ServiceComb在华为消费者云的亿级用户微服务实践

AGENDA

微服务化总体策略

微服务化实践

微服务化收益

website: http://servicecomb.incubator.apache.org/
gitter: https://gitter.im/ServiceCombUsers/Lobby

关于我

- 经历：2008年加入华为，从事多年平台中间件、云计算和PaaS平台设计和开发
- 目前在华为终端消费者云业务从事华为手机应用市场的云化、微服务化架构设计
- 《Netty权威指南》、《分布式服务框架原理与实践》作者
- 热爱技术写作和分享

联系方式：

Email：li_linfeng@huawei.com

微信号：右侧扫码

Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby

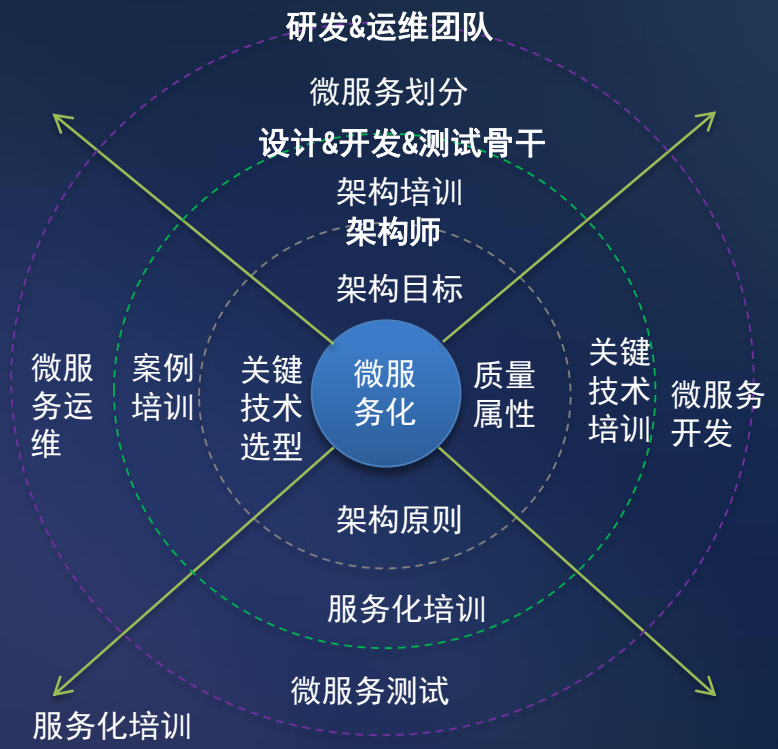
华为消费者云服务简介

Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby

总体策略-顶层统一设计

大兵团作战，统一认识，组织赋能

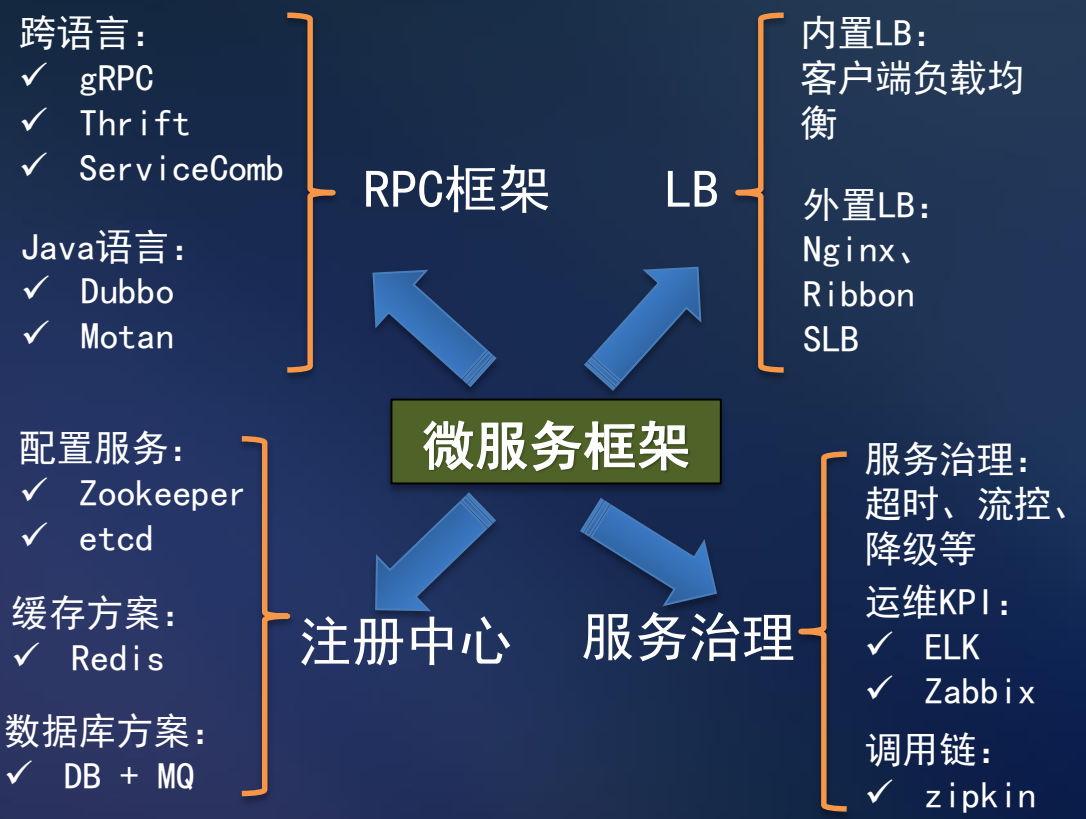


- **规范制定：** 微服务命名规范、微服务API定义规范、微服务验收规范、微服务运维和治理规范…
- **技术选型：** 统一的微服务框架技术选型，涵盖微服务框架、微服务流水线（CI/CD）、微服务自动化运维和部署框架
- **组织赋能：** 拉通设计、开发、测试和运维的跨团队培训、技术方案试点、案例分享、 ServiceComb统一预警体系，不同业务部门之间的合作和协同
- **服务化地图：** 业务部署上线后，对业务微服务API进行基线化，并发布服务化地图，涵盖微服务的划分原则、微服务流水线地址、微服务SLA指标等

Website: http://servicecomb.incubator.apache.org/
Gitter: https://gitter.im/ServiceCombUsers/Lobby

总体策略-技术选型

没有十全十美的微服务框架，适合的才是最好的



选择ServiceComb的原因：

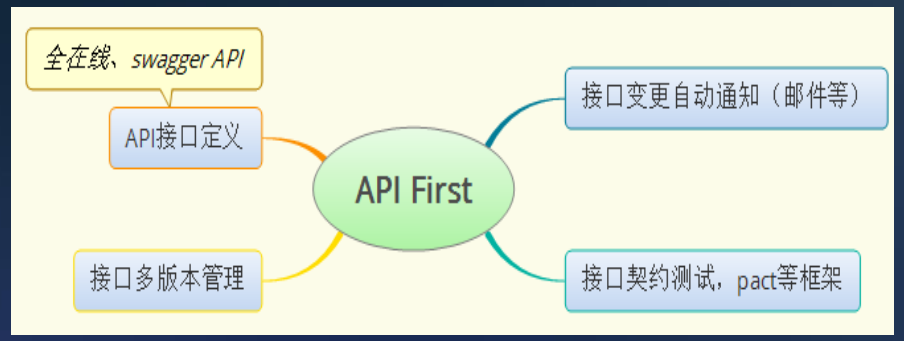
- 微服务API设计：支持Swagger API规范
- 开发方式：支持透明RPC、Spring MVC风格的微服务开发
- 编程模型：同时支持同步、异步的编程模型
- 性能：支持原生的Reactive模式（EventLoop），相比于传统的同步服务调用，性能更高（吞吐量N倍+提升、时延降低为原来的50%-）
- 轻量级、易集成：可以方便的与Spring MVC、Tomcat等已有技术和容器集成，支持轻量级的standalone部署
- 服务治理：可商用的服务治理能力
- 成熟和商用支持：Apache孵化项目，公司内部大规模使用，专职的团队支撑

Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby

实践- API First

遵循OpenAPI Specification，消费端、提供端只依赖API



- API描述：无论Rest API，或者RPC Highway API，统一使用Swagger YAML定义API
- 微服务代码生成：服务端和客户端都基于API定义，通过ServiceComb提供的工具生成不同语言的类库，客户端可以不导入服务端的类库定义，双方互相解耦
- 微服务接口测试：测试基于流水线的API定义生成自动化测试用例，防止开发本地随意修改API
- 微服务API依赖关系：通过微服务流水线编译、打包和系统集成测试，识别并展示微服务API之间的依赖关系（我依赖谁、谁依赖我）
- API变更管控：API变更邮件通知和审批，每日微服务流水线构建，及时发现接口不兼容问题

Website: http://servicecomb.incubator.apache.org/
Gitter: https://gitter.im/ServiceCombUsers/Lobby

实践-不同的微服务开发风格

不同团队，经验和开发习惯都有差异，选择适合业务的开发模式

1、传统使用Spring MVC开发的团队，继续使用Spring MVC风格：

```
@RestSchema(schemaId = "springmvcHello")
@RequestMapping(path = "/springmvchello", produces = MediaType.
        APPLICATION_JSON)
public class SpringmvcHelloImpl implements Hello {
  @Override
  @RequestMapping(path = "/sayhi", method = RequestMethod.POST)
  public String sayHi(@RequestParam(name = "name") String name) {
    return "Hello " + name;
  }
```

2、使用DSF等服务框架开发的，更熟悉透明RPC开发模式：

```
@RpcReference(microserviceName = "hello", schemaId = "hello")
private static Hello hello;

@RpcReference(microserviceName = "hello", schemaId = "codeFirstCompute")
public static Compute compute;

public static void main(String[] args)
    throws Exception {
  init();
  System.out.println(hello.sayHi( name: "Java Chassis"));
```

3、还可以使用JAX-RS模式开发微服务：

```
@RestSchema(schemaId = "jaxrsHello")
@Path("/jaxrshello")
@Produces(MediaType.APPLICATION_JSON)
public class JaxrsHelloImpl implements Hello {

  @Path("/sayhi")
  @POST
  @Override
  public String sayHi(String name) { return "Hello " + name; }
```

开发模式没有优劣之分，对于重构方式进行的微服务开发，选择业务熟悉的模式，可以更好的重用已有代码和开发经验，提升重构效率

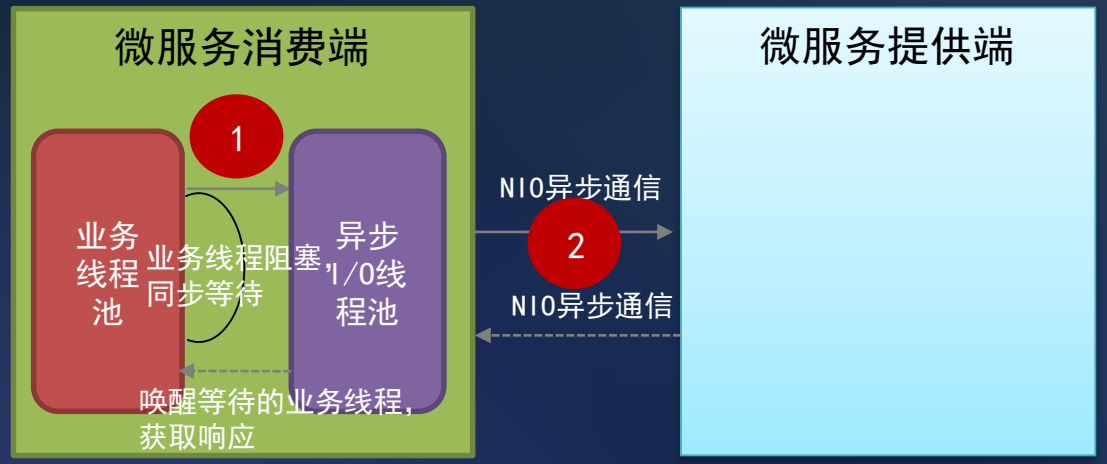
Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby

实践-同步和异步

不一刀切，根据业务实际需要选择同步或者异步微服务调用

微服务同步调用问题：



1、同步服务调用：业务线程将请求消息交给I/O线程之后，无论I/O线程是同步还是异步发送请求消息，业务线程都会同步阻塞，等待响应

2、异步I/O通信：I/O通信方式与服务调用方式没关联关系，无论是同步服务调用还是异步服务调用，I/O通信都可以采用异步非阻塞模式

同步服务调用的几个缺点：

1、业务线程利用率低：线程资源是系统中比较重要的资源，在一个进程中线程总数是有限制的。提升线程的使用率，就能够有效提升系统的吞吐量。在同步服务调用中，如果服务端没有返回响应，客户端业务线程就会一直阻塞（wait），傻等期间，无法处理其它业务消息。

2、纠结的超时时间：服务的超时时间配置是个比较纠结的事情，如果超时时间配置过大，如果响应慢，会导致线程被长时间挂住；如果配置过小，则会导致超时增多，成功率降低。

3、雪崩效应：假如超时时间配置较大（例如3S），服务端响应的平均时延达到了超时时间阈值，会导致业务线程长时间处于wait状态，工作效率降低，业务堆积，发生级联的雪崩效应。

Website: http://servicecomb.incubator.apache.org/
Gitter: https://gitter.im/ServiceCombUsers/Lobby

实践-全栈异步

适合异步的业务采用全栈异步架构，提升性能和可靠性

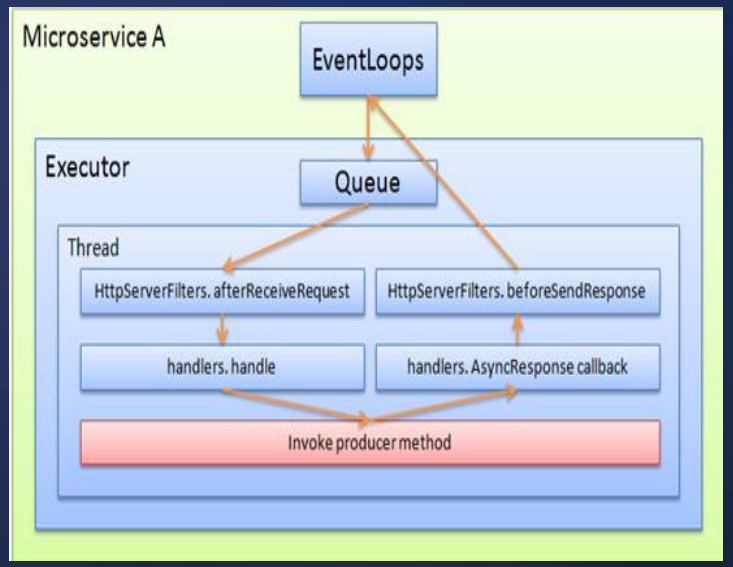
异步场景1：降低长流程/复杂业务流程时延：消费端需要调用多个微服务，进行业务逻辑编排，多个微服务之间没有执行先后顺序和参数依赖，可以通过异步微服务调用并行执行

异步场景2：性能提升：使用更少的线程处理更多的消息，提升线程和I/O利用效率

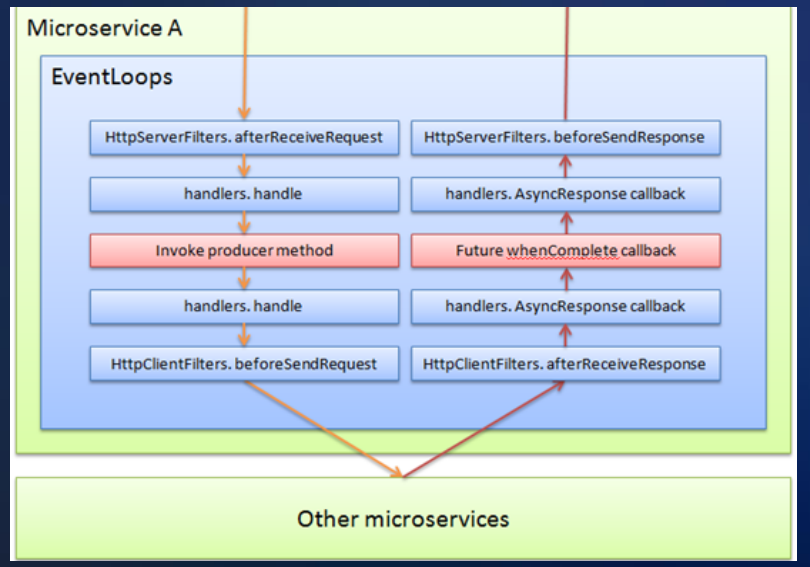
异步场景3：业务超时较长：业务上对服务调用时延不敏感（例如1-3S），如果采用同步调用 + 大超时时间，在业务高峰期，如果时延达到超时阈值，系统很容易被压挂

异步场景4：级联调用：需要级联调用多个微服务，希望提升可靠性，不会因为某个微服务处理慢而导致其它微服务调用被阻塞

传统I/O和业务线程分离技术：



纯Reactive异步：



性能对比测试：采用Reactive异步模式之后，TPS提升 43% 左右、时延降低 28% 左右，CPU占用降低 56% 左右

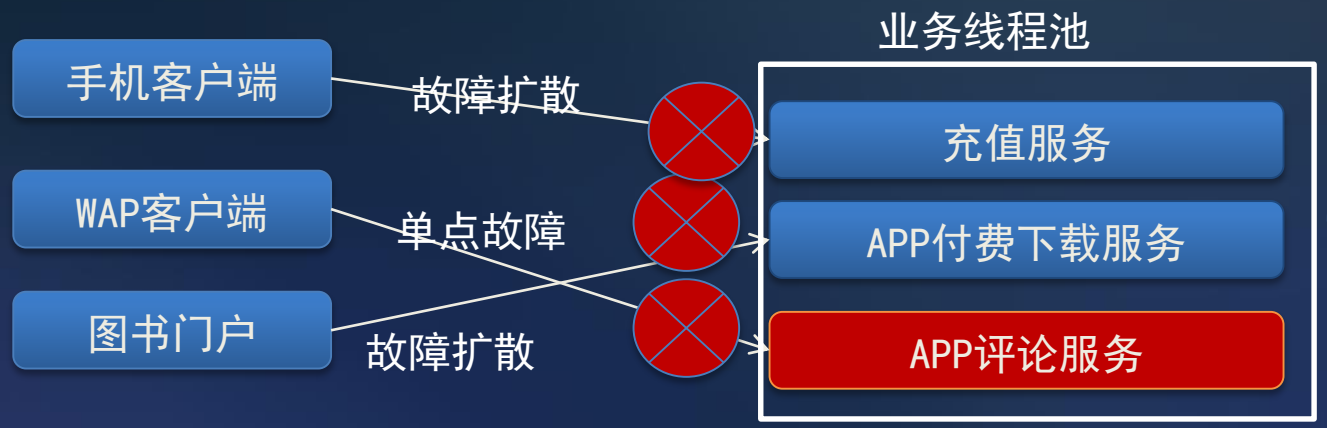
Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby

实践-故障隔离

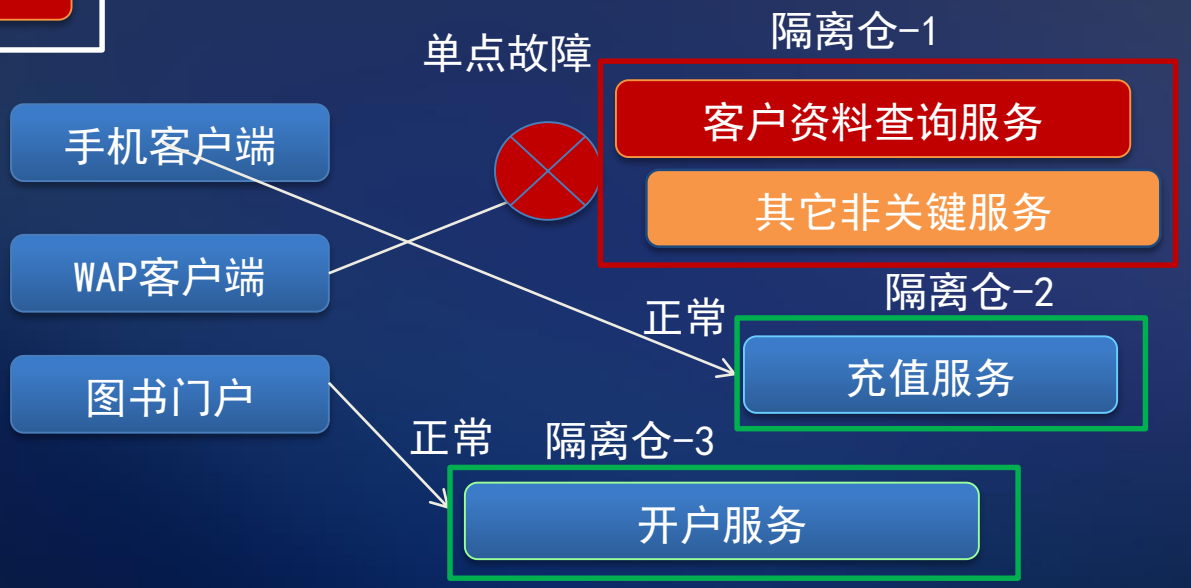
利用ServiceComb的隔离仓技术，实现业务不同接口的故障隔离

无线程池隔离仓：



配置隔离仓：

```
servicecomb:
  executors:
    Provider:
      [schemaId].[operation]
```



通过配置线程池隔离仓，实现快慢接口、读写接口、核心和非核心接口、管理和业务接口的调度隔离，提升微服务可靠性

Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby

实践-轻量化

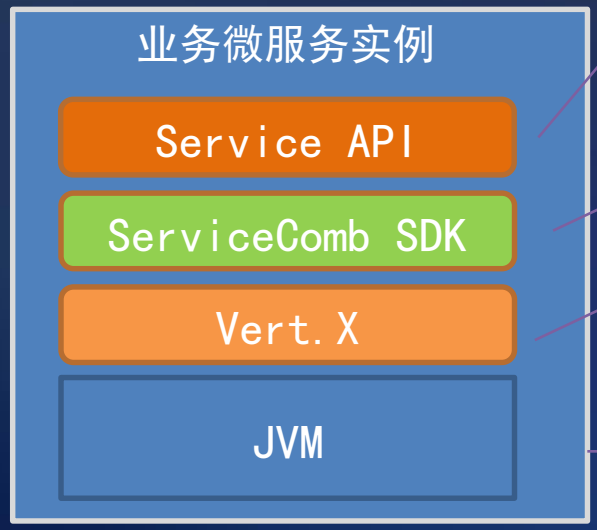
业务按需配置ServiceComb的类库依赖，基于standalone轻量级部署

开发态：按照业务实际需要配置ServiceComb的maven 依赖，只加载需要的类库：

如果需要使用ServiceComb的流控再配置

```
<dependency>
    <groupId>org.apache.servicecomb</groupId>
    <artifactId>handler-flowcontrol-qps</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.servicecomb</groupId>
    <artifactId>transport-highway</artifactId>
</dependency>
```

如果需要使用私有RPC协议时才需要配置依赖



业务微服务独立部署，一个微服务可对应多个Java Interface

对于纯微服务消费端，不需要额外监听端口

Standalone模式，后台应用不依赖Web容器

更轻量，启停速度更快，云端弹性伸缩效率更高

Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby

实践-微服务治理

通过ServiceComb的服务治理，保障业务的线上运行质量

- 服务目录
- 服务配置管理
- 服务灰度发布
- 微服务性能监控大盘
- 微服务治理策略

Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby

总结-微服务带来的收益

以ServiceComb为核心构建的微服务架构体系，极大的提升了消费者云业务研发效率

需求的研发和交付周期缩短，业务更敏捷应对变化

可独立交付和升级的服务越来越多，业务之间逐步解耦，可独立演进

服务之间的接口契约可视化、可度量和管控，架构看护和优化更有章可循

服务无状态，微服务数据自治，天生的弹性伸缩架构，高效支撑业务快速发展

提供微服务维度的监控大盘和细粒度的治理措施，运维更得心应手

个人级、微服务团队级、项目级三级微服务流水线体系，基础设施自动化

...

Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby

Thank You.

Website: http://servicecomb.incubator.apache.org/

Gitter: https://gitter.im/ServiceCombUsers/Lobby