计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(1):175−181 [doi: 10.15888/j.cnki.csa.008239]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于动态采样策略的微服务链路追踪方法①

颜复海[1,2], 李培军[1], 张泽华[3], 陈文辉[3], 许舒人[1]

[1](中国科学院 软件研究所 软件工程技术研究开发中心, 北京 100190)
[2](中国科学院大学, 北京 100490)
[3](厦门物之联智能科技有限公司, 厦门 361022)
通信作者: 许舒人, E-mail: shuren@iscas.ac.cn

摘　要: 针对现有开源分布式链路追踪系统或框架的采样策略存在收集过多无助于故障分析等任务的正常执行的跟踪的问题, 本文提出一种动态采样策略, 通过采样策略树和执行轨迹图两种数据结构解决了跟踪采样率自动调整问题和如何快速准确地找到需要调整跟踪采样率的服务的问题, 并通过两者的协作提高异常执行的跟踪占比. 实验结果表明, 本文方法在任一时间段内均有效提高了异常执行的跟踪占比.

关键词: 微服务; 分布式链路追踪; 故障分析; 采样策略

引用格式: 颜复海,李培军,张泽华,陈文辉,许舒人.基于动态采样策略的微服务链路追踪方法.计算机系统应用,2022,31(1):175–181. http://www.c-s-a.org.cn/1003-3254/8239.html

Microservices Tracing Based on Dynamic Sampling Strategy

YAN Fu-Hai[1,2], LI Pei-Jun[1], ZHANG Ze-Hua[3], CHEN Wen-Hui[3], XU Shu-Ren[1]

[1](Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)
[3](Xiamen Wuzhilian Intelligent Technology Co. Ltd., Xiamen 361022, China)

Abstract: To address the problem that the sampling strategies of existing open-source distributed tracing systems or frameworks collect redundant traces of normal executions that are less helpful to tasks such as fault analysis, we propose a dynamic sampling strategy. With two data structures of sampling strategy tree and execution trace graph, it realizes the automatic adjustment of trace sampling rate and finds the way to quickly and accurately determine services that need to adjust the trace sampling rate. The collaboration between the above data structures enhances the trace proportion of anomalous executions. The experimental results show that the proposed method effectively improves the trace proportion of anomalous executions in any time period.

Key words: microservices; distributed tracing; fault analysis; sampling strategy

1　引言

由于具备高可用性、支持动态伸缩与分布式部署等特点, 微服务架构已成为云原生时代设计与构建大型分布式应用系统的主要技术架构之一. 在微服务架构中, 应用系统被划分为多个高内聚、功能单一、可独立部署的服务, 并通过服务之间的轻量级远程调用来构建复杂的业务逻辑[1,2].

可观察性是指系统可由外部输出推断其内部状态的程度. 对微服务架构而言, 当服务数量庞大时, 服务之间的调用关系变得复杂, 使得故障分析与排查的困难程度显著上升, 因此微服务架构也面临着如何提高可观察性的问题. 分布式链路追踪 (distributed tracing) 是提升微服务架构的可观察性的关键技术之一, 其通过收集应用系统处理服务请求的详细信息来为系统行

① 收稿时间: 2021-03-14; 修改时间: 2021-04-07; 采用时间: 2021-04-20; csa 在线出版时间: 2021-12-17

© 中国科学院软件研究所 http://www.c-s-a.org.cn

为建模, 以帮助开发与运维人员获得系统运行时的全局视角[3].

应用系统对一次外部请求的完整处理称为一次执行 (execution). 以一次执行中是否出现异常为依据可将该次执行划分为正常执行或异常执行. 跟踪 (trace) 是对执行的记录, 一个跟踪对应一次执行, 其内容包含该次执行中各服务之间的调用关系以及各服务处理调用请求的详情与结果. 跟踪由插桩代码生成并上报至负责汇总处理的远程进程.

出于减小对应用系统的性能影响、节约存储资源等方面的考虑, 在分布式链路追踪中通过特定的采样策略来只收集一部分执行的跟踪[3–5]. 常见的开源链路追踪系统或框架实现的采样策略有固定采样、概率采样、限速采样、自适应采样等[6–8]. 以上采样策略虽然实现简单且有效避免了插桩代码对应用系统造成较大工作负载, 但是存在收集过多无助于故障分析与排查等任务的正常执行的跟踪等问题[4,5]. 以概率采样为例, 收集的跟踪当中各类执行的跟踪占比与其出现的概率一致, 因此高频的正常执行的跟踪占比将远大于低频的异常执行. 但是对于故障分析这一需求而言, 异常执行的跟踪占比越大、跟踪数量越多则提供给系统管理人员进行故障分析等任务的信息量越大[4,5].

针对以上问题, 本文提出一种动态采样策略: 通过采样策略树解决如何自动调整跟踪采样率的问题; 通过执行轨迹图解决如何快速准确地找到需要调整跟踪采样率的服务的问题; 通过以上两种数据结构的协作提高异常执行的跟踪占比. 基于以上工作, 本文实现了原型系统并通过实验对动态采样策略的有效性进行了验证.

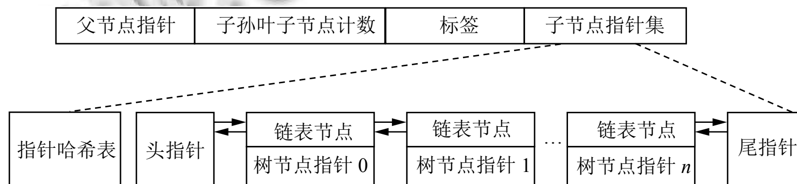

图 1 采样策略树的树节点结构

2 基于采样策略树的跟踪采样率自动调整方法

2.1 问题分析

任一执行类型的调用链路中最先处理外部请求的服务称为该类执行的入口服务. 提高入口服务的跟踪采样率将会增加其对应执行类型的跟踪数量, 从而消耗更多的存储资源. 若要提高存储资源的利用率, 那么当提升某一入口服务的跟踪采样率时就应适当减小其他部分入口服务的跟踪采样率, 使得跟踪数量保持动态平衡. 本文设计了一种称为采样策略树的数据结构来解决跟踪采样率的自动调整问题.

2.2 采样策略树的定义

定义 1. 一棵 m 阶的采样策略树是满足以下性质的 m 叉树:

1) 树中任一节点至多有 m 个子节点;

2) 除根节点外, 其他节点要么没有子节点, 要么子节点数大于 1.

图 1 所示为采样策略树的树节点结构. 父节点指针为指向该树节点的父节点的指针. 若树节点为根节点, 则父节点指针为空. 子孙叶子节点计数为以当前树节点为根节点的 m 叉树中所有叶子节点的计数. 树中只保存入口服务. 标签为当前树节点保存的服务的唯一标识符. 所有服务只保存在叶子节点中, 因此枝干节点的标签的值为空. 子节点指针集为固定容量的集合, 用于保存当前节点的所有子节点的指针, 其底层由一个指针哈希表和一个双向链表来实现.

2.3 采样策略树的操作

采样策略树支持的操作有插入、剪枝、生成和提升等. 插入操作将新服务的叶子节点插入到离根节点尽可能近的位置. 剪枝操作将某一服务对应的叶子节点从树中删除.

生成操作以路径概率积作为树中服务的跟踪采样率. 路径概率积的定义如下:

定义 2. 设叶子节点 l 且叶子节点的父节点不为根节点 r, 其到根节点的路径为 $s=(l,n_1,n_2,\cdots,n_k,r)$, n_i 为路径上的枝干节点, 则该叶子节点的路径概率积为:

$$p=\frac{1}{N_r}\cdot\prod_{i=1}^{k}\frac{1}{N_{n_i}} \tag{1}$$

© 中国科学院软件研究所 http://www.c-s-a.org.cn

其中, N_x 为节点 x 的子节点数量; 若叶子节点 l 的父节点为根节点, 则该节点的路径概率积为:

$$p = \frac{1}{N_r} \tag{2}$$

提升操作将提升树中服务的跟踪采样率, 其伪代码如算法 1 所示. 提升操作内部包含了降级操作, 遭到降级的叶子节点, 其保存的服务的跟踪采样率也随之降低.

算法 1. 采样策略树的提升操作算法

输入: 叶子节点 leafNode
1. **function** Promote(leafNode)
2. **if** leafNode 的父节点为根节点 **then**
3. **return**
4. **end if**
5. grandparent **:=** leafNode 的祖父节点
6. parent **:=** leafNode 的父节点
7. 从 parent 的子节点集合中删除 leafNode
8. **if** grandparent 的子节点数小于最大值 **then**
9. 将 leafNode 添加到 grandparent 的子节点集中
10. 将 leafNode 的父节点指针指向 grandparent
11. **if** parent 只剩一个子节点 **then**
12. 将 parent 唯一的子节点添加到 parent 的父节点的子节点集合当中, 删除 parent //路径压缩
13. **else**
14. 更新 parent 的子孙叶子节点计数值
15. **end if**
16. **else**
17. lruNode **:=** grandparent 的最近最久未使用子节点
18. **if** parent 的子节点数大于 2 **then**
19. Merge(lruNode, leafNode)
20. **else**
21. 替换 lruNode 与 leafNode 的位置
22. **enf if**
23. 更新 parent 的子孙叶子节点计数值
24. **end if**
25. **end function**
26. **function** Merge(a, b)
27. 在 a 的父节点与 a 之间插入一个新的枝干节点
28. 将 a, b 的父节点指针指向新的枝干节点
29. 将 a, b 移动到新枝干节点的子节点集合当中
30. **end function**

图 2 为一棵 3 阶采样策略树的发生降级的提升操作示例. 一个节点的子节点以从左到右的顺序表示其最近被使用过的时间顺序, 即最右边的为最近最久未被使用的子节点. 需要被提升的服务为 D. 由于 grandparent 的子节点数量已达最大值 3, 因此需要降级该节点的最近最久未被使用的子节点 C, 使其与 D 交换位置.

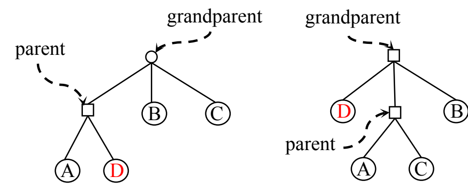

图 2 发生降级的提升操作

2.4 跟踪采样率的自动调整

给定服务并找到对应的叶子节点之后, 提升操作将改变该叶子节点处于树中的位置使其离根节点更近. 根据路径概率积的定义可推导出: 一个服务的叶子节点被提升之后, 该服务根据路径概率积计算所得的跟踪采样率随之提高. 提升操作可能触发降级操作, 即某一服务的叶子节点被下放到树的更深处, 因此被降级的叶子节点的服务的跟踪采样率会随之降低. 跟踪采样率的变化方向与其是否被提升有关且执行提升操作的条件可自行设定. 跟踪采样率变化的幅度与树结构的变化有关, 具体数值取决于所涉及树节点的子节点数.

在提升操作中加入降级操作是为了提高存储资源的利用率. 一个服务的吞吐量和跟踪采样率共同决定了以该服务作为入口服务的执行类型的跟踪数量. 当吞吐量不变时, 跟踪采样率与跟踪数量呈正相关. 根据路径概率积的定义可推导出: 采样策略树中所有服务根据路径概率积计算所得的跟踪采样率总和为 1. 因此, 当提高其中一个服务的跟踪采样率时, 必定导致树中另外一部分服务的跟踪采样率的降低, 使得跟踪数量保持动态平衡, 从而有效提高存储资源利用率.

3 基于流言协议的执行轨迹图全局更新算法

3.1 问题分析

入口服务中的插桩代码做出采样决策并将其写入请求上下文之后, 采样决策会随着调用链路传播给该入口服务递归调用的其他服务, 并且各服务根据请求上下文中的采样决策判断是否上报跟踪, 因此任一服务的跟踪采样率等于其入口服务的跟踪采样率. 当某一非入口服务出现异常时, 要提升其跟踪采样率以提高异常执行的跟踪占比, 则需找到与之对应的入口服务. 针对以上问题, 本文利用执行轨迹图这一数据结构来刻画服务之间的调用关系.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

3.2 执行轨迹图的定义

定义 3. 执行轨迹图 $G=(V,E)$ 是一个有向图, 其中, 节点集 V 表示服务集合, 边集 E 表示调用关系集合, 边的箭头指向被调用的服务.

图 3 为一个执行轨迹图示例. A 为入口服务, 且 A 调用服务 B 与服务 C 来完成其本身的请求处理. 图中其他服务的调用关系与之类似.

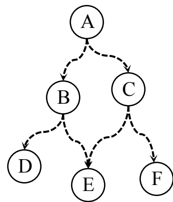

图 3　执行轨迹图示例

3.3 全局更新算法

执行轨迹图的更新是一个动态的过程, 即每次发现新的服务或调用关系, 就把表示新服务的顶点、表示新调用关系的边添加到执行轨迹图当中. 基于执行轨迹图, 在给定任一服务的前提下, 可通过图搜索算法快速找到该服务对应的入口服务. 在实际应用中, 出于容灾和性能上的考虑, 需要冗余地部署跟踪收集进程, 因此需要在多个跟踪收集进程之间同步执行轨迹图, 使得所有执行轨迹图的副本能够反映各服务之间的实时调用关系. 本文提出一种基于流言协议的执行轨迹图全局同步算法以解决上述问题.

图 4 为本文用到的基于流言协议的组件: 种子、注册中心, 以及两者的通信关系. 注册中心用于种子发现, 即当新的种子加入时向注册中心登记其路由信息. 种子定期向注册中心发送心跳信号并获得所有其他种子的实时路由信息, 以及将新调用关系封装成消息散播给其他种子. 在实现中, 每个跟踪收集进程都会维护一个种子, 而注册中心是全局唯一的. 当跟踪收集进程发现新调用关系时, 会向本地的种子发起消息散播请求, 将新调用关系同步给其他跟踪收集进程中的执行轨迹图. 执行轨迹图的全局更新算法伪代码如算法 2 所示. 算法 2 参照了流言协议的 SIR 模型[9], 即每个种子对于任一消息在任一时刻只对应以下 3 种状态之一: S (susceptible), 未收到该消息; I (infected), 已收到该消息且正在参与该消息的散播过程; R (removed), 已收到该消息且已退出该消息的散播过程.

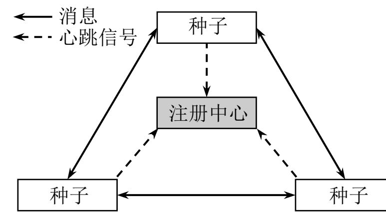

图 4　基于流言协议的组件及通信关系

算法 2. 执行轨迹图的全局更新算法

输入: 新调用关系 newRel
1. **function** Synchronize(newRel)
2. msgID := 由 snowflake 算法生成全局唯一消息 ID
3. msg := (msgID, newRel) //消息格式
4. Monger(msg)
5. **end function**
6. **function** Monger(msg)
7. **if not** msg 的 ID 存在于 LRU 缓存中 **then**
8. 提取调用关系, 更新执行轨迹图
9. 将 msg 的 ID 放入 LRU 缓存
10. **end if**
11. **if not** 本地节点对 msg 处于 R 状态 then
12. peers **:=** 随机选取 n 个对等节点
13. **for each** 对等节点 peer **in** peers:
14. 以 msg 为参数, 远程调用 peer 的 Monger 方法
15. **end for**
16. 以概率 p 转换成 R 状态, 否则为 I 状态
17. **end if**
18. **end function**

4 原型系统

4.1 系统架构

本文参考开源分布式链路追踪系统 Jaeger 的数据模型与架构并参照 OpenTracing 规范[10] 实现了原型系统, 其系统架构如图 5 所示. 插桩代码负责生成、上报跟踪以及周期性地向配置服务器拉取采样策略. 代理负责为插桩代码屏蔽配置服务器和收集器的路由信息. 配置服务器负责处理采样策略的拉取请求以及更新采样策略的请求并且通过策略管理器维护全局唯一的采样策略树. 注册中心用于种子发现, 维护所有种子的路由信息. 种子用于执行全局更新算法. 收集器负责收集、分析以及持久化跟踪.

4.2 跟踪收集过程

跨距 (span) 是跟踪的基本组成单元, 一个跨距记录一个服务处理一次请求处理的详细信息以及调用关系. 收集一个跟踪即收集该跟踪的所有跨距. 跟踪收集的详细过程如下:

(1) 当外部请求到达应用系统时, 入口服务中的插

© 中国科学院软件研究所　http://www.c-s-a.org.cn

桩代码将做出采样决策并将其写入请求上下文使其在调用链路中传播;

(2) 所有服务中的插桩代码都将在其嵌入的服务处理请求之前分析请求上下文并创建一个跨距, 且请求上下文中的采样决策会被写入跨距上下文;

(3) 待请求处理完成之后, 插桩代码将根据跨距上下文中的采样决策判断是否上报该跨距, 若执行采样则将跨距发送至代理, 再由代理上报至收集器;

(4) 收集器分析跨距获得调用关系, 判断是否为新调用关系, 若为新调用关系则执行全局更新算法;

(5) 收集器调用评估器判断该次执行是否为预先定义的执行类型, 若是则向配置服务器发送请求以提升该执行的入口服务的跟踪采样率;

(6) 收集器将跨距保存至数据库中.

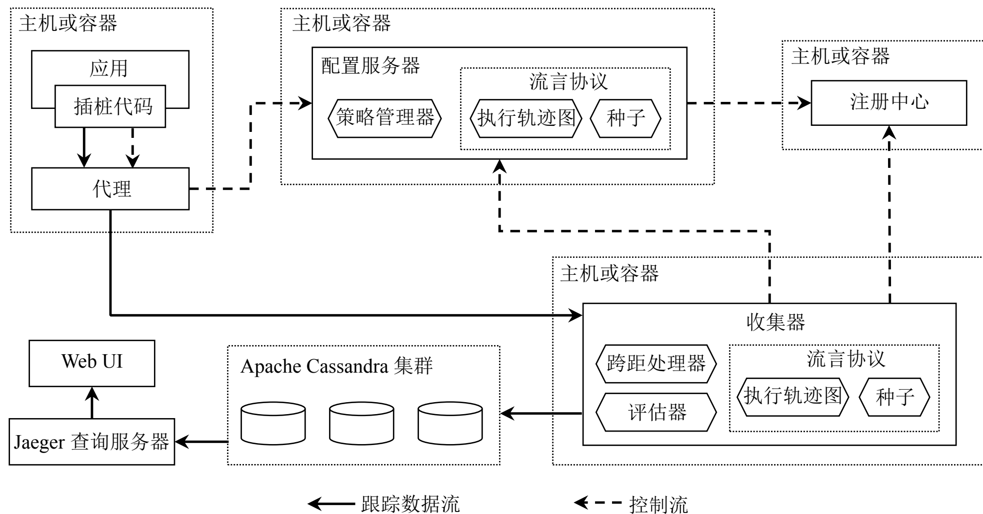

图 5　原型系统的架构

4.3　采样策略拉取过程

插桩代码周期性地发送拉取采样策略的请求, 该请求包含两个参数: 服务唯一标识符和该服务从上一次发送请求到目前为止这一时间段内的每秒请求数, 即 QPS (request per second) 值.

配置服务器为所有入口服务维护一个 QPS 值的哈希表. 当接收到采样策略拉取请求时, 配置服务器首先会用参数中的 QPS 更新该哈希表, 然后通过以下等式计算采样率:

$$S = \min\left(\max\left(O_{\rm sst}\cdot W_{\rm qps}\cdot\alpha, S_{\min}\right), 1\right) \tag{3}$$

其中, $O_{\rm sst}$为采样策略树的生成操作的路径概率积计算结果, $\alpha\in(0,+\infty)$为缩放因子, $S_{\min}$ 为最小采样率, $W_{\rm qps}$ 为 QPS 权重函数的输出, 其计算方式如下:

$$W(q_i)=\frac{\sum_{k=1}^{E}\frac{1}{q_k}}{q_i} \tag{4}$$

其中, q_i 表示第 i 个入口服务的实时 QPS 值, E 为入口服务数.

计算所得的采样率会被封装成概率采样为底层采样方式的采样策略作为响应返回给插桩代码, 由插桩代码分析响应结果并据此更新本地的采样策略.

4.4　采样策略树与执行轨迹图的协作过程

为评估器设置评估条件作为判断 HTTP 服务的某次执行是否为异常执行的依据: HTTP_STATUS_CODE≠200.

图 6 所示为 HTTP 服务出现一次异常执行时采样策略树与执行轨迹图的协作过程.

(1) 分析调用服务 E 时由插桩代码生成的跨距, 调用评估器分析其标签, 发现其满足评估条件;

(2) 调用执行轨迹图的查询接口, 由其利用图搜索算法找到服务 E 的入口服务, 即服务 A;

(3) 调用一棵 2 阶采样策略树的提升操作接口, 提升服务 A 的叶子节点并降级服务 X 的叶子节点以提升服务 A 的跟踪采样率、降低服务 X 的跟踪采样率.

实际上执行类型是高度可定制的, 可根据不同的任务与场景为评估器设置不同的评估条件, 并由插桩代码在预设的异常情况发生时为跨距设置与评估条件对应的标签名称以及标签值.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

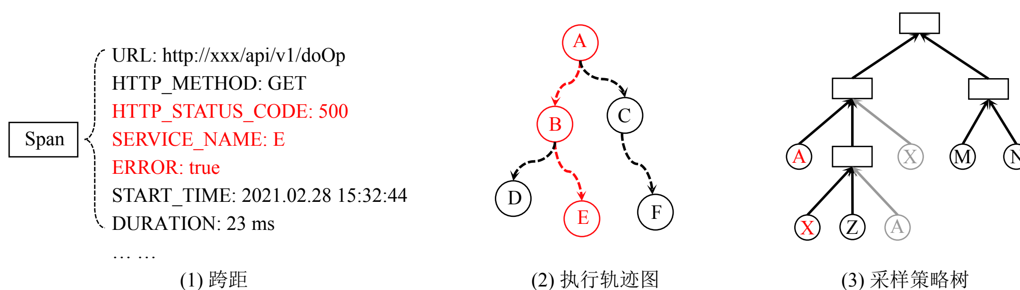

图 6　采样策略树与执行轨迹图的协作过程

5　实验验证与分析

5.1　实验方案

本文在 3 台主机上部署 Kubernetes 集群、Istio 服务网格并开发与部署对应 8 个执行类型的 32 个 HTTP 微服务作为模拟云环境部署的实验环境. 每台主机的操作系统为 CentOS 7.9, 处理器为 Inter(R) Xeon(R) E5-2620 @ 2.00 GHz, 内存容量为 32 GB, Kubernetes 版本为 1.19.0, Istio 版本为 1.7. 本文使用基于 Python 开发的压测工具 Locust 来模拟多用户的并发请求. 在实验中以容器组 (pod) 的形式冗余部署了 3 个收集器.

本文着重关注两个实验指标: 任一时间段内各执行类型的跟踪占比和跟踪数. 本文将比较固定采样、概率采样、限速采样、自适应采样以及动态采样的实际表现. 以上各采样策略的实验参数如表 1 所示.

表 1　实验中各采样策略的参数

采样策略	参数名	参数值	说明
固定采样	always_sample	true	为true时, 收集所有跟踪
概率采样	sampling_rate	0.01	采样率
限速采样	max_traces_per_second	20	每秒最大采样次数
自适应采样	min_sampling_rate	0.01	最低采样率
动态采样	order_sst	4	采样策略树的阶数, 即最大子节点数
	min_sampling_rate	0.01	最低采样率
	amplification_factor	10	缩放因子

在各采样策略的实验中, 每隔固定长度的时间段则为某一类执行注入故障且在对下一类执行注入故障之前, 修复上一类执行的故障. 注入的故障类型为内部错误, 具体表现形式为调用服务所返回的 HTTP 状态码为 500, 因此在实验中设定评估条件为: HTTP 状态码不等于 200. 当插桩代码捕捉到该异常时会将 HTTP 状态码以及对应的值 500 写入跨距标签.

5.2　实验结果

各采样策略的实验中跟踪占比的变化情况如图 7–图 11 所示. 图 11 表明在动态采样策略的实验中, 对于任一时间段内被注入故障的执行类型的跟踪占比显著上升. 各采样策略的实验中的跟踪数变化情况如图 12 所示. 在动态采样的实验中, 各时间段内被注入故障的执行类型的入口服务的跟踪采样率显著上升, 其对应的跟踪数也显著增多, 但是由于同时降低了另一部分入口服务的跟踪采样率, 因此任一时间段的跟踪数相对于前一时间段没有明显地上升或下降, 使得跟踪数保持动态平衡而异常执行的跟踪占比显著上升, 从而提高了存储资源的利用率.

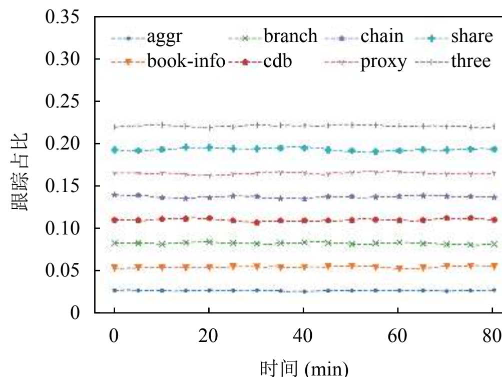

图 7　固定采样实验的跟踪占比变化

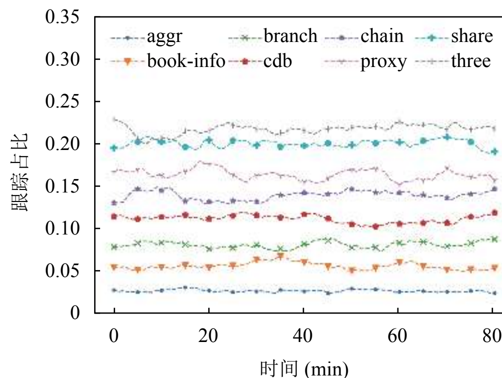

图 8　概率采样实验的跟踪占比变化

© 中国科学院软件研究所 http://www.c-s-a.org.cn

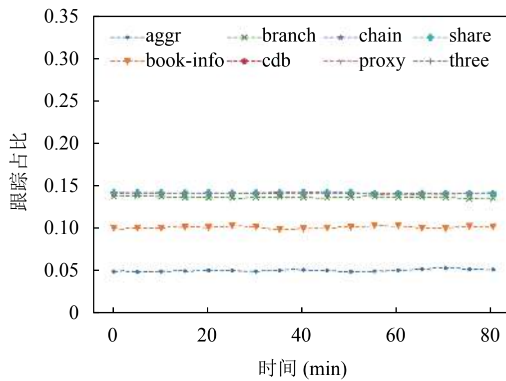

图 9　限速采样实验的跟踪占比变化

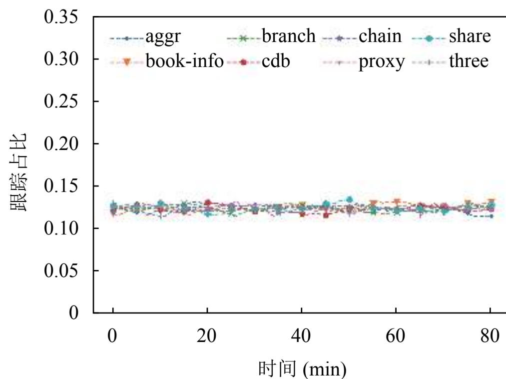

图 10　自适应采样实验的跟踪占比变化

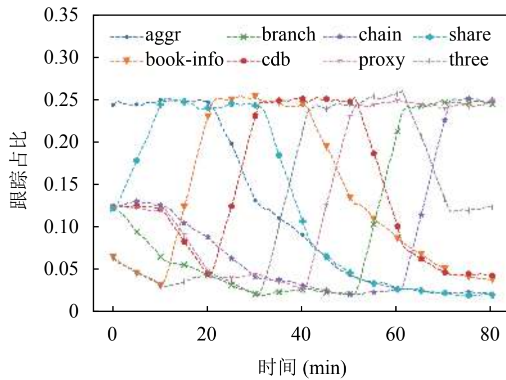

图 11　动态采样实验的跟踪占比变化

实验结果表明, 动态采样有效提升了任一时间段内异常执行的跟踪占比并且高效利用了存储资源.

6　总结

针对现有开源分布式链路追踪系统或框架的采样策略存在的收集过多无助于故障分析等任务的正常执行的跟踪这一问题, 本文提出一种基于动态采样策略的微服务链路追踪方法: 通过基于采样策略树的跟踪采样率自动调整方法解决了跟踪采样率的自动调整问题; 通过基于流言协议的执行轨迹图全局更新算法解决了如何快速准确地找到需要调整跟踪采样率的入口服务的问题. 实验结果表明, 本文方法在任一时间段内均有效地提升了异常执行的跟踪占比并且高效利用了存储资源.

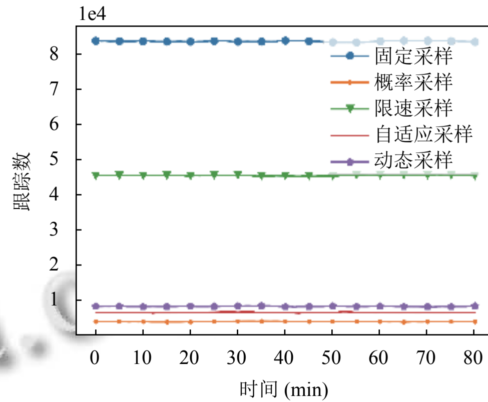

图 12　跟踪数随时间的变化情况

参考文献

1 Namiot D, Sneps-Sneppe M. On micro-services architecture. International Journal of Open Information Technologies, 2014, 2(9): 24–27.

2 Newman S. Building Microservices. Sebastopol: O’Reilly Media, Inc., 2015: 280.

3 Sigelman BH, Barroso LA, Burrows M, *et al*. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report, 2010.

4 Las-Casas P, Mace J, Guedes D, *et al*. Weighted sampling of execution traces: Capturing more needles and less hay. Proceedings of the ACM Symposium on Cloud Computing. Carlsbad: ACM, 2018. 326–332. [doi: 10.1145/3267809.3267841]

5 Las-Casas P, Papakerashvili G, Anand V, *et al*. Sifter: Scalable sampling for distributed traces, without feature engineering. Proceedings of the ACM Symposium on Cloud Computing. Santa Cruz: ACM, 2019. 312–324. [doi: 10.1145/3357223.3362736]

6 Jaeger. Uber technologies. https://www.jaegertracing.io. [2021-04-15].

7 Apache Software Foundation. Apache SkyWalking. https://skywalking.apache.org. [2021-04-15].

8 Twitter. OpenZipkin. https://zipkin.io. [2021-04-15].

9 Di Marzo Serugendo G, Gleizes MP, Karageorgos A. Self-organising software: From natural to artificial adaptation. Berlin Heidelberg: Springer, 2011. 139–162.

10 Cloud Native Computing Foundation. OpenTracing. https://opentracing.io/. [2021-04-15].

© 中国科学院软件研究所 http://www.c-s-a.org.cn