计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2024,33(9):140−152 [doi: 10.15888/j.cnki.csa.009628]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于微服务分布式链路的服务质量优化策略①

佟业新, 曲新奎, 杨皓然, 张军涛, 周明涛

(中航信移动科技有限公司, 北京 100101)
通信作者: 曲新奎, E-mail: xkqu@travelsky.com.cn

摘　要: 微服务架构作为一种敏捷而弹性的软件设计范式, 已经在当今的软件开发领域中取得了广泛的应用. 然而, 随着微服务数量的不断增加, 系统复杂度随之升高, 系统的服务质量随之下降, 如何提升微服务架构下的线上业务服务质量是一个重要命题, 而服务链路的优化是其中的关键挑战. 本文通过对微服务架构下服务链路的深入研究, 提出了链路抽样、链路拓扑生成、强弱依赖判定、循环调用识别、重复无效调用识别等链路分析方法, 并在此基础之上, 实践了一系列包括强弱演练、循环调用分拆、重复调用减支合并、故障自愈、链路溯源等在内的一系列有效的优化策略, 有效提升了微服务架构下的生产运行系统服务质量.

关键词: 微服务; 链路分析; 链路溯源; 强弱依赖; 重复调用

引用格式: 佟业新,曲新奎,杨皓然,张军涛,周明涛.基于微服务分布式链路的服务质量优化策略.计算机系统应用,2024,33(9):140–152. http://www.c-s-a.org.cn/1003-3254/9628.html

Service Quality Optimization Strategy Based on Microservice Distributed Link

TONG Ye-Xin, QU Xin-Kui, YANG Hao-Ran, ZHANG Jun-Tao, ZHOU Ming-Tao

(TravelSky Mobile Technology Ltd., Beijing 100101, China)

Abstract: Microservices architecture, as an agile and resilient software design paradigm, has been widely applied in the field of software development. However, with the increasing number of microservices, the complexity of the systems rises, and the service quality of the system decreases. Enhancing the quality of online business service under the microservices architecture is a critical challenge. Optimization of service links is a key aspect in addressing this challenge. This work conducts an in-depth study of service links under the microservices architecture and proposes various link analysis methods, including link sampling, link topology generation, strong and weak dependency determination, identification of cyclic calls, and recognition of redundant and ineffective calls. Building upon these methods, the study implements a series of effective optimization strategies, such as robust testing, disassembling cyclic calls, reducing and merging redundant calls, fault self-healing, and link tracing. These strategies effectively improve the service quality of production and operation system services under the microservices architecture.

Key words: microservice; link analysis; link tracing; strong and weak dependency; redundant call

微服务架构[1]有效地解决了大规模分布式系统下的服务自动发现与注册、互相通信等分布式扩展性问题[2], 但是随着业务规模的快速增长, 微服务节点数量逐渐增加到 10 万, 甚至百万级别, 微服务节点数越多, 请求路径可能就越长. 在一些复杂的分布式系统中, 一次请求的分布式链路可能涵盖上千个节点, 而请求经过的节点数量越多, 则越可能引发性能和稳定性下降等诸多问题, 因此面对微服务体系下复杂的分布式链

① 基金项目: 国家自然科学基金 (U2033205)
收稿时间: 2024-02-06; 修改时间: 2024-02-23; 采用时间: 2024-05-06; csa 在线出版时间: 2024-07-26
CNKI 网络首发时间: 2024-07-29

路, 如何保障业务服务质量是一个重要命题. 在实际生产微服务环境中往往缺少微服务之间的依赖[3]视图, 在快节奏的产品迭代中, 微服务的依赖关系随着产品需求的变化而频繁发生变动, 给生产服务质量保障带来了挑战, 甚至在不自觉的情况下引入了重复调用、循环调用等架构性问题, 而这些问题在目前生产系统错综中都是以黑盒形式存在, 研发和运维人员无法对这些问题形成直观的认知, 也缺少有效的手段提前发现和解决问题, 为生产系统的性能和稳定性留下了巨大的隐患.

为了解决上述问题, 提升微服务架构下的服务质量, 避免因局部问题导致大规模生产故障, 探索一种及时有效发现分布式架构隐患的方法是十分必要的, 本文深入研究微服务架构下的分布式链路[4]优化策略, 并提出了相应的评价指标. 本文针对上万服务节点规模的实际生产环境中碰到的实际问题, 研究链路抽样、链路拓扑生成、强弱依赖判定、循环调用识别、重复无效调用识别等链路分析方法, 并在此基础之上, 设计并实现了包括强弱依赖演练、循环调用分拆、重复调用减支合并、故障自愈、链路溯源等在内的一系列有效的优化策略, 这些研究在实际的项目中已经得到应用, 在服务质量提升方面起到了很好的效果.

1 背景及相关工作

1.1 分布式链路

随着微服务架构的兴起, 应用程序按功能被拆分成小型、独立并且能够多节点部署的服务, 随着服务数量的不断增多, 使得服务间的调用变得更加复杂, 迫切需要一种技术来追踪和监控这些调用. 分布式链路追踪技术便是为此而生, 它帮助开发者监控请求的调用路径, 分析系统性能瓶颈、故障和延迟, 从而提高系统的可靠性和效率.

分布式链路追踪技术[5]通过在请求进入系统时分配唯一的 traceId, 并在请求的开始、结束、服务调用、数据库查询等关键节点插入代码埋点, 记录请求数据上的 span (请求片段) 和 context (上下文信息), 实现对分布式系统请求的全流程监控. 每个请求的 span 包含了执行时间、操作名称以及时间戳等关键信息, 以确保在多服务器节点、中间件以及跨线程间通过请求头或者其他字段传递 traceId 和 SpanID 时, 保持上下文的一致性, 实现分布式系统中服务的全局追踪.

此外, 分布式链路追踪的数据通常通过集中式的追踪数据收集器对各个服务节点的追踪数据进行接收、存储和管理. 这些数据最终会被存储在专门的存储后端组件, 如 Elasticsearch[6]、Cassandra[7] 等. 通过这种集中式的数据存储, 系统能够更方便地进行查询和分析, 以获取有关请求的执行路径.

基于前述的技术理论, 目前有多种成熟的工具实现了分布式链路追踪, 包括 Jaeger[8]、OpenTracing[9]、SkyWalking[10]、Zipkin[11]等, 以 Zipkin 为例子, 该工具的设计和实现受到 Twitter 的 Dapper[5]论文启发, 通过客户端插桩技术实现数据的收集. 每个服务节点都需要集成 Zipkin 客户端库生成 span 并传递给 Zipkin 数据收集器. Zipkin 的系统架构由数据收集器、查询服务和存储组件组成, 数据收集器接收追踪数据, 查询服务提供查询界面, 而存储模块用于数据的持久化. Zipkin 支持插件、扩展机制以及自定义存储后端等, 以适应不同的存储需求. 如图 1 所示.

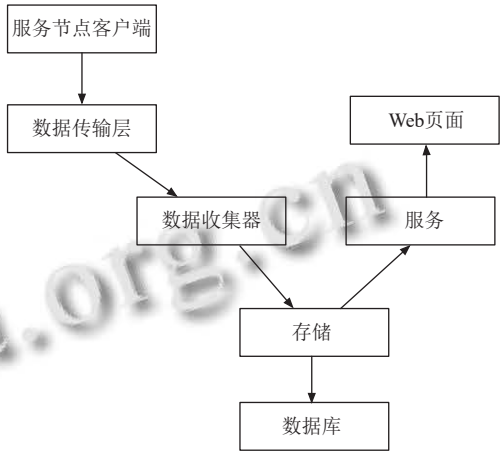

图 1 Zipkin 分布式链路架构

1.2 链路抽样

随着分布式追踪技术在实际生产环境中的应用, 在微服务架构下的上万个微服务节点之间形成了一个个复杂的调用链路并产生了海量的链路数据, 每日产生的链路数据存储量已达到 600 亿条, 占用超过 10 TiB 的存储空间. 这种庞大的数据量以及复杂的链路分支给性能分析和服务依赖分析带来了挑战. 因此, 制定链路抽样策略成为处理和分析海量链路数据的一项关键基础工作.

链路抽样[12]的目的是在减轻数据收集负担, 同时

保持链路数据可用性和准确性, 这通常通过精选一部分进行跟踪来实现. 抽样策略的关键在于决定哪些数据值得追踪以及采用哪种方法, 会直接影响到链路的分析和优化的质量和最终效果. 恰当的抽样策略可以让研发人员在较低的开销下获取到有代表性的链路跟踪数据, 从而有效地优化服务质量. 反之, 如果抽样策略选择不当会导致关键信息的遗漏, 从而影响服务质量优化的效果.

1.3 分布式链路下的服务质量分析

实际生产系统中故障的定位与恢复以及日常服务质量保障都离不开对分布式链路的分析. 故障的发生往往是系统健康状况的指示器, 可能表明系统中存在潜在的问题. 分布式链路追踪作为重要的监控和诊断工具, 提供了帮助定位问题的能力. 通过追踪请求在分布式系统中的路径, 提供了对系统运行情况的全面视图, 有助于快速发现和解决潜在的故障点, 提高系统的可靠性.

面对复杂的分布式链路, 经常遇到容器[13]单点问题频发引起的服务质量问题, 在没有控制住故障爆炸半径时, 可能引起整个服务链路的响应时间延长, 甚至导致服务的不可用. 同时增加了系统的维护难度, 如故障排查、性能优化和监控等. 此外带来的系统的安全风险也比较突出, 可能导致资源利用不均衡和服务调用异常等问题. 提升系统的稳定性、可用性和可维护性, 需要基于分布式链路追踪实时数据及时识别问题并动态地进行容器调度策略调整,

在微服务场景下, 单节点故障问题难发现且隐蔽, 这主要由于微服务架构的分布式特性和异步通信机制. 分散部署且数量庞大的微服务以及复杂的异步通信导致故障不容易被立即发现或追踪. 此外, 微服务系统中存在复杂的服务依赖关系和容器化环境下的动态调整, 这些都增加了系统的不确定性. 此外, 系统的复杂性和节点故障的随机性也加剧了问题的隐蔽性, 使得在系统中调试、监测和发现单节点故障变得更加困难, 应对这些挑战需要采用全面的监控、追踪和测试手段, 以提高对系统内部状态的可见性和诊断能力. 针对分布式链路追踪数据进行抽样和全面的分析, 为研发和运维人员提供可视化、有指导意义的解决方案, 是保障微服务架构下服务质量的关键.

1.4 分布式链路下的信息安全

在微服务架构中, 核心服务有信息安全类的防护需求[14], 但是在实际生产系统环境中, 经常发现来自外部的请求调用接口, 无法分辨这些请求的真实目的, 给系统带来了很大的安全风险. 为了确保系统的安全, 需要了解系统的行为、业务指标并对系统进行分析和改进. 为发现安全风险, 本文提供了基于 RPCContext 的全链路溯源技术框架, 构建一个基于链路统计数据业务流量视图, 实现分布式全链路的业务数据、流量、网络溯源, 全方面监控民航系统的各种指标, 通过业务流量溯源的方式, 排查恶意的请求隐患, 同时也能发现是否有桥接流量进入系统, 全方面地保证系统的可用性和安全性.

1.5 研究思路

基于前文所述的背景及相关工作, 本文采用了从链路的抽样生成、链路问题分析、链路优化、拓展应用逐层递进的形式展开论述, 如图 2 所示.

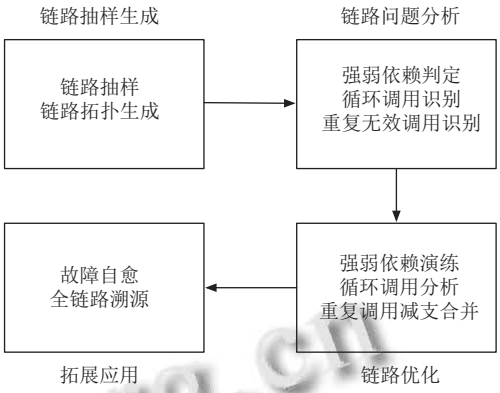

图 2 总体框架

文中首先通过对链路数据抽样并生成链路拓扑, 然后以此为基础, 对链路拓扑中的强弱依赖不合理、循环调用、重复无效调用等问题进行判定和识别, 接下来给出了具体的强弱依赖演练、循环调用分拆、重复调用减支合并等优化方法和机制, 最后在故障自愈、全链路溯源中进行拓展应用. 最终实现了可靠性、性能方面的提升.

2 服务链路分析与优化

2.1 链路抽样分析

要进行服务质量的优化, 需要深入理解和掌握链路抽样的原理和方法. 包括设计和实现有效的抽样策略, 效果的评估, 并据此进行服务质量优化.

传统头部抽样方法[12]是在请求的根源处根据 traceId 的值和全局抽样率做出抽样, 这种方法虽能在

保证链路完整性的情况下减少数据量, 但无法确保业务的覆盖面和异常链路的留存. 请求量大的业务抽样后的数据过多, 会降低分析效率; 反之, 请求量小的业务则因数据过少导致样本不足, 影响分析准确性. 异常链路则反映了系统中存在的问题或故障. 数量虽少但是对故障定位、系统诊断和优化至关重要, 例如, 一个异常的链路由于某个服务响应时间过长而导致整个请求的延迟增加. 本文基于 Dapper[5]改造链路数据, 提出一种既能保留异常链路又能确保业务覆盖完整性的抽样方案.

2.1.1 traceId 设计

本文基于 HTTP 请求的路径对 traceId 进行改造, 使 traceId 包含业务信息, 以在抽样时根据不同的业务采用不同的抽样策略. 通过 trace client 捕获异常并生成异常标识向上传递, 实现了对链路的染色, 使得染色链路在抽样时被直接保留.

HTTP 请求连接客户端和服务端, 每个请求的路径代表着一个业务. 通过将请求的路径信息生成唯一标识并整合入 traceId, 使得在抽样时可以根据 traceId 中读的路径信息为不同业务根据其请求量分配合适的抽样率.

本文设计的 traceId 共 128 位, 分高 64 位和低 64 位两部分. 每 4 位转换为十六进制字符串用于传输和存储, 例如 c4a8e8e606550210d27f2b5a142f7ca6. 改造前高 64 位中的 32 位用于存储 ip, 其余 32 位未使用, 设为 0. 为了在保证 traceId 具有随机性并包含路径信息, 我们重新设计了高 64 位, 如表 1 所示, 利用原本未使用的 32 位存储路经信息, 并保留 ip 和 64 随机数进行组合以保证了链路的唯一性.

表 1 高 64 位 traceId 设计

位数	内容
32	ip
20	路径
4	运行环境
4	抛弃mark
4	留存mark

考虑到请求路径长度不等且信息含量多导致无法编码进 32 位. 本文采用将路径映射到数据库中的自增主键 id, 并选取其中的 20 位以代表最多 1 048 576 (2^{20}) 路径. 当前系统的路径有 4 000 余个, 该方法不仅覆盖了现有路径而且还预留了未来增长的空间. 此外, 我们还利用剩余的 12 位增加额外功能, 包括标记测试环境数据以避免抽样, 以及标记没有价值和具备特殊价值的追踪数据.

traceId 在分布式系统网关请求入口生成, 并随着服务调用向下传递. 服务间调用通过 RPCContext 中的 attachments 或 HTTP 调用的 headers 进行 traceId 传递, 确保了在分布式环境中的有效追踪.

2.1.2 链路抽样

本文实现的链路抽样地过程中, 实时监听并同步路径数据至数据库, 建立 id 与路径的映射并缓存至内存中. 针对不同的路径根据其请求量配置响应的抽样率, 在接收到链路数据后, 取 traceId 的前 16 个字符转为 64 位 bit, 将 64 位右移 12 位, 与 0xfffff 做与操作得到路径 id, 即 traceIdHigh >>> 12 & 0xfffff, 进而根据该 id 获得抽样率进行抽样决策. 如图 3 所示.

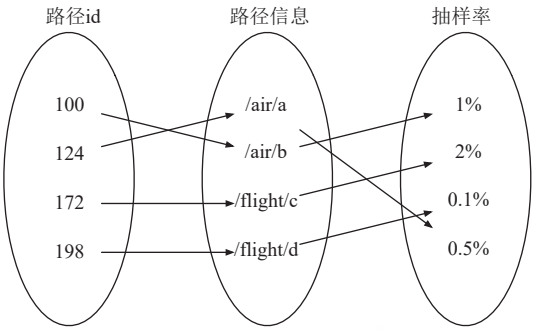

图 3 路径信息映射图

抽样使用 traceId 的低 64 位随机数完成, 并确保其均匀分布在 64 位能表达的数值范围内. 将抽样率乘以 2^{64} 计算得到的最大值作为抽样阈值. 如果低 64 位值小于阈值, 其 trace 将被抽样. 例如当抽样率为 0.1% 时, 只有数值在[0, (2^{64})×0.1%]范围内的 trace 会被抽样, 基于随机数是均匀分布的, 这样可以保证抽样数量的准确性.

2.1.3 异常链路留存策略

traceId 在链路的首个节点生成, 但异常可能发生在链路下游的任何节点, 从而错过 traceId 的初识标记时机. 因此我们通过 span 来携带异常以标记异常链路.

链路由多个 span 构建, 每个 span 记录服务调用数据, 形成跟踪链. 每次服务调用生成的 span 携带唯一的 traceId, 确保从源头到下游节点的一致性. 当发生异常时, 异常信息会被捕获并标记在 span 及响应中. 异常信息像 traceId 一样沿着链路反向传播, 实现异常信息逐级上报, 为故障诊断提供关键数据. 异常链路的标

记引入了一种新的分布式追踪模型, 如图 4 所示, 异常链路被染色, 使得异常路径可以被清晰地识别.

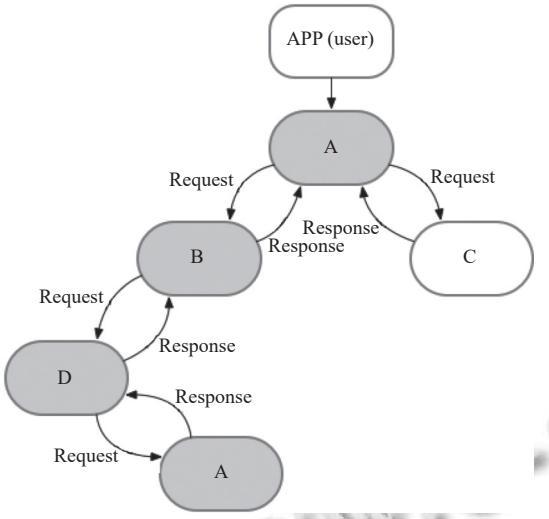

图 4 异常链路染色图

异常留存策略优先级高于抽样机制, 在持久化链路前先判断 span 是否有异常标记, 若有, 该 span 直接保留, 跳过抽样逻辑. 异常留存策略执行后, 使分布式链路避免了因抽样导致异常链路缺失, 为问题排查和服务质量分析提供了关键数据.

2.1.4 全链路拓扑生成

全链路拓扑的生成基于链路抽样之后的分布式链路结构[5]. 首先将分布式链路结构拆解, 获取节点之间的关系, 根据关系形成有向图, 再丰富有向图的服务信息, 形成全链路拓扑图.

如图 5 所示, 该图展示了请求经过的所有服务节点以及服务间的调用关系. 全链路拓扑图为图数据结构[15], 定义为有向图[16]以模拟现实生产环境可能存在循环调用情况.

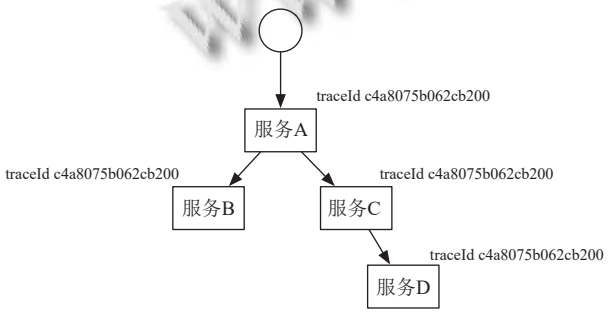

图 5 分布式链路

如图 6 所示, 有向图的方向可以帮助判定顶点 3 和 4 之间有循环调用的情况存在.

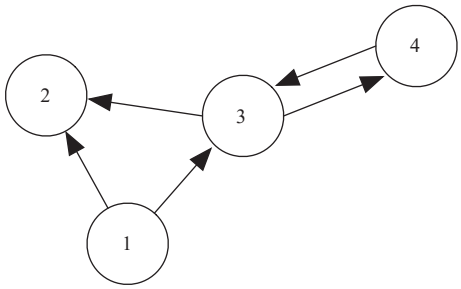

图 6 有向图

首先是生成有向图, 在分布式链路中每个服务调用关系被定义为有向图的边 $v \to w$, 其中 v 和 w 是定点. 通过统计服务节点数量, 确定有向图的顶点数量 Digraph (vertexAmount), 随后将 $v \to w$ 关系添加至 Digraph 内, 即 addEdge($v \to w$) 据此生成有向图.

创建有向图后, 需要映射服务信息以生成全链路拓扑图. 过程中, 实现服务节点与顶点的映射, 如服务节点 a 依赖 b 被映射为 $v \to w$, 保证了无论有向图如何操作, 全链路拓扑图的完整性都得以维持.

2.2 服务链路分析

与经典 Doxygen+Graphviz 工具相比, 本文将要提到的全链路拓扑是通过基于链路数据生成, 侧重于分布式环境下运行时服务见的动态调用关系, 用于理解系统行为、诊断和性能优. 而前者主要基于源代码生成类和继承图等静态结构图, 旨在文档化代码结构和关系, 提升代码可理解性.

2.2.1 重复调用识别

重复调用在单次请求内体现为服务 A 对服务 B 的多次调用, 会导致网络层通信、序列化与反序列化和路由选址的多次开销, 对系统性能造成负面影响, 应当识别并妥善处理.

本文根据分布式链路结构的底层存储, 准确识别重复调用情况. 实现链路由多次请求聚合, 每次请求包含多个服务调用, 每个服务调用由两个名为 span 的数据结构表示, 分别对应客户端 client 和服务端 server, 例如服务 A 调服务 B 会产生两个 span 数据, 标识调用双方. 如图 7 所示.

在一次服务调用中, 若涉及下游调用, 将为这些下游调用生成两个 span, 这 2 个 span 不仅共享一个 span id, 且共享一个 parent id, 指向他们上游 span 的 id. 因此, 一次调用内发生多次下游调用会产生多个 span, 它们具有不同的 id 但相同的 parent id, 通过这种方式可

以追踪调用链中每个请求的来源和结构.

为了检测重复调用, 可以采用如下计算方法.

● 定义服务调用集合 S: 包含所有服务调用.

● 定义 parent id 集合 P: 包含所有的 parent id.

● 定义函数 f: $S\to P$: 将每个服务调用映射到其 parent id.

● 定义计数函数 g: $P\to N$: 计算每个 parent id 的服务调用数量.

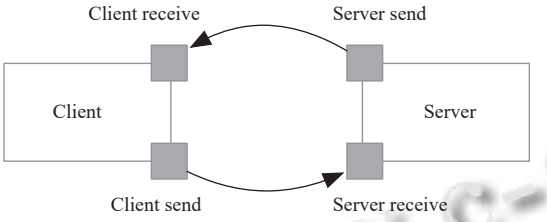

图 7 单次服务调用

通过映射和计数, 如果 $g(f(S)) > 1$, 则标识有来自于同一个上游 span 的多次调用 即发现了重复调用.

2.2.2 循环调用识别

循环调用意味着服务间出现了循环依赖关系. 假设服务 A 依赖 B 且 B 依赖 A, 导致部署时需要 A 和 B 同时部署, 否则会报错. 这种依赖增加了系统的耦合度, 降低了模块内聚, 严重影响了研发效率. 此外循环依赖也影响了业务扩展性, 考虑新的业务场景时, 需确保与现有依赖应用的兼容性, 增加了研发成本, 不利于业务得发展.

本文基于全链路拓扑图, 完成了循环调用识别. 循环调用在图中表现为环. 我们采用深度优先搜索算法来检测有向无环图 (DAG), 利用递归调用栈遍历有向路径. 只要发现有向边 $v\to w$ 并且 w 已经在栈中, 则确认存在环, 标示出循环调用. 反之, 若没有找到这样的边, 便表示图中没有环.

在此基础上, 通过计算强连通分量[17], 对全链路拓扑图中循环调用的服务节点做出了分组.

首先, 通过计算全链路拓扑图的反向图并得出逆后序排列, 可以确定遍历的顺序. 然后, 在拓扑图中按此顺序执行深度优先搜索得出强连通分量, 这些分量通过映射函数转换为对应的服务节点以实现分组. 分组后, 可以直观地从图中看出需要优化处理的服务关系以及健康的、无需处理的服务关系. 如图 8 所示, 例如 0、2、3、4、5 这 5 个节点形成的强连通分量中, 0→5→4→2→0 出现了循环调用, 2→3→2 出现了循环调用, 而节点 6 则很健康, 因为节点 6 依赖的节点 0、4、9 并被没有被依赖节点 6 的节点 7 调用, 没有出现循环调用.

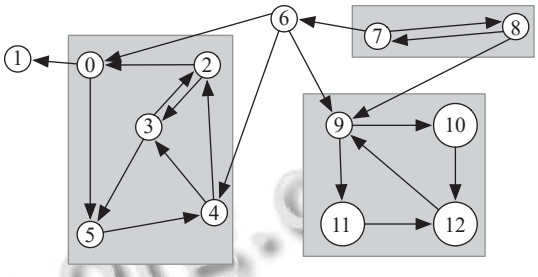

图 8 强连通分量

2.2.3 强弱依赖关系判定

本文探讨强弱依赖判定旨在优化微服务之间的依赖关系, 构建更可靠、可维护且高效的系统. 主要目的包括:

1) 耦合度: 识别强弱依赖关系有助于降低微服务之间的耦合度, 使得单个微服务可以独立地进行修改和测试, 而不会影响到其他微服务.

2) 可维护性: 理解强弱依赖关系有助于简化分布式系统架构, 提高系统的维护性和扩展性.

3) 可靠性: 强依赖的失败可能会导致依赖于它的服务失败. 通过识别这些依赖, 有助于设计出更健壮的错误处理和恢复机制.

强依赖是业务核心流程所必需的依赖, 其调用异常会直接导致业务流程中断. 而弱依赖的异常不会影响业务核心流程的完成. 以查询航班信息为例, 起降时间和地点的查询是强依赖, 因为它们是业务的首要目标. 而机上 WiFi 和电源的查询是弱依赖, 即使出现异常, 也不会中断业务流程, 仅进行日志记录用于后续问题排查即可.

本文定义了两个强弱依赖的概念.

1) 代码强弱依赖: 是指在实际生产环境, 代码执行层面上服务之间的强弱依赖关系.

2) 业务设计强弱依赖: 是在业务设计抽象层面的服务强弱依赖关系.

鉴于业务设计的实施依靠代码的实现, 而两者之间可能因多种因素出现偏差, 为了缩小这种差异, 本文采取的方法是首先基于分布式链路数据生成代码层面的强弱依赖视图. 然后, 通过强弱依赖演练机制验证

代码强弱依赖的准确性, 据此得到实际的代码强弱依赖数据. 这一结果帮助研发团队和业务设计师进行对比、调整和对齐, 以实现业务设计与代码实现的一致性.

本文提出的一种结合全链路拓扑图和微服务演练的方法, 用于对服务的强弱关系进行判定.

首先, 通过在海量拓扑链路中随机抽取 n 条路径, 计算每个服务节点的去重总次数及其在路径中的占比, 以此生成代码的强弱依赖数据. 若某节点在路径中的出现占比超过 95%, 则认为是强依赖; 否则为弱依赖.

$$\left(\frac{count(unique(serviceName)bytrace)}{n} \geqslant 95\%\right) \approx StrongDep \tag{1}$$

然后通过服务演练机制, 基于代码强弱依赖数据进行故障注入实验[18], 模拟依赖服务故障情况, 通过观察是否中断后续请求来判定强弱依赖. 例如 A 依赖于 B 和 C, 且请求会从 A 先调用 B 再调用 C, 当 B 用异常中断请求时、即 A 不再调用后续的 C, 那么判定 B 为 A 的强依赖, 反之为弱依赖. 为了确保实验的通用性、结果一致性, 同时降低实验流量成本, 所有实验均在稳定的生产环境中进行, 并利用微服务泳道机制区分实验与普通用户请求, 避免对用户造成影响. 实验中, 对每个下游依赖服务单独注入故障, 并在连续实验之间设置 10 s 间隔以消除干扰.

2.3 基于服务链路的优化机制

2.3.1 重复调用处理机制

重复调用的优化可以通过两种主要方法来解决: 请求合并和功能内聚.

1) 请求合并: 此策略通过整合多个小型请求为单一大请求以降低网络负担, 特别适合频繁调用多服务的场景, 以减少传输延时. 例如, 在查询航班列表时, 若 B 服务仅支持单一航班查询, 将导致 A 与 B 之间多次通信. 改进后, B 服务支持批量航班查询, 使得 A 仅需一次调用即可获取整个列表. 如图 9 所示.

2) 功能内聚: 该策略将同一个功能对下游的依赖放到同一个服务内调用, 如图 10 所示, 将 A 调用 C 的需求, 由 B 调用 C 合并进行满足.

2.3.2 循环调用处理机制

面对微服务中循环调用可能引发的性能问题、死锁及响应延迟, 本文提出两种主要优化策略: 服务合并与拆分, 以及引入异步通信.

1) 服务合并与拆分: 首先对涉及循环调用的业务逻辑进行深入分析, 然后根据实际情况选择合并或拆分服务. 以 A、B 之间循环调用为例子, 如图 11 所示.

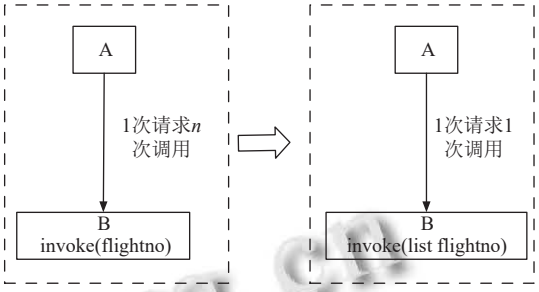

图 9 请求合并

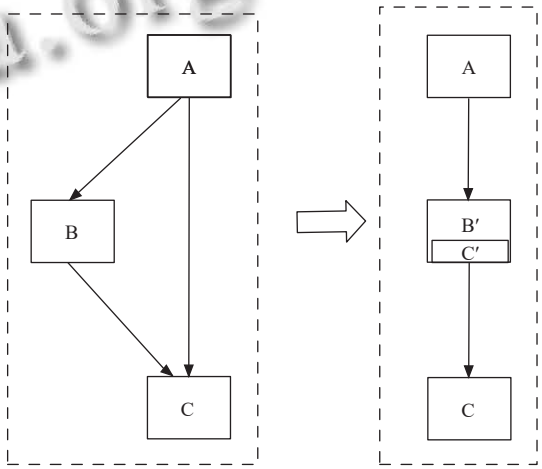

图 10 功能内聚

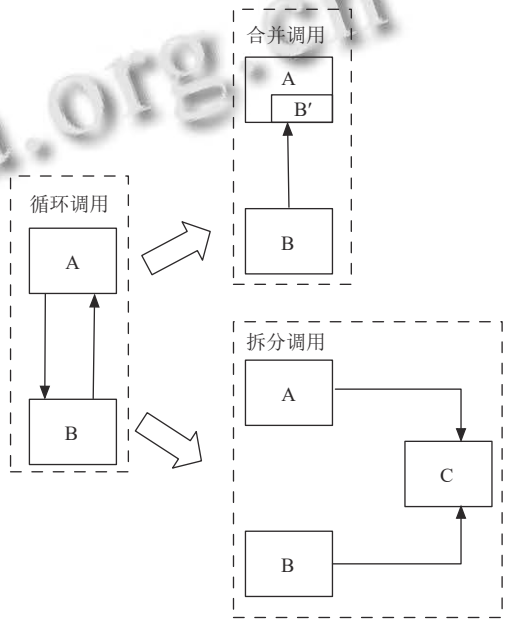

图 11 服务合并与拆分

合并调用: 以 A 和 B 之间的循环调用为例, 可以将 B 服务中 A 所需的功能直接整合到 A 中, 以此去除 A

对 B 的远程调用需求, 仅保留 B 对 A 的调用. 这种方法直接简化了服务间的调用关系, 减少了循环依赖的可能性.

拆分调用: 是将 A 和 B 共享的逻辑抽离, 形成一个独立的服务模块 C. 这样, A 和 B 都可以依赖于 C, 而不直接相互调用, 有效地打破了循环调用的模式.

2) 引入异步通信: 将同步的循环调用改为异步通信, 使用消息队列 MQ[19]、旁路数据同步等技术来解耦服务之间的直接依赖, 这有助于避免死锁和提高系统的弹性, 如图 12 所示.

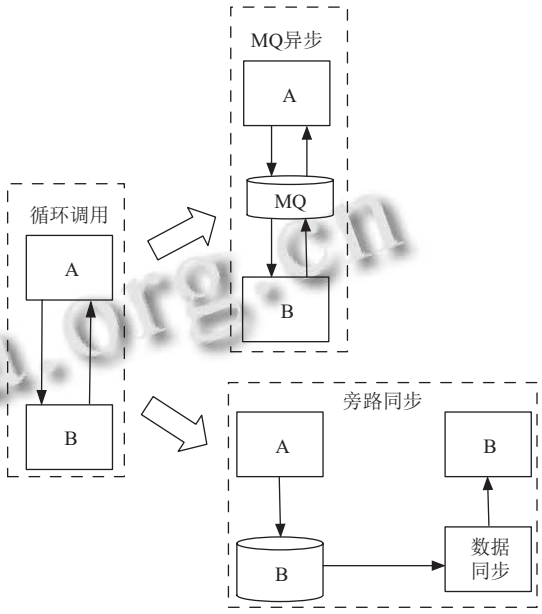

图 12 异步通信

2.3.3 强弱依赖处理机制

第 1.1.4 节介绍的强弱依赖判定方法不仅为业务优化指出了具体方向, 还为验证强弱依赖判定的准确性及代码优化效果提供了依据. 为此, 本文提出了一种基于服务链路的强弱依赖演练方法, 该方法利用拓扑链路强弱依赖分析、微服务泳道隔离、混沌工程[20]、容器调度[21]等技术, 持续稳定地验证服务间的强弱依赖关系, 及时发现架构问题, 预防潜在故障, 确保故障不影响用户体验, 从而增强系统的稳定性.

如图 13 所示, 通过对生产环境中各请求入口 API 进行详尽的强弱依赖演练验证, 本文旨在揭示并解决编码过程中可能出现的、因对下游微服务依赖判断失误或业务设计偏差而造成的系统隐患.

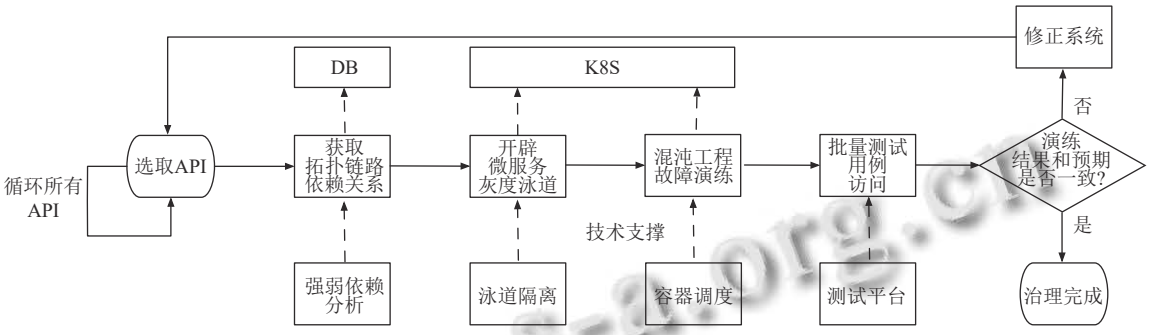

图 13 强弱依赖处理流程

每日定时批量执行如下强弱依赖演练操作, 见算法 1.

算法 1. 强弱依赖演练步骤

1) 从一组 API 中选取一个目标 API 进行测试, 例如, 专注于行程业务的 API.

2) 利用 API 代码分析工具, 获取该 API 的依赖关系, 这包括它所依赖的其他服务或接口. 然后, 根据这些依赖关系, 在一个隔离的环境中 (如 K8S 集群) 为这个 API 及其依赖创建独立的服务环境.

3) 通过故障注入技术, 模拟各种依赖服务故障的情况, 然后执行 API 测试用例. 根据测试结果, 如果 API 在某些依赖故障下表现异常, 那么这些依赖被视为“强依赖”; 否则, 视为“弱依赖”.

4) 将测试结果中的依赖属性 (强依赖或弱依赖) 与先前通过代码分析获得的预期依赖属性进行对比. 存在不一致时, 提供机制让 API 的负责人手动调整依赖属性.

5) 在依赖属性调整后, 重复上述步骤, 直到测试结果与代码分析的依赖属性一致.

6) 业务研发工程师根据代码强弱依赖演练数据, 与业务实际进行比对, 修正代码, 保持与业务设计一致.

3 基于服务链路的拓展应用

3.1 基于服务链路分析的故障自愈

故障自愈和离群摘除是在服务链路分析的一个有效的应用场景. 分布式服务中出现个别节点的宕机或不可预见的故障时, 上报的分布式链路数据中会记录节点异常信息, 通过实时分析带有异常的链路数据可以发现这些疑似故障节点, 把它定义为离群节点, 后续

服务调用过程中排出对离群节点的调用, 即离群摘除. 这个故障出现、链路分析和摘除过程称为故障自愈, 这个机制可以减少个别节点故障带来的失败, 提升系统整体稳定性.

如图 14 所示, 本文结合微服务调用过程中产生的分布式链路数据, 实时上报给时序数据库[22], 自愈模块会对该数据进行实时分析, 以及时发现分布式链路中的坏点服务提供方节点 D, 在必要时进行更新替换节点 D.

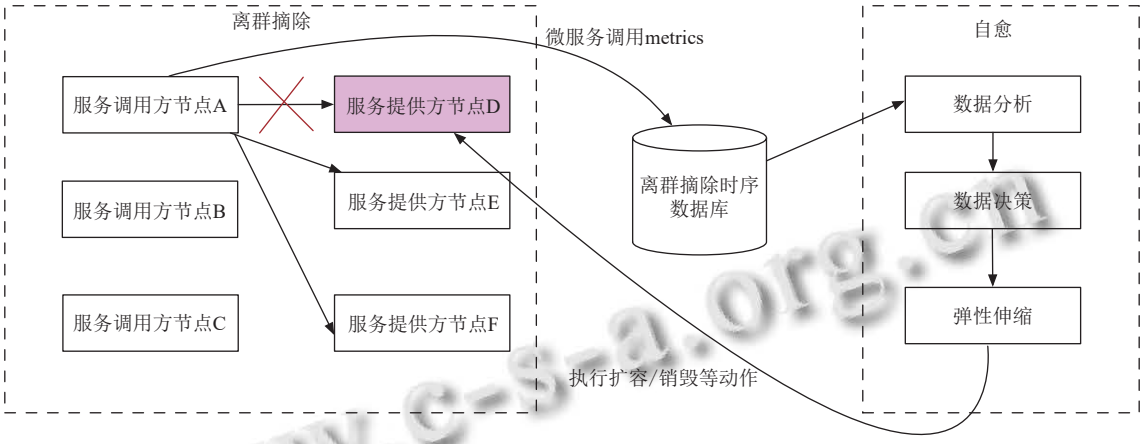

图 14 基于服务链路分析的故障自愈

在分布式链路中, 当单个服务节点出现异常, 如响应延时或高失败率时, 消费者节点能够自动识别并将其临时从调用列表中移除, 此过程称为离群摘除 (算法 2). 这意味着异常节点会被暂时禁用, 避免进一步的调用, 直到后续重试证明其已恢复正常. 若重试成功, 节点重新被认定为可用; 否则, 它将继续被禁用. 此外, 离群摘除机制还包括对应用实例可用性的监测与动态调整, 确保调用的成功执行, 从而显著提高业务稳定性和服务品质.

算法 2. 离群摘除过程

```
输入: 服务资源名称 name, 当前服务提供方节点数量 count, 时间窗口内该服务资源的 metrics, 已经被禁用的服务节点列表 isolationList, 禁用上限百分比 limit, 禁用阈值 thresholds.
IF count ≤ 1 return
 IF isolationList.size < count × limit
   IF metrics.totalRequest > thresholds.requestThreshold
     IF metrics.errorRate > thresholds.errorThreshold
       isolationList.add(name)
     END IF
   IF metrics.avgRT < thresholds.avgRT
     isolationList.add(name)
   END IF
   END IF
   离群摘除实时数据上报
 END IF
END IF
离群恢复过程:
WHILE(true) {
for (isolationService: isolationList) {
  IF isolationService.expireTime() < currentTimeMillis
    isolationList.remove(isolationService);
  END IF
}
}
```

在图 14 的示例中, 一个业务服务集群由节点 D、E、F 组成, 服务调用方 A 分别请求这 3 个节点. 若节点 D 表现异常 (如失败率增加、响应时间延长), 而 A 无法识别并剔除这一异常节点, 其对服务集群的调用失败率可能高达 33.3%. 通过实施离群摘除, 即时识别并移除节点 D, 失败率可显著降低至约 0.1%, 极大提升了服务质量.

服务调用方会将请求后的数据指标 (如响应时间、请求成功与否、服务名称、ip 地址和端口等) 异步记录到根据式 (3) 计算得出的唯一标识符的滑动时间窗内. 每次调用结束后, 系统将触发一次离群摘除计算, 利用表 2 中预设的指标阈值进行判断, 确保异常节点的及时发现和处理.

$$(serviceName : version : group : ip : port) \quad (2)$$

以图 14 所示的离群摘除过程中, 服务节点 A 以随机轮询的方式调用 D、E、F, 如果节点 D 在运行中每分钟调用数超过 20, 失败率大于 20%, 或者响应时间超过 1 s, 则会被自动识别为异常节点并加入不可用列表. 在后续 A 将不会再给 D 分配流量, 直至 D 恢复或者被销毁, 具体流程如下所示.

在上述离群摘除逻辑中, 首先检查服务节点的数量. 若服务节点仅有一个, 表明是单点服务, 不适用于离群摘除, 以确保服务的可用性. 若节点数多于一个, 进一步考虑已隔离节点数是否达到上限, 以防摘除过多节点导致剩余节点过载. 若未达上限, 检查节点的失败率和响应时间是否超出正常阈值. 超出阈值的节点将被隔离, 避免在负载均衡中被选择, 以减少请求失败率. 隔离后的节点在设定时间后可恢复正常, 此过程中收集表 3 中所示的数据支持未来的弹性调整.

表 2　离群摘除各参数阈值

限制指标	默认值
每分钟请求数下限	20 个
每分钟失败率下限	20%
平均响应时间阈值	1 s
摘除比例上限	30%
节点单次摘除时长	60 s
未恢复累计次数上限	10
摘除恢复后的检测时长	60 s

表 3　离群摘除统计数据

指标统计阶段	统计指标 (统计时间窗口内)
离群摘除	摘除开始时间
	失败率
	失败请求总数
	请求总数
	连续摘除次数
	服务提供方已被摘除节点数量
	服务提供方总节点数
	服务提供方摘除节点上限
离群恢复	节点恢复时间
	本次摘除时长

容器自愈模块在接收到离群摘除的异常节点数据之后, 再进一步对性能指标进行确认, 综合比较响应时间、oldgc 时长, 线程状态、线程数量等数据指标, 对容器节点进行更新操作. 在本文的设计中服务调用方节点会离散上报离群摘除的数据, 在服务节点都出现问题的情况下, 按照图 15 所示, 离散上报后的服务提供方节点可能会涵盖所有的服务提供方节点列表 (10.1.1.1–10.1.1.10), 本文针对这种情况, 进行按比例分批恢复节点, 直至业务集群恢复到正常状态.

3.2　基于 RPCContext 的全链路溯源

全链路溯源的核心目标是创建一个业务流量视图, 利用链路统计数据来监测并区分非法请求及识别非标准途径进来的桥接流量是否侵入核心业务环节. 本文提出了一种基于 RPCContext 的全链路溯源技术框架, 如图 16 所示, 旨在构建完整的业务流量视图. 该框架从调用链的最底层获取 RPCContext 数据, 进行数据处理和分析, 而在调用链的最上层, 对数据源进行标注, 包括用户信息、设备信息、网络信息等, 并将这些信息存储于微服务调用框架的 RPCContext 中.

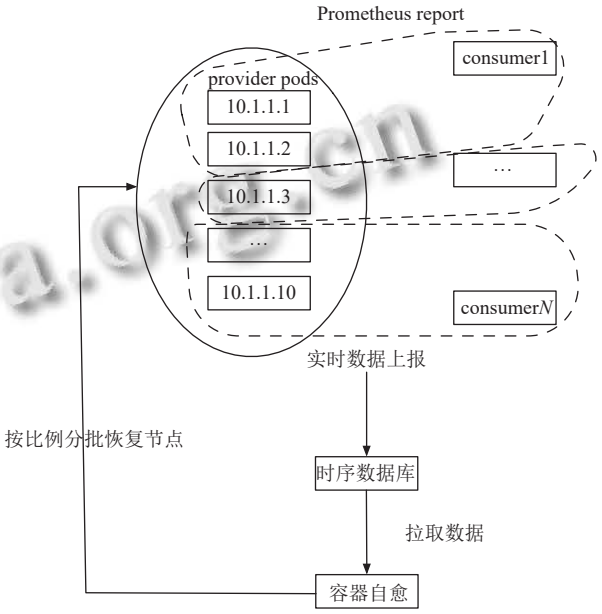

图 15　按比例分批恢复节点

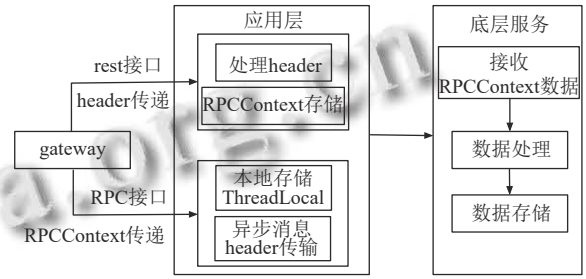

图 16　基于 RPCContext 的全链路溯源的技术架构

现有开源框架中 RPCContext 只能进行一次数据的传递, 这意味着从网关入口 gateway 往下游传递时, 下游服务调用的顺序为 gateway→A→B→C. 数据只能通过 gateway 传递到 A 服务, B 和 C 服务都不能拿到数据的状态. 如图 17 所示.

通过本文的设计优化, 数据能够从网关 (gateway) 顺畅传递至整个服务调用链的最末端服务 (C), 实现了数据在 gateway 到 A、B 乃至 C 服务的连续传递. 此外, 在数据传递过程中, 系统会从全局配置中心获取指定的数据键 (key), 确保所需的数据能够根据配置准确地向下游服务传递. 这样的改进不仅拓宽了 RPCContext

的应用范围, 也增强了服务间的互联互通能力, 如图 18 所示.

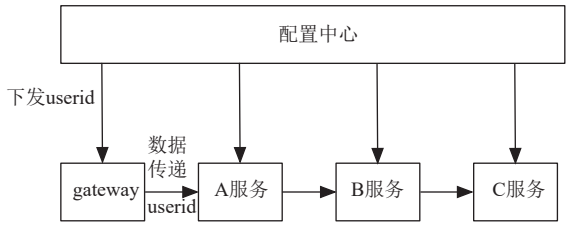

图 17 RPCContext 部分传递示意图

全链路溯源所需传递的数据采用 key-value 形式, 在构建溯源系统时, 面临一个挑战是多线程调用场景问题, 在多线程环境中, 如主线程 A 调用 RPC 服务 M1, 随后创建线程 B 调用 M2 和 M3, 目标是确保数据能顺利传递到 M2 和 M3. 但由于线之间线程隔离的问题, 无法完成跨线程的信息传递.

本文采用封装线程池技术的方案, 能够在多线程环境下实现数据的顺畅透传, 即从一个服务调用 (如 M1) 中提取必要的上下文信息 (如 S1), 并将这些信息在后续的多线程或异步调用 (如 M2、M3) 中保持和传递, 保证了数据的一致性和完整性. 这对于确保分布式系统中服务调用的连贯性、跟踪和安全性至关重要, 如图 19 所示.

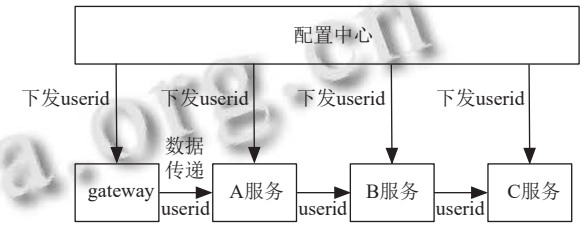

图 18 RPCContext 完整传递示意图

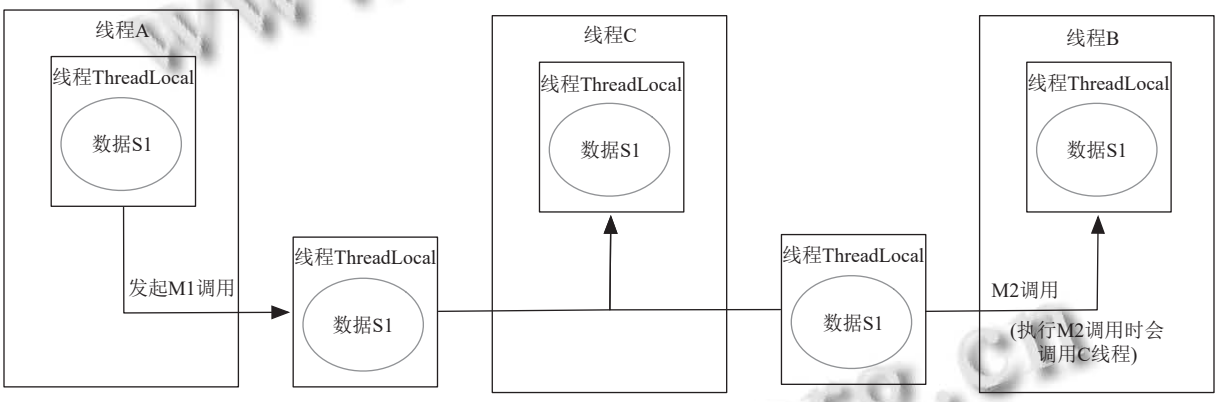

图 19 跨线程 Context 传递

为发现桥接流量, 主要策略是引入特征码 T1 标记上游流量. 下游服务接收并分析这一特征码, 通过将其与安全特征库中的数据进行对比来验证流量的合法性. 流程如算法 3 所示.

算法 3. 桥接流量识别过程

输入: 特征码 T1, 用户信息 userinfo, 设备信息 deviceinfo, 网络信息 networkinfo, 链路流程 linkSet, 特征禁用阈值 thresholds, 特征库 featureMap, 流量链路 trace.

```
IF T1 IS EMPTY return
userinfo = T1.subString(0, 8);
deviceinfo = T1.subString(8, 16);
networkinfo = T1.subString(16, 24);
link = trace.getLink()
IF featureMap.contains(userInfo, deviceinfo, networkinfo)
  IF linkSet.contains(link)
    IF featureMap.get(userInfo).thresholds > thresholds
      return true
    END IF
    return false
  END IF
  return false
END IF
return false
```

在桥接流量的识别过程中, 首先检查特征码 T1 是否存在. 若 T1 为空, 流量识别失败. 使用 T1, 通过字符串分割技术提取用户、设备、网络信息, 与 trace 流量链路信息进行比对. 这些信息在特征库中验证; 若不匹配, 或链路信息在正常链路集合中不存在, 流量识别视为失败. 此外, 根据用户信息在特征库中确定用户的禁用阈值, 与标准阈值比较. 超过标准阈值标记请求非法; 低于则视为合法请求.

4 应用效果

4.1 可靠性提升

应用链路分析技术构建完成之后, 在航旅纵横抽取了 5 个领域进行了全面分析共发现重复调用的链路共计 175 个, 发现循环调用 2 个, 发现并解决了系统的潜在隐患, 如表 4 所示.

表 4 无效调用分析结果

链路分析	重复调用	循环调用
首页	10	0
航班动态	38	0
值机	70	0
行程	50	1
权益	7	1

链路抽样之后, 每日链路存储数据下降 80%, 大大地节省了存储空间, 如表 5 所示.

表 5 链路抽样所节省的存储空间对比 (TiB)

抽样前/后	链路抽样所占存储空间
抽样前	10
抽样后	2

在应用了链路溯源之后, 链路各层的服务来源方清晰可见. 如表 6 所示, 最底层接口“行程信息查询”的访问源头可以得到明确的计量, 目前溯源的情况表明:

- 内部 job 和 flink 流式计算调用量占比较大, 需要进一步优化减少此类内部调用量, 以减轻系统压力, 提升系统稳定性.
- ToB 用户的请求量在外部流量中占比最大, 在运维和研发层面, 需要制定维护等级策略, 提高 ToB 请求链路的运维等级, 提高研发测试交付质量, 以优先保障 ToB 渠道的服务质量以保证 ToB 服务的 SLA.

表 6 最底层接口“行程信息查询”源头调用量统计

请求源头	调用量 (日)	占总调用量百分比 (%)
批量job/flink	3 976 519	48.2
ToB	3 536 576	42.9
安卓客户端	400 549	4.8
iOS客户端	324 105	3.9
微信小程序	3 763	0.04
支付宝小程序	156	0.001
总调量	8 241 688	99.84

通过以上分布式链路的溯源调用方式的统计, 能够帮助业务剖析出关键着力方向, 全方位地保证了系统的安全性.

4.2 性能提升

上述研究方法在航旅纵横已经投入到实际应用, 经过近一年的实践, 对上百个业务领域, 数千个对外 API 进行了优化改造, 整体实现 API 成功率高于 3 个 9 的 API 百分比由 99.957% 上升到 99.996%, 数百个关键 API 的服务质量得到有效提升. 其中首先改造的内容是根据分布式链路的循环调用识别、重复无效调用识别结果, 进行业务代码级别上的改进, 基本消灭了重复调用和循环调用, 解决了系统隐患, 提高了系统的稳定性, 如表 7 所示.

表 7 无效调用优化后结果

链路分析	重复调用	循环调用
首页	0	0
航班动态	0	0
值机	0	0
行程	1	0
权益	1	0

在上述改造完成之后, 进一步利用本文提供的强弱依赖演练方法, 对首页、航班动态等领域进行了 5 轮以上的强弱依赖演练, 业务根据演练结果进一步进行业务代码优化, 经过多轮优化之后, 如表 8 和表 9 所示, 针对航旅纵横 APP 的首页、航班动态两个核心业务领域优化前后进行了对比, 首页核心领域, 失败率下降为 0, 99 线响应时间降低 31%, 首页全链路系统稳定性提升 99.9%; 航班动态成功率提高到 99.99%, 99 线响应时间降低了 40%, 首页全链路系统稳定性提升 77.8%, 旅客出行体验得到了有效的保障, 效果十分显著.

表 8 服务链路优化前

业务	qps	avg (ms)	99 line (ms)	std	失败率 (%)
首页	303	108.9	595.3	97.8	0.002 8
航班动态	312	96.5	534.8	154.4	0.003 7

表 9 服务链路优化后

业务	qps	avg (ms)	99 line (ms)	std	失败率 (%)
首页	434.4	104.4	407	0.01	0
航班动态	346.8	83.5	316	34.2	0.001 3

5 结论

本文围绕微服务架构体系下复杂分布式系统的服务链路优化展开了研究, 提出了提高系统稳定性和性能的具体方法, 并在航旅纵横实际生产环境中进行了广泛的实践应用, 取得了良好的效果. 本文提出的服务链路优化方法和实践, 为研发运维人员在保障生产稳定性过程中提供了有效的指导.

参考文献

1 Nadareishvili I, Mitra R, McLarty M, *et al*. Microservice Architecture: Aligning Principles, Practices, and Culture. Sebastopol: O'Reilly Media, Inc., 2016.

2 Zimmermann O. Microservices tenets: Agile approach to service development and deployment. Computer Science-research and Development, 2017, 32(3): 301–310.

3 张齐勋, 吴一凡, 杨勇, 等. 微服务系统服务依赖发现技术综述. 软件学报, 2024, 35(1): 118–135. [doi: 10.13328/j.cnki.jos.006827]

4 Liu HF, Zhang JJ, Shan HS, *et al*. JCallGraph: Tracing microservices in very large scale container cloud platforms. Proceedings of the 12th International Conference on Cloud Computing. San Diego: Springer, 2019. 287–302.

5 Sigelman BH, Barroso LA, Burrows M, *et al*. Dapper, a large-scale distributed systems tracing infrastructure. https://www.researchgate.net/profile/Luiz-Barroso/publication/239595848_Dapper_a_Large-Scale_Distributed_Systems_Tracing_Infrastructure/links/5474acdc0cf29afed60f9031/Dapper-a-Large-Scale-Distributed-Systems-Tracing-Infrastructure.pdf. [2023-12-19].

6 Elasticsearch. https://github.com/elastic/elasticsearch. [2023-12-19].

7 Cassandra. https://github.com/apache/cassandra. [2023-12-19].

8 Jaeger. https://www.jaegertracing.io/. [2023-12-19].

9 OpenTracing. https://opentracing.io/. [2023-12-19].

10 SkyWalking. https://skywalking.apache.org/. [2023-12-19].

11 Zipkin. https://zipkin.io/. [2023-12-19].

12 Li BW, Peng X, Xiang QL, *et al*. Enjoy your observability: An industrial survey of microservice tracing and analysis. Empirical Software Engineering, 2022, 27(1): 25. [doi: 10.1007/s10664-021-10063-9]

13 Kubernetes. https://kubernetes.io/. [2023-12-19].

14 Mateus-Coelho N, Cruz-Cunha M, Ferreira LG. Security in microservices architectures. Procedia Computer Science, 2021, 181: 1225–1236. [doi: 10.1016/j.procs.2021.01.320]

15 严蔚敏, 吴伟民. 数据结构: C 语言版. 北京: 清华大学出版社, 1997.

16 Bang J, Gutin G, 著; 姚兵, 张忠辅, 译. 有向图的理论、算法及其应用. 北京: 科学出版社, 2009.

17 李义军, 任子真. 拓扑排序和强连通算法在源代码分析中的应用. 计算机系统应用, 2009, 18(1): 96–98. [doi: 10.3969/j.issn.1003-3254.2009.01.025]

18 Akuthota A. Chaos engineering for microservices [Master's thesis]. St. Cloud: St. Cloud State University, 2023.

19 Vinoski S. Advanced message queuing protocol. IEEE Internet Computing, 2006, 10(6): 87–89. [doi: 10.1109/MIC.2006.116]

20 殷康璘, 杜庆峰. 基于混沌工程的微服务韧性风险识别和分析. 软件学报, 2021, 32(5): 1229–1255. [doi: 10.13328/j.cnki.jos.006231]

21 Zhong ZH, Buyya R. A cost-efficient container orchestration strategy in Kubernetes-based cloud computing infrastructures with heterogeneous resources. ACM Transactions on Internet Technology, 2020, 20(2): 15.

22 Rhea S, Wang E, Wong E, *et al*. LittleTable: A time-series database and its uses. Proceedings of the 2017 ACM International Conference on Management of Data. Chicago: ACM, 2017. 125–138.

(校对责编: 孙君艳)