计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2020,29(4):84−91 [doi: 10.15888/j.cnki.csa.007362]

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

# 基于 ServiceComb 的多语言微服务平台①

赵　昱

(武汉邮电科学研究院, 武汉 430074)
通讯作者: 赵　昱, E-mail: zylong1995@163.com

**摘　要**: 现有的网络管理软件多用 C++开发, 服务之间相互依赖较为严重, 而运维管理软件多使用 Java 开发. 为实现网管服务和运维服务统一管理, 本文提出了一种基于 ServiceComb 微服务架构开发微服务底座, 统一管理多语言应用服务的解决方案. 系统使用服务底座的方式实现不同语言服务的统一接入管理, 通过实践证明该系统能够兼容两种不同语言的服务, 满足对不同语言服务的监控和管理.

**关键词**: 微服务平台; ServiceComb 架构; 底座

引用格式: 赵昱.基于 ServiceComb 的多语言微服务平台.计算机系统应用,2020,29(4):84–91. http://www.c-s-a.org.cn/1003-3254/7362.html

## Multi-Language Micro-Service Platform Based on ServiceComb

ZHAO Yu

(Wuhan Research Institute of Posts and Telecommunications, Wuhan 430074, China)

**Abstract**: The existing network management software is mostly developed in C++, and the interdependence among services is relatively serious, while the operation and maintenance management software mostly based on Java development. In order to realize unified management of network management services and operation and maintenance services, this study proposes a solution for developing micro-service bases based on ServiceComb micro-service architecture to manage multi-programming language application services. The system uses the service base to realize unified access management of different language services. It is proved by practice that the system can be compatible with services in two different programming languages to meet the monitoring and management of different programming language services.

**Key words**: micro-service platform; ServiceComb architecture; chassis

随着云技术和通信网络的快速发展, 通信网络管理系统一方面需要承载新开发的网络运维服务, 另一方面需要兼容原有的网络管理服务, 将原有服务和新开发的服务统一管理[1].

在微服务平台上实现已有网管功能的融合管理和新开发微服务架构功能的发布和服务治理, 如何将 C/S 架构的网管平台中的 C++服务发布到使用 B/S 架构开发的 Java 语言平台[2], 是现有通信运维管理平台最关键的工作之一. 目前对于通信网运维平台的研究主要是使用 Java 或其他语言进行单语言开发与维护[3], 对多种语言微服务统一管控的研究较少.

针对以上问题, 本文在前有研究的基础上, 提出了使用 ServiceComb 架构的微服务平台, 通过开发 Java 底座和 C++底座的模式, 为现有两种语言的共同治理提供共同的环境. 而对于传统网管服务较多、服务之间相互依赖的情况, 使用边车模式, 将互相依赖的服务编入同一边车内, 从而使已有网管软件的功能不受到影响. 平台面向通信提供商运维人员, 将已有的通信网管软件进行融合一体化, 在满足运营商要求的同时, 减少运维人员的工作难度.

① 收稿时间: 2019-09-18; 修改时间: 2019-10-15; 采用时间: 2019-10-22; csa 在线出版时间: 2020-04-05

## 1 ServiceComb 架构

### 1.1 微服务架构介绍

传统软件多使用单体架构实现, 单体架构也被称为单体系统或者是单体应用, 就是一种系统中所有的功能、模块耦合在一个应用中的架构方式. 用简单的方式理解就是将整个应用包括应用、数据库等都在同一个服务器上. 而分布式从简单的角度上理解就是将应用和数据等分开到不同的服务器上, 就然后对于应用和数据库进行不同方向上的性能优化等等操作. 单一架构相比于微服务架构, 具有测试成本高、可伸缩性差、可靠性差、迭代困难、跨语言程度差、团队协作难等缺点.

微服务是一种架构风格, 一个大型的复杂软件应用, 由一个或者多个微服务组成, 系统中的各个微服务可以被独立部署, 各个微服务之间是松耦合的, 每个微服务仅仅关注于完成一件任务并很好的完成该任务. 将一个复杂的软件系统进行拆分, 通过拆分之后, 这个复杂的应用系统变的更加的高效.

微服务架构优点可以概括为以下几点:

(1) 降低复杂度: 将原来偶合在一起的复杂业务拆分为单个服务, 规避了原本复杂度无止境的积累. 每一个微服务专注于单一功能, 并通过定义良好的接口清晰表述服务边界. 每个服务开发者只专注服务本身, 通过使用缓存、DAL 等各种技术手段来提升系统的性能, 而对于消费方来说完全透明.

(2) 可独立部署: 由于微服务具备独立的运行进程, 所以每个微服务可以独立部署. 当业务迭代时只需要发布相关服务的迭代即可, 降低了测试的工作量同时也降低了服务发布的风险.

(3) 容错: 在微服务架构下, 当某一组件发生故障时, 故障会被隔离在单个服务中. 通过限流、熔断等方式降低错误导致的危害, 保障核心业务正常运行.

(4) 扩展: 单块架构应用也可以实现横向扩展, 就是将整个应用完整的复制到不同的节点. 当应用的不同组件在扩展需求上存在差异时, 微服务架构便体现出其灵活性, 因为每个服务可以根据实际需求独立进行扩展.

### 1.2 ServiceComb 架构

目前常见的微服务架构有 SpringCloud 和 Dubbo, Spring Cloud 是一系列框架的有序集合. 它利用 Spring Boot 的开发便利性巧妙地简化了分布式系统基础设施的开发, 如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等, 都可以用 Spring Boot 的开发风格做到一键启动和部署. Dubbo 是阿里巴巴公司开源的服务治理框架, 使用人数众多, 提供了优秀的分布式解决方案. 但是 SpringCloud 和 Dubbo 在 Java 之外的其他编程语言的支持较差, 为了解决这一问题, 本文采用 ServiceComb 架构进行开发.

ServiceComb 是华为云于 2017 年 6 月开源的微服务框架, 并于 2017 年 12 月正式进入 Apache 软件基金会孵化. 其包括一站式的服务注册、服务治理、动态配置功能, 具备服务化契约增强、多语言 SDK 支持、多通信协议支持等优势特性, 并提供 SAGA 数据最终一致性方案解决微服务架构数据一致性难题. ServiceComb 兼容 Spring Cloud 等业界流行微服务框架, 互通业界生态.

ServiceComb 相比与其他微服务架构的主要优点是解决了存量应用和新开发应用共存的问题, 对于新开发的 Java 应用, 使用 Java-chassis (Java 底座) 完成服务治理功能, 而对于已经存在的, 使用其他语言开发的应用, 使用底座的方式实现业务代码的 0 侵入改造, 从而完成混合部署, 统一治理. ServiceComb 特性如图 1 所示.

## 2 系统分析与架构设计

### 2.1 系统需求分析

2018 年 5 月中国移动集团发布了最新的《OMC 系统通用技术规范》, 将管控一体、微服务和云架构的技术要求纳入规范, 规范的核心关键是 OMC 的微服务架构.

随着编程语言的发展, 网络管理和运维软件的开发环境越来越多样, 需要建立一个可以统一管理多种编程语言服务的微服务平台. 多语言微服务平台应该满足的需求:

(1) 统一服务注册中心, 对平台上的微服务提供统一的服务注册与发现、以及服务治理能力的支撑.

(2) 统一服务配置中心, 为微服务提供集中式的服务配置存储、以及客户端服务配置的自动拉取, 实现服务配置的动态更新支持.

(3) 集中式的服务运维监管中心, 实现服务健康检查与运行监控、服务配置动态变更与管理, 对服务进

行可视化的监管.

(4) 多语言微服务底座, 实现微服务的统一注册和发现、配置动态更新、健康检查、服务 RPC 通信. 微服务的统一治理问题, 如: 负载、熔断、容错、限流等.

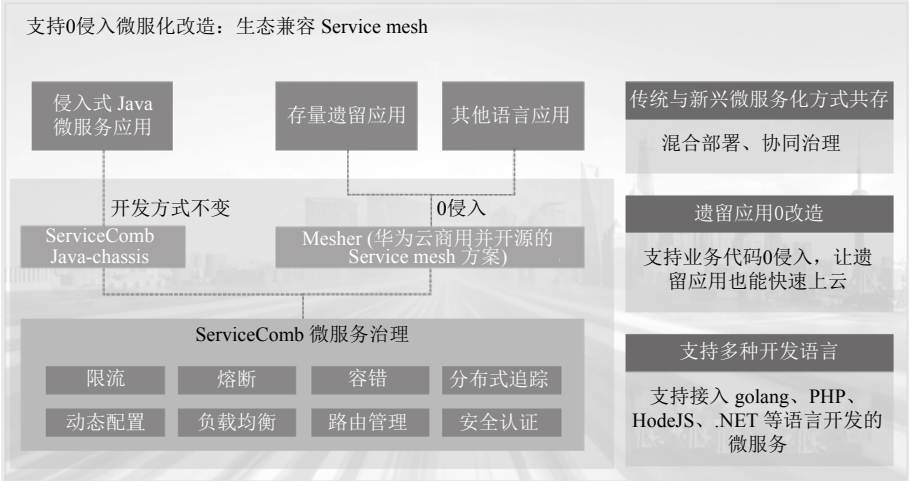


图 1 ServiceComb 特性图

### 2.2 系统模块拆分

根据系统需求分析, 将平台设计拆分为 3 个功能模块来完成开发. 分别是服务注册和配置中心、运维监管中心、微服务底座. 3 个模块功能如下:

服务注册和配置中心负责提供微服务的注册和发现功能支撑, 支持服务配置的动态配置.

运维监管中心主要是对平台和平台上的微服务进行监控和管理, 对系统运维功能支持.

多语言微服务底座是核心模块, 提供对多种编程语言微服务的支持, 实现服务治理、服务间通信、服务的注册与发现等功能. 目前提供了对 Java 和 C++的支持.

### 2.3 整体架构设计

为了使多语言微服务平台的每个服务之间的分工和开发界限清晰, 提高开发的效率, 将平台分为应用层、接口层、服务层、数据层和物理层, 整体架构设计图如图 2 所示.

各层包含内容如下:

(1) 应用层: 包括对服务的监控功能、配置的发布功能、服务的发布功能.

(2) 接口层: 接口层主要是 API Gateway 实现的网关服务, 基于 JWT 实现的权限管理功能和基于 RocketMQ 实现的消息中间件服务.

(3) 服务层: 主要是平台提供的核心服务, 如服务注册和配置中心、运维监管中心、Java 微服务底座, C++微服务底座和边车. 还包括平台微服务架构的基础服务, 由 ServiceComb 对 SpringCloud 有着良好的支持性, 使用 Consul 作为服务注册和配置中心, 自动化配置服务 SpringCloud Config. 在微服务底座中, 基于 Hystrix 实现的容错处理器, 实现了服务的容错处理和服务质量信息上报. 基于 Ribbon 实现的负载均衡处理器, 有效提高服务的承载能力. 通过 RPC 调用链模式, 将负载均衡、熔断等服务治理功能通过实现单独的处理器来实现, 自定义选择处理器, 形成链式调用, 从而完成服务治理的功能, 也为后期拓展提供了简单的模式. 平台的基础服务使用 Consul 作为服务注册和配置中心, 简化了设计. 使用 SpringCloud Config 处理服务的配置动态更新.

(1) 数据层: 数据层主要包括服务实例的数据、配置信息数据、配置缓存.

(2) 物理层: 包括服务器、网络设备、安全组件等, 支撑系统的物理组件.

## 3 微服务平台的设计

### 3.1 注册与配置中心

为构建高效注册和配置中心, 微服务平台采用 HashiCorp 公司 Consul 注册配置中心组件[4], Consul 作为微服务组件能同时提供注册中心和配置中心的功能.

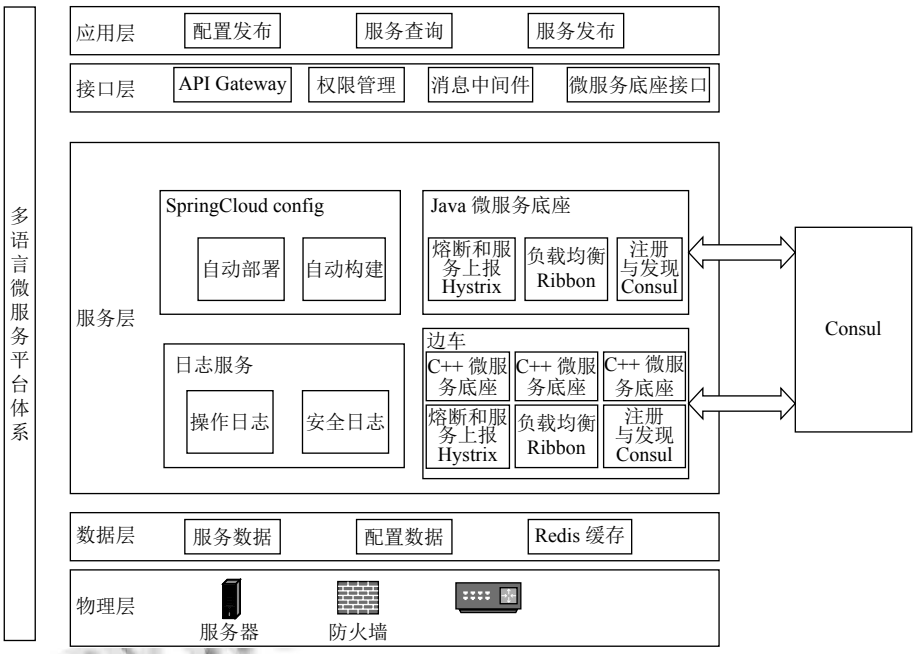


图 2 系统架构图

注册中心服务端实现:

启动使用 Consul 的应用程序, 在 hashicorp 官网下载 consul.exe, 配置 consul 到环境变量启动即可完成服务端.

注册中心客户端实现:

ecwid 是 Consul 客户端的一种, 在 Pom 依赖中添加 ecwid 的 consul-api. 通过 ConsulClient 的 newService 对象的构造方法传入注册 IP 和端口. 然后设置 newService 对象的服务名、服务 ID、服务标签和服务端口信息, 最后调用 ConsulClient.agent-ServiceRegister方法完成服务注册.

在分布式应用场景中, 微服务实例对于配置中心应该具有弱依赖和稀疏变更特点[5], 弱依赖性表示微服务与配置中心依赖性应该降至最低, 在配置中心发生故障情况下, 提高系统容灾性. 稀疏变更表示动态配置在部署至微服务实例后, 很少发生改变. 而一旦发生变更, 则需要配置客户端从配置中心快速地同步和部署配置变更.

针对上述特点, 配置客户端引入本地快照机制, 将配置保存值本地文件中, 提高微服务容灾性. 引入本地缓存机制, 将本地微服务最新配置保存至缓存中, 提高微服务获取实时配置的性能.

配置中心客户端变更流程: 用户在监控中心发布配置变更后, 变更被同步至配置中心, 客户端与的配置中心之间以长轮询方式维持着心跳, 在获取到变更通知后, 重新从配置中心拉取配置, 更新本地缓存和快照文件.

Consul 配置中心以键值 K/V 方式存储配置信息, key 值以文件夹路径表示. 为实现配置管理与共享, 配置数据按照微服务隔离层次 (环境---服务组---服务---服务实例) 来存储, 形成树型多层级文件夹结构, 结构中子节点能够访问父节点中的配置, 同层级节点中配置相互隔离, 如图 3 所示.

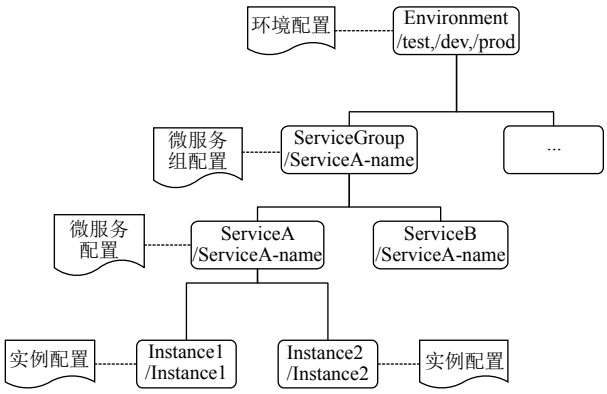


图 3 配置隔离示意图

根节点“环境”(Environment) 下包含了若干个服务组, 它的配置对所有微服务共享;

节点“服务组”(ServiceGroup) 对环境中服务按照功能特性进行了分组, 配置对组内所有微服务及其实例共享, 不同微服务组配置相互隔离;

节点“服务”(Service) 中的配置对该服务所有实例共享, 不同微服务间配置相互隔离;

节点“服务实例”(Instance) 中存储某个微服务实例配置, 不同实例间配置相互隔离.

### 3.2 运维监控中心

运维监控中心与 Java 底座中的 Hystrix 组件进行通信, 获取服务的服务质量信息并在前端以图表的形式显示.

运维监管中心在启动后就按 5 s 的周期对全部的服务实例进行性能监控, 与服务实例进行通信获取服务质量数据, 如果是 java 微服务, 则建立和该服务实例的 hystrix.stream 流的连接, 读取服务质量数据, 如果是网管微服务, 则建立和该服务实例所在边车的 hystrix.stream 流的连接, 读取服务质量数据. 读取到的服务质量数据包含了该服务实例上应用了 hystrix 的所有方法的服务质量数据, 对每一个服务提供者提供的每一个方法均记录该方法的请求流量 (qps)、请求的失败/异常数[6]、请求的错误比例、请求的平均响应时间、服务熔断状态等服务质量参数[7].

获取到服务质量数据后, 对数据进行处理, 处理的内容包括: 整理数据、判断告警状态、生成或恢复服务质量告警, 处理完成后将服务质量数据保存到缓存中, 在每个获取服务质量性能数据的周期里都对此缓存进行更新, 此外, 每隔 5 分钟将一次获取到的服务质量数据保存到数据库. 当查看指定服务实例的服务质量数据时, 后台根据用户请求的参数从缓存中查找到对应的服务质量数据并返回给前端.

### 3.3 微服务底座

#### 3.3.1 Java 微服务底座

Java 微服务底座的提供的负载均衡、熔断、容错、限流等功能使用目前已有的 SpringCloud 组件即可完成.

服务的注册和发现通过 Consul 完成, 考虑到服务间的通信是通过底座完成, 所以需要对注册发现进行监听.

(1) 服务注册设计

服务需要通过根据配置文件, 自动将服务注册到 Consul. 在获取服务的时候, 需要及时获取到最新的服务, 在获取服务的时候需要使用缓存, 避免频繁的和 Consul 进行调用. 服务注册和健康检查流程如下:

1) Spring 容器启动的时候, 读取配置的 spring.factories, 对于配置的 bean 进行初始化, 将 Discovery AutoConfiguration 里的配置 bean 进行初始化.

2) 在 Spring 容器初始化完成后, 会发送 Context RefreshedEvent, 监听这个 Event, 可以启动服务端, 如 IceServer 或者 RestServer.

3) 启动服务以后发送事件, 监听到服务启动成功以后, 将服务注册到 Consul.

注册的时候, 默认注册地址为: "http://"+服务 IP+"/actuator/health", 通过 Actuator 组件, consul 直接向该服务发起心跳, 可以得到服务的健康状态. 监控通信通过读取数据, 可以判断服务是否存活.

(2) 服务发现设计

1) 客户端调用通过实现负载均衡功能的处理器上获取服务列表, 当处理器调用 Consul 获取服务列表的时候, 首先去访问缓存, 获取当前的缓存状态, 若缓存状态为 success, 则返回服务列表, 若缓存状态为 False, 则去 Consul 上获取最新的服务列表, 并更新缓存.

2) 初始化的时候启动 Consul 监听任务去 Consul 上监听是否有服务的变更.

3) 当 Consul 的服务发生变更以后, 向缓存发送缓存失效信息, DiscoveryCache(发现缓存) 此时设置缓存可读状态为 False.

4) 当下次请求访问到, DiscoveryCache 的时候, 获取服务列表时, 首先去获取缓存可读的状态, 当为 False 的时候, 直接向 Consul 获取最新的服务列表.

(3) 服务 RPC 调用链设计

微服务底座的服务治理使用调用链方式进行, 调用链是类似 ServiceComb 的 Chassis 组件中的 Handler. 将负载均衡、熔断等服务治理功能通过实现单独的处理器来实现, 自定义选择处理器, 形成链式调用, 从而完成服务治理.

调用链采用分层设计, 分别为 API 接入层、治理层、协议层. Spring 应用或者是网管客户端通过调用接入层的 API 接口引入服务治理功能, 服务治理的整体结构采用链式服务, 方便以后的服务治理功能的拓展, 协议层封装了客户端发送请求和接收请求的操作.

Spring-support 提供 Spring 注解, 注解名称 @Reference, 便于上层透明调用远程端口, 注解属性

serviceName(服务名称), identity(访问), protocol(协议类型). Spring 容器扫描到@Reference 注解后针对接口生成代理, 并交由 spring 容器进行管理. 代理的核心实现在拦截器 DefaultMethod-Interceptor 中, 将所有参数封装为 Invocation, 传给服务治理模块.

治理层中业务的 Handler 采取单例模式, 保证每一个服务在调用链中的治理是独立的. LoadBalancer 从注册中心 consul 上根据路由和负载均衡策略找出一个地址发起调用. HystrixHandler 对服务调用进行相关统计, 然后根据统计数据判断是否熔断. 熔断指定时间后会尝试恢复. TransportClient-Handler 负责根据协议找到具体的 client 然后发起调用, 如 BusClient.

协议层中 TransportHandler 根据协议名称选取 protocol 实现, 在 protocol 中通过相应 Client 发起远程调用并接收响应.

自定义容器管理 ExtensionLoader, 通过 JavaSpi 机制实现一个自定义容器. 同步调用方式中 DefaultMethod Interceptor 将相关参数封装为请求, 经过 Handler 链处理, 使用对应 Protocol 将请求发送出去, 将当前线程阻塞在控制器, 服务端有响应后通知解除阻塞.

#### 3.3.2 C++微服务底座

C++微服务底座 (CppChassis) 负责承载 C++微服务, 并负责与 SideCar 之间的微服务发现的通信、服务治理的通信、配置获取的通信与配置变化消息的处理和心跳的发送.

CppChassis RPC 调用流程图如图 4 所示.

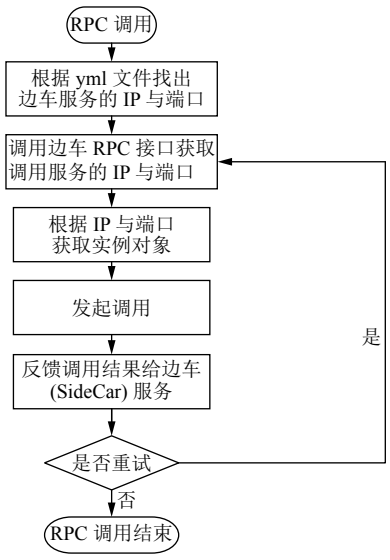


图 4 RPC 调用流程图

C++网管微服务调用其他 C++网管服务时, 在 CppChassis 里面会将调用过程分为 3 个步骤:

(1) Before Invoke: 该调用是通过 CppChassis 从 SideCar 获取 C++Provider 服务的 IP 端口等定位信息;

(2) Ice Invoke: 该调用通过指定的 IP 端口直接发起向其他 C++Provider 服务的 Ice 调用;

(3) After Invoke: 该调用是将调用结果通过 Cpp-Chassis 反馈给 SideCar, 用于服务治理相关信息的收集.

#### 3.3.3 边车

对于 C++服务, 平台使用边车模式支持服务治理功能, Sidecar 模式是一种将应用功能从应用本身剥离出来作为单独进程的方式. 该模式允许我们向应用无侵入添加多种功能, 避免了为满足第三方组件需求而向应用添加额外的配置代码.

边车负责进行控制, 提供负载均衡, 容错控制, 熔断控制, 健康检查等功能, 服务间实际的远程调用由服务自己完成.

(1) 边车的注册设计

项目启动的时候通过 SPI 机制启动扫描类, 扫描环境变量 C++服务对应的路径下的 yml, 过滤出所有 yml 格式的文件分别批量注册. 注册完毕后运维监控中心发送注册通知, 等待边车返回健康检查数据.

(2) 边车的健康检查设计

边车的健康检查通过调用 C++微服务底座健康检查接口实现, consul 的心跳检查会通过边车调用 C++接口完成数据采集.

(3) 边车的 RPC 调用设计

边车提供了两个方法: BeforeInvoke, AfterInvoke; C++服务在发起远程调用前, 先从 BeforeInvoke 方法获取服务实例, 调用完成后再用 AfterInvoke 方法通知边车调用结束, 以及调用的结果.

BeforeInvoke 方法中, 边车会创建一个 Handler 链, 与 Java 底座实现类似.

LoadBalanceHandler 负责负载均衡功能和容错功能, HystrixHandler 负载熔断控制和服务质量统计, 从而与 Java 底座使用同样方式实现服务治理.

AfterInvoke 方法会把远程调用结果通知给对应的处理器 Handler, 从而完成服务调用监控.

## 4 系统测试

### 4.1 系统功能测试

由于平台设计中, 主要功能是对服务的监控和发

布, 通过查看对服务的监控, 可以测试平台的监控能力和底座的承载能力.

当平台检查到服务时, 会在页面上展示服务监控数据, 服务监控情况如图 5 所示. 需要了解异常服务详情时, 使用监控可以了解具体的服务 IP 和端口以及健康检查端口返回的元数据, 异常服务详情如图 6 所示.


图 5 服务监控图


图 6 异常服务详情

## 4.2 性能测试

系统性能测试主要分为两个部分, 第一部分为采用微服务架构的服务较原有架构服务的性能提升, 另一部分为原有服务采用微服务底座承载之后性能劣化是否在 20% 以内.

本文采用 Apache Bench 模拟 100 个用户对微服务发出访问请求, 分析内存占用情况如图 7 所示, 其中上方虚线表示传统单架构服务的内存占用, 下方实线表示采用本文微服务底座的服务内存占用. 横轴表示请求发出后服务运行时间 (s), 纵轴表示内存占用大小 (MB).

对于原有 C++语言开发的服务, 由于 RPC 操作通过边车进行, 每次调用服务在原有调用逻辑上增加了 BeforeInvoke 和 AfterInvoke 操作, 可能会造成性能劣化, 通过模拟调用同一接口多次, 得到调用时间如表 1 所示.

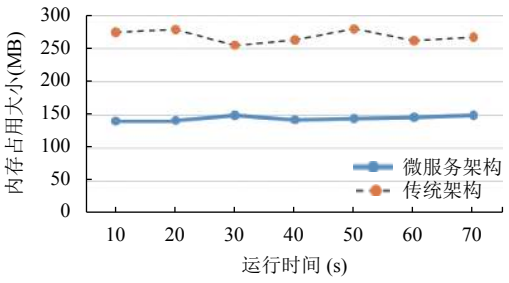


图 7 内存占用对比图

表 1 多次调用 C++接口响应时间对比分析

| RPC 调用次数 | 原始调用耗时 (s) | 微服务调用耗时 (s) |
|---|---|---|
| 1000 | 1.22 | 1.21 |
| 2000 | 2.07 | 2.21 |
| 3000 | 2.98 | 3.35 |
| 5000 | 4.97 | 5.52 |
| 10 000 | 9.77 | 10.95 |
| 20 000 | 19.79 | 22.25 |

可以看出, 在高并发情况下, 采用本文微服务架构的服务内存占用较传统架构服务有着明显优势, 并且内存占用变化更为平稳.

从表中可以看出, 原有 C++服务通过引入底座模块后, RPC 调用耗时有所增加, 但劣化范围在 12% 左右, 在期望范围之内.

通过将不同语言语言开发的服务置于不同的底座之中, 统一交由平台监控管理, 对于微服务的并发环境下内存占用, 由于使用基于 SpringCloud Robbin 的负载均衡组件, 使得服务更加平稳且内存占用更小, 对于 C++服务, 在引入 C++底座和边车后被改造为微服务, 使其具有了服务治理的功能, 在 RPC 调用时性能略有下降但无明显劣化.

## 5 结束语

通信业务的快速发展, 网络管理软件和运维系统也在不断更新, 但统一管理的难度却越来越大, 运维人员需要统一的管理控制平台应对愈发复杂的网络情况. 多语言微服务平台作为承载 EMS 系统和 SDN 控制器的拓展融合平台, 实现了微服务的注册与发现、质量监控、服务定制化发布、多种语言微服务共同治理的一体化集成. 该平台能帮助新开发的 Java 微服务发布到管控平台进行发布与管理, 也能将传统的 C++服务通过底座发布到平台中并实现统一管控. 为网络方案供应商解决满足中国移动 OMC 技术规范中的要求提供了新的思路和示范性的系统, 为传统网络管理软件实现服务治理和应用监控起到了关键性作用, 同时也保证了已有功能再利用, 减少了资源浪费. 但是对于其他编程语言的融入没有进行相应测试, 下一步将以承载 Golang、Scala 微服务作为研究目标.

## 参考文献

1 闫培平. 智慧城市路灯远程管控系统关键技术的研究及实现[硕士学位论文]. 西安: 西安理工大学, 2019.
2 雷东卿. 消防应急通信中计算机及云平台技术特征及应用研究. 信息通信, 2019, (2): 181–182.
3 高扬. 浅谈通信运营商云平台网络子系统的设计. 中国新通信, 2018, 20(14): 28–29. [doi: 10.3969/.issn.1673-4866.2018.14.025]
4 李晓明, 应毅, 曾岳. 基于 Java 的微服务技术在构建企业智能大数据平台下的应用与开发研究. 现代电子技术, 2019, 42(15): 165–169.
5 戴安康. 浅析基于云平台的新能源汽车电控系统通信设计. 电子测试, 2018, (20): 78–79. [doi: 10.3969/j.issn.1000-8519.2018.20.033]
6 章慧滔. 基于云平台的视频会议系统的架构及 MCU 调度算法的研究[硕士学位论文]. 南昌: 南昌大学, 2018.
7 谭一鸣. 基于微服务架构的平台化服务框架的设计与实现[硕士学位论文]. 北京: 北京交通大学, 2017.