产生的, 从而把这些相邻的叶子节点合并为一个叶子节点, 追踪数据也随着合并. 完成合并后, 一个叶子节点中保存的数据为一类数据. 对这些分好类的数据使用自适应采样算法, 在使用时只需设置置信度δ和均值误差ε.

4 案例学习

分布式请求追踪技术为运维人员观测系统、了解系统提供了极大的方便, 特别是对于具有复杂服务依赖性的微服务系统, 服务间的调用关系以及服务响应时间在诊断故障时具有非常重要的作用. 另一方面, 请求追踪带来的性能开销和存储开销也不可忽视, 如果系统已经能够长期稳定运行, 关闭请求追踪功能可以有效降低系统的运行成本. Trace++的自动化请求追踪能力满足了运维人员对系统可观测性和追踪功能启停的诉求, 并且让开发者可以根据实际的需要动态地改变追踪粒度的粗细.

TrainTicket 是一个模拟火车票售票的微服务 benchmark, 由 41 个服务构成, 一次请求处理涉及上百次服务调

用. 理解 TrainTicket 中各个服务之间的调用关系, 如果不借助请求追踪技术, 采取源码分析的方式费时且费力. 然而, 即使采用了分布式请求追踪技术, 源代码插桩的方式也会使请求追踪受限于开发阶段开发人员对于系统的理解, 后续如要根据系统的运行时行为调整请求追踪将十分麻烦. 使用 Trace++能够在不对 TrainTicket 的源码做任何修改的情况下得到请求的追踪路径. 图 10 展示处理一次火车票检索请求的服务调用, 如果依靠源代码插桩实现请求追踪, 就需要对图中的每一个服务都插桩追踪代码, 这个过程不仅要求开发者要熟悉源码, 还要熟悉 Opentracing 的实现原理. 一旦有服务在插桩时被遗漏或者没有正确地生成 span, 就会产生不完整的服务调用图, 不能准确地反映请求的执行路径.

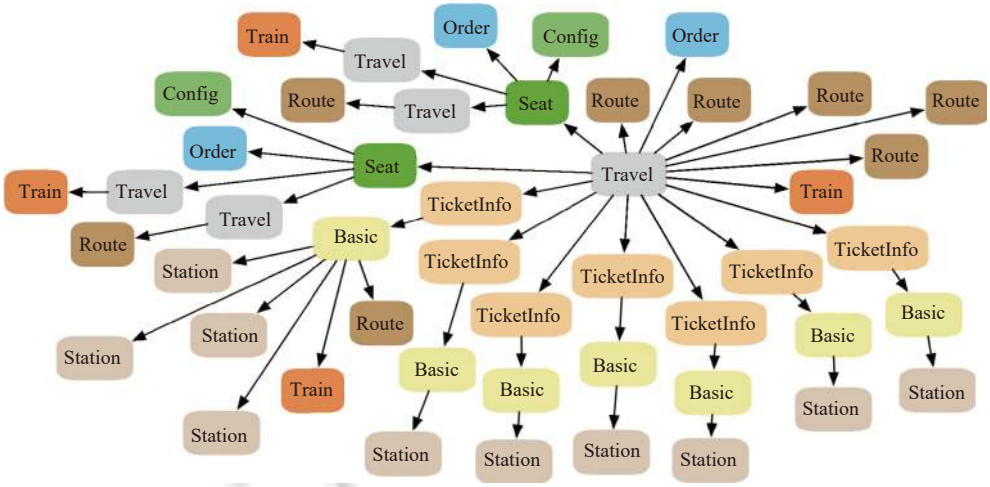
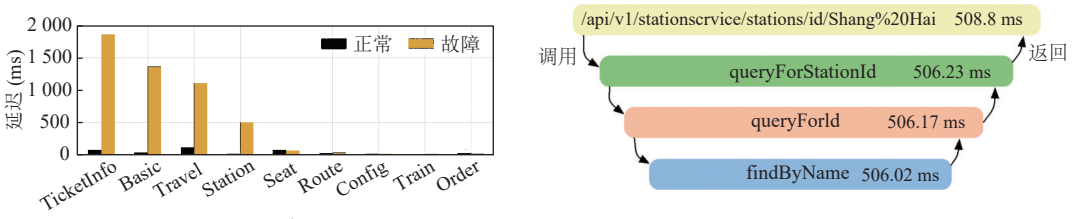


图 10 火车票检索的服务调用图

Trace++可以控制追踪功能的启停, 运维人员可以根据系统运行时的状态决定是否启动请求追踪. 一个典型的场景是系统出现性能故障导致请求的响应时间显著提升, 这时候开启请求追踪, 获取服务的响应时间, 从而定位发生性能故障的服务. 为了验证 Trace++辅助性能故障定位的能力, 我们使用混沌工程的方法模拟 TrainTicket 中出现性能故障的场景, 具体的实施方案为: 开启 Trace++的请求追踪功能获取追踪数据, 对 station 服务注入 500 ms 的网络延迟故障, 观察对比故障注入前后各个服务的请求处理时间. 对故障注入前后的请求追踪数据进行分析得到图 11(a) 所示结果, 由于故障的传播性, 下游服务的故障会导致上游服务也出现故障, 在分析故障发生的源头时需要结合服务调用关系进行追溯. 由图 10 可知 travel、ticketInfo、basic、station 这 4 个出现延迟异常的服务中, station 处在服务调用的最下游, 可以合理地推断是 station 服务首先出现异常延迟, 从而拖慢上游的 basic、ticketInfo、travel 服务的处理. 定位到异常源头的服务后, 了解服务源码的运维人员计划以更细的粒度定位故障根源, 比如定位到代码中的函数. 传统的追踪技术因为需要事先在源码中埋点, 所以无法在程序运行时动态调整追踪的粒度, 而 Trace++的动态插桩能力则非常适用于这种场景, 只需提供追踪的函数名以及函数所属的类名, Trace++就能生成规则脚本, 并实施动态插桩往程序中添加函数级别的执行路径追踪. Station 服务在处理请求过程中调用了 3 个函数, 追踪这些函数的执行时间得到图 11(b). 由图 11(b) 可以发现 station 服务的延迟主要是来自执行 findByName 函数, 这个函数的功能是在数据库中检索火车站, 访问数据库时需要建立网络连接, 过慢的网络访问是导致 findByName 函数执行时间长的原因之一, 因此下一步排查的范围可以缩小到 station 与数据库之间的访问过程.

由上述案例可以看到, Trace++为微服务系统提供了灵活且强大的监控和调试能力, 特别是对于微服务快速迭代的特性, Trace++不需要开发者在每一次的代码变更中还要兼顾追踪功能的正确性, 而且还能够满足开发阶段未覆盖的追踪需求. 在 DevOps 文化日益盛行的软件世界里, 开发者身兼运维的角色, 熟悉源码的开发者能够使用 Trace++轻易地获取程序运行时的细粒度追踪信息, 而不需要他们去了解请求追踪是如何实现.



(a) 服务级别 (b) 函数级别

图 11 不同粒度的追踪

5 实 验

5.1 实验环境

为验证 Trace++自动化请求追踪能力和自适应采样方法, 本文选取开源的火车票售票微服务系统 TrainTicket 作为测试平台. 相比其他开源的微服务系统, TrainTicket 的系统规模较大, 微服务的个数达到 41 个, 具有复杂的服务调用关系, 比较贴近真实的微服务系统, TrainTicket 的系统架构如图 12 所示. 但是, TrainTicket 在实现时没有做性能上的优化, 无法处理高并发量的请求. TrainTicket 中的微服务之间使用 http 协议进行通信, 为了验证 Trace++对 gRPC 请求的追踪能力, 我们修改了 baisc 和 price 这两个微服务的实现, 将 http 调用改为 gRPC 调用. 修改后的 TrainTicket 部署在 Kubernetes 集群上, 集群配置以及 TrainTicket 的部署配置如表 1 所示.

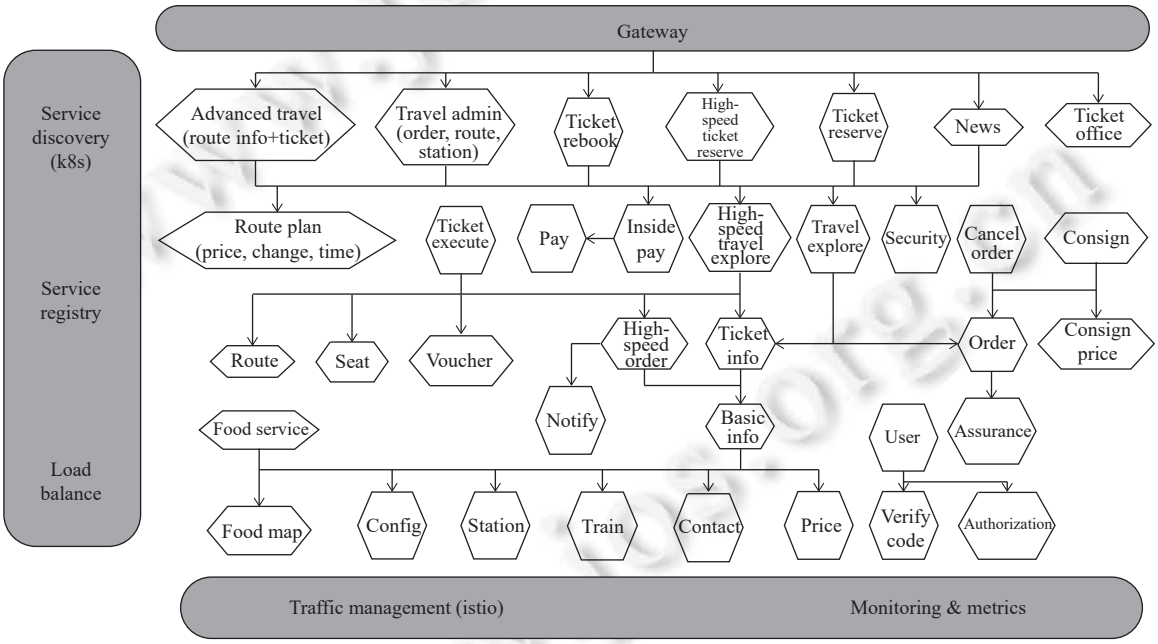


图 12 TrainTicket 架构

表 1 Kubernetes 集群配置和 TrainTicket 部署配置

名称	节点	CPU	存储	版本
Kubernetes集群	节点数量 9	节点CPU个数 8	节点memory 15.6 GB	Linux内核版本 5.4.0-050400-generic
TrainTicket	服务实例数量 1	容器CPU个数 0.2	容器memory 500 MB	Docker版本 19.3.5

访问 TrainTicket 的请求由负载生成器产生, 负载生成器构建了一个用户行为模型, 模拟用户访问 TrainTicket 时的行为模式, 比如用户会执行“登录→检索→订票→填写信息→提交订单→支付”的操作序列, 负载生成器按照操作序列依次生成对应的请求.

5.2 自动化请求追踪

5.2.1 调用链构建

利用 Trace++对 TrainTicket 的微服务实例动态插桩追踪代码, 启动负载生成器, 收集追踪数据. 本文另外部署了一套采用源代码插桩实现请求追踪的 TrainTicket 系统, 以该系统生成的追踪数据为基准, 检验 Trace++生成的追踪数据能否正确反映 TrainTicket 中的依赖关系. 表 2 展示了不同请求下由源码插桩和 Trace++这两种方法生成的 trace 所具有的 span 数量, 从结果可知两种方法产生的 trace 在 span 数量上是一致的. 我们进一步比较 span 间连接关系, 这代表服务间的调用关系, 最终确定了 Trace++的追踪效果与源码插桩的追踪效果一致, 能够准确地捕获服务间依赖关系.

表 2 不同请求下源码插桩和 Trace++生成的 trace 具有的 span 数量

服务	请求URI	Trace的span数量	
		源码插桩	Trace++
assurance	/api/v1/assuranceservice/assurances/types	1	1
auth	/api/v1/users/login	3	3
cancel	/api/v1/cancelservice/cancel/{orderId}/{loginId}	9	9
consign	/api/v1/consignservice/consigns/account/{accountId}	1	1
contact	/api/v1/contactservice/contacts/account/{accountId}	1	1
execute	/api/v1/executeservice/execute/execute/{orderId}	5	5
execute	/api/v1/executeservice/execute/collected/{orderId}	5	5
food	/api/v1/foodservice/foods/{date}/{startStation}/{endStation}/{tripId}	13	13
insidePayment	/api/v1/inside_pay_service/inside_payment	7	7
order	/api/v1/orderservice/order/refresh	3	3
orderOther	/api/v1/orderOtherService/orderOther/refresh	3	3
travel	/api/v1/travelservice/trips/left	95	95
travel2	/api/v1/travel2service/trips/left	23	23
travelPlan	/api/v1/travelplanservice/travelPlan/quickest	577	577
travelPlan	/api/v1/travelplanservice/travelPlan/minStation	595	595
travelPlan	/api/v1/travelplanservice/travelPlan/cheapest	577	577

5.2.2 性能开销

请求追踪会为服务带来额外的性能开销, 本文选择 TrainTicket 中被频繁访问和被访问次数较少的服务作为监控对象, 观察这些服务在开启和关闭请求追踪情况下的 CPU 和内存使用率的变化. 由表 2 可知, 访问 travelPlan 服务的/api/v1/travelplanservice/travelPlan/minStation 服务端点时, 产生的 trace 具有 595 个 span, 说明处理一次该服务端点的请求需要大量微服务的参与, 其中 basic 服务、ticketInfo 服务、travel 服务被访问次数都超过了 80 次. 因此, 选择这 3 个服务作为被频繁访问服务的代表, 而 config 服务、order 服务、routePlan 服务因为访问次数低于 20 次, 被选为低频访问服务的代表, 下文用“6 个服务”指代这些被选中的服务. 先测量原生的未经过任何插桩的 6 个服务的 CPU 和内存使用率, 然后使用 Trace++对 TrainTicket 做动态插桩并开启请求追踪, 测量开启请求追踪时 6 个服务的 CPU 和内存使用率, 之后关闭请求追踪, 测量关闭请求追踪时的 CPU 和内存使用率. 本文还测量了使用源代码插桩方式的请求追踪的 CPU 和内存使用率. 配置负载生成器使其不断发起访问/api/v1/travelplanservice/travelPlan/minStation 服务端点的请求, 使用 docker stats 命令实时监控上述 6 个服务所在容器的内存和 CPU 使用率, 结果分别如图 13 和图 14 所示.

服务的请求响应时间对用户体验有直接的影响, 为了测量 Trace++对服务响应时间的影响程度, 本文测量在无插桩, 源代码插桩, Trace++动态插桩但关闭请求追踪, Trace++动态插桩且开启请求追踪这 4 种情况下的请求响

应时间. 由表 2 可知/api/v1/travelplanservice/travelPlan/quickest、/api/v1/travelplanservice/travelPlan/minStation、/api/v1/travelplanservice/travelPlan/cheapest 这 3 种请求的执行路径最长, 这意味着这 3 种请求的响应时间受请求追踪的影响最大, 因此, 本文以这 3 种请求的响应时间作为测量对象, 取每种请求 100 次响应时间的平均值作为测量结果. 图 15 展示了 4 种情况下 3 种请求的响应时间, 在关闭请求追踪的情况下, Trace++对服务的请求响应时间影响较小, 带来的响应时间增加低于 5.37%. 在开启请求追踪的情况下, Trace++会为请求响应时间带来一定的延迟, 增幅低于 29%, 接近于源码插桩带来的开销.

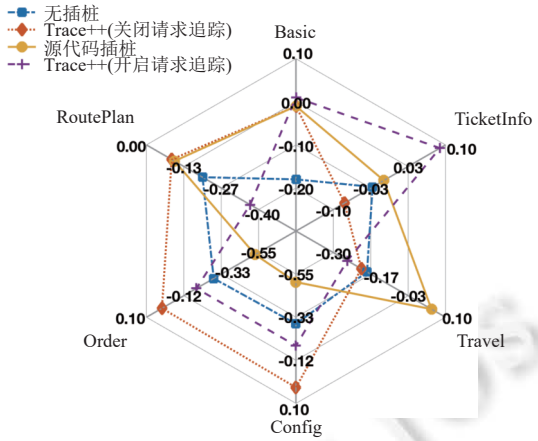


图 13 服务的内存使用量变化

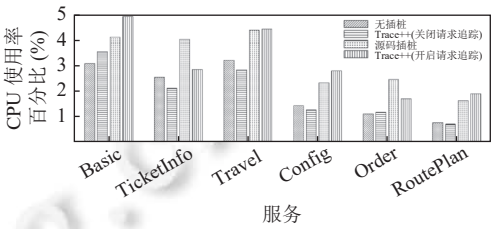


图 14 不同情况下的 CPU 使用率

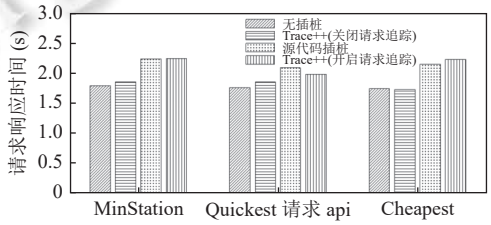


图 15 不同情况下的请求响应时间

5.3 自适应采样

配置负载生成器产生访问表 2 中 api 的请求, 收集追踪数据. 为了使追踪数据多样化, 每隔一小时对 TraintTicket 中的 route 服务或者 train 服务注入 500 ms 的网络延迟故障, 持续时间为 5 min. 总共采集追踪数据 49 526 条, 经过分类得到不同请求对应的追踪数据. 不同种类追踪数据的时延大小不同, 时延较大的数据其均值也比较大, 同样的均值误差 ϵ, 对于时延较大的数据和时延较小的数据, 严重程度不相同. 因此, 本文将不同种类追踪数据的时延取值通过缩放统一映射到 [0, 1000] 的范围内, 设置自适应采样算法的置信度为 0.95, 设置均值误差为 3. 在对追踪数据进行分类时, 根据 URI 和追踪数据中的 span 数量完成对追踪数据的分类, 对每个类别的追踪数据进行采样得到表 3 所示结果.

由表 3 的追踪数据分类结果可知, 使用前缀树分类算法, 16 种请求 URI 被分为 14 个类别, 其中除了类别 13 外的分类都是正确的. 类别 14 虽然包含两种 URI 的请求, 但是查看源码发现这两种请求的处理代码是相同的, 因此产生的追踪数据也相同. 对于类别 13 中两类 URI 的追踪数据, 虽然具有相同的 span 数量, 但是包含的 span 种类并不相同, 将这两类 URI 的追踪数据分为一类会影响采样的效果. 解决这个问题需要在分类时考虑追踪数据中 span 的种类以及每种种类的数量. 根据 URI 和 span 种类以及数量对追踪数据进行分类, 表 3 中类别 13 的数据被分为两类, 对这两类数据分别进行采样后得到的样本数量为 2 321, 减少了 55% 的样本. 由表 3 的追踪数据采样结果可知, 每个类别的压缩率受追踪数据的总量和追踪数据的延迟分布的影响而不同, 类别 10 的追踪数据压缩率最低, 达到 2.15%, 这是因为该类数据的总量大, 而且延迟的分布集中, 采样时只需要在大量集中分布的数据中选取少量样本点, 如图 16(a) 所示. 类别 9 的追踪数据压缩率最高, 达到 39.86%, 这是因为该类数据的总量较小, 延迟的分布相对分散, 需要采集较多数据以覆盖不同的延迟分布, 如图 16(b) 所示. 总体来看, 使用自适应采样方法减少了 89.4% 的追踪数据.

为了验证自适应采样方法能够得到有代表性的样本, 本文与随机采样方法做比较. 实验先使用自适应采样方

法对追踪数据采样, 得到采样样本, 然后使用随机采样方法对同一批追踪数据进行采样, 得到相同数量的样本. 图 17(a) 展示了随机采样方法对类别 2 的追踪数据的采样结果, 样本中延迟在 [0, 2000] ms 范围内的数据占据 93.35%, 而明显具有异常延迟的样本数量非常少, 如图中黑色虚线框所示. 这就导致了大量对微服务系统监控有重要价值的追踪数据丢失, 保留下来的追踪数据高度冗余. 在图 17(b) 中, 可以看到自适应采样方法对具有异常延迟的数据具有显著的采样偏好, 如图中黑色虚线框所示. 延迟高于 2 000 ms 的追踪数据中有 79.82% 的数据被采样, 延迟在 [0, 2000] ms 范围内的数据中有 23.28% 的数据被采样, 可以得到结论, 自适应采样方法采样的数据有代表性.

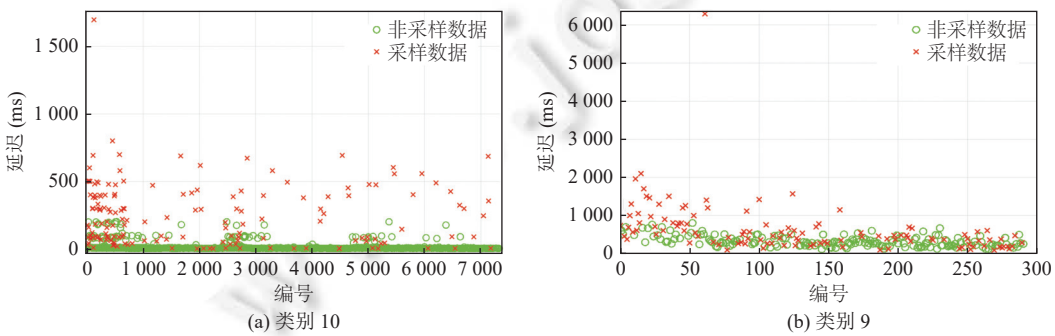
表 3　根据 URI 和 span 数量分类的追踪数据采样结果

类别	请求URI分组	总量	样本数	压缩率 (%)
1	/api/v1/users/login	5 985	212	3.54
2	/api/v1/travelservice/trips/left	1 479	406	27.45
3	/api/v1/travel2service/trips/left	1 520	193	12.70
4	/api/v1/executeservice/execute/collected/{orderId}	1 170	142	12.14
5	/api/v1/executeservice/execute/execute/{orderId}	1 170	208	17.78
6	/api/v1/travelplanservice/travelPlan/minStation	1 149	243	21.15
7	/api/v1/foodservice/foods/{date}/{startStation}/{endStation}/{tripId}	1 461	196	13.42
8	/api/v1/inside_pay_service/inside_payment	1 170	283	24.19
9	/api/v1/cancelservice/cancel/{orderId}/{loginId}	291	116	39.86
10	/api/v1/consignservice/consigns/account/{accountId}	7 363	158	2.15
11	/api/v1/assuranceservice/assurances/types	1 461	170	11.64
12	/api/v1/contactservice/contacts/account/{accountId}	1 461	204	13.96
13	/api/v1/orderOtherService/orderOther/refresh/api/v1/orderservice/order/refresh	21 542	5160	23.95
14	/api/v1/travelplanservice/travelPlan/cheapest/api/v1/travelplanservice/travelPlan/quickest	2 304	395	17.14

为了探究均值误差和置信度这两个参数对自适应采样方法的影响, 进行了以下实验: 从类别 12 获取 1 300 条数据, 其中的 1 287 条数据的延迟范围在 [0, 100] ms, 标记为正常数据, 剩下的 13 条数据的延迟范围在 [150, 800] ms, 标记为异常数据.

设置置信度为 0.95, 在不同的均值误差下使用自适应采样方法对 1 300 条数据采样, 计算压缩率和异常样本占异常数据的比例, 异常样本占异常数据的比例越高, 说明样本的代表性越好. 由图 18(a) 可以看到, 压缩率先随着均值误差的增大而急剧下降, 后面均值误差的增加对压缩率的下降影响极小. 异常样本比例随着均值误差的增大而呈现阶梯式下降, 考虑到压缩率先急后缓的下降趋势, 使用比较小的均值误差能够同时保证样本大小和样本代表性.

设置均值误差为 3, 在不同的置信度下使用自适应采样方法对 1 300 条数据采样, 图 18(b) 展示了置信度对压缩率和异常样本比例的影响. 当置信度很低时, 异常样本比例仍可以维持较高水平, 并且置信度达到 0.37 后, 就可以采样到所有的异常样本. 另一方面, 压缩率受置信度增大的影响会逐渐上升, 增加的幅度较小. 因此, 使用较高的置信度能够在牺牲少量的压缩性能的情况下保证样本代表性.



(a) 类别 10　　(b) 类别 9

图 16　两类追踪数据的采样结果

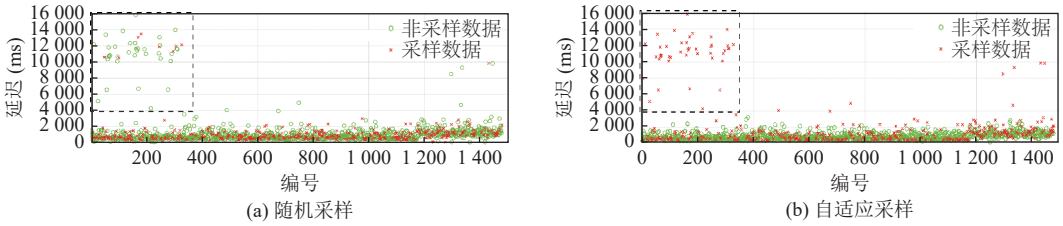


图 17 不同方法的采样结果

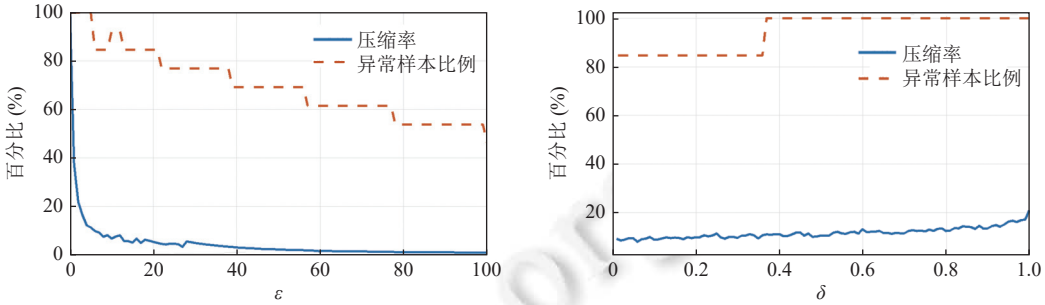


图 18 参数对采样的影响

6 总结和下一步工作

本文面向 Java 微服务系统, 设计和实现具有自动化、对开发者透明的请求追踪能力和自适应采样能力的请求追踪工具 Trace++. Trace++利用动态代码插桩技术实现了对开发者透明, 对应用程序低侵入, 能够灵活启停的分布式请求追踪方案. 借助容器目录动态挂载技术, Trace++能够对容器中的应用程序实施动态代码插装. 对于生成的追踪数据, Trace++使用自适应采样方法保留对微服务系统监控有价值的追踪数据, 丢弃冗余追踪数据. 自适应采样方法使用前缀树算法对追踪数据进行分类, 利用数据稀疏性, 在每一类追踪数据中将数据分为多个桶, 使用霍夫丁不等式计算得到每个桶的采样率, 在桶中进行随机采样.

目前, 使用前缀树算法在分类时没有考虑到追踪数据中的时间信息, 会影响分类的准确性, 进而影响采样的效果. 后续工作将探索能够利用追踪数据时间信息的分类算法, 以获得更加准确的追踪数据分类.

References:

[1] Lin JJ, Chen PF, Zheng ZB. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In: Proc. of the 16th Int'l Conf. on Service-oriented Computing. Hangzhou: Springer, 2018. 3–20. [doi: 10.1007/978-3-030-03596-9_1]

[2] Yu GB, Chen PF, Zheng ZB. Microscaler: Automatic scaling for microservices with an online learning approach. In: Proc. of the 2019 IEEE Int'l Conf. on Web Services. Milan: IEEE, 2019. 68–75. [doi: 10.1109/ICWS.2019.00023]

[3] Yu GB, Chen PF, Chen HY, Guan ZJ, Huang ZC, Jing LX, Weng TJ, Sun XM, Li XY. MicroRank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In: Proc. of the 2021 Web Conf. Ljubljana: ACM, 2021. 3087–3098. [doi: 10.1145/3442381.3449905]

[4] Yang Y, Li Y, Wu ZH. Survey of state-of-the-art distributed tracing technology. Ruan Jian Xue Bao/Journal of Software, 2020, 31(7): 2019–2039 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6047.htm [doi: 10.13328/j.cnki.jos.006047]

[5] Chanda A, Cox AL, Zwaenepoel W. Whodunit: Transactional profiling for multi-tier applications. In: Proc. of the 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems. Lisbon: ACM, 2007. 17–30. [doi: 10.1145/1272996.1273001]

[6] Sambasivan RR, Fonseca RLC, Shafer I, Ganger RG. So, you want to trace your distributed system? Key design insights from years of practical experience. Technical Report, Pittsburgh: Carnegie Mellon University, 2014.

[7] He ZL, Chen PF, Li XY, Wang YF, Yu GB, Chen CL, Li XR, Zheng ZB. A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems. IEEE Trans. on Neural Networks and Learning Systems, 2020: 1–15. [doi: 10.1109/TNNLS.2020.

3027736]

[8] Barham P, Donnelly A, Isaacs R, Mortier R. Using magpie for request extraction and workload modelling. In: Proc. of the 6th Conf. on Symp. on Operating Systems Design and Implementation. San Francisco: USENIX Association, 2004. 18. [doi: 10.5555/1251254.1251272]

[9] Chen MY, Accardi A, Kıcıman E, Lloyd J, Patterson D, Fox A, Brewer E. Path-based failure and evolution management. In: Proc. of the 1st Symp. on Networked Systems Design and Implementation. San Francisco: DBLP, 2004. 309–322.

[10] Fonseca R, Freedman MJ, Porter G. Experiences with tracing causality in networked services. In: Proc. of the 2010 Internet Network Management Conf. on Research on Enterprise Networking. San Jose: USENIX Association, 2010. 10. [doi: 10.5555/1863133.1863143]

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

[12] Reynolds P, Killian CE, Wiener JL, Mogul JC, Shah MA, Vahdat A. Pip: Detecting the unexpected in distributed systems. In: Proc. of the 3rd USENIX Conf. on Networked Systems Design and Implementation. San Jose: USENIX Association, 2006. 115–128.

[13] 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. [doi: 10.5555/1972457.1972463]

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

[15] Fonseca R, Dutta P, Levis P, Stoica I. Quanto: Tracking energy in networked embedded systems. In: Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 323–338. [doi: 10.5555/1855741.1855764]

[16] Thereska E, Salmon B, Strunk J, Wachs M, Abd-El-Malek M, Lopez J, Ganger GR. Stardust: Tracking activity in a distributed storage system. ACM SIGMETRICS Performance Evaluation Review, 2006, 34(1): 3–14. [doi: 10.1145/1140103.1140280]

[17] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. In: Proc. of the 15th European Conf. Budapest on Object-oriented Programming. Hungary: Springer, 2001. 327–354. [doi: 10.1007%2F3-540-45337-7_18]

[18] Mace J, Roelke R, Fonseca R. Pivot tracing: Dynamic causal monitoring for distributed systems. In: Proc. of the 25th Symp. on Operating Systems Principles. Monterey: ACM, 2015. 378–393. [doi: 10.1145/2815400.2815415]

[19] Erlingsson Ú, Peinado M, Peter S, Budiu M, Mainar-Ruiz G. Fay: Extensible distributed tracing from kernels to clusters. In: Proc. of the 23rd ACM Symp. on Operating Systems Principles. Cascais: ACM, 2011. 311–326. [doi: 10.1145/2043556.2043585]

[20] Aguilera MK, Mogul JC, Wiener JL, Reynolds P, Muthitacharoen A. Performance debugging for distributed systems of black boxes. ACM SIGOPS Operating Systems Review, 2003, 37(5): 74–89. [doi: 10.1145/1165389.945454]

[21] Koskinen E, Jannotti J. Borderpatrol: Isolating events for black-box tracing. ACM SIGOPS Operating Systems Review, 2008, 42(4): 191–203. [doi: 10.1145/1357010.1352613]

[22] Reynolds P, Wiener JL, Mogul JC, Aguilera MK, Vahdat A. WAP5: Black-box performance debugging for wide-area systems. In: Proc. of the 15th Int'l Conf. on World Wide Web. Edinburgh: ACM, 2006. 347–356. [doi: 10.1145/1135777.1135830]

[23] Tak BC, Tang CQ, Zhang C, Govindan S, Urgaonkar B, Chang RN. vPath: Precise discovery of request processing paths from black-box observations of thread and network activities. In: Proc. of the 2009 Conf. on USENIX Annual Technical Conf. San Diego: USENIX Association, 2009. 259–272.

[24] Anandkumar A, Bisdikian C, Agrawal D. Tracking in a spaghetti bowl: Monitoring transactions using footprints. ACM SIGMETRICS Performance Evaluation Review, 2008, 36(1): 133–144. [doi: 10.1145/1384529.1375473]

[25] Sengupta B, Banerjee N, Bisdikian C, Hurley P. Tracking transaction footprints for non-intrusive end-to-end monitoring. Cluster Computing, 2009, 12(1): 59–72. [doi: 10.1007/s10586-008-0066-7]

[26] Wang T, Perng CS, Tao T, Tang CQ, So E, Zhang C, Chang R, Liu L. A temporal data-mining approach for discovering end-to-end transaction flows. In: Proc. of the 2008 IEEE Int'l Conf. on Web Services. Beijing: IEEE, 2008. 37–44. [doi: 10.1109/ICWS.2008.59]

[27] 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 IEEE/ACM Int'l Conf. on Software Engineering: Companion (ICSE-Companion). Gothenburg: ACM, 2018. 323–324. [doi: 10.1145/3183440.3194991]

[28] Basiri A, Behnam N, De Rooij R, Hochstein L, Kosewski L, Reynolds J, Rosenthal C. Chaos engineering. IEEE Software, 2016, 33(3): 35–41. [doi: 10.1109/MS.2016.60]

[29] 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]

[30] Lai CA, Kimball J, Zhu T, Wang QY, Pu C. milliScope: A fine-grained monitoring framework for performance debugging of n-tier Web services. In: Proc. of the 37th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS). Atlanta: IEEE, 2017. 92–102. [doi: 10.1109/ICDCS.2017.228]

[31] Chen MY, Kiciman E, Fratkin E, Fox A, Brewer E. Pinpoint: Problem determination in large, dynamic internet services. In: Proc. of the 2002 Int'l Conf. on Dependable Systems and Networks. Washington: IEEE, 2002. 595–604. [doi: 10.1109/DSN.2002.1029005]

[32] Killian CE, Anderson JW, Braud R, Jhala R, Vahdat AM. Mace: Language support for building distributed systems. ACM SIGPLAN Notices, 2007, 42(6): 179–188. [doi: 10.1145/1273442.1250755]

[33] Pham C, Wang L, Tak BC, Baset S, Tang CQ, Kalbarczyk Z, Iyer RK. Failure diagnosis for distributed systems using targeted fault injection. IEEE Trans. on Parallel and Distributed Systems, 2017, 28(2): 503–516. [doi: 10.1109/TPDS.2016.2575829]

[34] 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 2018 ACM Symp. on Cloud Computing. Carlsbad: ACM, 2018. 149–161. [doi: 10.1145/3267809.3267823]

[35] Liu HF, Zhang JJ, Shan HS, Li M, Chen Y, He XF, Li XW. JCallGraph: Tracing microservices in very large scale container cloud platforms. In: Proc. of the 12th Int'l Conf. on Cloud Computing. San Diego: Springer, 2019. 287–302. [doi: 10.1007%2F978-3-030-23502-4_20]

[36] Bauer M, van der Aa H, Weidlich M. Estimating process conformance by trace sampling and result approximation. In: Proc. of the 7th Int'l Conf. on Business Process Management. Vienna: Springer, 2019. 179–197. [doi: 10.1007%2F978-3-030-26619-6_13]

[37] Las-Casas P, Mace J, Guedes D, Fonseca R. Weighted sampling of execution traces: Capturing more needles and less hay. In: Proc. of the 2018 ACM Symp. on Cloud Computing. Carlsbad: ACM, 2018. 326–332. [doi: 10.1145/3267809.3267841]

[38] Las-Casas P, Papakerashvili G, Anand V, Math J. Sifter: Scalable sampling for distributed traces, without feature engineering. In: Proc. of the 2019 ACM Symp. on Cloud Computing. Santa Cruz: ACM, 2019. 312–324. [doi: 10.1145/3357223.3362736]

[39] Yan Y, Chen LJ, Zhang Z. Error-bounded sampling for analytics on big sparse data. Proc. of the VLDB Endowment, 2014, 7(13): 1508–1519. [doi: 10.14778/2733004.2733022]

[40] Hoeffding W. Probability inequalities for sums of bounded random variables. In: Fisher NI, Sen PK, eds. The Collected Works of Wassily Hoeffding. New York: Springer, 1994. 409–426. [doi: 10.1007%2F978-1-4612-0865-5_26]

[41] Dinn AE. Flexible, dynamic injection of structured advice using byteman. In: Proc. of the 10th Int'l Conf. on Aspect-oriented Software Development Companion. Porto de Galinhas: ACM, 2011. 41–50. [doi: 10.1145/1960314.1960325]

附中文参考文献:

[4] 杨勇, 李影, 吴中海. 分布式追踪技术综述. 软件学报, 2020, 31(7): 2019–2039. http://www.jos.org.cn/1000-9825/6047.htm [doi: 10.13328/j.cnki.jos.006047]

黄梓程(1996－), 男, 硕士, 主要研究领域为微服务, 分布式追踪, 故障注入.

余广坝(1995－), 男, 博士, CCF 学生会员, 主要研究领域为分布式系统, 云计算, 智能运维.

陈鹏飞(1987－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为分布式系统, 智能运维, 云计算, 微服务, 区块链.

陈泓仰(1997－), 男, 博士生, CCF 学生会员, 主要研究领域为分布式系统, 微服务, 云计算.