



# 21天微服务实战营-Day1

华为云DevCloud & 微服务联合出品



# Day1 微服务架构知识介绍

- 微服务简介
- 容器与容器平台
- 微服务架构模式
- 微服务开发框架
- Service Mesh
- 微服务平台

# 什么是微服务

微服务架构是一种架构模式，它要求开发者以一种不同于以往的开发方式进行软件开发，设计功能比较单一，拥有接口的服务，他们都可以被独立的构建，测试，部署。

微服务是得益于DevOps文化的发展，持续集成工具的成熟，越来越多的公司向敏捷转型，微服务架构模式可以指导企业开发出具有可伸缩，弹性，高可用的系统，从以往的几个月的上线频率，缩短为几周甚至几天。

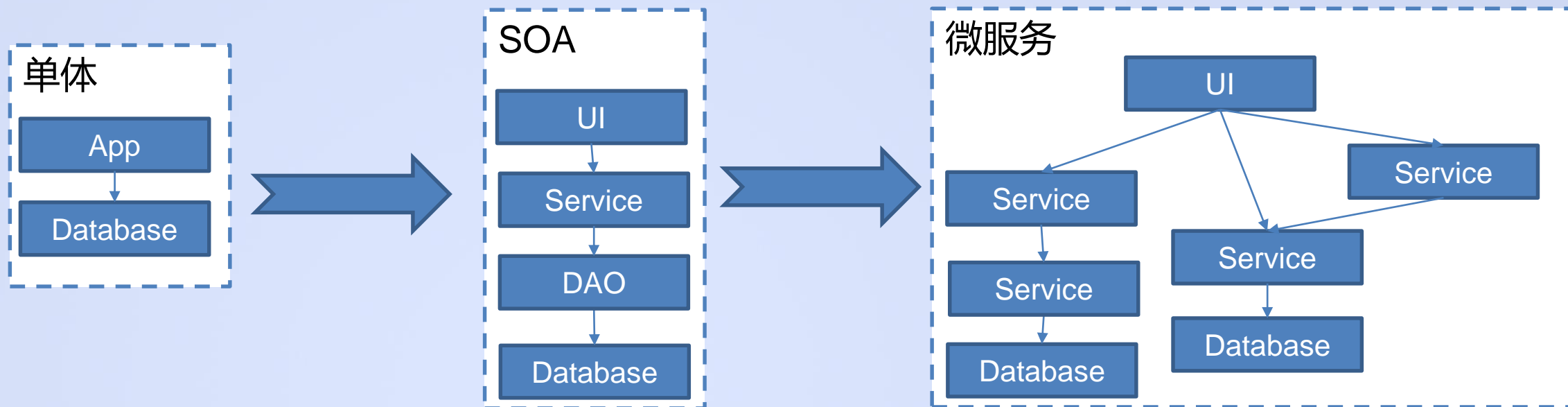
传统软件是由单一服务构成，微服务提倡将一个软件按照功能模块进行划分

# 为什么使用微服务

- 独立运行：服务异常不再彼此影响，必要时将非核心功能隔离，不影响主要功能运转。一个服务实例崩溃不会影响到其他实例，整体系统依然正常。按功能伸缩，当某个模块算力需求变化时只进行该功能实例的伸缩，而不是整个系统的伸缩，减少资源浪费。
- 独立升级：一个小特性的更改或者bug fix不会影响大部分功能的正常运转
- 代码复用：一套代码可以用于不同的独立系统中，在公司内部或者开源社区中进行分享。比如，支付服务，用户管理服务，认证鉴权。
- 技术演进：单体服务使用陈旧的技术，想象你过去使用struts1+spring，你想升级struts2来获得一定的收益，接着你想整体切换到Spring MVC,彻底摆脱struts框架，不断地切换框架为工程稳定性带来风险，而陈旧的框架又无人维护。而微服务项目不受旧代码拘束。
- 语言限制：当你发现某个新功能更适合使用Go而不是java时该怎么办，Java也不是万金油，每种语言都有适合自己的场景，微服务使开发者能根据服务场景选择语言。招聘开发者也不必局限于语言
- 团队：小团队运作更加敏捷，配合紧密，开发周期短，组织扩张灵活



# 历史



微服务的演进历史是漫长的，从单体的MVC架构到分布式SOA架构，在结合了敏捷开发，DevOps等理念后最终诞生了微服务。一个很好的印证是，在我深入的实践了DevOps和敏捷开发后，自发地开始萌芽了微服务的思想理论。

微服务最早出现在国内是在2015年初的时候，成功的案例有AWS以及Netflix等公司。

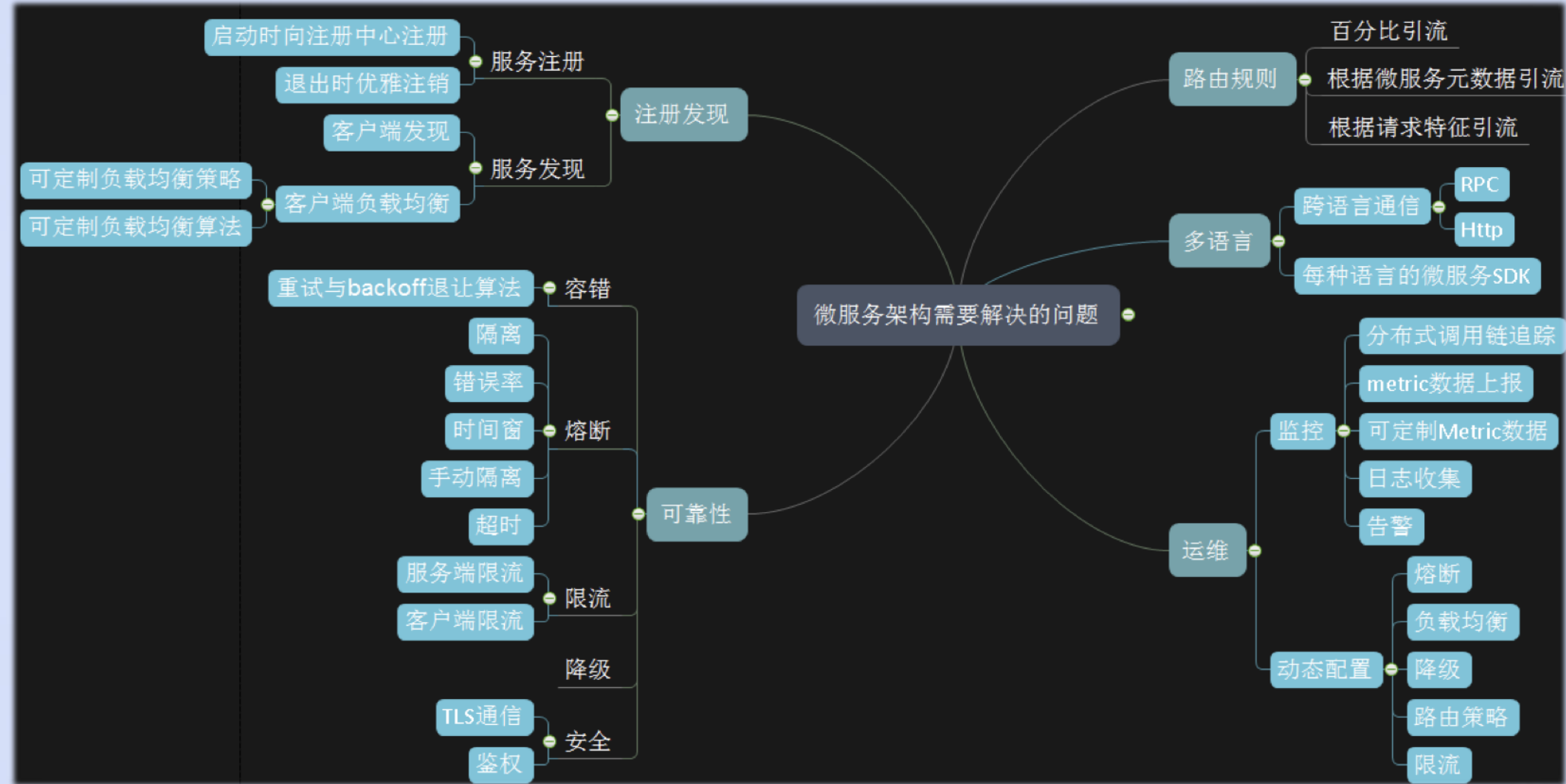
EC2最早是亚马逊内部使用的一个服务，最终被作为一种服务对外提供成为AWS，而基于微服务架构，AWS基于现有的服务之上快速迭代新的产品，丰富AWS能力，现在已经拥有100多种不同的服务，回报是巨大的。

华为很早便践行了微服务理论并对外开源了微服务相关项目，华为云得益于微服务架构快速推出大量新的云服务。

# 微服务面临的挑战

1. 持续集成：大量的工程，需要一个持续集成工具自动完成编译，打包，发布，部署等工作
2. 版本管理：大量的版本，就会遇到兼容性问题。你需要让项目可控
3. 文档管理：版本在持续升级，服务接口不匹配。你需要一个文档管理系统，并让开发者严格遵守文档进行开发
4. 生命周期管理：服务运行期，需要一个平台管理服务，除了部署，启停，还要能够在服务崩溃时自动拉起服务
5. 运维：运维人员操作服务，查看指标，日志，分布式调用链，更改配置项都由于微服务架构而变得比以往更加复杂
6. 调试：在开发期你如果依赖于很多微服务，如何方便地在本地去调用依赖的服务。
7. 网络调用：从过去本地的内存栈调用变为了网络调用，不再可靠
8. 安全：如何控制不让未经授权的调用者访问到自己的数据
9. 如何云服务化：转型微服务涉及一系列的工作，处理以上复杂的问题需要大量的基础代码研发，如何能驾驭诸多的技术和文化变更

# 构建微服务系统是困难的



接下来的章节中将简单介绍微服务模式带来的问题的解决方式，并在后续课程中进行实战

# 容器与容器平台

可以学习day1-day4的内容来了解相关技术

[https://activity.huaweicloud.com/21days\\_cce/index.html](https://activity.huaweicloud.com/21days_cce/index.html)

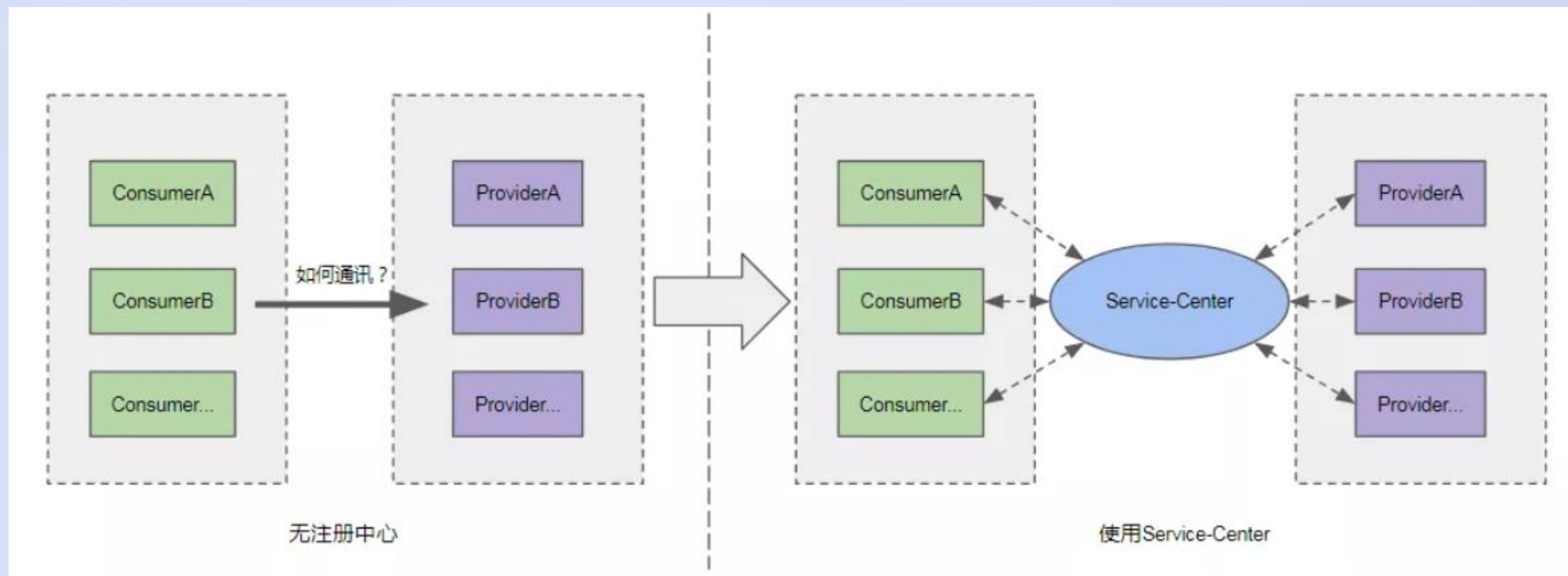
容器的出现帮助了微服务技术体系的成熟，这是至关重要的一环

容器和微服务是天生一对，在过去的虚拟机时代中，VM的启停甚至长达数分钟，在进行弹性伸缩时，假设你设置了这样的伸缩条件“CPU负载高于80%时，新增一台机器”，那么很可能在你没有得到一台新机器前，现存的VM中的服务就已经被压垮，继而引发级联崩溃。

容器的启停只需要数秒，并且由统一的容器平台管理全生命周期，为微服务系统提供了运行条件

从以往以虚拟机为中心的管理变为了以服务为中心的管理。让开发者专注于应用本身

# 微服务模式—注册发现



注册发现是微服务的基础，每个微服务实例都要注册自己的信息与地址到注册中心中，每个实例都在注册中心中查询自己需要访问的实例信息与真实地址。

Service center是华为云提供的一个典型的注册中心，帮助服务间进行注册与发现

注册中心优点：

1. 解耦服务提供者与服务消费者，服务消费者不需要硬编码服务提供者地址。
2. 服务动态发现及可伸缩能力，服务提供者实例的动态增减能通过注册中心动态推送到服务消费者端。
3. 通过注册中心可以动态地监控服务运行状态

# 微服务模式—路由管理

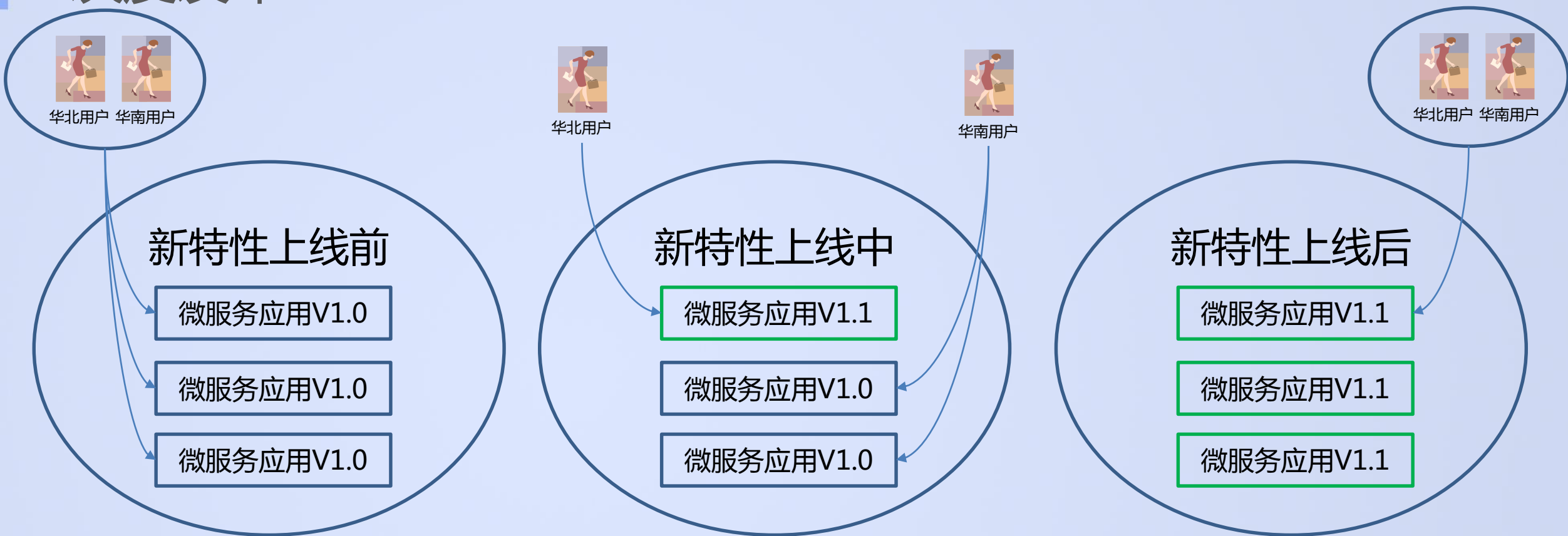
## ServiceA

微服务	地址	版本	数据中心
ServiceB	10.4.1.1:8080	1.0	北京
ServiceB	10.4.1.2:8080	1.0	深圳
ServiceB	10.4.1.3:8080	2.0	北京

微服务A已经根据微服务名查询到了相关的实例信息并放置于本地内存，如上图表格，那么如何决定自己到底要选择哪一批实例进行访问呢？

可以看到每个实例都具有一定的属性，上表中有2个属性版本与数据中心。那么编写算法根据微服务名和属性筛选服务实例，最终就可以决定要访问的服务实例的集合了

# 灰度发布



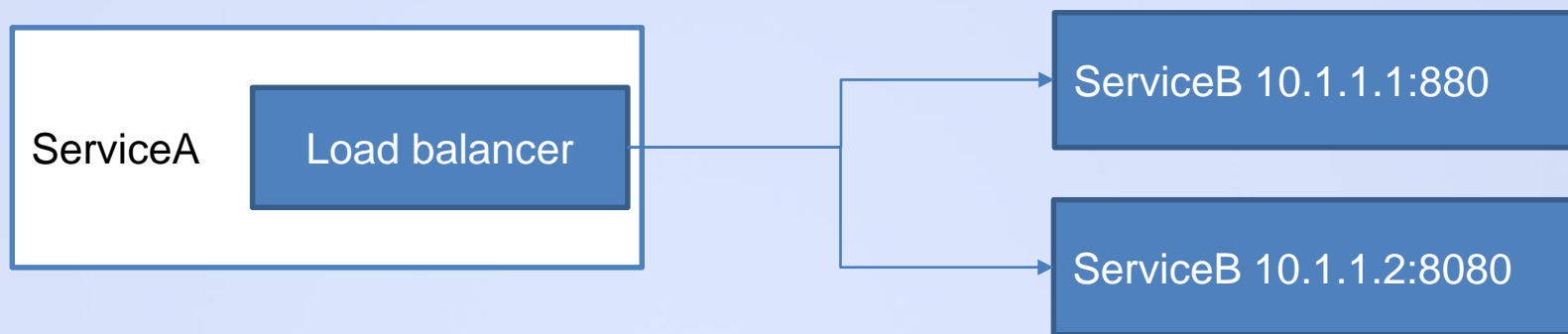
路由管理可以为我们解决什么问题？

灰度发布就是一个典型的场景

原本系统中运行着1.0版本的服务，我们可以将部分用户的流量迁移到新版本1.1中让部分用户优先试用。最终将所有流量迁移到新版本中

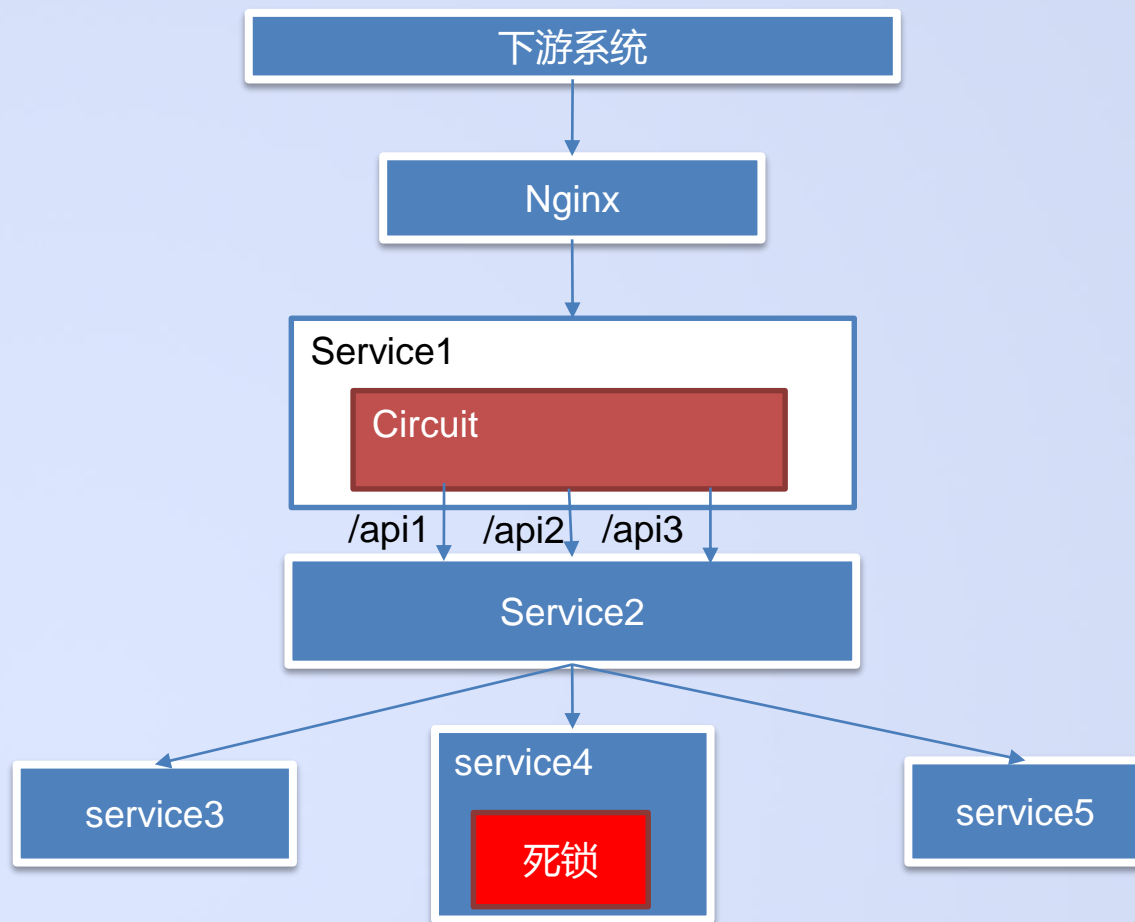


# 微服务模式—客户端负载均衡



当在路由过程中决定了一个实例集合后，就可以对集合实行负载均衡算法，选择其中一个实例访问，即客户端负载均衡，区别于nginx这样的传统负载均衡组件，负载均衡直接在客户端，也就是你的业务进程内部进行。

# 微服务模式—熔断



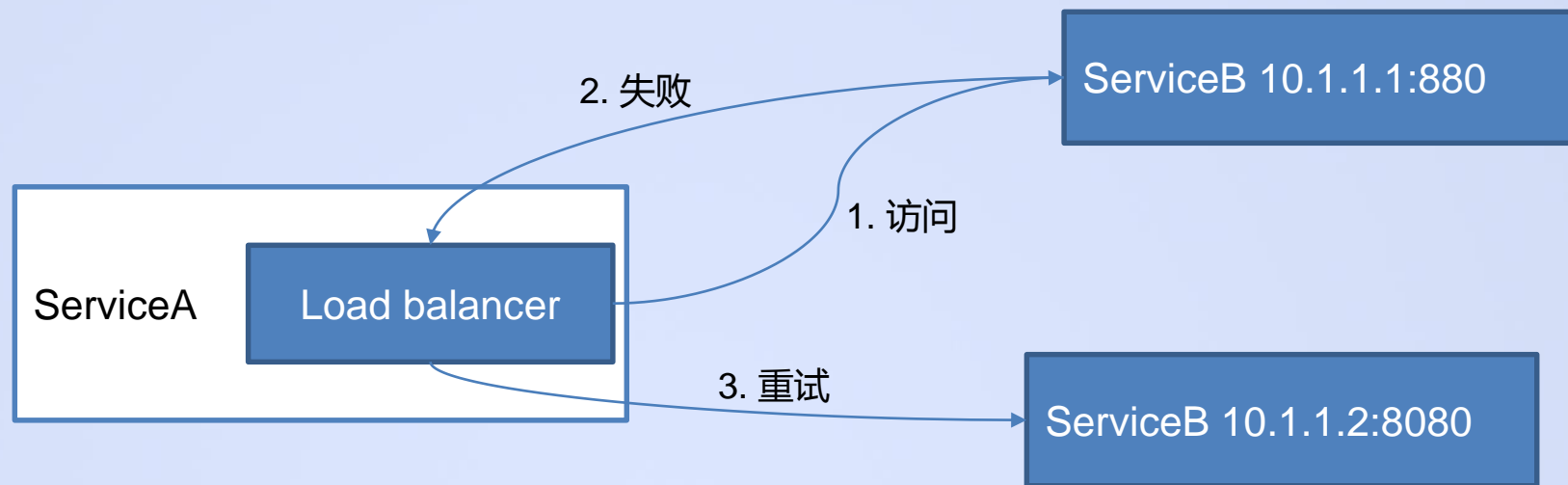
当服务发生错误，超时等问题，系统需要将这部分非核心功能隔离，以免引起级联崩溃

上图，Service1，调用api2，访问service4完成一个业务，当这个api2出现死锁时，将引起下游未做超时处理服务沾满线程池，最终大范围瘫痪，导致其他功能失效，想想如果更加庞大的系统，是一个多大的灾难，异常将被放大。

如果能够在访问api2达到一定超时次数就将api2隔离掉，不再发生网络调用，那么就不再产生新的死锁，系统稳定性就会提升

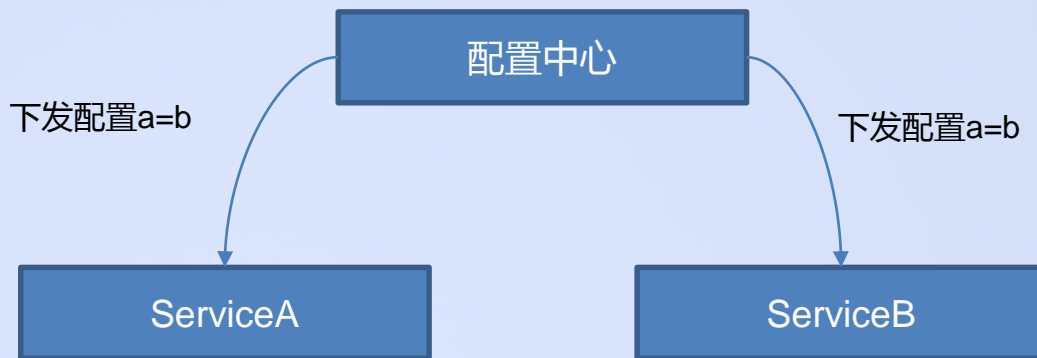
道理就像战舰的隔离仓，当遇到漏水的船舱时要将有问题的船舱隔离以避免灾难蔓延。

# 微服务模式—容错



当一个请求失败时，可以尝试对同一个地址进行重试或者从负载均衡中选取一个新的地址进行重试。

# 配置管理



服务分布在各个服务器中，在需要更改配置时，我们不想登录到每个机器上进行文件编辑，并进行服务重启，配置中心能够解决这个问题

当你遇到配置变更时，只需要在配置中心中进行更改，各个服务受到变更消息后，进行更改并在运行时生效

Day5将详细介绍

# 监控

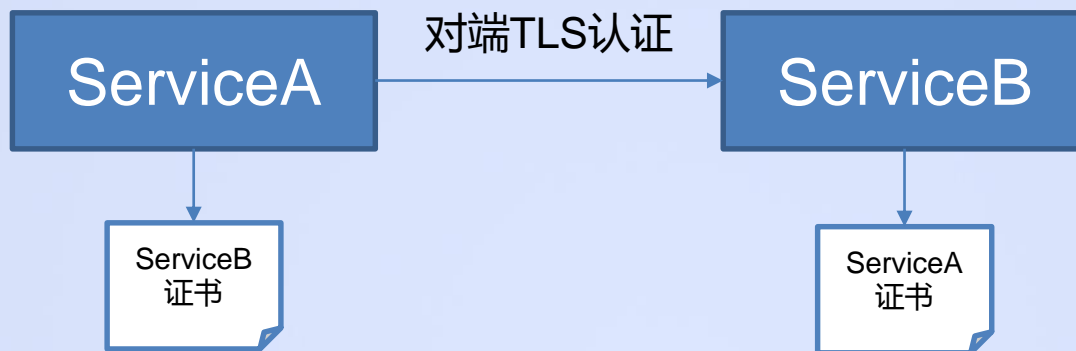
**日志：**分布式系统庞大，无法再传统的登录到服务器上去查看日志，需要将日志上报并统一汇聚到一个监控系统中

**分布式调用链追踪：**网络的调用不同于本地调用，就像本地调用可以用工具分析，分布式的网络调用也需要被监控起来，并在监控系统中进行分析以便随时掌握系统调用状况，分析服务性能瓶颈等。

**指标：**指标分为通用指标如cpu，内存，请求数量，延迟等。以及自定义指标如用户注册量，商品购买量等

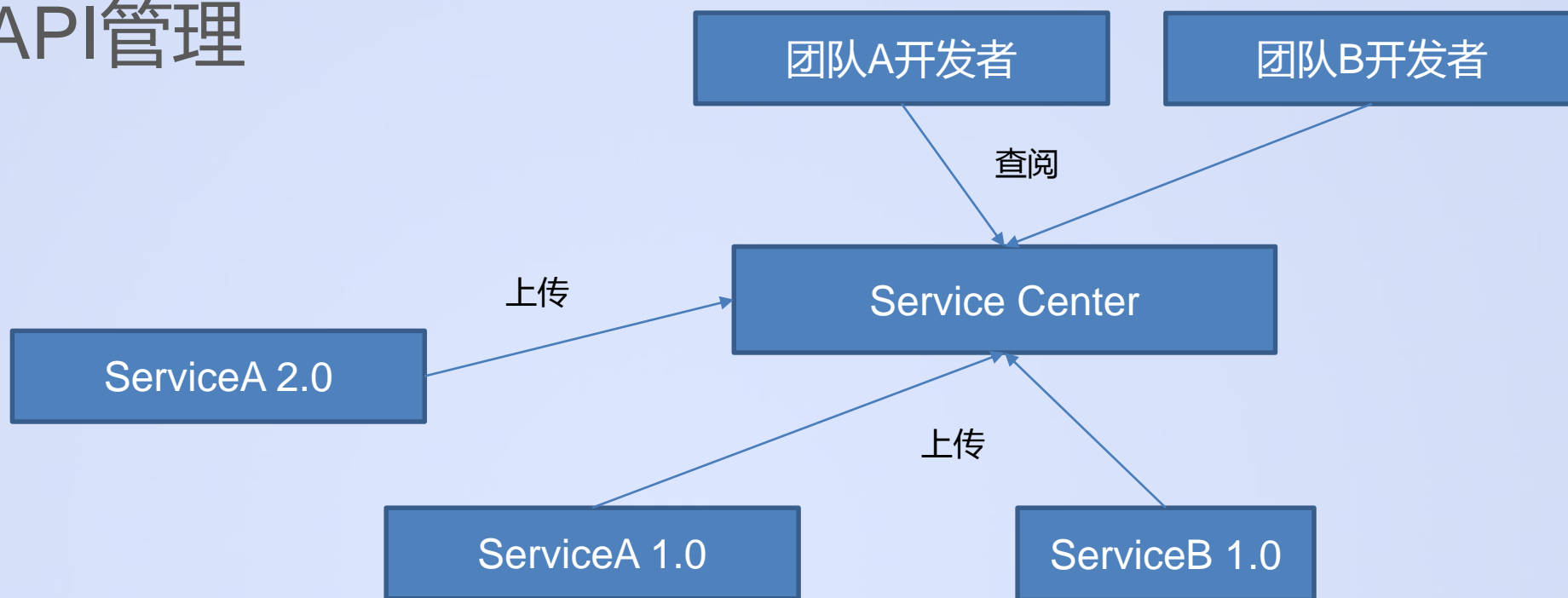
华为的APM服务是一个监控平台，能够支撑微服务系统的运行。开源中有Promethues，Zipkin，elastic search，kafka，TSDB等大量服务可用于搭建一个监控系统。

# 安全



网络调用引起了安全访问的问题，使用对端TLS认证可以解决这个问题。由ServiceB，ServiceA的开发者都签发证书，分发给彼此，两者加载对方证书，对彼此进行认证，以确定彼此真实身份。

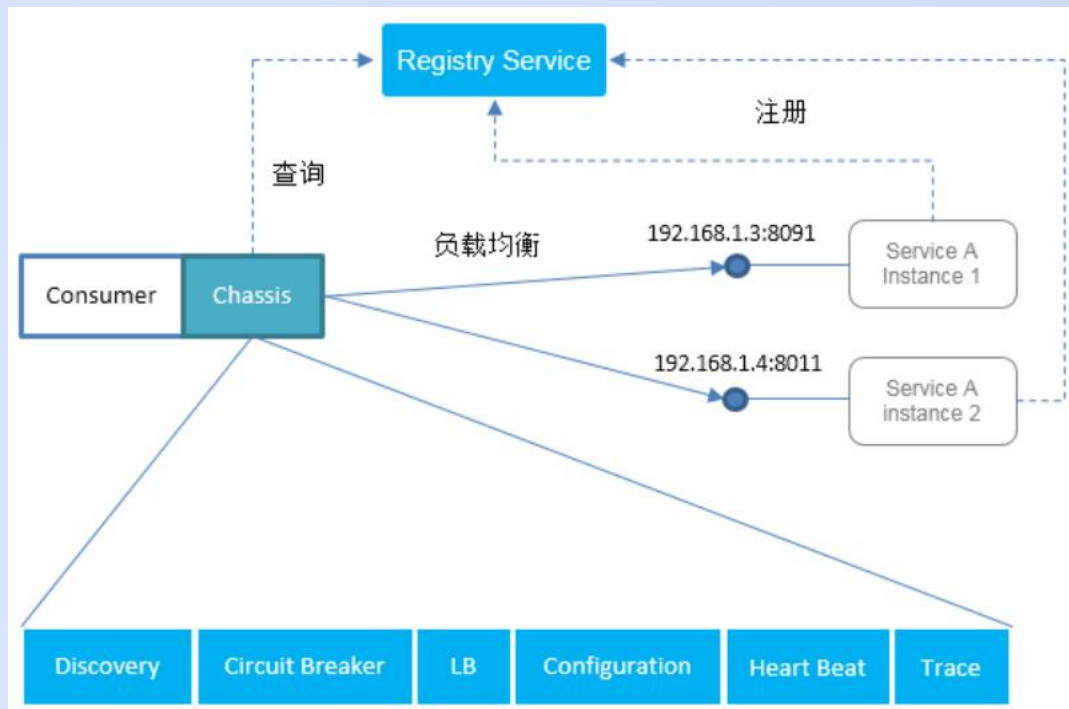
# API管理



Service center不同于竞品独有的API文档管理能力，可以托管系统中所有服务的API文档，各个团队成员或者管理者可以在service center中查看服务文档，并以此为设计，开发依据。微服务版本与文档为绑定关系，提高了沟通效率以及可靠性。



# 开发框架

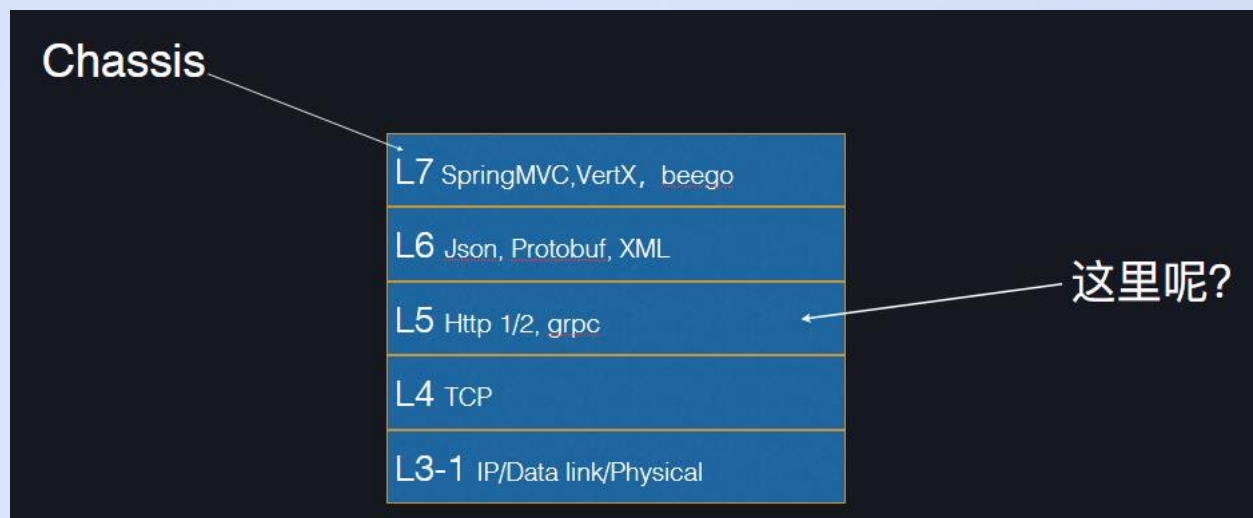


要完成以上的功能需要在微服务中编写代码，而这些代码都可以作为通用库来提供，这就是微服务开发框架。

开发者引入框架并学习开发方式，配置方式，就可以快速开发出具备微服务特性的应用。我们称这种模式为chassis，华为云提供java-chassis与go-chassis 2种语言框架，供用户选择

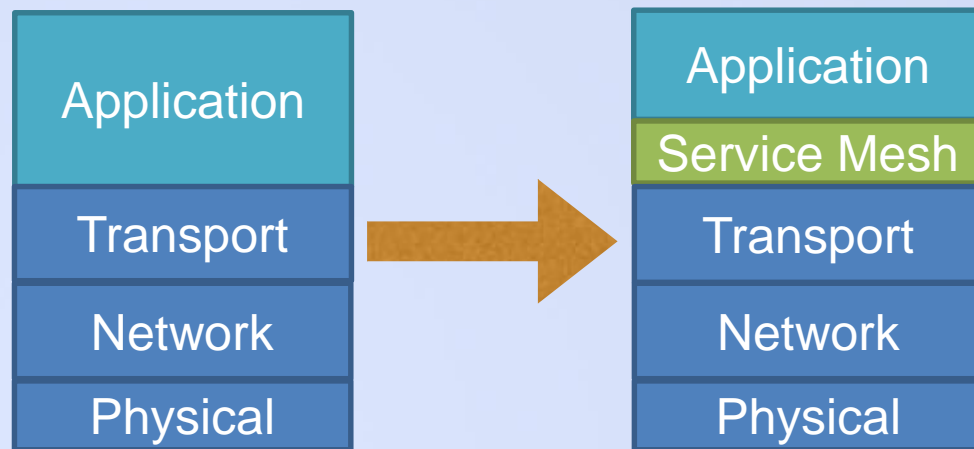
Day2-14课程将介绍如何使用框架进行开发。

# Service Mesh



Service Mesh同样解决微服务面临的问题，但是以另一种解题思路完成的。我们在第7层使用各种开发框架，如传统开发框架Spring MVC，微服务开发框架java-chassis，go-chassis等等，编码时直接调用。那么微服务开发框架的功能下沉到5层呢？

# Service Mesh



2017年由William Morgan提出，一种基础设施层，服务间通信通过Service Mesh进行，一种TCP/IP之上的网络模型，轻量网络代理，与业务部署在一起，负责可靠传输复杂拓扑网络中的请求，将应用变为现代的云原生应用。

可以简单地理解为以前应用跑在TCP/IP之上，现在跑在Service mesh之上以处理微服务模式带来的问题。

Service Mesh的专题将在Day6的直播中介绍。

# 微服务平台

除了开发，微服务还需要解决的是将持续集成，基础设施，监控，中间件等功能组合在一起的平台为开发者提供良好的服务，这部分将在Day15-21介绍

# Thank You





# 21天微服务实战营-Day2

华为云DevCloud & ServiceStage服务联合出品



# Day2 微服务入门之编写HelloWorld

## 大纲

- 开发第一个微服务
- 服务契约
- 开发服务调用者



# 开发第一个微服务

创建一个空的maven工程，然后在pom文件中加入如下依赖：

```
<properties>
... <cse.version>2.3.62</cse.version>
... </properties>

<dependencyManagement>
... <dependencies>
... <dependency>
... <groupId>com.huawei.paas.cse</groupId>
... <artifactId>cse-dependency</artifactId>
... <version>${cse.version}</version>
... <type>pom</type>
... <scope>import</scope>
... </dependency>
... </dependencies>
... </dependencyManagement>

<dependencies>
... <dependency>
... <groupId>com.huawei.paas.cse</groupId>
... <artifactId>cse-solution-service-engine</artifactId>
... </dependency>
... </dependencies>
```

创建一个简单的CSEJavaSDK微服务只需要引入cse-solution-service-engine包，但我们仍然推荐大家使用<dependencyManagement>管理依赖依赖，这在项目依赖关系复杂时可以有效降低依赖管理复杂度。

# 开发第一个微服务

## 创建一个main类：

```
public class AppMain {  
    public static void main(String[] args) throws Exception {  
        Log4jUtils.init(); // 初始化默认的日志组件  
        BeanUtils.init(); // 加载Spring bean定义文件，正式开始启动流程  
    }  
}
```

CSEJavaSDK使用的默认日志组件是Log4J，并在此基础上进行了一些默认的配置，可以开箱即用。

## 创建服务的REST接口类：

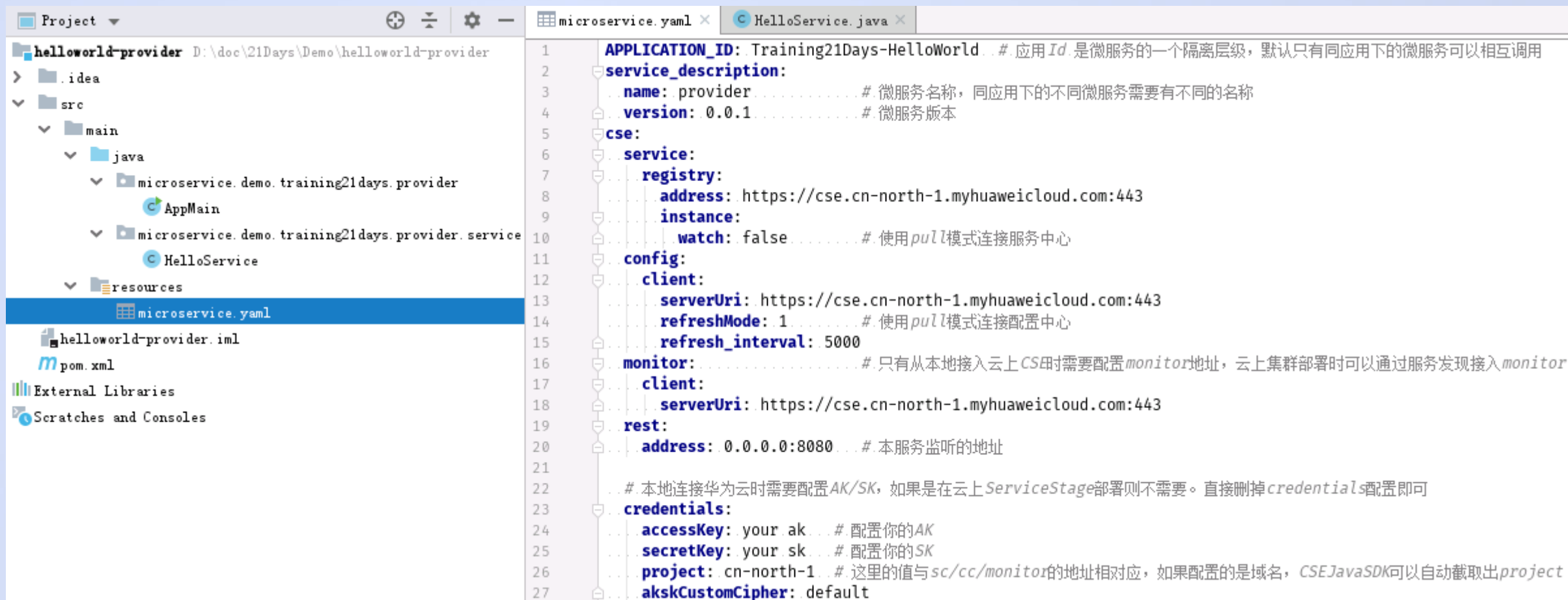
```
@RestSchema(schemaId = "hello") // 该注解声明这是一个REST接口类，CSEJavaSDK会扫描到这个类，根据它的代码生成接口契约  
@RequestMapping(path = "/provider/v0") // @RequestMapping是Spring的注解，这里在使用Spring MVC风格开发REST接口  
public class HelloService {  
    @RequestMapping(path = "/hello/{name}", method = RequestMethod.GET)  
    public String sayHello(@PathVariable(value = "name") String name) {  
        return "Hello," + name;  
    }  
}
```

一个微服务可以有多个接口契约。这里使用@RestSchema注解声明HelloService是一个契约id为hello的REST接口，同时会在启动时生成相应的契约。

这里的REST接口是以Spring MVC风格开发的。CSEJavaSDK支持的开发风格有REST([JAX-RS](#)、[Spring MVC](#))和[RPC](#)，开发者可以自由选用。

# 开发第一个微服务

在src\main\resources\目录下创建一份microservice.yaml文件



The screenshot shows an IDE with two panes. The left pane displays the project structure for 'helloworld-provider' located at 'D:\doc\21Days\Demo\helloworld-provider'. The structure includes a 'src' directory with 'main' and 'resources' subdirectories. The 'main' directory contains 'java' and 'resources' subdirectories. The 'resources' directory contains 'microservice.yaml'. The 'main' directory also contains 'AppMain' and 'microservice.demo.training21days.provider.service' subdirectories. The 'resources' directory also contains 'helloworld-provider.iml' and 'pom.xml' files. The right pane shows the content of the 'microservice.yaml' file, which is a YAML configuration for a microservice. The configuration includes fields for 'APPLICATION\_ID', 'service\_description', 'cse', 'config', 'monitor', 'rest', and 'credentials'. The 'APPLICATION\_ID' is 'Training21Days-HelloWorld'. The 'service\_description' includes 'name' (provider), 'version' (0.0.1), and 'cse' (service). The 'cse' section includes 'registry' (address: https://cse.cn-north-1.myhuaweicloud.com:443) and 'instance' (watch: false). The 'config' section includes 'client' (serverUri: https://cse.cn-north-1.myhuaweicloud.com:443, refreshMode: 1, refresh\_interval: 5000) and 'monitor' (serverUri: https://cse.cn-north-1.myhuaweicloud.com:443). The 'rest' section includes 'address' (0.0.0.0:8080). The 'credentials' section includes 'accessKey', 'secretKey', 'project', and 'akskCustomCipher'.

```
1 APPLICATION_ID: Training21Days-HelloWorld... # 应用Id 是微服务的一个隔离层级，默认只有同应用下的微服务可以相互调用
2 service_description:
3   name: provider... # 微服务名称，同应用下的不同微服务需要有不同的名称
4   version: 0.0.1... # 微服务版本
5   cse:
6     service:
7       registry:
8         address: https://cse.cn-north-1.myhuaweicloud.com:443
9         instance:
10          watch: false... # 使用pull模式连接服务中心
11       config:
12         client:
13           serverUri: https://cse.cn-north-1.myhuaweicloud.com:443
14           refreshMode: 1... # 使用pull模式连接配置中心
15           refresh_interval: 5000
16         monitor: ... # 只有从本地接入云上CSE时需要配置monitor地址，云上集群部署时可以通过服务发现接入monitor
17         client:
18           serverUri: https://cse.cn-north-1.myhuaweicloud.com:443
19       rest:
20         address: 0.0.0.0:8080... # 本服务监听的地址
21
22   # 本地连接华为云时需要配置AK/SK，如果是在云上ServiceStage部署则不需要。直接删掉credentials配置即可
23   credentials:
24     accessKey: your ak... # 配置你的AK
25     secretKey: your sk... # 配置你的SK
26     project: cn-north-1... # 这里的值与sc/cc/monitor的地址相对应，如果配置的是域名，CSEJavaSDK可以自动截取出project
27     akskCustomCipher: default
```

# 开发第一个微服务

运行AppMain类，可以在ServiceStage的微服务控制台看到provider服务

 华为云

北京一

控制台 总览 资源管理 应用开发 应用上线 应用运维 软件中心

费用 资源 工单 备案 yhs0092

76



微服务控制台  
Cloud Service Engine

仪表盘

服务目录

服务治理

全局配置

事务看板

←

服务目录

您可以从应用、微服务和实例的维度查看微服务详细信息。如果微服务较多，您可以通过搜索功能查找目标微服务。[如何维护微服务？](#)

应用列表 微服务列表 实例列表

创建微服务 删除

Training21Days-Hello... 微服务名称

<input type="checkbox"/>	微服务名称	所属应用	版本数	实例数	创建时间	操作
<input type="checkbox"/>	provider	Training21Days-Hell...	1	1	2019/02/12 20:26:10 GMT+0...	删除

体验新版

# 你已经开发出第一个微服务了！

调用 <http://127.0.0.1:8080/provider/v0/hello/Bob> ，可以得到provider服务的应答

The screenshot displays a REST client interface with the following components:

- URL Bar:** Shows the URL `http://127.0.0.1:8080/provider/v0/hello/Bob`.
- Method and URL:** A dropdown menu is set to `GET`, and the URL `http://127.0.0.1:8080/provider/v0/hello/Bob` is entered.
- Buttons:** `Send` and `Save` buttons are visible.
- Tabs:** `Params`, `Authorization`, `Headers`, `Body`, `Pre-request Script`, and `Tests` are listed. `Params` is the active tab.
- Params Table:**

KEY	VALUE	DESCRIPTION
Key	Value	Description
- Response Section:**
  - `Status:` 200 OK
  - `Time:` 4 ms
  - `Size:` 97 B
  - `Download` button
- Response Body:** The response is displayed in JSON format: `{\"Hello,Bob\"}`.

# 服务契约

The screenshot shows the CSE Java SDK console interface. On the left, there's a sidebar with navigation links: 基本信息 (Basic Information), 动态配置 (Dynamic Configuration), and 灰度发布 (Canary Release). The main area displays details for a service named 'Training21Days-HelloWo...'. The status is '在线' (Online), with 1 normal instance and 0 abnormal instances. The version is 0.0.1, and the creation time is 2019/02/12 20:26:10 GMT+08:00. Below this, there are tabs for 实例列表 (Instance List), 被调用服务 (Called Service), 调用服务 (Called Service), and 服务契约 (Service Contract). The '服务契约' tab is selected, showing a dropdown menu with 'hello' and two tabs: 'Swagger' and 'Yaml'. The 'Yaml' tab is active, displaying the following YAML content:

```
1 ---
2 swagger: "2.0"
3 info:
4   version: "1.0.0"
5   title: "swagger definition for microservice.demo.training21days.provider.service.HelloService"
6   x-java-interface: "cse.gen.Training21Days_HelloWorld.provider.hello.HelloServiceIntf"
7 basePath: "/provider/v0"
8 consumes:
9   - "application/json"
10 produces:
11   - "application/json"
12 paths:
13   /hello/{name}:
14     get:
15       operationId: "sayHello"
16       parameters:
17         - name: "name"
18           in: "path"
19           required: true
20           type: "string"
21       responses:
22         200:
23           description: "response of 200"
24           schema:
25             type: "string"
```

点击provider服务的记录查看详情，我们可以看到一份服务契约。

服务契约描述了微服务的接口，是在启动过程中由CSEJavaSDK根据微服务REST接口类（这里是HelloService.java）自动生成的。如果你观察一下provider服务的启动日志，会发现在日志里也将生成的契约打印出来了。

服务契约不仅仅是一份接口文档，它也约束了CSEJavaSDK运行时接收请求和返回应答的行为。

CSEJavaSDK使用的服务契约是Swagger契约，用户可以从网上搜索到相关资料。关于REST接口定义的约束，可以参考[接口定义和数据类型](#)。

## TIPS：

服务契约描述了服务的接口，因此契约内容的变化可以认为是服务接口变化了。在正式的生产环境中这应该是不允许随意发生的。修改provider服务的接口，重启服务，可以发现服务启动失败，因为它的契约与服务中心中保存的契约不一致。如果碰到这种问题，在正式的生产环境中推荐的处理方式是在microservice.yaml文件中升级微服务版本；开发环境也可以考虑删除服务中心里的provider服务记录，或者配置service\_description.environment=development。

# 服务契约



参见ServiceComb开源资料[ServiceComb-Java-Chassis微服务系统架构](#)，服务契约的作用贯穿ServiceComb的三个模型，而不是简单地作为接口文档。服务契约将三个模型解耦，这使得运行模型中的同一套微服务治理逻辑既可以用于不同的开发风格代码，也可以用于不同的通信方式，让框架的功能扩展能力更好。同时接口契约规范了provider和consumer双方的交互行为，让开发与测试之间、不同微服务的开发之间的沟通协作效率更高。

CSEJavaSDK作为一个带服务契约的REST开发框架，在使用上不会像传统的Servlet开发方式那么随心所欲，但随着系统规模扩大、复杂度提升，服务契约带来的好处将会明显大于其在开发方式上的限制。



# 开发微服务调用者

开发一个consumer服务来调用provider服务，pom.xml文件和main类完全相同，定义一个REST接口类接收外部请求并调用provider服务：

```
@RestSchema(schemaId = "helloConsumer")
@Path("/consumer/v0") // 这里使用 JAX-RS 风格开发的 consumer 服务
public class HelloConsumerService {
    // RPC调用方式需要声明一个 provider 服务的 REST 接口代理
    @RpcReference(microserviceName = "provider", schemaId = "hello")
    private HelloService helloService;

    // RestTemplate调用方式需要创建一个 ServiceComb 的 RestTemplate
    private RestTemplate restTemplate = RestTemplateBuilder.create();

    @Path("/hello")
    @GET
    public String sayHello(@QueryParam("name") String name) {
        // RPC 调用方式体验与本地调用相同
        return helloService.sayHello(name);
    }

    @Path("/helloRT")
    @GET
    public String sayHelloRestTemplate(@QueryParam("name") String name) {
        // RestTemplate 使用方式与原生的 Spring RestTemplate 相同，可以直接参考原生 Spring 的资料
        // 注意 URL 不是 http://{IP}:{port}，而是 cse://{provider端服务名}，其他部分如 path/query 等与原生调用方式一致
        ResponseEntity<String> responseEntity =
            restTemplate.getForEntity("cse://provider/provider/v0/hello/" + name, String.class);
        return responseEntity.getBody();
    }
}
```

```
// 定义 RPC 调用方式所使用的代理接口
public interface HelloService {
    // 方法名称与 provider 服务契约中的 operationId 保持一致
    // 参数顺序与 provider 服务契约中定义的顺序保持一致
    String sayHello(String name);
}
```

CSEJavaSDK支持[JAX-RS](#)、[Spring MVC](#)和[RPC](#)三种开发风格，一般我们推荐用户使用前两种，配合CSEJavaSDK自动生成服务契约的能力开发更方便。

CSEJavaSDK提供了两种微服务调用方式，[RPC方式](#)和[RestTemplate方式](#)。

# 开发微服务调用者

consumer服务的microservice.yaml文件与provider服务基本一致，但是注意它的服务名称需要改为“consumer”，服务监听地址改为“0.0.0.0:9090”。

启动consumer服务，可以在ServiceStage的微服务控制台看到consumer服务。调用consumer服务的两个接口，可以看到通过RPC方式和RestTemplate方式都能够成功调用provider。

The screenshot displays two REST client interactions. Each interaction shows a GET request to a specific URL, followed by a response with a 200 OK status, 8 ms execution time, and 99 B size. The response body for both is a JSON array containing the string "Hello,Alice".

**Request 1:**

```
GET http://127.0.0.1:9090/consumer/v0/hello?name=Alice
```

**Response 1:**

```
[{"name": "Alice"}]
```

**Request 2:**

```
GET http://127.0.0.1:9090/consumer/v0/helloRT?name=Alice
```

**Response 2:**

```
[{"name": "Alice"}]
```

# Thank You





# 21天微服务实战营-Day3

华为云DevCloud & ServiceStage服务联合出品



# Day3 感知微服务和CSE的交互

## 大纲

- 服务中心
- 配置中心
- monitor
- 通过代理连接CSE服务

# 支撑微服务运行的服务

```
cse:
  service:
    registry:
      address: https://cse.cn-north-1.myhuaweicloud.com:443
      instance:
        watch: false
    config:
      client:
        serverUri: https://cse.cn-north-1.myhuaweicloud.com:443
        refreshMode: 1
        refresh_interval: 5000
    monitor:
      client:
        serverUri: https://cse.cn-north-1.myhuaweicloud.com:443
```

在microservice.yaml文件中，我们配置了三个地址，微服务启动的时候，会从配置文件中读取这些地址，分别连接华为云微服务引擎（Cloud Service Engine，CSE）的三个服务：

- 服务中心（service center，sc）
- 配置中心（config center，cc）
- monitor

# 服务中心

服务中心提供了微服务注册、管理、发现功能。

- 当一个provider微服务实例启动时，会将自己的服务信息、实例信息等注册到sc。
- 当consumer服务需要调用provider时，会去sc查询provider的服务契约、实例列表等信息，将请求发往实例列表中记录的实例。
- 没有服务中心，微服务无法实现相互调用，因此服务中心是微服务实例启动时必须连接的服务。

# 服务中心

配置项	默认值	说明
servicecomb.service.registry.address	https://127.0.0.1:30100	服务中心的地址
servicecomb.service.registry.instance.watch	true	是否采用watch模式监听实例变化。推荐将这个选项显式配置为false，即使用pull模式。
servicecomb.service.registry.instance.pull.interval	30	pull模式下consumer服务刷新查询provider服务实例列表的时间间隔
servicecomb.service.registry.instance.healthCheck.interval	30	心跳时间间隔，单位为秒，正常运行的微服务实例每隔这么长时间向sc发送一次心跳请求以维持自己的在线状态
servicecomb.service.registry.instance.healthCheck.times	3	允许的连续心跳失败次数，如果出现n+1次连续心跳失败则sc认为该实例已异常关闭，将自动下线该实例
servicecomb.service.registry.instance.empty.protection	true	是否启用空实例保护，true表示如果从sc查询到的provider实例列表为空，将会尝试沿用本地缓存的provider实例列表



# 配置中心

- 配置中心提供了存储、管理配置项的功能。
- 通过连接配置中心，微服务可以获得运行时动态变更配置的能力。
- 服务治理功能的配置也是由配置中心下发的。
- 配置中心客户端包含在org.apache.servicecomb:config-cc包中。

# 配置中心

配置项	默认值	说明
servicecomb.config.client.serverUri	无	配置中心地址，如果为空则微服务实例不会连接配置中心
servicecomb.config.client.refreshMode	0	配置刷新方式（0/1），0表示watch模式，1表示pull模式。推荐显式配置为1
servicecomb.config.client.refresh_interval	30000	配置刷新的时间间隔，单位是毫秒

# monitor

monitor服务允许微服务实例上报吞吐量等数据，并在CSE的服务治理页面上展示相关数据。Monitor功能包含在com.huawei.paas.cse:cse-handler-cloud-extension包中。



# monitor

配置项	默认值	说明
servicecomb.monitor.client.serverUri	无	如果没有配置该地址，则会从服务中心通过服务发现的方式获取monitor地址。从华为云外部连接CSE时需要显式配置地址，在华为云上部署时推荐不配置，走sc服务发现。
servicecomb.monitor.client.enabled	true	是否连接monitor

# 通过代理连接CSE服务

如果开发调试环境无法直接连接华为云CSE服务，CSEJavaSDK也提供了代理配置，允许通过代理连接sc/cc/monitor服务。参见[代理设置](#)。

代理配置在microservice.yaml文件中，示例如下所示：

```
servicecomb:
  proxy:
    enable: true           #是否开启代理
    host: yourproxyaddress #代理地址
    port: 80               #代理端口
    username: yourname     #用户名
    passwd: yourpassword   #密码
```

# Thank You





# 21天微服务实战营-Day4

华为云DevCloud & ServiceStage服务联合出品



# Day4 微服务实例的生命周期分析

## 大纲

- 服务启动流程
- 服务发现
- 服务退出



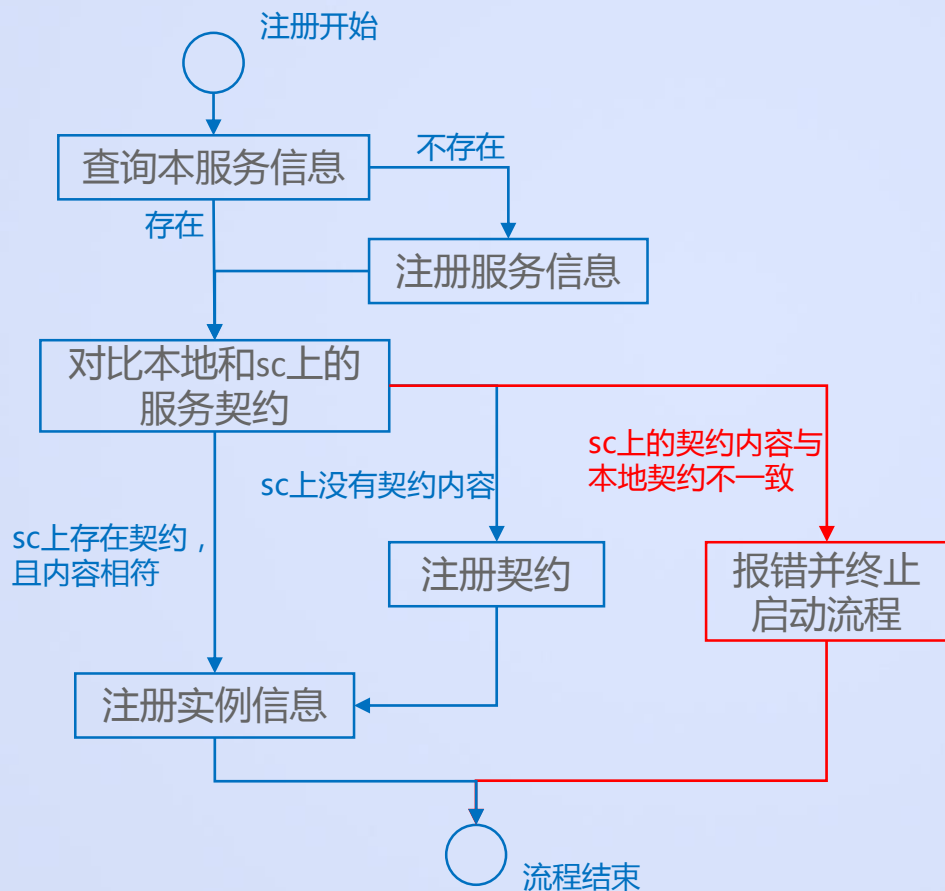
# 服务启动流程

一个微服务实例在启动过程中主要经历的流程有：

- 初始化日志框架
- 加载本地配置（包括System Property、环境变量、配置文件）
- 实例化Spring Bean
- 初始化SCBEngine
- 注册服务

# 服务启动流程

注册服务的流程如下：



- 当env为默认环境或production环境时，不允许出现服务实例本地与sc上的服务契约不一致的情况（一旦契约不一致会走左图中的红色路径）
- env=development时，服务实例会将不一致的接口契约注册到sc，覆盖sc上原有的契约（如下图）

```
APPLICATION_ID: HelloWorld
service_description:
  name: provider
  version: 0.0.1
  environment: development
```

# 服务启动流程

当微服务实例注册到sc上去后，启动流程完成。如果有一些操作需要在服务启动完成时执行，可以定义一个org.apache.servicecomb.core.BootListener去监听事件，并在接收到AFTER\_REGISTRY事件时触发操作的执行。

```
@Component
public class CustomBootEventListener implements BootListener {

    private static final Logger LOGGER = LoggerFactory.getLogger(CustomBootEventListener.class);

    public void onBootEvent(BootEvent bootEvent) {
        if (!EventType.AFTER_REGISTRY.equals(bootEvent.getEventType())) {
            return;
        }

        LOGGER.info("=====");
        LOGGER.info("Service startup completed!");
        LOGGER.info("=====");
    }
}
```

```
receive MicroserviceInstanceRegisterTask event, check instance Id... org.apache.servicecomb.core.SCBEngine$1.afterRegistryInstance(SCBEngine.java:182)
instance registry succeeds for the first time, will send AFTER_REGISTRY event. org.apache.servicecomb.core.SCBEngine$1.afterRegistryInstance(SCBEngine.java:184)
===== microservice.demo.training21days.provider.bootevent.CustomBootEventListener.onBootEvent(CustomBootEventListener.java:18)
Service startup completed! microservice.demo.training21days.provider.bootevent.CustomBootEventListener.onBootEvent(CustomBootEventListener.java:19)
===== microservice.demo.training21days.provider.bootevent.CustomBootEventListener.onBootEvent(CustomBootEventListener.java:20)
```

# 服务发现

Consumer服务调用provider服务时，需要去服务中心查询provider服务的契约、实例列表等信息，然后才能对provider发起调用：

- 查询条件包括AppID、serviceName、environment、versionRule，如果碰到consumer端找不到provider服务的问题，除了检查provider服务的实例有没有注册到sc，还需要检查这四个配置项是否有问题
- 查询到provider端服务信息后，consumer会从sc下载该provider服务的全部契约，加载到本地
- Consumer端加载provider服务信息的过程发生在consumer第一次调用该provider的时候，如果consumer服务的实例在启动后一直没有调用provider，则它一直不会去sc查询和加载provider服务信息

# 服务退出

CSEJavaSDK向JVM注册了一个shutdown hook，以实现优雅停机，在JVM进程退出时进行一系列的清理操作，其中包括：

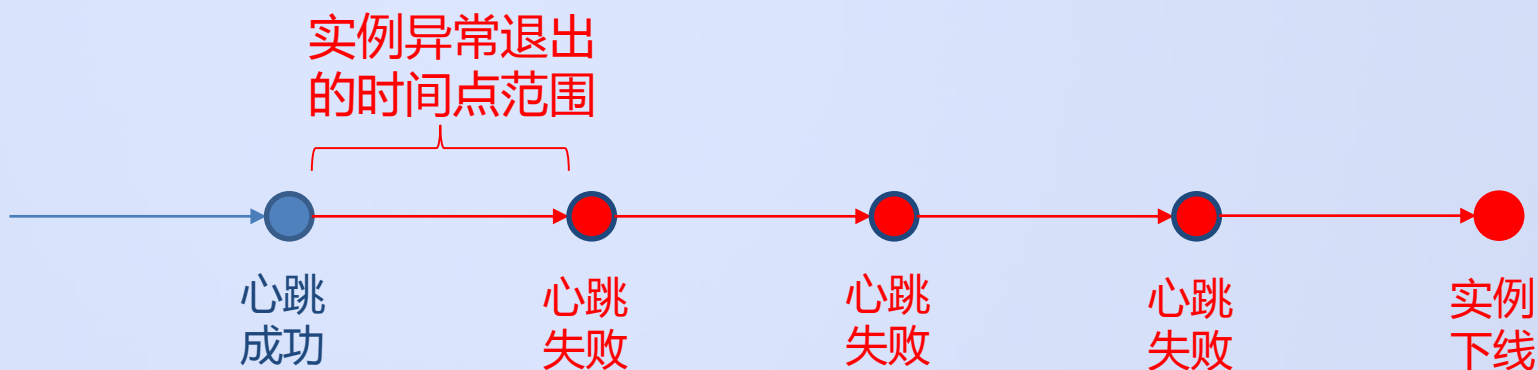
- 向服务中心注销本实例
- 停止接收请求，并等待已接受的请求处理完成

由于JVM的限制，要确保优雅停机功能正常触发，需要用户正常停止JVM进程，而不能强制杀进程。以Linux操作系统为例：

- kill \${PID} 的方式停止微服务进程可以触发优雅停机
- kill -0 \${PID} 的方式强制停止微服务进程则不会触发优雅停机

# 服务退出

如果微服务实例遭遇异常情况，没有调用sc接口注销自身实例就停止运行。sc会通过感知心跳超时的方式下线实例。前面的课程中提到微服务连接sc的配置中有心跳时间间隔和允许连续心跳失败次数这两个配置，假设心跳时间间隔为 $t$ ，允许心跳失败次数为 $n$ ，则sc检测到实例连续心跳失败 $n+1$ 次的时候下线实例，从实例异常退出到sc下线实例的时延 $T$ 的取值范围是  $t*n < T < t*(n+1)$ ，按照默认值计算为90-120秒



# Thank You





# 21天微服务实战营-Day5

华为云DevCloud & ServiceStage服务联合出品





# Day5 教你如何配置你的微服务

## 大纲

- microservice.yaml配置文件
- 环境变量、System property
- 动态配置
- 通过API获取配置
- 日志配置

# microservice.yaml配置文件

微服务实例启动时会从classpath下加载microservice.yaml配置文件：

- 如果多个jar包下都有microservice.yaml文件，那么他们都会被加载
- 磁盘目录下的microservice.yaml配置文件的优先级高于jar包内的配置文件
- 可以通过在microservice.yaml文件内配置servicecomb-config-order来指定优先级

# 环境变量、System property

微服务实例启动时也会从环境变量、系统属性中加载配置：

- Linux系统的环境变量不允许有点号“.”，但CSEJavaSDK框架会自动将配置项key中的下划线映射为点号，因此我们可以将点转换为下划线来配置环境变量
- 环境变量的优先级高于配置文件，system property的优先级高于环境变量

# 动态配置

微服务实例连接配置中心后，可以从配置中心获取动态配置：

- 动态配置的优先级是最高的，并且可以在运行时刷新
- 服务治理所使用的诸多控制逻辑也是由配置项来控制的。实现服务动态治理的方式就是通过配置中心动态下发配置项

# 通过API获取配置

CSEJavaSDK使用统一的API来获取配置，用户使用配置的时候，不需要关心从环境变量或者配置中心来读取配置，框架已经自动为用户从各个配置来源读取配置，并根据优先级规则将所有配置进行了合并和覆盖。

优先级：动态配置 > system property > 环境变量 > 配置文件

# 通过API获取配置

将provider服务的sayHello方法的应答的前缀从固定的"Hello,"改为从配置项获取：

```
private DynamicStringProperty sayHelloPrefix = DynamicPropertyFactory
    .getInstance().getStringProperty( propName: "hello.sayHelloPrefix", defaultValue: "");

@RequestMapping(path = "/hello/{name}", method = RequestMethod.GET)
public String sayHello(@PathVariable(value = "name") String name) {
    return sayHelloPrefix.getValue() + name;
}
```

在microservice.yaml文件中加上配置：

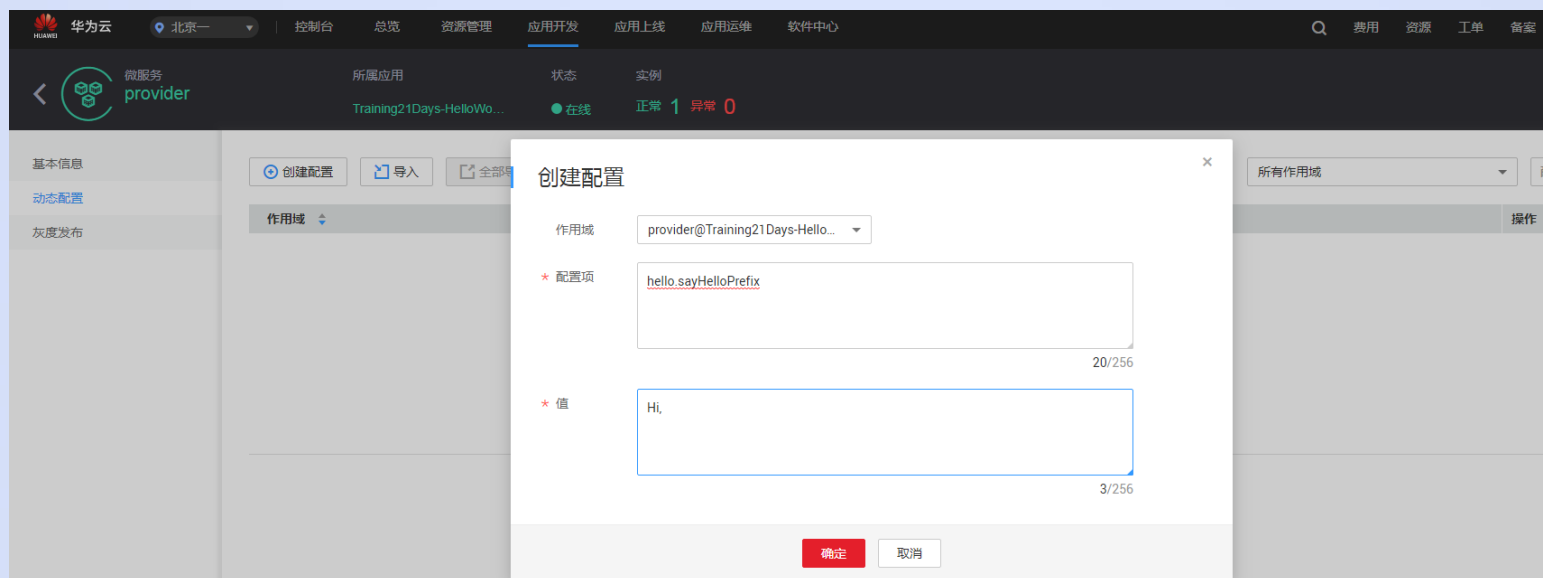
```
hello:
  sayHelloPrefix: "Hello "
```

此时启动服务进行调用，情况如下：



# 通过API获取配置

将provider服务的详情页面配置hello.sayHelloPrefix=Hi, , 再次调用服务发现应答已经产生变化：



# 日志配置

- CSEJavaSDK默认使用的日志框架是Log4j，并且给出了一份默认的配置，在org.apache.servicecomb:foundation-common包的log4j.properties文件内。
- CSEJavaSDK提供了accesslog功能，可以在传输方式为[REST over Vertx](#)的条件下使用，accesslog默认也是基于Log4j打印的，配置文件在org.apache.servicecomb:transport-rest-vertx包的log4j.properties文件内。
- 如果要覆盖默认的日志配置，在项目的resources/config目录下配置一份log4j.properties文件即可。



# Thank You





# 21天微服务实战营-Day6

华为云DevCloud & ServiceStage服务联合出品



# Day6 CSE实战之开发网关

## 大纲

- 使用DefaultEdgeDispatcher开发网关服务
- 使用URLMappedEdgeDispatcher开发网关服务

# EdgeService网关介绍

EdgeService是CSEJavaSDK提供的网关服务。网关服务可以作为微服务系统的边界，对外提供REST接口，将用户发送的RESTful请求转发给内部微服务。

EdgeService预置了默认的请求转发模块，可以根据简明的规则将请求转发到后端微服务，也允许用户开发更复杂的自定义规则转发机制。

EdgeService本身也是一个微服务，开发和配置方式与普通的微服务类似。用户可以通过扩展机制在EdgeService中定制各种业务功能。

# 使用DefaultEdgeDispatcher开发网关服务

开发一个EdgeService网关服务所需引入的maven配置与开发普通微服务基本一致，只是需要多引入一个edge-core包的依赖：

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.huawei.paas.cse</groupId>
      <artifactId>cse-dependency</artifactId>
      <version>${cse.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-solution-service-engine</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.servicecomb</groupId>
    <artifactId>edge-core</artifactId>
  </dependency>
</dependencies>
```

# 使用DefaultEdgeDispatcher开发网关服务

定义服务的main类和microservice.yaml配置文件：

- Main类的定义与普通微服务完全相同
- microservice.yaml配置文件与普通微服务基本相同，只是需要额外增加一些请求转发规则的配置

```
cse:
  http:
    dispatcher:
      edge:
        default: ..... # 使用DefaultEdgeDispatcher开发网关服务
        enabled: true ..... # 开启DefaultEdgeDispatcher
        prefix: rest ..... # 匹配请求路径前缀为/rest
        withVersion: true ..... # 请求带版本号，例如v1表示[1.0.0,2.0.0)范围内的微服务版本
        # prefixSegmentCount默认值就是1，这里只是展示一下，实际上可以省略该配置
        prefixSegmentCount: 1 ..... # 前缀长度，例如/rest/provider/v0/hello/Bob，去掉第一段，将/provider/v0/hello/Bob转发到后端
```

# 使用DefaultEdgeDispatcher开发网关服务

启动edge、provider、consumer服务，通过edge分别调用provider和consumer，调用成功：

The image displays two screenshots of a REST client interface, likely Swagger Client or Postman, showing successful GET requests.

**Top Screenshot:**

- Method: GET
- URL: 127.0.0.1:8000/rest/provider/v0/hello/Bob
- Status: 200 OK
- Time: 8 ms
- Size: 97 B
- Body: "Hello Bob"

**Bottom Screenshot:**

- Method: GET
- URL: 127.0.0.1:8000/rest/consumer/v0/hello?name=Bob
- Status: 200 OK
- Time: 12 ms
- Size: 97 B
- Body: "Hello Bob"

# 使用URLMappedEdgeDispatcher开发网关服务

除了DefaultEdgeDispatcher，CSEJavaSDK也提供了URLMappedEdgeDispatcher，允许用户指定请求URL和微服务的映射关系。

使用URLMappedEdgeDispatcher开发EdgeService的方法和使用DefaultEdgeDispatcher基本一致，只在microservice.yaml中的请求转发规则上有所不同。

```
cse:
  http:
    dispatcher:
      edge:
        url: ..... # 使用URLMappedEdgeDispatcher开发网关服务
        enabled: true ..... # 开启URLMappedEdgeDispatcher
        mappings:
          providerV0: ..... # 定义名为providerV0的映射规则
            prefixSegmentCount: 1 ..... # 前缀长度为1，例如接到url为/hello/provider/v0/hello/Bob的请求，截去第一段发给provider
            path: "/hello/.*" ..... # 映射规则匹配请求url的正则表达式
            microserviceName: provider ..... # 映射规则对应转发的后端微服务
            versionRule: 0.0.0-1.0.0 ..... # 匹配的微服务版本范围，大于或等于0.0.0，小于1.0.0
          consumerV0:
            prefixSegmentCount: 1
            path: "/client/.*"
            microserviceName: consumer
            versionRule: 0.0.0-1.0.0
```



# 使用URLMappedEdgeDispatcher开发网关服务

启动edge、provider、consumer服务，通过edge分别调用provider和consumer，调用成功：

The image displays two screenshots of a REST client interface, likely Swagger Client or Postman, showing successful GET requests.

**Top Screenshot:**

- Method: GET
- URL: 127.0.0.1:8000/hello/provider/v0/hello/Bob
- Status: 200 OK
- Time: 17057 ms
- Size: 97 B
- Body: "Hello Bob"

**Bottom Screenshot:**

- Method: GET
- URL: 127.0.0.1:8000/client/consumer/v0/hello?name=Alice
- Status: 200 OK
- Time: 3655 ms
- Size: 99 B
- Body: "Hello Alice"

# EdgeService网关转发机制

- 由前文可以看出，DefaultEdgeDispatcher的配置较为简单，但对于EdgeService接收的请求URL格式有一定的要求；URLMappedEdgeDispatcher的转发规则较为灵活，但配置相对更繁琐一点。
- 推荐用户在设计微服务的时候预先规划好各微服务的请求URL格式，可以减少后期服务演进过程中碰到的接口兼容性问题。

# Thank You





# 21天微服务实战营-Day7

华为云DevCloud & ServiceStage服务联合出品



# Day7 CSE实战之框架扩展机制

## 大纲

- Handler扩展机制
- Filter扩展机制
- 异常转换扩展机制
- 请求处理流程简介

# Handler扩展机制

Handler机制工作于用户业务代码接收REST请求之前和发送REST请求之后，支持默认/服务两个级别的配置。

多个handler之间是链式工作的，每个handler的handle方法处理完成后，由下一个handler继续处理该次请求。

```
public interface Handler {  
    /**  
     * 每次有请求经过handler链时，都会被这个方法处理一次  
     * @param invocation invocation中记录了本次请求相关的信息  
     * @param asyncResp asyncResp用于异步返回处理结果  
     */  
    void handle(Invocation invocation, AsyncResponse asyncResp) throws Exception;  
}
```

# Handler扩展机制——开发一个Handler

```
package microservice.demo.training21days.provider.handler;

import javax.ws.rs.core.Response.Status;

import org.apache.servicecomb.core.Handler;
import org.apache.servicecomb.core.Invocation;
import org.apache.servicecomb.swagger.invocation.AsyncResponse;
import org.apache.servicecomb.swagger.invocation.exception.CommonExceptionData;
import org.apache.servicecomb.swagger.invocation.exception.InvocationException;

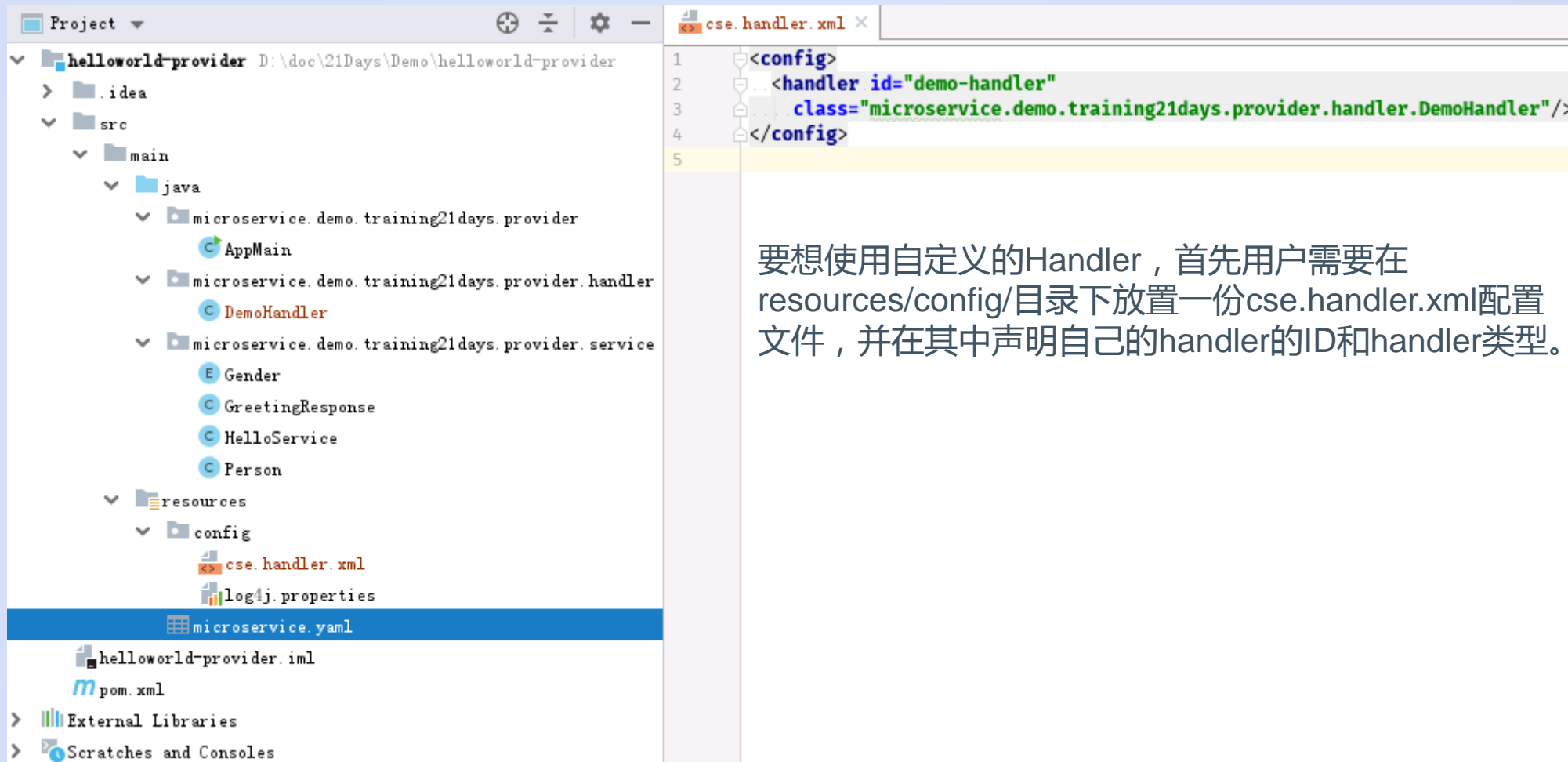
public class DemoHandler implements Handler {
    @Override
    public void handle(Invocation invocation, AsyncResponse asyncResp) throws Exception {
        // 从这里可以取出本次请求调用的方法的完整名字，格式是 serviceName.schemaId.operationId
        String operationName = invocation.getOperationMeta().getMicroserviceQualified_name();
        // 这里我们只检查 sayHello 方法的参数
        if ("provider.hello.sayHello".equals(operationName)) {
            Object name = invocation.getSwaggerArgument("idx: 0");
            // 如果 name=stranger，则拒绝请求，返回 403
            if ("stranger".equalsIgnoreCase((String) name)) {
                asyncResp.producerFail(new InvocationException(Status.FORBIDDEN, new CommonExceptionData("Don't know you :(")));
                return;
            }
        }
        // 通过检查，继续执行后面的逻辑
        invocation.next(asyncResp);
    }
}
```

这里在provider服务里开发了一个handler，该handler会检查调用sayHello方法的请求：

- 如果name是stranger，则返回403错误
- 如果是其他名字则允许调用

注意：handle方法内要么调用Invocation#next方法将请求向后传递；要么调用AsyncResponse的各种方法终止请求处理流程，返回一个应答。但是绝不能存在逻辑分支，既不向后传递请求，也不返回应答，这样会将该请求挂住，导致请求无法发往下游服务。

# Handler扩展机制——开发一个Handler



The screenshot displays an IDE interface. On the left, the 'Project' view shows a directory tree for 'helloworld-provider'. The tree includes folders like '.idea', 'src', and 'main', with sub-packages such as 'microservice.demo.training21days.provider.handler'. The 'resources' folder is expanded, showing 'config' and 'cse.handler.xml'. The 'cse.handler.xml' file is selected and its content is shown in the editor on the right. The XML content defines a handler with the ID 'demo-handler' and the class 'microservice.demo.training21days.provider.handler.DemoHandler'.

```
<config>
  <handler id="demo-handler"
    class="microservice.demo.training21days.provider.handler.DemoHandler"/>
</config>
```

要想使用自定义的Handler，首先用户需要在resources/config/目录下放置一份cse.handler.xml配置文件，并在其中声明自己的handler的ID和handler类型。



# Handler扩展机制——开发一个Handler

要在handler链中加载并使用handler，还需要在microservice.yaml配置文件中显式声明handler链的配置，将demo-handler加进去。

在一开始搭建微服务时引入的cse-solution-service-engine包内带有一份默认的handler链配置，大家可以打开这个包观察一下它提供的microservice.yaml配置文件。为了不损失原有的功能，我们复制一份默认的provider端handler链配置，并将自己的demo-handler加在handler链的开头。

```
cse:
  # 处理链配置
  handler:
    chain:
      Provider:
        default: demo-handler,qps-flowcontrol-provider,bizkeeper-provider
```

注意：consumer端handler链中有一个比较特殊，就是loadbalance，该handler在每次consumer服务准备发送请求时，从provider服务实例列表中选取一个实例作为本次调用的目标。如果consumer端handler链没有loadbalance handler，就会在调用时碰到找不到provider服务实例的问题。

# Handler扩展机制——开发一个Handler

启动provider服务，调用它的sayHello方法，如果传入的名字不是stranger，则provider返回200的响应；如果传入的名字是stranger，则provider返回403响应。

The screenshot displays two HTTP requests in a browser's developer tools. The first request is a GET to `http://127.0.0.1:8080/provider/v0/hello/bob` with a status of `200 OK` and a response body of `"Hello bob"`. The second request is a GET to `http://127.0.0.1:8080/provider/v0/hello/stranger` with a status of `403 Forbidden` and a JSON response body: `{ "message": "Don't know you :(" }`.

Method	URL	Status	Time	Size	Response Body
GET	<code>http://127.0.0.1:8080/provider/v0/hello/bob</code>	200 OK	6 ms	97 B	<code>"Hello bob"</code>
GET	<code>http://127.0.0.1:8080/provider/v0/hello/stranger</code>	403 Forbidden	138 ms	124 B	<code>{ "message": "Don't know you :(" }</code>

# Filter扩展机制

- Filter机制有两个接口，即HttpServerFilter和HttpClientFilter。
- Filter扩展机制工作于Handler扩展机制的外层，HttpServerFilter在provider端handler前工作，HttpClientFilter在consumer端handler后工作。
- Filter机制只有全局级别的生效范围。

# Filter扩展机制

HttpServerFilter中的常用方法介绍如下：

```
public interface HttpServerFilter {  
    /**  
     * HttpServerFilter的优先级  
     */  
    int getOrder();  
  
    /**  
     * 是否启用该Filter，默认启用  
     */  
    default boolean enabled() {  
        return true;  
    }  
  
    /**  
     * 微服务作为provider接到请求后，会依次调用各HttpServerFilter的afterReceiveRequest方法进行处理。  
     * 注意：如果您不了解框架的底层逻辑，建议对于invocation和requestEx两个参数只读不写，否则很容易导致意料之外的问题。  
     * @param invocation 包含了本次请求的相关信息  
     * @param requestEx 包含了一些请求的原始信息。例如，不在服务契约中声明的header是不会存放在Invocation中传递给下游的provider端  
     * Handler链的，但是在requestEx里我们可以拿到这些header信息  
     * @return 如果返回null，则该请求还会继续向下执行；否则会将该方法返回的Response作为应答发给调用方，不再执行接下来的请求处理逻辑  
     */  
    Response afterReceiveRequest(Invocation invocation, HttpServletRequestEx requestEx);  
  
    /**  
     * 在应答发送给调用方之前，依次调用各HttpServerFilter的beforeSendResponse方法进行处理。默认不处理。  
     * 注意：如果您不了解框架的底层逻辑，建议对于invocation和requestEx两个参数只读不写。  
     * @param invocation 与afterReceiveRequest方法接收到的invocation相同  
     * @param responseEx 即将发送给调用方的应答。responseEx内可以自定义一些返回的header信息，但建议不要修改body。  
     */  
    default void beforeSendResponse(Invocation invocation, HttpServletResponseEx responseEx) {  
    }  
}
```

# Filter扩展机制

HttpClientFilter中的常用方法介绍如下：

```
public interface HttpClientFilter {  
    /**  
     * 是否启用该Filter，默认启用  
     */  
    default boolean enabled() {  
        return true;  
    }  
  
    /**  
     * HttpClientFilter的优先级  
     */  
    int getOrder();  
  
    /**  
     * 微服务作为consumer发送请求时，会依次调用各HttpClientFilter的beforeSendRequest方法进行处理。  
     * @param invocation 包含了本次请求的相关信息，包括服务契约相关的信息  
     * @param requestEx 本次请求相关的参数信息，可以在这里自定义一些header信息，但建议不要修改body  
     */  
    void beforeSendRequest(Invocation invocation, HttpServletResponse requestEx);  
  
    /**  
     * 接收到服务端的应答后，会依次调用各HttpClientFilter的afterReceiveResponse方法进行处理。  
     * @param invocation 与afterReceiveRequest方法接收到的invocation相同  
     * @param responseEx 这里包含了一些应答的原始数据信息，如header等  
     * @return 如果返回null则继续调用下一个HttpClientFilter处理，否则将该方法返回的Response作为应答返回给业务逻辑  
     */  
    Response afterReceiveResponse(Invocation invocation, HttpServletResponse responseEx);  
}
```

# Filter扩展机制——开发一个HttpServerFilter

现在我们在edge服务中定义一个HttpServerFilter：

```
public class DemoFilter implements HttpServerFilter {  
  
    private static final String LET_STRANGER_PASS = "LetStrangerPass";  
  
    @Override  
    public int getOrder() {  
        return 0;  
    }  
  
    @Override  
    public Response afterReceiveRequest(Invocation invocation, HttpServletRequestEx httpServletRequestEx) {  
        // 从请求中取出一个header  
        String letStrangerPass = httpServletRequestEx.getHeader(LET_STRANGER_PASS);  
        if (!StringUtils.isEmpty(letStrangerPass)) {  
            // 如果此header存在则将它存入到InvocationContext中，InvocationContext可以从上游服务自动传递给下游服务  
            invocation.addContext(LET_STRANGER_PASS, letStrangerPass);  
        }  
        return null;  
    }  
}
```

这里的Filter的目的是从请求中获取名为LetStrangerPass的header，将其存入InvocationContext中向下游的provider服务传递。

当provider服务的DemoHandler从InvocationContext中取出LetStrangerPass并且其值为true时，就允许stranger请求访问sayHello方法。

# Filter扩展机制——开发一个HttpServerFilter

为了edge服务能够加载该filter，我们还需要定义一份SPI配置文件：



SPI机制 ( Service Provider Interface ) 是一种JDK内置的服务提供发现机制。  
开发者扩展了哪个接口，那么对应的SPI配置文件的名字必须是与该接口相同的，文件的内容是该接口的实现类的名字。

# Filter扩展机制——开发一个HttpServerFilter

我们修改一下provider服务中的DemoHandler的逻辑，如果能从InvocationContext中取出LetStrangerPass，并且其值为true，则允许stranger访问sayHello方法：

```
public class DemoHandler implements Handler {
    @Override
    public void handle(Invocation invocation, AsyncResponse asyncResp) throws Exception {
        // 从这里可以取出本次请求调用的方法的完整名字，格式是 serviceName.schemaId.operationId
        String operationName = invocation.getOperationMeta().getMicroserviceQualified_name();
        // 这里我们只检查 sayHello 方法的参数
        if ("provider.hello.sayHello".equals(operationName)) {
            Object name = invocation.getSwaggerArgument( idx: 0);
            // 如果 name=stranger，则拒绝请求，返回403
            if (!"true".equalsIgnoreCase(invocation.getContext( key: "LetStrangerPass")))
                && "stranger".equalsIgnoreCase((String) name)) {
                asyncResp.producerFail(new InvocationException(Status.FORBIDDEN, new CommonExceptionData("Don't know you :(")));
                return;
            }
        }
        // 通过检查，继续执行后面的逻辑
        invocation.next(asyncResp);
    }
}
```



# Filter扩展机制——开发一个HttpServerFilter

The image displays two screenshots of the Edge browser's developer tools, specifically the Network tab, illustrating the effect of the `LetStrangerPass` header.

**Top Screenshot (Failed Request):**

- Method: GET
- URL: `http://localhost:8000/rest/provider/v0/hello/stranger`
- Status: 403 Forbidden
- Time: 6 ms
- Size: 124 B
- Body (JSON): 

```
{  "message": "Don't know you :( "
```

**Bottom Screenshot (Successful Request):**

- Method: GET
- URL: `http://localhost:8000/rest/provider/v0/hello/stranger`
- Status: 200 OK
- Time: 6 ms
- Size: 102 B
- Body (Text): `"Hello stranger"`
- Headers (1): 

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> LetStrangerPass	true	
Key	Value	Description

通过edge调用provider时，如果设置header `LetStrangerPass=true`，则使用`name=stranger`的参数也可以调用`sayHello`方法。

# 异常转换扩展机制

如果用户的业务代码抛出了InvocationException异常，则框架会将InvocationException中的data直接序列化为响应消息的body。如果是其他的异常，则返回的响应消息状态码为490/590，响应body为CommonExceptionData。我们也提供了异常转换扩展机制，允许用户捕获不同类型的异常后，将其转换为响应消息：

- 该机制允许按优先级排列和选取异常转换器
- 可以定义特定类型的异常转换器，转换该类型及其子类的异常

# 异常转换扩展机制——开发一个异常转换器

我们来定义一个IllegalArgumentException的转换器作为例子。

首先我们在provider服务的greeting方法中增加一个检查，如果person参数的属性缺失，则抛出一个IllegalArgumentException。

```
@PostMapping(path = "/greeting")
public GreetingResponse greeting(@RequestBody Person person) {
    if (StringUtils.isEmpty(person.getName()) || null == person.getGender()) {
        throw new IllegalArgumentException("Lack of property");
    }
    GreetingResponse greetingResponse = new GreetingResponse();

    if (Gender.MALE.equals(person.getGender())) {
        greetingResponse.setMsg("Hello, Mr." + person.getName());
    } else {
        greetingResponse.setMsg("Hello, Ms." + person.getName());
    }
    greetingResponse.setTimestamp(new Date());

    return greetingResponse;
}
```

# 异常转换扩展机制——开发一个异常转换器

调用provider服务的greeting方法，不传name字段，可以看到provider服务返回的状态码是590，错误信息是Cse Internal Server Error，此时请求错误的信息没有展示出来。

The screenshot displays a REST client interface with the following details:

- Request:**
  - Method: POST
  - URL: 127.0.0.1:8080/provider/v0/greeting
  - Body: 

```
{
  "gender": "MALE"
}
```
- Response:**
  - Status: 590 Cse Internal Server Error
  - Time: 693 ms
  - Size: 148 B
  - Body: 

```
{
  "message": "Cse Internal Server Error"
}
```

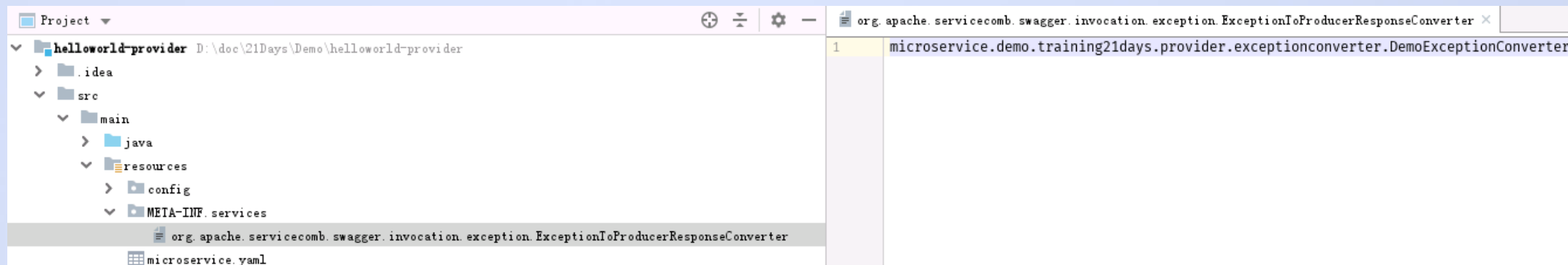
# 异常转换扩展机制——开发一个异常转换器

现在我们来定义一个针对IllegalArgumentExpection的转换器

```
public class DemoExceptionConverter implements ExceptionToProducerResponseConverter<IllegalArgumentException> {  
    /**  
     * 该方法的返回值表明DemoExceptionConverter处理IllegalArgumentException及其子类的异常。  
     */  
    @Override  
    public Class<IllegalArgumentException> getExceptionClass() {  
        return IllegalArgumentException.class;  
    }  
  
    /**  
     * 当业务代码抛出的IllegalArgumentException被捕获后，会传入该方法进行处理。  
     * @param swaggerInvocation 本次业务调用相关的信息  
     * @param e 被捕获的异常  
     * @return 转换后的响应消息，将会发送给调用方  
     */  
    @Override  
    public Response convert(SwaggerInvocation swaggerInvocation, IllegalArgumentException e) {  
        return Response.consumerFailResp(  
            new InvocationException(Status.BAD_REQUEST,  
                new CommonExceptionData(  
                    swaggerInvocation.getInvocationQualifiedName() + " gets illegal param: " + e.getMessage()))  
        );  
    }  
}
```

# 异常转换扩展机制——开发一个异常转换器

ExceptionToProducerResponseConverter的扩展类也是通过SPI机制加载的，因此需要定义一份SPI配置文件



# 异常转换扩展机制——开发一个异常转换器

再次调用provider服务的greeting方法，此时响应消息是转换后的错误信息

POST 127.0.0.1:8080/provider/v0/greeting

Send Save

Params Authorization Headers (1) Body Pre-request Script Tests Cookies Code Comments (0)

none form-data x-www-form-urlencoded raw binary JSON (application/json) Beautify

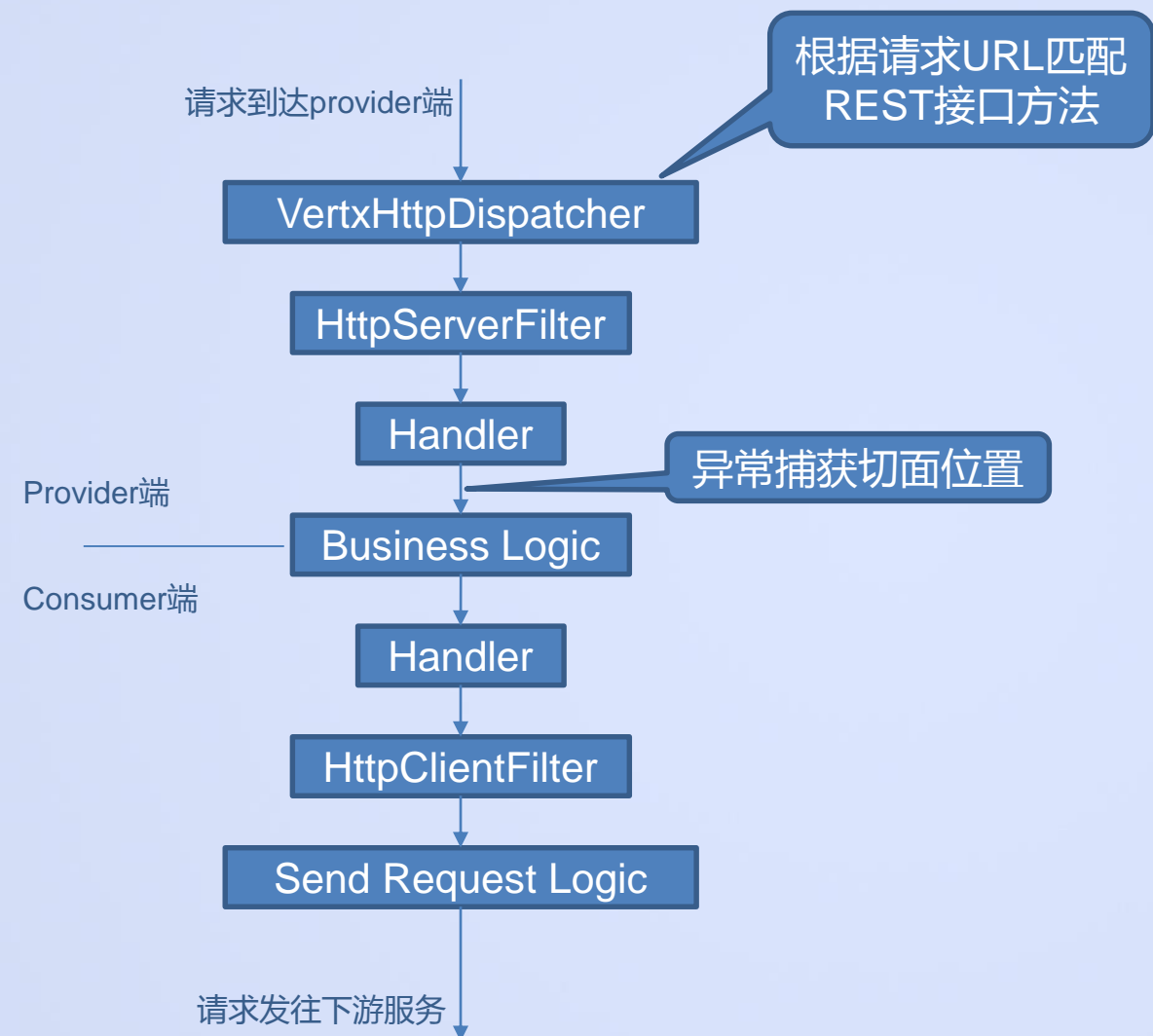
```
1 {  
2  
3   "gender": "MALE"  
4 }
```

Body Cookies Headers (2) Test Results Status: 400 Bad Request Time: 4 ms Size: 183 B Download

Pretty Raw Preview JSON

```
1 {  
2   "message": "PRODUCER rest provider.hello.greeting gets illegal param: Lack of property"  
3 }
```

# 请求处理流程简介



一个请求发送到某微服务，触发服务执行业务逻辑，调用下游服务方法的总体流程如图所示。

- Filter机制只有全局作用范围，Handler机制有全局和服务级作用范围
- 注意异常转换扩展捕获异常的位置，如果异常不是由业务逻辑抛出，而是由Handler等抛出的，则不在它的处理范围内
- Handler由handler.xml定义加载，Filter和异常转换机制由SPI机制加载，这些机制的实现类都不是由Spring Bean机制加载和管理的，因此@Autowired等Bean自动注入功能无法在这些扩展类里使用
- 如果要在这些扩展类里获取Spring Bean，可以考虑使用BeanUtils#getBean方法，但要注意获取时机不能太早，否则可能对应的Spring Bean还没有被Spring框架实例化
- EdgeService网关服务只有consumer端handler，没有provider端handler



# Thank You





# 21天微服务实战营-Day8

华为云DevCloud & ServiceStage服务联合出品



# Day8 CSE实战之负载均衡

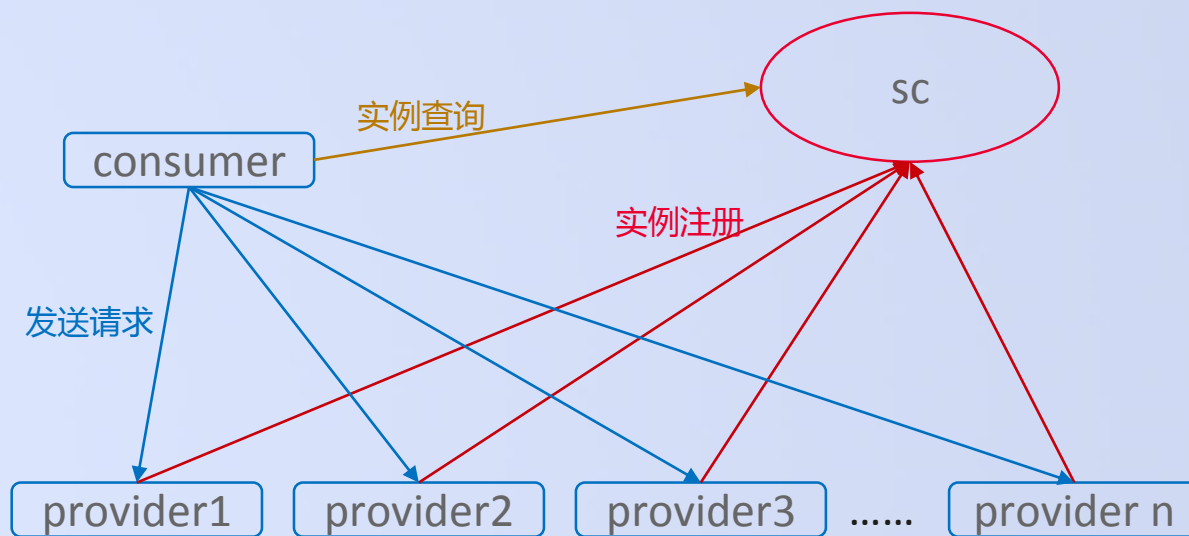
## 大纲

- 负载均衡策略
- 请求重试机制
- 实例隔离机制

# 负载均衡策略

CSEJavaSDK的负载均衡机制是客户端负载均衡：

- 微服务实例启动时会将自己的实例信息（包括IP、端口号等）注册到sc，并且通过心跳机制维持本实例的在线状态。
- consumer定时去服务中心查询provider的实例（第一次查询发生在consumer第一次调用provider的时候，之后默认30秒去sc查一次），并缓存在本地。
- consumer调用provider时会通过负载均衡机制从缓存的provider实例列表中选取一个作为本次请求发送的地址。
- 内置的负载均衡策略有RoundRobin、Random、WeightedResponse、SessionStickiness，其中默认使用的是RoundRobin。



# 负载均衡策略

启动一个consumer实例，三个provider实例，调用consumer的greeting方法，可以看到consumer默认是使用RoundRobin策略调用三个provider的。连续调用consumer6次，三个provider的日志记录分别是：

```
2019-02-25 11:38:30,347 [INFO] 192.168.0.45 - - Mon, 25 Feb 2019 11:38:30 CST /provider/v0/greeting 200 65 8 5c73633634a31c51
2019-02-25 11:38:33,615 [INFO] 192.168.0.45 - - Mon, 25 Feb 2019 11:38:33 CST /provider/v0/greeting 200 65 3 5c73633902d84f8a
```

实例1

```
2019-02-25 11:38:31,495 [INFO] 192.168.0.45 - - Mon, 25 Feb 2019 11:38:31 CST /provider/v0/greeting 200 65 3 5c7363376ed08941
2019-02-25 11:38:34,674 [INFO] 192.168.0.45 - - Mon, 25 Feb 2019 11:38:34 CST /provider/v0/greeting 200 65 3 5c73633a652f94c0
```

实例2

```
2019-02-25 11:38:32,625 [INFO] 192.168.0.45 - - Mon, 25 Feb 2019 11:38:32 CST /provider/v0/greeting 200 65 2 5c736338a0353b2e
2019-02-25 11:38:35,697 [INFO] 192.168.0.45 - - Mon, 25 Feb 2019 11:38:35 CST /provider/v0/greeting 200 65 3 5c73633b85f29c50
```

实例3

从时间顺序上可以看出，consumer服务的6次请求是由三个provider实例轮流处理的。

# 负载均衡策略

The screenshot shows the Huawei Cloud ServiceStage console interface. The top navigation bar includes '应用开发' (Application Development), '应用上线' (Application Deployment), '应用运维' (Application Operations), and '软件中心' (Software Center). The left sidebar contains '应用开发' (Application Development), '微服务开发' (Microservice Development), '无服务器应用开发' (Serverless Application Development), '微服务评估' (Microservice Evaluation), '帮助中心' (Help Center), and '体验中心' (Experience Center). The main content area displays the '引擎列表' (Engine List) page, which includes a table of engines and a '控制台' (Control Console) button. The table has columns for '名称' (Name), '状态' (Status), '版本' (Version), '已用/套餐总实例数' (Used/Total Instance Count), '服务中心连接地址' (Service Center Connection Address), '创建时间' (Creation Time), and '操作' (Operations). The table lists one engine: 'Cloud Service Engine' with status '可用' (Available), version '专业版' (Professional Edition), and 4/100 instances. The '操作' column has a '控制台' (Control Console) button. Below the table, the '服务治理' (Service Governance) page is shown, featuring a 'Training21Days-Hello...' dropdown menu and a '在线' (Online) status indicator. The '服务治理' page also displays a '仪表盘' (Dashboard) with '微服务数量 2' (Microservice Count 2) and a '服务目录' (Service Directory) section showing 'consumer#0.0.1' and 'provider#0.0.1'.

应用开发

微服务引擎（Cloud Service Engine）是企业级微服务应用管理平台，包含微服务SDK、服务中心、配置中心、治理中心，帮助您实现微服务应用的快速构建、实时监控和高可用。且CSE兼容主流开源生态，不绑定特定开发框架和平台，支持已有应用业务代码零修改接入。可免费使用10个实例，超出部分按需抵扣。

购买微服务引擎专享版 创建免费引擎（0个）

引擎列表 NEW!

您还可以购买 5 个微服务引擎专享版。

所有状态(1) 名称

名称	状态	版本	已用/套餐总实例数	服务中心连接地址	创建时间	操作
Cloud Service Engine	可用	专业版	4/100 (4%)	https://cse.cn-north-1.myhu...		控制台 更多

服务治理

您可以在微服务部署完成后，根据微服务的运行情况对服务进行治理。 [如何治理微服务？](#)

Training21Days-Hello...

Training21Days- 在线

微服务数量 2

离线 0 正常 2 异常 0

1 实例数 consumer#0.0.1

3 实例数 provider#0.0.1

登陆ServiceStage页面，依次点击“应用开发”->“引擎列表”->“控制台”->“服务治理”，在应用选择下拉框里选择“Training21Days-HelloWorld”，进入该应用的治理页面

# 负载均衡策略

consumer#0.0.1 Training2...

监控

阈值

0 | 0 | 0%

0 | 0 | 0%

0 | 0 | 0%

吞吐量 0/s

熔断状态 Closed

实例数 1 90th 0ms

中位数时延 0ms 99th 0ms

平均时延 0ms 99.5th 0ms

负载均衡(0) 限流(0) 降级(0) 容错(0) 熔断(0)

错误注入(0) 黑白名单(0)

+ 新增

选择微服务 所有微服务

负载均衡策略 随机

确定 取消



```
graph TD; C((1 consumer#0.0.1)) --> P((3 provider#0.0.1));
```

在服务治理页面的右边点击选择consumer服务，将其负载均衡策略修改为“随机”

重新调用consumer的greeting方法，可以看到三个provider实例接收greeting请求的分布情况变为随机的了

# 请求重试机制

当遭遇网络连接超时、实例下线等问题时，consumer服务默认会尝试选择下一个provider服务实例进行调用，来尽量保证业务调用的成功。

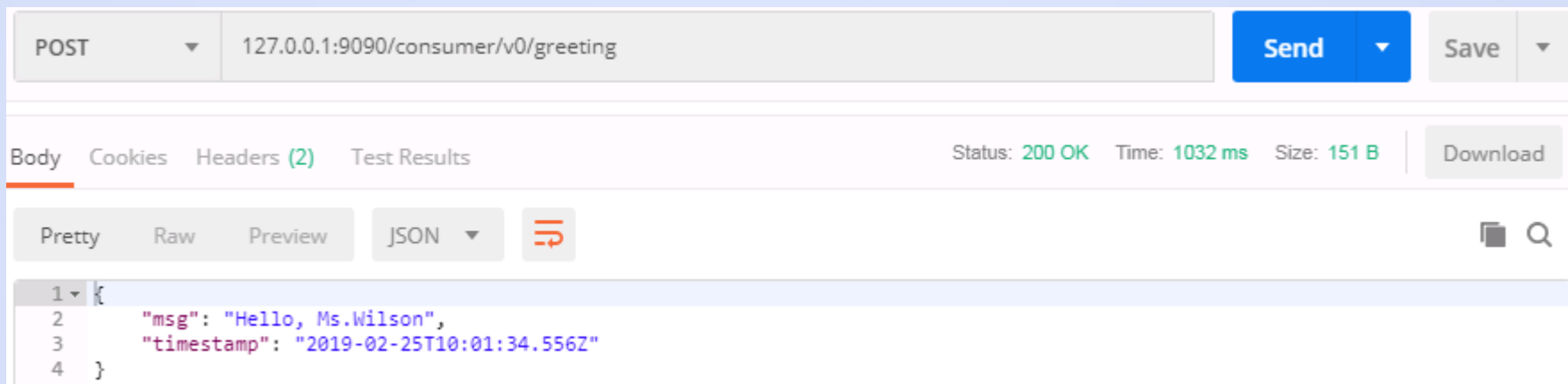
- 默认的重试机制配置是retryOnSame=0,retryOnNext=1，即在同一个provider实例上重试零次，选择下一个provider实例重试一次
- 并不是所有的错误都会让consumer端进行重试，具体判断逻辑请参考ServiceComb-Java-Chassis中的DefaultRetryExtensionsFactory类
- 默认的重试机制配置在cse-solution-service-engine包的microservice.yaml文件中



# 请求重试机制

我们可以模拟一下实例意外停止的场景来测试重试机制。

强制停止一个provider实例（注意不要是正常退出，否则触发实例优雅停机的话我们的可用时间窗口太短），然后调用consumer的greeting方法：



可以看到，虽然有的请求花费的时间长一点，但是仍然会调用成功的。如果观察consumer服务的日志，可以发现有时候consumer会选取那个被强制关闭的provider实例，但是在抛出 `java.net.ConnectException: Connection refused` 异常后，consumer会向另一个provider实例发送请求并调用成功。

# 请求重试机制

在治理页面上我们可以动态地调整重试机制的配置，治理页面的重试机制名为容错。大家可以在页面上选择一下容错策略为Failover，前往consumer服务的详情页面查看配置项


The screenshot displays the 'consumer#0.0.1' configuration page in a service governance console. A red box highlights the breadcrumb 'consumer#0.0.1' at the top left, with a red arrow pointing to it and the text '点击这里进入consumer服务详情页' (Click here to enter the consumer service detail page). The page is divided into several sections:

- Summary:** Shows status indicators (0 green, 0 yellow, 0 red) and metrics like '吞吐量' (Throughput) and '熔断状态' (Circuit Breaker Status) as NaN.
- Table:** A table with 4 columns showing instance counts and latency percentiles (90th, 99th, 99.5th) for 'consumer#0.0.1', with all values being NaN.
- Configuration Tabs:** Includes tabs for '负载均衡(0)', '限流(0)', '降级(0)', '容错(0)' (which is selected), and '熔断(0)'. Below these are '错误注入(0)' and '黑白名单(0)'.
- Form:** A configuration form with a '+ 新增' (Add) button. It includes:
  - '容错对象' (Fault Tolerance Object) set to '所有微服务' (All Microservices).
  - '是否开启容错' (Whether to enable fault tolerance) with '开启' (Enable) selected.
  - '容错策略' (Fault Tolerance Strategy) with 'Failover' selected, and other options 'Failfast', 'Failback', and 'custom'.
  - '确定' (Confirm) and '取消' (Cancel) buttons.
- Diagram:** A service dependency diagram on the right showing 'consumer#0.0.1' (node 1) depending on 'provider#0.0.1' (node 3).

# 请求重试机制

从配置项页面可以看到，Failover策略就是默认的tryOnSame=0，tryOnNext=1策略。

<

 微服务 consumer

所属应用

Training21Days-HelloWo...

状态

● 在线

实例

正常 1 异常 0

基本信息

动态配置

灰度发布

创建配置

导入

全部导出

所有作用域

配置项或值

作用域	配置项	值	操作
consumer@Training21Days-HelloWorld#0.0.1	cse.loadbalance.retryOnNext	1	<a href="#">编辑</a> <a href="#">删除</a>
consumer@Training21Days-HelloWorld#0.0.1	cse.loadbalance.retryOnSame	0	<a href="#">编辑</a> <a href="#">删除</a>
consumer@Training21Days-HelloWorld#0.0.1	cse.loadbalance.retryEnabled	true	<a href="#">编辑</a> <a href="#">删除</a>

Tips: 大家可以在页面上尝试一下其他的几种重试策略。CSE的大部分治理策略都是通过配置项管理的，在治理页面上进行治理操作后，可以到这里查看对应的配置项。

# 实例隔离机制

如果consumer在调用某个provider实例时频繁报错，则consumer会将该provider实例隔离，等待一段时间后再尝试向其发送请求。

- 默认隔离规则是连续5次调用某实例失败后隔离该实例
- 默认隔离60秒后将被隔离实例放回可选provider实例列表中，但当次请求是否被路由到该实例仍取决于路由策略的选择
- 如果尝试对被隔离实例进行调用时失败了，则该实例立即重新进入隔离状态，等待下一个60秒重试机会；调用成功则从隔离状态恢复
- 并非所有的调用失败都会计入实例隔离的判断机制里

# 实例隔离机制

我们仍然启动一个consumer实例、三个provider实例，先调用consumer的方法，测试consumer会正常地将请求路由到三个provider实例上去。

接着强制关闭一个provider实例，连续调用consumer服务，可以看到consumer在调用强制关闭的provider实例几次之后，将这个实例隔离并且不再调用了。

```
Caused by: java.net.ConnectException: Connection refused: no further information
->... 11 more
2019-02-26 11:52:20,410 [WARN] bizkeeper command CONSUMER rest provider.hello.greeting failed due to InvocationException: code=490;msg=CommonExceptionData [message=Cse Internal Bad Request] org.apache.servicecomb.bizkeeper.BizkeeperCommand.
2019-02-26 11:52:20,410 [WARN] bizkeeper execution error, info=cause:InvocationException,message:InvocationException: code=490;msg=CommonExceptionData [message=Cse Internal Bad Request];cause:AnnotatedConnectException,message:Connection refu
2019-02-26 11:52:20,410 [ERROR] service CONSUMER rest provider.hello.greeting, call error, msg is cause:InvocationException,message:InvocationException: code=490;msg=CommonExceptionData [message=Cse Internal Bad Request];cause:CseException,r
You can add fallback logic by catching this exception.
info: operation=provider.hello.greeting.;cause:InvocationException,message:InvocationException: code=490;msg=CommonExceptionData [message=Cse Internal Bad Request];cause:AnnotatedConnectException,message:Connection refused: no further info
2019-02-26 11:52:20,410 [ERROR] Invoke server failed. Operation CONSUMER rest provider.hello.greeting; server rest://192.168.0.45:8080; 0-0 msg cause:InvocationException,message:InvocationException: code=490;msg=CommonExceptionData [message:
You can add fallback logic by catching this exception.
info: operation=provider.hello.greeting.;cause:InvocationException,message:InvocationException: code=490;msg=CommonExceptionData [message=Cse Internal Bad Request];cause:AnnotatedConnectException,message:Connection refused: no further info
2019-02-26 11:52:20,415 [ERROR] Invoke server success. Operation CONSUMER rest provider.hello.greeting; server rest://192.168.0.45:8081 org.apache.servicecomb.loadbalance.LoadbalanceHandler$3.onExecutionSuccess(LoadbalanceHandler.java:306)
2019-02-26 11:52:20,416 [INFO] 127.0.0.1 - - Tue, 26 Feb 2019 11:52:19 CST /consumer/v0/greeting 200 65 1011 5c74b7f3a20b5de3 org.apache.servicecomb.transport.rest.vertx.accesslog.impl.AccessLogHandler.lambda$handle$0(AccessLogHandler.java:4
2019-02-26 11:52:21,251 [WARN] Isolate service provider's instance bda7e921396311e99b640255ac105554. org.apache.servicecomb.loadbalance.filter.IsolationDiscoveryFilter.allowVisit(IsolationDiscoveryFilter.java:142)
2019-02-26 11:52:21,257 [INFO] 127.0.0.1 - - Tue, 26 Feb 2019 11:52:21 CST /consumer/v0/greeting 200 65 7 5c74b7f532df3540 org.apache.servicecomb.transport.rest.vertx.accesslog.impl.AccessLogHandler.lambda$handle$0(AccessLogHandler.java:42)
2019-02-26 11:52:22,401 [INFO] 127.0.0.1 - - Tue, 26 Feb 2019 11:52:22 CST /consumer/v0/greeting 200 65 6 5c74b7f6803019a3 org.apache.servicecomb.transport.rest.vertx.accesslog.impl.AccessLogHandler.lambda$handle$0(AccessLogHandler.java:42)
2019-02-26 11:52:23,426 [INFO] 127.0.0.1 - - Tue, 26 Feb 2019 11:52:23 CST /consumer/v0/greeting 200 65 6 5c74b7f767922a84 org.apache.servicecomb.transport.rest.vertx.accesslog.impl.AccessLogHandler.lambda$handle$0(AccessLogHandler.java:42)
2019-02-26 11:52:24,250 [INFO] 127.0.0.1 - - Tue, 26 Feb 2019 11:52:24 CST /consumer/v0/greeting 200 65 6 5c74b7f88988d822 org.apache.servicecomb.transport.rest.vertx.accesslog.impl.AccessLogHandler.lambda$handle$0(AccessLogHandler.java:42)
2019-02-26 11:52:25,245 [INFO] 127.0.0.1 - - Tue, 26 Feb 2019 11:52:25 CST /consumer/v0/greeting 200 65 7 5c74b7f972e44db0 org.apache.servicecomb.transport.rest.vertx.accesslog.impl.AccessLogHandler.lambda$handle$0(AccessLogHandler.java:42)
2019-02-26 11:52:26,132 [INFO] 127.0.0.1 - - Tue, 26 Feb 2019 11:52:26 CST /consumer/v0/greeting 200 65 6 5c74b7fa27645f90 org.apache.servicecomb.transport.rest.vertx.accesslog.impl.AccessLogHandler.lambda$handle$0(AccessLogHandler.java:42)
```

实例隔离触发机制除了连续调用失败，还有错误率阈值。但在某些问题场景下，出错实例会积累很高的错误率。此时如果要想每隔60秒的尝试调用机制将错误率慢慢拉低到阈值以下，来解除实例的隔离状态，会花费相当长的时间。因此我们推荐大家使用默认连续调用失败机制判断实例隔离，方便问题实例的快速隔离和正常实例的快速恢复。

# 小结

- CSEJavaSDK的负载均衡策略是客户端负载均衡，默认策略是轮询
- 重试策略可以在provider端实例意外退出、网络断连时保证consumer端业务调用不出错。默认的重试策略是tryOnNext=1,tryOnSame=0
- 实例隔离机制可以快速将问题实例从consumer端的实例缓存列表中排除，减小业务调用受问题实例影响的概率
- 今天的课程所讲的内容都在ServiceComb开源文档中有说明，推荐阅读：  
[https://docs.servicecomb.io/java-chassis/zh\\_CN/references-handlers/loadbalance.html](https://docs.servicecomb.io/java-chassis/zh_CN/references-handlers/loadbalance.html)

# Thank You







# 21天微服务实战营-Day9

华为云DevCloud & ServiceStage服务联合出品





# Day9 CSE实战之服务治理

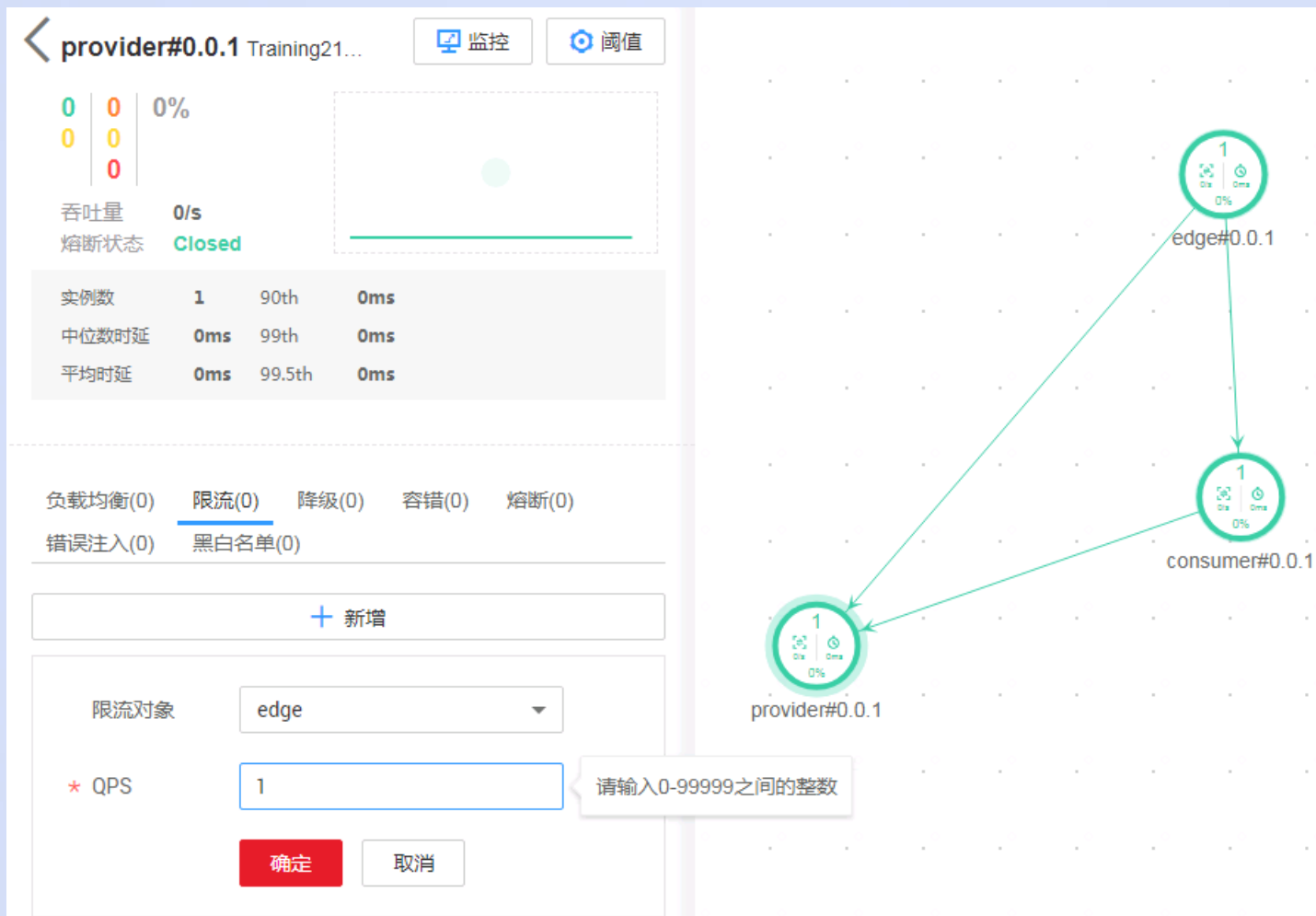
## 大纲

- 限流策略
- 服务熔断
- 服务降级
- 灰度发布

# 限流策略

- CSEJavaSDK支持客户端和服务端的限流策略，限制每秒钟最大请求数
- 支持default、微服务、schema（ 契约 ）、operation（ 接口 ）限流四个粒度的限流
- 治理页面支持的是服务端微服务级限流配置
- 限流策略是实例级限流，例如，配置provider服务的服务端流控为1000QPS，如果有两个provider服务实例，则他们可以接收共计2000QPS的流量

# 限流策略

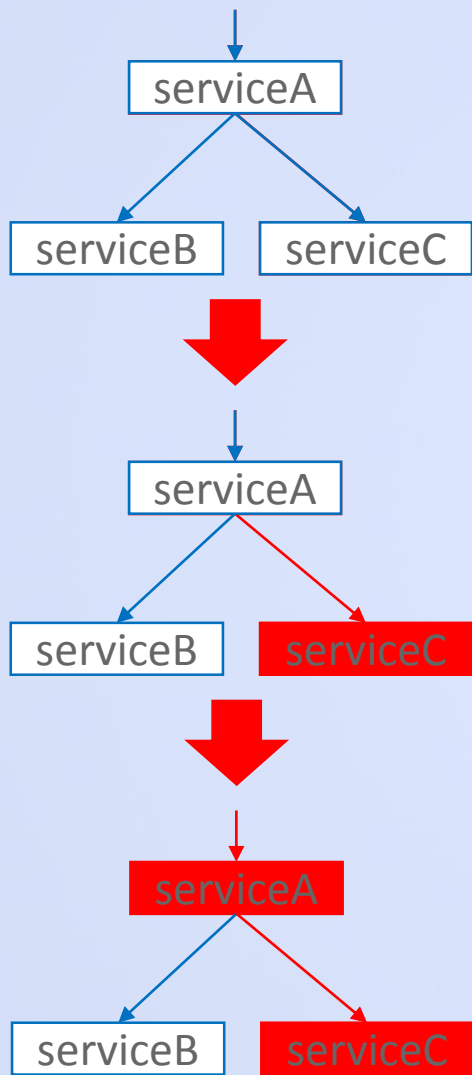


在provider服务上配置限流策略为对edge服务限流至1QPS，通过edge服务连续调用consumer和provider服务。

观察provider服务中打印的日志，可以看到每秒钟provider只处理一次来自edge服务的请求，其他请求都以429状态码返回的。而对于来自consumer的请求，provider正常处理并返回200状态码。

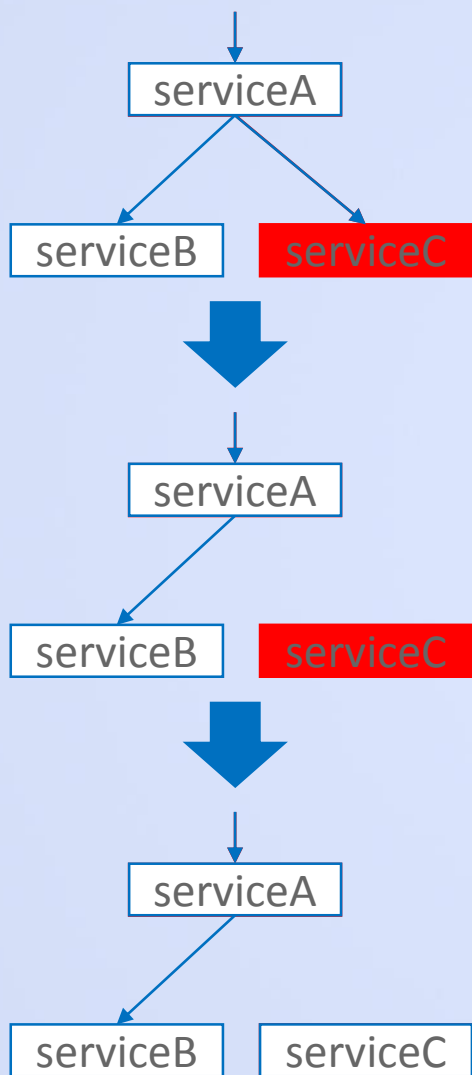
```
09:44:37 CST /provider/v0/hello/Alice 200 13 2 5c75eb8566ff9bf7
09:44:38 CST /provider/v0/hello/Bob 200 11 1 5c75eb86c817a894 01
09:44:38 CST /provider/v0/hello/Alice 200 13 1 5c75eb866c170711
09:44:38 CST /provider/v0/hello/Bob 429 41 1 5c75eb86281fa768 01
09:44:38 CST /provider/v0/hello/Alice 200 13 1 5c75eb868440acd4
09:44:38 CST /provider/v0/hello/Bob 429 41 1 5c75eb860ea48971 01
09:44:38 CST /provider/v0/hello/Alice 200 13 2 5c75eb867246b63d
09:44:38 CST /provider/v0/hello/Bob 429 41 0 5c75eb86255404de 01
09:44:38 CST /provider/v0/hello/Alice 200 13 2 5c75eb86f17d2d98
09:44:38 CST /provider/v0/hello/Bob 429 41 0 5c75eb86f6becc4a4 01
09:44:38 CST /provider/v0/hello/Alice 200 13 1 5c75eb86a680379a
09:44:38 CST /provider/v0/hello/Bob 429 41 1 5c75eb8686971aac 01
09:44:39 CST /provider/v0/hello/Alice 200 13 1 5c75eb8762f30c07
09:44:39 CST /provider/v0/hello/Bob 200 11 1 5c75eb87567287cd 01
09:44:39 CST /provider/v0/hello/Alice 200 13 1 5c75eb877cd27728
09:44:39 CST /provider/v0/hello/Bob 429 41 1 5c75eb8715293b46 01
09:44:39 CST /provider/v0/hello/Alice 200 13 1 5c75eb87d02ef8c3
09:44:39 CST /provider/v0/hello/Bob 429 41 1 5c75eb87a133c166 01
```

# 服务熔断

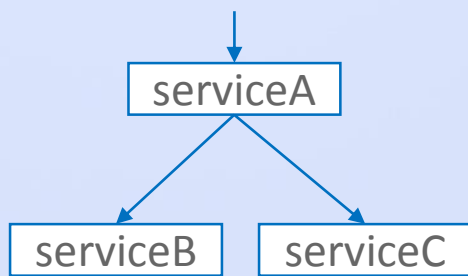


- 对于一个分布式系统，如果某个请求的调用链中的某个服务出现故障，响应变慢，会导致整个链路的响应变慢，请求堆积。
- 当这种情况变得越来越严重的时候，占用的资源会越来越多，到达系统瓶颈，造成整个系统崩溃，所有请求都不可用。

# 服务熔断



- 熔断可以将问题服务隔离开，令请求可以快速返回
  - 待问题服务变为正常状态后，再从熔断状态中恢复过来
- 通过这种机制，我们可以临时断开次要业务路径，保障系统整体的可用性。



# 服务熔断——手动熔断

edge#0.0.1 Training21Days-...

监控

阈值

0 | 0 | 0%

0 | 0 | 0%

0 | 0 | 0%

吞吐量 0/s

熔断状态 Closed

实例数	1	90th	0ms
中位数时延	0ms	99th	0ms
平均时延	0ms	99.5th	0ms

负载均衡(0)

限流(0)

降级(0)

容错(0)

熔断(0)

错误注入(0)

黑白名单(0)

+ 新增

熔断对象

consumer

helloConsumer.sayHello

触发条件

☒ 手动熔断

☐ 取消熔断

☐ 自动熔断

确定

取消



熔断可以手动开启，也可以自动开启。其触发方式不同，但效果相同。

这里我们选择edge服务，手工熔断其调用consumer服务的sayHello方法的路径。

# 服务熔断——手动熔断

edge#0.0.1 Training21Days-...

监控

阈值

0 | 0 | 0%

0 | 0 | 0%

0 | 0 | 0%

吞吐量 0/s

熔断状态 Closed

实例数	1	90th	0ms
中位数时延	0ms	99th	0ms
平均时延	0ms	99.5th	0ms

负载均衡(0)

限流(0)

降级(0)

容错(0)

熔断(0)

错误注入(0)

黑白名单(0)

+ 新增

熔断对象

consumer

helloConsumer.sayHello

触发条件

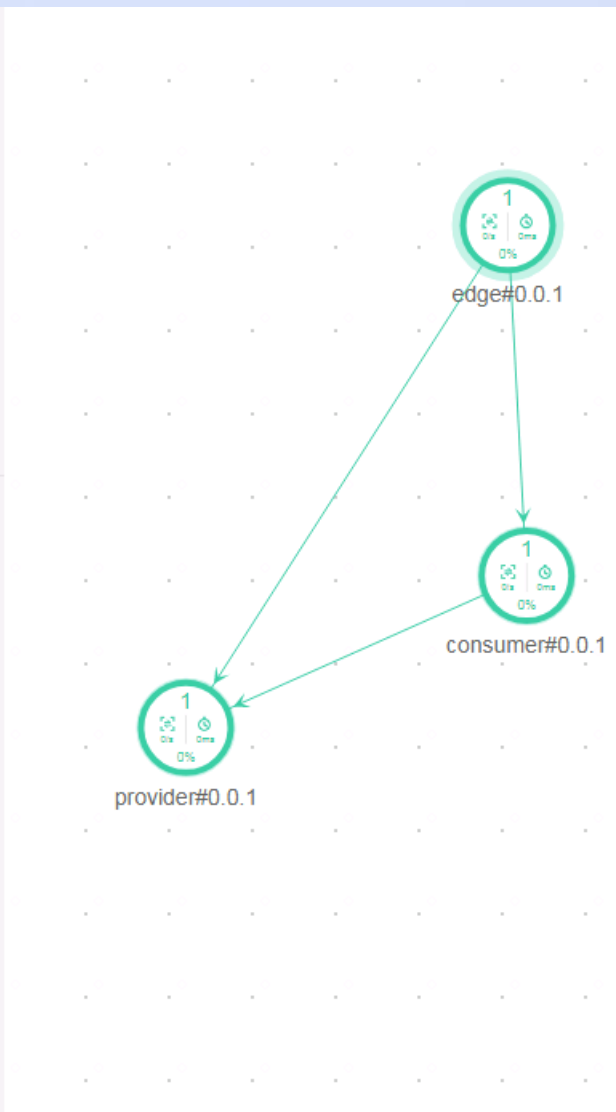
☒ 手动熔断

☐ 取消熔断

☐ 自动熔断

确定

取消



熔断可以手动开启，也可以自动开启。其触发方式不同，但效果相同。

这里我们选择edge服务，手工熔断edge服务调用consumer服务sayHello方法的路径。

# 服务熔断——手动熔断

通过edge服务调用consumer服务的sayHello方法和greeting方法，可以看到sayHello方法已经调不通了，但greeting方法仍然能够调通

The screenshot displays two sequential API requests in a REST client interface.

**Request 1:**

- Method: GET
- URL: 127.0.0.1:8000/rest/consumer/v0/hello?name=Alice
- Status: 490 Cse Internal Bad Request
- Time: 53 ms
- Size: 146 B
- Body (JSON):

```
{  "message": "Cse Internal Bad Request"}
```

**Request 2:**

- Method: POST
- URL: 127.0.0.1:8000/rest/consumer/v0/greeting
- Status: 200 OK
- Time: 17 ms
- Size: 151 B
- Body (JSON):

```
{  "msg": "Hello, Ms.Wilson",  "timestamp": "2019-02-27T04:50:30.911Z"}
```



# 服务熔断——自动熔断



我们也可以在页面上配置自动熔断规则。

默认的自动熔断规则需要在10秒内存在20次以上的调用，错误率达到50%时触发熔断，在我们的实验中较难触发。

这里我们将熔断规则改为：

- 熔断效果维持30秒
- 错误率达到50%时触发熔断
- 10秒内有3个以上的请求时开始判断错误率

# 服务熔断——自动熔断

微服务  
edge

所属应用  
Training21Days-HelloWo...

状态  
● 在线

实例  
正常 1 异常 0

创建配置

导入

全部导出

所有作用域

配置项或值

作用域	配置项	值	操作
edge@Training21Days-HelloWorld#0.0.1	cse.circuitBreaker.Consumer.requestVolumeThreshold	3	编辑 删除
edge@Training21Days-HelloWorld#0.0.1	cse.circuitBreaker.Consumer.errorThresholdPercentage	50	编辑 删除
edge@Training21Days-HelloWorld#0.0.1	cse.circuitBreaker.Consumer.sleepWindowInMilliseconds	30000	编辑 删除
edge@Training21Days-HelloWorld#0.0.1	cse.circuitBreaker.Consumer.forceOpen	false	编辑 删除
edge@Training21Days-HelloWorld#0.0.1	cse.circuitBreaker.Consumer.forceClosed	false	编辑 删除
edge@Training21Days-HelloWorld#0.0.1	cse.circuitBreaker.Consumer.enabled	true	编辑 删除
edge@Training21Days-HelloWorld	cse.loadbalance.isolation.enabled	false	编辑 删除

为了避免实例隔离机制对熔断现象的干扰，我们这里将实例隔离关闭。

TIPS：这里对熔断机制、隔离机制的调整是为了更方便地展示熔断的现象，并不是推荐的配置。如果您对于微服务熔断能力还不熟悉，建议先维持默认配置，在实践中根据实际需求再进行调整。

# 服务熔断——自动熔断

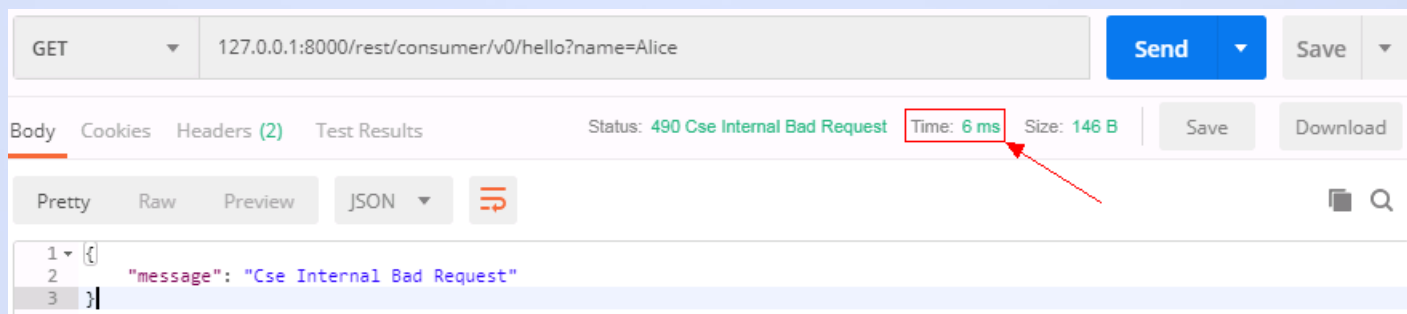
```
private DynamicLongProperty helloDelay = DynamicPropertyFactory.getInstance().getLongProperty(propName: "delay.sayHello", defaultValue: 0);

@Path("/hello")
@GET
public String sayHello(@QueryParam("name") String name) {
    // RPC 调用方式体验与本地调用相同
    if (helloDelay.get() > 0) {
        try {
            Thread.sleep(helloDelay.get());
        } catch (InterruptedException e) {
            LOGGER.error("sayHello sleeping is interrupted!", e);
        }
    }
    return helloService.sayHello(name);
}
```

我们在consumer服务的sayHello方法里增加线程sleep的逻辑

启动provider、consumer、edge服务后，在配置中心设置delay.sayHello=40000，多次发请求通过edge调用consumer的sayHello方法，触发edge服务到consumer服务sayHello方法请求路径的熔断。

可以看到熔断发生后，edge调用consumer的sayHello方法不会等待30秒超时后再返回错误，而是立即就返回错误了



# 服务熔断——自动熔断



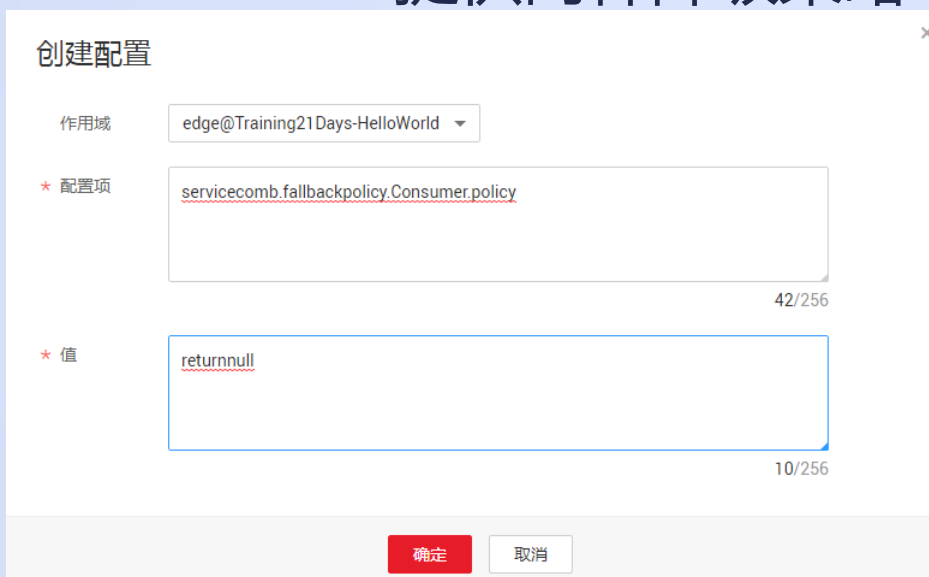
- 熔断发生后，edge服务的熔断状态变为“Open”（可能会延迟几秒，微服务实例上报状态到monitor有时间间隔）
- 熔断期间，通过edge调用consumer服务的方法仍然是通的，只有consumer服务的sayHello方法被熔断
- 修改consumer服务的delay.sayHello配置项为0，等熔断时间窗过后，再次调用consumer的sayHello方法，edge的熔断状态变回“Closed”
- 注意，edge的熔断时间窗结束后，需要等到edge调用consumer的sayHello方法成功，熔断状态才会结束。否则页面上仍然会一直显示熔断状态为“Open”

# 服务降级

这里讲的降级策略是指服务调用出错时的处理策略，可以对应参考ServiceComb文档资料中的[降级策略](#)的容错配置。

CSEJavaSDK提供两种降级策略，即抛出异常和返回null，默认为抛出异常

为了让降级的效果更明显，我们将降级策略修改为returnnull



创建配置

作用域: edge@Training21Days-HelloWorld

\* 配置项: servicecomb.fallbackpolicy.Consumer.policy (42/256)

\* 值: returnnull (10/256)

确定 取消

# 服务降级



← 请求超时

可以看到，当edge调用consumer超时时，返回的响应不再是490错误，而是200，消息体为空  
熔断发生后，edge服务返回的也是空消息体



← 触发熔断

# 灰度发布

- CSEJavaSDK支持灰度发布功能，可以实现版本平滑过渡升级
- 支持按百分比引流和按请求参数特征引流两种方式

```
// for microservice version 0.0.1
// @RequestMapping(path = "/hello/{name}", method = RequestMethod.GET)
// public String sayHello(@PathVariable(value = "name") String name) {
//     return sayHelloPrefix.getValue() + name;
// }

// for microservice version 0.0.2
@RequestMapping(path = "/hello/{name}", method = RequestMethod.GET)
public String sayHello(@PathVariable(value = "name") String name) {
    return sayHelloPrefix.getValue() + name + "." + generateGreeting();
}

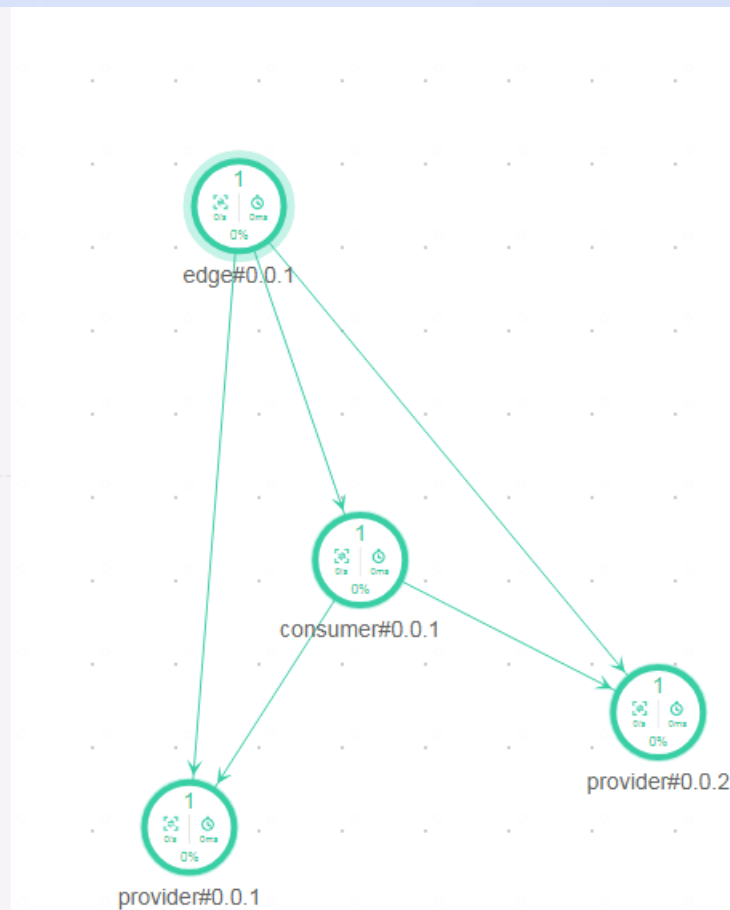
private String generateGreeting() {
    Calendar calendar = new GregorianCalendar();
    int hourOfDay = calendar.get(Calendar.HOUR_OF_DAY);
    if (hourOfDay < 12) {
        return "Good morning.";
    }
    if (hourOfDay < 18) {
        return "Good afternoon.";
    }
    if (hourOfDay < 22) {
        return "Good evening.";
    }
    return "Good night.";
}
```

为了验证灰度发布的效果，我们开发一个0.0.2版本的provider服务，它的sayHello方法会根据时间返回不同的问候信息

修改sayHello方法后，升级provider服务为0.0.2版本来启动provider服务

```
APPLICATION_ID: Training21Days-HelloWorld
service_description:
  name: provider
  #version: 0.0.1
  version: 0.0.2
```

# 灰度发布



将新旧版本的provider服务各启动一个实例，在治理页面看到的微服务情况如左图所示

此时通过edge连续调用sayHello方法，应该是新旧两个版本的应答交替返回，即两个版本的provider服务实例都是可用的服务实例，在轮询策略下依次被调用

127.0.0.1:8000/rest/consumer/v0/hello?name=Alice

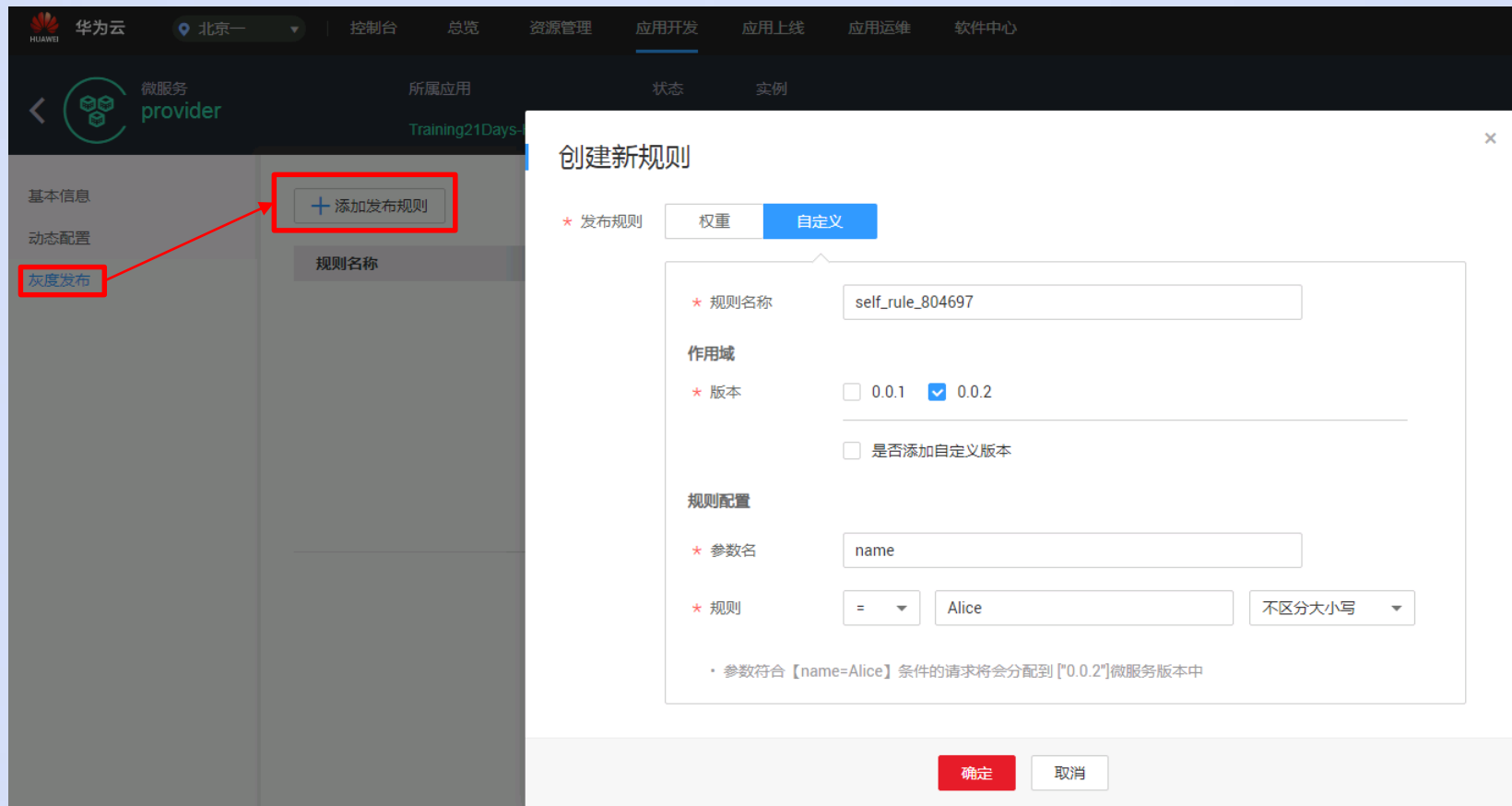
"Hello Alice. Good evening."

127.0.0.1:8000/rest/consumer/v0/hello?name=Alice

"Hello Alice"



# 灰度发布



进入provider服务的详情页面->灰度发布页面，点击添加发布规则，添加如左图所示的自定义灰度规则

# 灰度发布

The image displays three sequential screenshots of a REST client interface, showing the process of testing a hello service with different names.

**First Screenshot:**

- Method: GET
- URL: 127.0.0.1:8000/rest/consumer/v0/hello?name=Alice
- Status: 200 OK
- Time: 8 ms
- Size: 114 B
- Response Body: "Hello Alice. Good evening."

**Second Screenshot:**

- Method: GET
- URL: 127.0.0.1:8000/rest/consumer/v0/hello?name=Bob
- Status: 200 OK
- Time: 32 ms
- Size: 97 B
- Response Body: "Hello Bob"

**Third Screenshot:**

- Method: GET
- URL: 127.0.0.1:8000/rest/consumer/v0/hello?name=Cara
- Status: 200 OK
- Time: 15 ms
- Size: 98 B
- Response Body: "Hello Cara"

再次调用sayHello方法，可以看到当name=Alice时，请求始终是由0.0.2版本的provider服务处理的；而name为其他参数时，请求都是由0.0.1版本的provider服务处理的。

# Thank You





# 21天微服务实战营-Day10

华为云DevCloud & ServiceStage服务联合出品



# Day10 CSE实战之微服务线程模型和性能统计

## 大纲

- 线程模型简介
- 性能统计 ( Metrics )

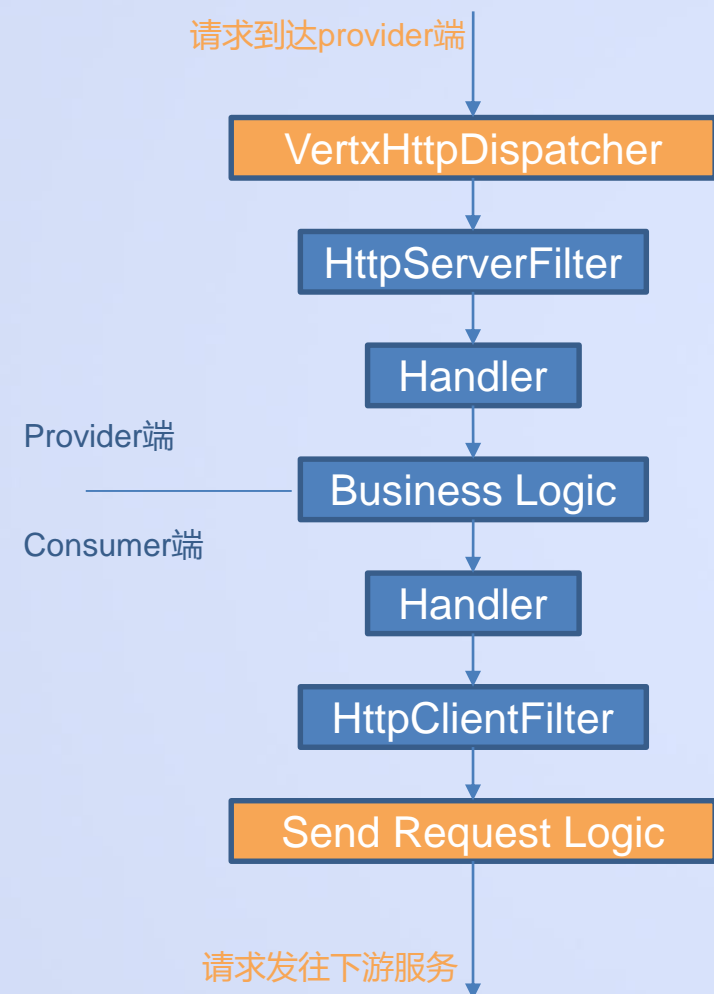
# 线程模型简介

ServiceComb ( CSEJavaSDK ) 是基于Vert.x开发的。

Vert.x是一个依赖Netty，具有异步非阻塞特点的框架，它是CSEJavaSDK高性能的基础，但也让CSEJavaSDK的线程模型看上去与传统的服务框架有所不同。

CSEJavaSDK线程模型的说明可以参考[开源文档](#)，使用CSEJavaSDK原生的默认开发方式时，其传输方式为[Rest over Vertx传输方式](#)。

# 线程模型简介——同步模型



原生的CSEJavaSDK框架开发的微服务默认工作于同步模式，传输方式为Rest over Vertx模式，基于Vert.x进行网络通信。

在此模式下，左图中橙色的部分是在网络线程中处理的，该部分逻辑代码是异步的，以避免阻塞网络线程；蓝色的部分是在业务线程池中处理的，可以是同步代码。

服务端方面，当请求到达微服务实例时，首先是网络线程从网络连接中接收到请求，经过一些处理后切换到业务线程运行provider端handler链、HttpServerFilter、用户的业务逻辑。切换到业务线程后，网络线程就可以去处理下一个请求了。这样可以使网络线程一直处于处理请求的状态，**开发者要避免做阻塞网络线程的操作**，如访问数据库、以同步逻辑代码发送REST请求等。

客户端方面，业务线程发送请求时，首先会在业务线程中对请求做一些处理（包括consumer端handler链、HttpClientFilter），然后转移到网络线程中进行发送。在等待应答的过程中，业务线程会一直处于阻塞状态。等到网络线程返回应答后，会通知业务线程继续运行后面的逻辑。

# 线程模型简介——reactive模型

除了同步模式，CSEJavaSDK也支持[Reactive模式](#)，该模式下所有的处理逻辑都运行在网络线程中。为避免阻塞网络线程，provider端服务接口代码和consumer端调用代码都需要是异步风格的。

Reactive模式的开发较为复杂，用户有兴趣的话可以查阅ServiceComb-Java-Chassis的资料了解相关信息。

Reactive在性能方面有着巨大的优势，但是却并非完美无缺的。它最大的问题就是要求整个项目的代码都运行于异步非阻塞的状态。一旦有一些第三方系统只有同步接口，比如某些数据库驱动三方件，那么这些地方的调用就不能直接放在业务逻辑中，否则会造成网络线程阻塞，性能打折扣。而即使使用线程池将其隔离，也会因为线程上下文的切换而带来额外开销。同时，异步风格的代码有违一般开发人员的习惯，写出来的代码可能不如传统的同步风格代码那么容易理解、调试和定位问题。因此，是否使用Reactive模式进行开发，需要设计和开发人员结合实际情况进行取舍。



# 线程模型简介——reactive模型

## 同步风格

```
@Path("/hello")
@GET
public String sayHello(@QueryParam("name") String name) {
    // RPC 调用方式体验与本地调用相同
    if (helloDelay.get() > 0) {
        try {
            Thread.sleep(helloDelay.get());
        } catch (InterruptedException e) {
            LOGGER.error("sayHello sleeping is interrupted!", e);
        }
    }
    return helloService.sayHello(name);
}
```

- Provider端业务接口直接返回响应消息
- Consumer端业务代码所使用的RPC代理接口也是直接返回响应消息的



## reactive风格

```
@Path("/hello")
@GET
public CompletableFuture<String> sayHello(@QueryParam("name") String name) {
    if (helloDelay.get() > 0) {
        try {
            Thread.sleep(helloDelay.get());
        } catch (InterruptedException e) {
            LOGGER.error("sayHello sleeping is interrupted!", e);
        }
    }

    CompletableFuture<String> response = new CompletableFuture<>();
    helloService.sayHello(name)
        .whenComplete((result, throwable) -> {
            if (null != throwable) {
                LOGGER.error("invoke sayHello failed!", throwable);
                response.completeExceptionally(throwable);
                return;
            }

            LOGGER.info("get response: {}", result);
            response.complete(result);
        });
    return response;
}
```

- Provider端业务接口返回CompletableFuture
- Consumer端业务代码使用的RPC代理接口返回的也是CompletableFuture。如果要处理应答的话，在应答异步返回的时候处理

# 线程模型简介——reactive模型

普通微服务默认都是工作于同步模式的，但是EdgeService网关服务默认是工作于Reactive模式的。因此，在EdgeService网关进行handler和filter扩展定制时需要注意不能阻塞线程，如进行数据库查询、文件IO、同步网络调用等。如果有需要，可以考虑将这部分工作移到其他线程池中处理，网络调用可以改为reactive调用模式，或者直接将EdgeService网关服务的默认线程池设为普通服务所使用的同步线程池，在microservice.yaml文件中进行如下配置：

cse:

executors:

default: servicecomb.executor.groupThreadPool

# 线程模型简介

判断一个微服务的工作模式是否是Reactive模式，最直接的办法是在本地以Debug模式启动该服务，在Filter/Handler扩展类或者业务代码里打断点，观察线程名。例如在edge和consumer服务的HttpServerFilter扩展类里打断点，可以看到filter逻辑的执行线程名的命名格式不同：

```
> Thread.currentThread() = {VertxThread@5257} "Thread[transport-vert.x-eventloop-thread-4,5,main]"
> this = {DemoFilter@6149}
> invocation = {Invocation@6151}
> httpServletRequest = {VertxServerRequestToHttpServletRequest@6152}
```

edge服务，reactive模式

```
> Thread.currentThread() = {Thread@6123} "Thread[pool-2-thread-1,5,main]"
> this = {ServerRestArgsFilter@6125}
> invocation = {Invocation@6128}
> request = {VertxServerRequestToHttpServletRequest@6129}
```

consumer服务，同步模式

TIPS：在同一个微服务中，同步模式和Reactive模式可以并存，一部分REST接口方法以同步模式处理请求，另一部分以Reactive模式处理请求

# 性能统计

CSEJavaSDK自带了一个简单好用的性能统计模块，只需要在maven依赖中引入metrics-core即可使用：

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>metrics-core</artifactId>
</dependency>
```

为开启metrics日志打印功能，还需要在microservice.yaml文件中配置：

```
cse:
  metrics:
    publisher:
      defaultLog:
        enabled: true # 是否在默认的日志中打印metrics日志
      window_time: 10000 # metrics日志打印周期，单位ms
```

**TIPS：**由于引入metrics-core模块会增加两个契约healthEndpoint和metricsEndpoint，分别描述的是健康检查和性能数据的REST接口，因此需要删除服务中心里的旧服务记录以更新契约。

# 性能统计

我们在consumer服务中开启metrics日志，重启并连续调用consumer服务，可以在consumer的日志中看到如下内容：

```
vertx:
  instances:
    name      eventLoopContext-created
    registry  2
    transport 10
    monitor-center 2
    config-center 2
  transport:
    client.endpoints:
      remote      connectCount disconnectCount connections send(Bps) receive(Bps)
      (summary)   0              0              1              0          795
    server.endpoints:
      listen      connectCount disconnectCount rejectByLimit connections send(Bps) receive(Bps)
      0.0.0.0:9090 0              0              0              1          795      0
      (summary)   0              0              0              1          795      0
  threadPool:
    corePoolSize maxThreads poolSize currentThreadsBusy queueSize taskCount completedTaskCount name
    8             8          0          0              0          0.0          0.0          cse.executor.groupThreadPool-group1
    8             8          0          0              0          30.4         30.4         cse.executor.groupThreadPool-group0
consumer:
  simple:
    status      tps      latency      operation
    rest.200    30      2.459/5.911 provider.hello.sayHello
    (summary)   30      2.459/5.911
  details:
    rest.200:
      provider.hello.sayHello:
        prepare: 0.010/0.018 handlersReq : 0.533/3.978 cFiltersReq: 0.017/0.084 sendReq : 0.390/0.564
        getConnect : 0.216/0.316 writeBuf : 0.174/0.290 waitResp: 1.263/2.355 wakeConsumer: 0.045/0.285
        cFiltersResp: 0.109/0.198 handlersResp: 0.090/0.285
producer:
  simple:
    status      tps      latency      operation
    rest.200    30      3.045/6.470 consumer.helloConsumer.sayHello
    (summary)   30      3.045/6.470
  details:
    rest.200:
      consumer.helloConsumer.sayHello:
        prepare: 0.048/0.082 queue : 0.133/0.217 filtersReq : 0.037/0.091 handlersReq: 0.113/0.220
        execute: 2.557/6.002 handlersResp: 0.019/0.046 filtersResp: 0.083/0.156 sendResp : 0.048/0.081
```

Consumer服务既作为服务端接受edge服务的请求，也作为客户端调用provider服务，所以它的metrics日志会打印consumer/provider两方面的内容。

这里详细打印了连接建立、线程池工作状态、吞吐量、请求在内部各阶段的平均处理时间、最大处理时间等数据。

进行压测和性能调优时，可以打开metrics日志作为判断依据。

# Thank You







# 21天微服务实战营-Day11

华为云DevCloud & ServiceStage服务联合出品



# Day11 CSE实战之使用CSEGoSDK开发微服务

## 大纲

- 开发CSEGoSDK微服务
- 开发CSEGoSDK服务调用者



# 开发CSEGoSDK微服务

准备工作：

- ❑ 创建一个新的GO项目
- ❑ 从华为云上下载CSEGoSDK压缩包，并解压到项目的vendor下



# 开发CSEGoSDK微服务

## ➤ 创建Rest接口

我们创建一个struct，名为XXX,然后创建XXX的Hello方法，该法必须参数为\*restful.Context

```
type Service struct {}  
  
func (*Service) Hello(ctx *restful.Context) {  
    name := ctx.ReadPathParameter( name: "name")  
    ctx.WriteJSON(fmt.Sprintf( format: "hello, %s", name), contentType: "application/json")  
}
```

## ➤ 编写url patterns.

```
// URLPatterns  
func (*Service) URLPatterns() []restful.Route {  
    return []restful.Route {  
        {Method: http.MethodGet, Path: "/provider/v0/hello/ {name}", ResourceFuncName: "Hello"},  
    }  
}
```

# 开发CSEGoSDK微服务

创建一个main.go：

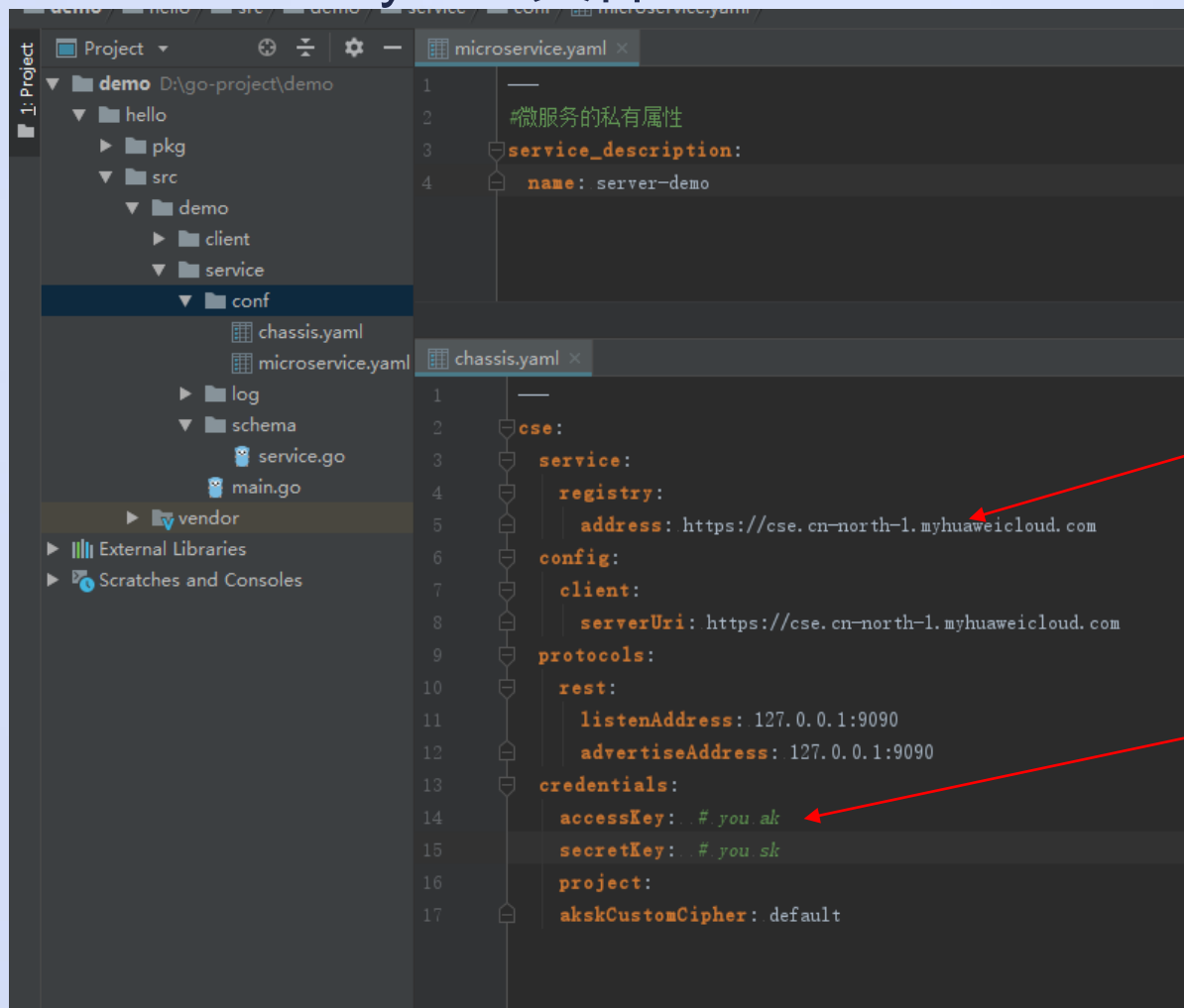
在main.go中我们需要将我们刚刚创建的Service进行注册

```
func main() {  
    chassis.RegisterSchema( serverName: "rest", &schema.Server{}, server.WithSchemaID( schemaID: "server_go"))  
    err := chassis.Init()  
    if err != nil {  
        openlogging.GetLogger().Error(err.Error())  
        return  
    }  
    chassis.Run()  
}
```

在main.go中使用chassis.RegisterSchema方法进行注册，使用chassis.Init()进行初始化，使用chassis.Run()进行启动服务，CSEGoSDK支持rest协议和grpc协议，开发者可以自由选择

# 开发CSEGoSDK微服务

在main.go同级目录下创建conf目录，并在conf目录下创建下 chassis.yaml 和 microservice.yaml 文件



servercenter地址，该地址必须填写

配置ak/sk，ak/sk为必填项，如不配置将无法连接到华为云

# 开发CSEGoSDK微服务

运行main.go,可以在ServiceStage的微服务控制台上可以开到server-demo 服务



微服务控制台  
Cloud Service Engine

仪表盘

服务目录

服务治理

全局配置

事务看板

服务目录

您可以从应用、微服务和实例的维度查看微服务详细信息。如果微服务较多，您可以通过搜索功能查找目标微服务。[如何维护微服务？](#)

应用列表 微服务列表 实例列表

[+ 创建微服务](#) [删除](#)

全部应用(1) 微服务名称

<input type="checkbox"/> 微服务名称	所属应用	版本数	实例数	创建时间	操作
<input type="checkbox"/> server-demo	default	1	1	2019/02/25 09:08:16 GMT+08:00	<a href="#">删除</a>

注意：如果直接使用IDE启动，需要配置环境变量  
CHASSIS\_HOME=/path/to/demo/service/

# 你已成功开发出CSEGoSDK服务

至此，你已经使用CSEGoSDK成功开发出了微服务。调用  
<http://127.0.0.1:9090/provider/v0/hello/Bod>，得到server以下的回复

The screenshot displays a REST client interface with the following components:

- Request Bar:** Method `GET` and URL `http://127.0.0.1:9090/provider/v0/hello/Bod`.
- Request Tabs:** Params, Authorization, Headers (1), Body, Pre-request Script, Tests. The `Headers (1)` tab is currently selected.
- Request Headers Table:**

KEY	VALUE
Key	Value
- Response Tabs:** Body, Cookies (1), Headers (3), Test Results. The `Body` tab is currently selected.
- Response Format:** Pretty, Raw, Preview. The format is set to `JSON`.
- Response Content:**

```
1  "hello , Bod"
```

# 开发微服务调用者

开发一个consumer服务来调用provider服务，定义一个REST接口类接收外部请求并调用provider服务：

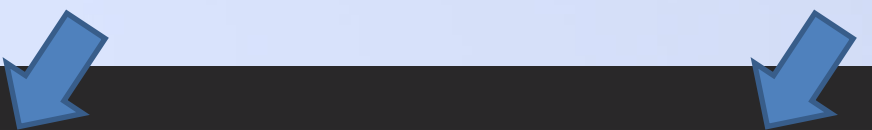
```
14
15     type Client struct {
16     }
17
18     func (*Client) Hello(ctx *restful.Context) {
19         //可以使用 cse:// 和 http://作为前缀
20         //req, err := rest.NewRequest(http.MethodGet, fmt.Sprintf("http://server-demo/provider/v0/hello/%s", ctx.ReadPathParameter("name")), nil)
21         req, err := rest.NewRequest(http.MethodGet, fmt.Sprintf("cse://server-demo/provider/v0/hello/%s", ctx.ReadPathParameter("name")), body: nil)
22         if err != nil {
23             ctx.WriteError(http.StatusInternalServerError, err)
24             return
25         }
26         resp, err := core.NewRestInvoker().ContextDo(context.TODO(), req)
27         if err != nil {
28             ctx.WriteError(http.StatusInternalServerError, err)
29             return
30         }
31         ctx.Write(httputil.ReadBody(resp))
32     }
```

通过core.NewRestInvoker()，创建新的restInvoker(CSEGoSD所有请求均抽象为 invocation),通过restInvoker下的ContextDo方法来对服务发起请求

# 开发微服务调用者

创建main.go:

main.go和provider端的main.go基本一致，修改以下两个箭头所指地方即可



```
2 func main() {  
3     chassis.RegisterSchema( serverName: "rest", &schema.Client {}, server.WithSchemaID( schemaID: "client-demo"))  
4     if err := chassis.Init(); err != nil {  
5         openlogging.GetLogger().Error("Init failed." + err.Error())  
6         return  
7     }  
8     chassis.Run()  
9 }
```



# 开发微服务调用者

配置文件chassis.yaml和microservice.yaml文件基本一直，我们只要修改chassis.yaml中的监听地址，监听地址我们改为：127.0.0.1:8080,以及microservice.yaml中的服务名,服务名修改为client-demo。启动consumer端服务，同样在serverstage看到该服务。调用consumer服务

The screenshot displays a REST client interface with the following details:

- Method:** GET
- URL:** http://192.168.0.131:8081/consumer/v0/hello?name=Bod
- Buttons:** Send, Save
- Tabs:** Params, Authorization, Headers (1), Body, Pre-request Script, Tests, Cookies, Code, Comments
- Params Table:**

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> name	Bod	
Key	Value	Description
- Body Tab:** Pretty, Raw, Preview, JSON (selected), and a refresh icon.
- Response Body:**

```
1 "hello , Bod"
```
- Status Bar:** Status: 200 OK, Time: 3 ms, Size: 121 B, Save, Download

# Thank You





# 21天微服务实战营-Day12

华为云DevCloud & ServiceStage服务联合出品



# Day12 CSE实战之CSEGoSDK场景实战

## 大纲

- ❑ 熔断
- ❑ 限流
- ❑ 容错
- ❑ 灰度

# 熔断

熔断：该功能在服务运行时，能很好的隔离上游服务。在熔断的保护下，如果请求错误过多，服务将停止对该服务访问。

熔断支持：

- 全局配置。该配置对所有的服务的所有请求都起作用，只要服务任何一个接口达到熔断条件将导致该服务所有接口都被熔断隔离。
- 服务级别配置。此时熔断只对所配置的服务器作用，没有配置的服务将使用默认配置或全局。此时与全局类似，某个服务的一个接口导致熔断也会影响其他接口不同正常使用。
- API级别配置。API级别的配置仅对API生效，即使同一个服务API之间互相不影响。如/hello接口发生错误导致熔断，/hi接口依旧可以正常使用

# 熔断

## 配置示例

### 全局配置示例：

```
---
cse:
  isolation:
    Consumer:
      timeoutInMilliseconds: 10
      maxConcurrentRequests: 1000
  circuitBreaker:
    Consumer:
      enabled: true
      forceOpen: false
      forceClosed: false
      sleepWindowInMilliseconds: 10000
      requestVolumeThreshold: 10
      errorThresholdPercentage: 50
```

1，表示是否开启熔断功能

2，表示是否强制开启熔断，开启后，将无法正常进行访问，请求被直接隔离

3，表示是否强制关闭熔断，强制关闭后，无论发生什么错误都不会产生熔断

4，表示熔断后多少毫秒后重新尝试访问服务，如果失败继续熔断，如果成功则恢复原来的访问

5，表示多少请求后计算错误请求占比，此时需要1为true以及2,3为false才生效

6，表示请求错误数占比达到该值后，开启熔断。同样需要1为true以及2,3为false

# 熔断

## 服务级别配置示例：

```
---
cse:
  isolation:
    Consumer:
      timeoutInMilliseconds: 10
      maxConcurrentRequests: 1000
  circuitBreaker:
    Consumer:
      enabled: true
      forceOpen: false
      forceClosed: false
      sleepWindowInMilliseconds: 10000
      requestVolumeThreshold: 10
      errorThresholdPercentage: 50
      server_name:
        enabled: false
        forceOpen: false
        forceClosed: false
        sleepWindowInMilliseconds: 10000
        requestVolumeThreshold: 20
        errorThresholdPercentage: 50
```

server\_name 为服务名，请根据实际配置的服务名进行填写

# 限流

## ➤ 介绍

用户可以通过配置限流策略限制Provider端或Consumer端的请求频率，使每秒请求数限制在最大请求量的大小。其中Provider端的配置可限制接收处理请求的频率，Consumer端的配置可限制发往指定微服务的请求的频率。使用是，需要添加以下 handler：provider添加ratelimiter-provider，consumer添加ratelimiter-consumer。

## ➤ 配置

限流配置在rate\_limiting.yaml中，同时需要在chassis.yaml的handler chain中添加ratelimiter-provider。其中qps.limit.[service] 是指限制从service 发来的请求的处理频率，若该项未配置则global.limit生效。Consumer端不支持global全局配置，其他配置项与Provider端一致， handler chain添加ratelimiter-consumer。



# 限流

## ➤ 配置演示

客户端配置，限流设置为10  
那么访问任何服务时流量都是每  
秒10次

consumer端：

```
---
cse:
  flowcontrol:
    Consumer:
      qps:
        enabled: true # enable rate limiting or not
        limit:
          server_name: 10 # rate limit for request to a provider
```

```
handler:
  chain:
    Consumer:
      default: bizkeeper-consumer, router, loadbalance, ratelimiter-consumer, transport
#ssl:
```

provider端

全局配置，限  
流设置为100

```
---
cse:
  flowcontrol:
    Provider:
      qps:
        enabled: true # enable rate limiting or not
        global:
          limit: 100 # default limit of provider
        limit:
          sever_name: 1 # rate limit for request from a provider
```

```
20 handler:
21   chain:
22     Provider:
23       incoming: ratelimiter-provider
```

# 容错

在CSEGoSDK提供自动重试的容错能力，用户可配置retry及backOff策略自动启用重试功能

## ➤ 类型

默认情况下CSEGoSDK支持三种容错类型

- ❑ zero：固定重试时间为0的重试策略，即失败后立即重试不等待
- ❑ constant：固定时间为backoff.minMs的重试策略，即失败后等待backoff.minMs再重试。
- ❑ jittered：按指数增加重试时间的重试策略，初始重试时间为backoff.minMs，最大重试时间为backoff.MaxMs。推荐此方法

# 容错

## 配置实例

```
cse:
  loadbalance:
    retryEnabled: true
    retryOnNext: 0
    retryOnSame: 1
  backoff:
    kind: jittered # 重试策略: [jittered或constant或zero]。
    MinMs: 200
    MaxMs: 400
```

是否使用容错

请求失败后向其他实例重试的次数。

请求失败后向同一个实例重试的次数。

重试策略: [jittered或constant或zero]。

重试最小时间间隔, 单位ms。

重试最大时间间隔, 单位ms。

# 灰度发布

应用于AB测试场景和新版本的灰度升级等相关场景。CSEGoSDK通过对路由的管理实现以上场景，以下重点讲述CSEGoSDK如何通过router的管理实现灰度发布。

## □ 路由管理

路由策略可应用于AB测试场景和新版本的灰度升级，主要通过路由规则来根据请求的来源、目标服务、Http Header及权重将服务访问请求分发到不同版本的微服务实例中。

# 灰度

- 配置路由管理：路由管理使用router.yamll 进行配置。同时也支持API方式进行设置，本文并不介绍，推荐使用router.yamll进行配置。
- 路由规则说明
  - 匹配特定请求由routeRule.{targetServiceName}.match配置，匹配条件是：source（源服务名）、source tags及headers，另外也可以使用refer字段来使用source模板进行匹配。
  - Match中的Source Tags用于和服务调用请求中的。sourceInfo中的tags进行逐一匹配。
  - Header中的字段的匹配支持正则、=、!=、>、<、>=、<=七种匹配方式。
  - 如果未定义match，则可匹配任何请求。
  - 转发权重定义在routeRule.{targetServiceName}.route下，由weight配置。
  - 服务分组定义在routeRule.{targetServiceName}.route下，由tags配置，配置内容有version和app。

# 灰度-配置示例

## ➤ 示例一：

每个路由规则的目标服务名称都由routeRule中的Key值指定。例如下表所示，所有以 “Carts”服务为目标服务的路由规则均被包含在以 “Carts”为Key值的列表中。

```
routeRule:
  Carts:
    - precedence: 1
      route:
        - weight: 100 #percent
          tags:
            version: 0.0.1
```

Key值（目标服务名称）应该满足是一个合法的域名称。例如，一个在服务中心中注册的服务名称。

# 灰度-配置示例

## ➤ 示例二：

每个在一个服务拥有多个不同版本的实例时，我们可以通过precedence进行设置优先级，默认为 0，该配置值越大，优先权越高。

```
routeRule:
  Carts:
    - precedence: 2
      match:
        headers:
          Foo:
            exact: bar
      route:
        - weight: 100
          tags:
            version: 2.0
    - precedence: 1
      route:
        - weight: 100
          tags:
            version: 1.0
```

图一

图一：在match到bar的请求，会优先将请求分流之version为2.0的实例

# Demo演示

## ➤ Demo访问规则：

<http://127.0.0.1:8080/consumer/v0/delay?t=1s&delay=20&c=5>

t 表示对provider发起访问的时间

delay表示服务延时的时长，单位ms

c 表示开启的goroutine数

上面的URL表示开启5个goroutine进行访问1s，每次请求延时20ms



# Demo演示

## ➤ 熔断：熔断配置

```
---
cse:
  isolation:
    Consumer:
      timeoutInMilliseconds: 1
      maxConcurrentRequests: 1000
  circuitBreaker:
    # scope: api
    Consumer:
      enabled: true
      forceOpen: false
      forceClosed: false
      sleepWindowInMilliseconds: 10000
      requestVolumeThreshold: 10 # 请求次数达到10次后, 若失败率高于50%服务开始熔断
      errorThresholdPercentage: 50
```

配置说明：该配置中，请求达到十次后将会开始计算错误率，如果错误率达到50%将会开启熔断。如下图，服务在连续访问十次都失败后，第十一开始熔断

访问URL: <http://127.0.0.1:8080/consumer/v0/delay?t=1s&delay=10&c=5>

```
36      "Error": "Get http://127.0.0.1:9092/provider/v0/delay/11: net/http: request canceled (Client.Timeout exceeded while awaiting headers)"
37    },
38    {
39      "Reply": "",
40      "Error": "Get http://127.0.0.1:9092/provider/v0/delay/11: net/http: request canceled (Client.Timeout exceeded while awaiting headers)"
41    },
42    {
43      "Reply": "",
44      "Error": "API provider:rest:/provider/v0/delay/11 is isolated because of error: circuit open"
45    }
46  ]
```

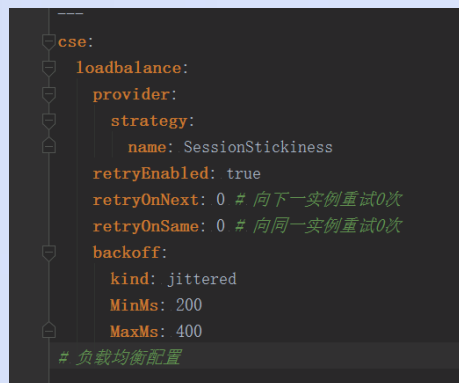
第十一次  
出现熔断

第十次错误方  
向

# Demo演示

容错：配置设置为0，并且配置会话保持策略，然后像服务发起数次访问，请求。效果如下右下图。访问URL如下：

<http://127.0.0.1:8080/consumer/v0/delay?t=1s&delay=10&c=1>



有图中我们可以看出失败后并没有继续访问统一实例，而是访问下一个实例

```
2019-03-21 16:30:13.056 +08:00 INFO schema/consumer.go:69 launched one http benchmark thread
2019-03-21 16:30:13.058 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.060 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.062 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9091/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.064 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9091/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.066 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.068 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.070 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9091/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.072 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.074 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.076 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.078 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9091/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.080 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9091/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.082 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.084 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.086 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9091/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.088 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9091/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.090 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.092 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9092/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
2019-03-21 16:30:13.093 +08:00 ERROR handler/transport_handler.go:48 Call got Error, err [Get http://127.0.0.1:9091/provider/v0/delay/1: net/http: request canceled (Client.Timeout exceeded while awaiting headers)]
```

# Demo演示

限流：在任意一个provider端设置限流，限流为10，开启十个线程访问1s

```
1  ---
2  cse:
3    flowcontrol:
4      Provider:
5        qps:
6          enabled: true # enable rate limiting or not
7          global:
8            limit: 10
```

首次访问由于框架内有需要进行初始化操作，导致访问不准确，二次访问后，每次访问次数都固定为20.

# Demo演示

## ➤ 灰度

完成目标：配置router.yaml文件。文件配置要求在header设施 key:XXX后，并且启动 provider\_v1和provider\_v2。访问规则会优先访问高版本实例，配置后，需要达到请求 100%请求version 1.0。配置如下：

```
1 routeRule:
2   provider:
3     - precedence: 2
4       match:
5         headers:
6           Foo:
7             exact: bar
8         route:
9           - weight: 100
10            tags:
11              version: 1.0
12     - precedence: 1
13       route:
14         - weight: 100
15         tags:
16           version: 2.0
```

配置说明：左侧配置中我们在头部只要携带了 Foo:bar 该请求将全部访问 version 1.0的实例，头部信息设置如下图

```
req, err := rest.NewRequest(method, url, body: nil)
if err != nil {
    panic(err)
}
//req.Header.Set("Foo", "bar") ← 取消注释
resp, err := restInvoker.ContextDo(ctx, req)
if err != nil {
    // ...
}
```

# Demo演示

## ➤ 灰度

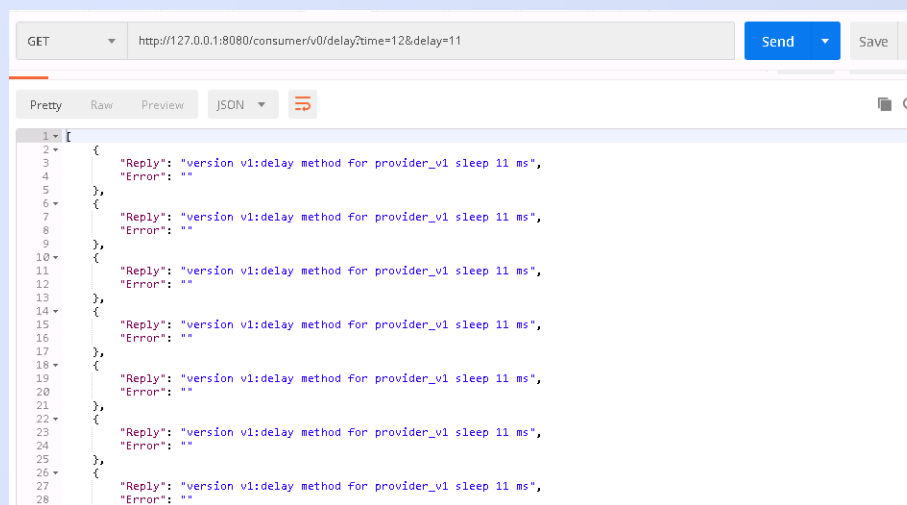
配置好所有的配置后，我们分别将配置注释和开启各访问一次，下方左侧图为没有使用router规则时访问，可以看到请求到达了version 2.0的实例。在使用router规则后，并且header包含Foo : bar时，请求将全部访问version 1.0。如下方右侧图



```
GET http://127.0.0.1:8080/consumer/v0/delay?time=12&delay=11
Send Save

Pretty Raw Preview JSON

1 [
2   {
3     "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
4     "Error": ""
5   },
6   {
7     "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
8     "Error": ""
9   },
10  {
11    "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
12    "Error": ""
13  },
14  {
15    "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
16    "Error": ""
17  },
18  {
19    "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
20    "Error": ""
21  },
22  {
23    "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
24    "Error": ""
25  },
26  {
27    "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
28    "Error": ""
29  },
30  {
31    "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
32    "Error": ""
33  },
34  {
35    "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
36    "Error": ""
37  },
38  {
39    "Reply": "version v2:delay method for provider_v1 sleep 11 ms",
40    "Error": ""
41  }
42 ]
```



```
GET http://127.0.0.1:8080/consumer/v0/delay?time=12&delay=11
Send Save

Pretty Raw Preview JSON

1 [
2   {
3     "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
4     "Error": ""
5   },
6   {
7     "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
8     "Error": ""
9   },
10  {
11    "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
12    "Error": ""
13  },
14  {
15    "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
16    "Error": ""
17  },
18  {
19    "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
20    "Error": ""
21  },
22  {
23    "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
24    "Error": ""
25  },
26  {
27    "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
28    "Error": ""
29  },
30  {
31    "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
32    "Error": ""
33  },
34  {
35    "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
36    "Error": ""
37  },
38  {
39    "Reply": "version v1:delay method for provider_v1 sleep 11 ms",
40    "Error": ""
41  }
42 ]
```

# 参考文献

## 配置参考

- ❑ [熔断配置](#)
- ❑ [限流配置](#)
- ❑ [容错配置](#)
- ❑ [灰度配置](#)

## 推荐文档:

- ❑ [微服务分布式系统熔断实战-为何我们需要API级别熔断？](#)
- ❑ [Go语言微服务开发框架实践-go chassis（上篇）](#)
- ❑ [Go语言微服务开发框架实践-go chassis（中篇）](#)
- ❑ [使用go chassis进行微服务路由管理](#)

# Thank You





# 21天微服务实战营-Day13

华为云DevCloud & ServiceStage服务联合出品





# Day13 CSE实战之异构技术栈相互调用

## 大纲

- 启动CSEJavaSDK开发的provider
- 启动CSEGoSDK开发的consumer互相调用

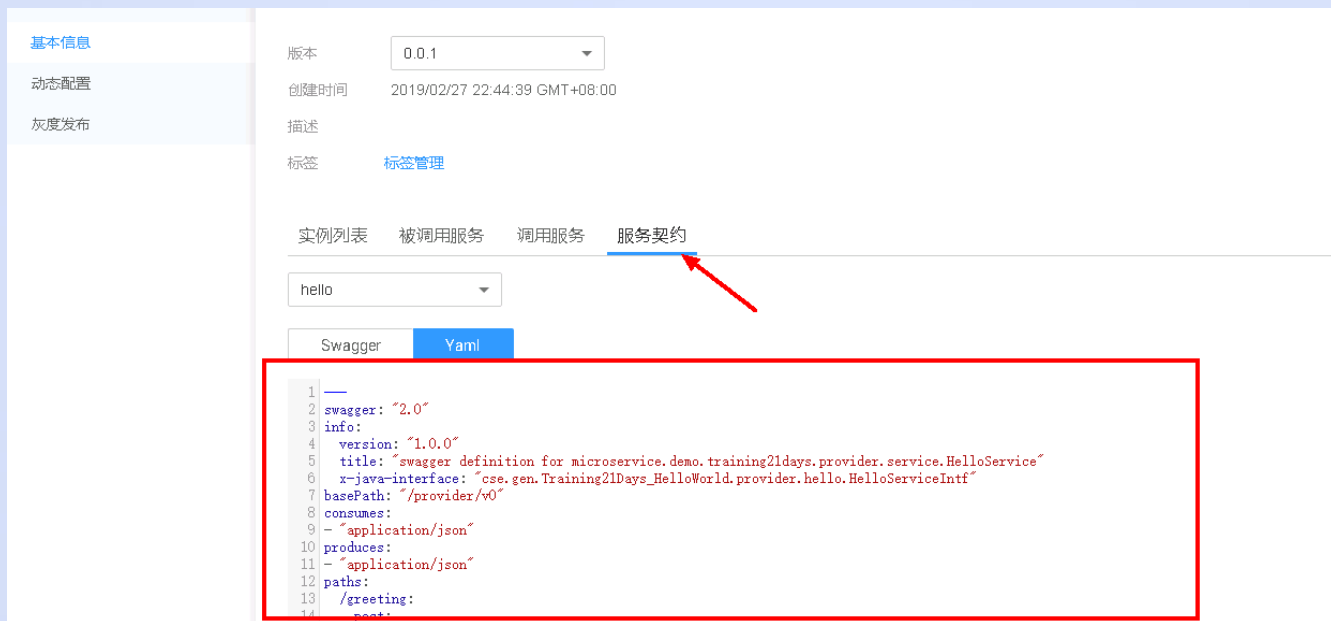
# 启动CSEJavaSDK开发的consumer和provider

在Day6时，我们已经详细的介绍了如何开发一个基于CSEJavaSDK的provider微服务，以及如何开发consumer微服务调用provider，同时也完成了打卡任务。

首先我们运行基于CSEJavaSDK的provider服务，并且启动edge服务。成功运行后请在ServiceStage下确保服务已成功注册。

# 启动CSEGoSDK开发的consumer

运行Day11 demo前我们更改Day11中demo服务的监听地址。将consumer的监听端口改为8081。同时我们也需要为Day11中的demo配置契约文件。契约可以在你启动Java后，在ServiceStage上获取，进入服务详情后，点击服务契约并选择Yaml显示,如下图:



在本地任意地方创建  
`{path}/{server_name}/schema/{server_name}.yaml`,将契约复制到`{server_name}.yaml`下。在启动Day11的demo之前，设置环境变量  
`SCHEMA_ROOT`，如：  
`export SCHEMA_ROOT = {path}/`

TIPS：服务契约是CSEJavaSDK和CSEGoSDK开发的微服务相互调用的基础。CSEJavaSDK对于契约的检查较严，如果启动时发现sc上注册的契约与本地生成的不同就会报错。因此，为保证Java服务能够正常启动，需要先启动Java微服务，确保它的契约先注册到sc，再启动Go微服务。

# 调用CSEGoSDK consumer

调用 127.0.0.1:8000/rest/consumer/v0/hello?name=Alice 调用成功后，查看Java-provider日志，如果成功看到下图证明成功。

```
2019-03-04 11:47:06,571 [INFO] 192.168.0.45 - - Mon, 04 Mar 2019 11:47:05 CST /provider/v0/hello/Alice 200 13 632
2019-03-04 11:47:19,406 [INFO] 192.168.0.131 - - Mon, 04 Mar 2019 11:47:19 CST /provider/v0/hello/Alice 200 13 3
```

查看Day6中的edge日志。下面日志为同时启动了Java和Go的consumer

```
2019-03-04 11:46:42,751 [INFO] find instances[2] from service center success. service=Training21Days-HelloWorld/consumer/0.0.0+, old revision=null, new revision=6622e345639cf581af89c82f852e79b7f3eeda6 or
2019-03-04 11:46:42,751 [INFO] service id=df84e45987f11364a8e99724248cbc1a718fd988, instance id=c41821583e2f11e9900e0255ac105166, endpoints=[rest://192.168.0.45:9090] org.apache.servicecomb.serviceregistr
2019-03-04 11:46:42,752 [INFO] service id=df84e45987f11364a8e99724248cbc1a718fd988, instance id=f2cacda23e2f11e9900e0255ac105166, endpoints=[rest://192.168.0.131:8081] org.apache.servicecomb.serviceregistr
```

成功访问后，可以看到以下访问日志

```
2019-03-04 11:47:06,620 [INFO] 192.168.0.131 - - Mon, 04 Mar 2019 11:46:42 CST /rest/consumer/v0/hello?name=Alice 200 13 24010 5c7c9fa3e48b952b.d
2019-03-04 11:47:17,087 [INFO] 192.168.0.131 - - Mon, 04 Mar 2019 11:47:16 CST /rest/consumer/v0/hello?name=Alice 200 15 204 5c7c9fc482463d64 org
2019-03-04 11:47:18,298 [INFO] 192.168.0.131 - - Mon, 04 Mar 2019 11:47:18 CST /rest/consumer/v0/hello?name=Alice 200 15 13 5c7c9fc63199e423 org.
2019-03-04 11:47:19,407 [INFO] 192.168.0.131 - - Mon, 04 Mar 2019 11:47:19 CST /rest/consumer/v0/hello?name=Alice 200 13 10 5c7c9fc7678a50d0 org.
```

# Thank You





# 21天微服务实战营-Day14

华为云DevCloud & ServiceStage服务联合出品



# Day14 CSE实战之其他服务何如接入CSE

## 大纲

- ❑ 创建springboot服务(默认熟练运用springboot)
- ❑ 为Springboot服务绑定mesher
- ❑ CSEGoSDK开发的provider和consumer

## □创建springboot服务

- 创建一个maven项目，引入springboot所需的jar包，为了方便测试，该服务需要具有两个接口一个作为服务提供者接口，另外一个作为客户端接口，该服务将和使用CSEGoSDK开发的服务端和客户端进行访问。

```
8  @RestController
9  @RequestMapping("/consumer/v0")
10 public class ConsumerController {
11     @RequestMapping("/hello/{name}")
12     public String ProviderHello(@PathVariable(value = "name") String name) {
13         RestTemplate restTemplate = new RestTemplate();
14         String s = restTemplate.getForObject( url: "http://provider/provider/v0/hello/" + name, String.class);
15         return s;
16     }
17 }

ConsumerController > ConsumerGreeting()

ProviderController.java x
8  @RestController
9  @RequestMapping("/provider/v0")
10 public class ProviderController {
11     @RequestMapping("/hello/{name}")
12     public String ConsumerHello(@PathVariable(value = "name") String name) { return "hello , " + name; }
15 }
```



# □创建springboot服务

在创建好springboot服务后，我们需要为刚刚创建好的绑定一个mesher，在绑定mesher之前，我们需要将springboot的代理设置为mesher的监听地址。本文通过Java的ProxySelector进行代理设置

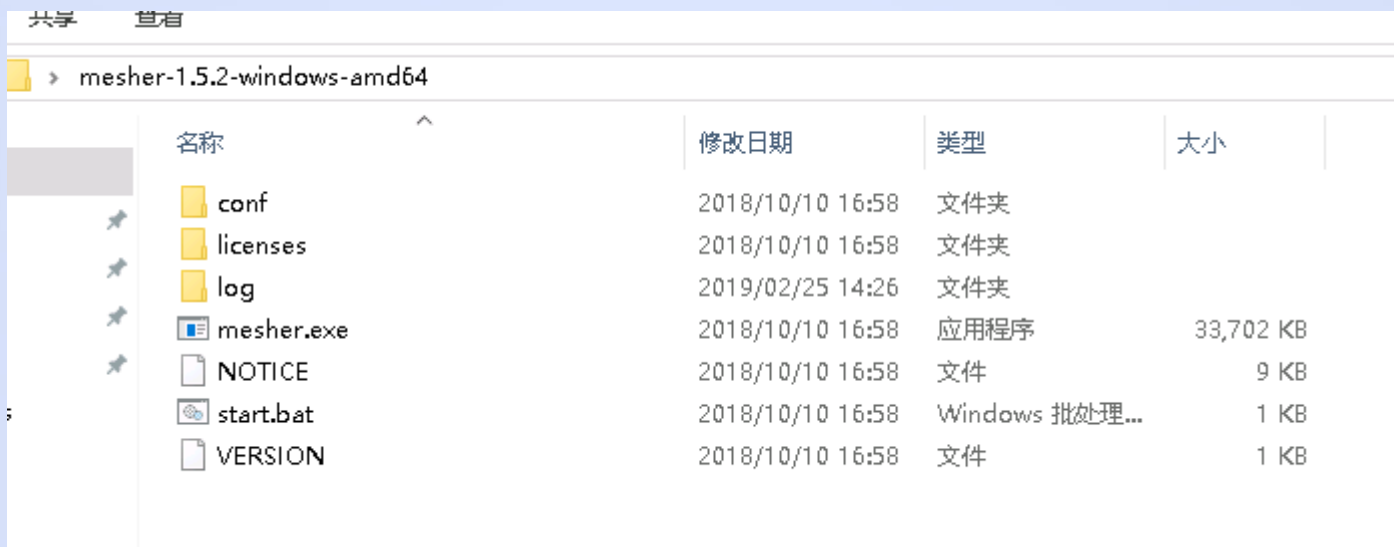
```
public class MyProxySelector extends ProxySelector {  
    @Override  
    public List<Proxy> select(URL uri) {  
        List<Proxy> list = new ArrayList<>();  
        // 设置需要进行代理的地址，此处为mesher的host和port  
        String address = "127.0.0.1";  
        int port = 30101;  
        InetSocketAddress socketAddress = new InetSocketAddress(address, port);  
        Proxy proxy = new Proxy(Proxy.Type.HTTP, socketAddress);  
        list.add(proxy);  
        return list;  
    }  
  
    @Override  
    public void connectFailed(URL uri, SocketAddress sa, IOException ioe) {  
    }  
}
```

```
@SpringBootApplication  
public class Application {  
    public static void main(String[] args) {  
        ProxySelector.setDefault(new MyProxySelector());  
        SpringApplication.run(Application.class, args);  
    }  
}
```

## □绑定mesher

Springboot服务接口以及代理均已开发好时，我们就可以为springboot服务绑定一个mesher了

1. 从华为云下载mesher，请点击[mesher](#)下载  
下载解压如下：



名称	修改日期	类型	大小
conf	2018/10/10 16:58	文件夹	
licenses	2018/10/10 16:58	文件夹	
log	2019/02/25 14:26	文件夹	
mesher.exe	2018/10/10 16:58	应用程序	33,702 KB
NOTICE	2018/10/10 16:58	文件	9 KB
start.bat	2018/10/10 16:58	Windows 批处理...	1 KB
VERSION	2018/10/10 16:58	文件	1 KB

# □绑定mesher

## mesher配置

修改microservice.yaml,将服务名修改为springboot , 并新增APPLICATION\_ID : Training21Days-HelloWorld

```
1  ## microservice property
2  APPLICATION_ID: Training21Days-HelloWorld
3  service_description:
4    name: springboot
5    version: 0.0.1
6    environment: #microservice environment
7    properties:
8      allowCrossApp: false #whether to allow calls across applications
9
```

服务名

mesher版本

修改chassis.yaml,将监听地址改为机器的外部访问地址 , 如 : 192.168.0.1,  
并在该配置文件中添加ak/sk

```
1  ---
2  cse:
3    credentials:
4      accessKey: # you ak
5      secretKey: # you sk
6      project:
7      akskCustomCipher: default
```

## □绑定mesher

### ➤ 启mesher:

除了使用mesher提供的启动脚本启动外我们亦可以直接运行mesher.exe(mesher), 在运行之前, 我们需要设置环境变量 SPECIFIC\_ADDR, SPECIFIC\_ADDR支持格式: rest:port,grpc:port, http:port。可同时写两个如下, 设置后运行mesher.exe(mesher)。

e.g. export SPECIFIC\_ADDR=rest:80

# 绑定mesher

➤ 修改完并启动mesher后，可以在serverstage下看到如下



微服务控制台  
Cloud Service Engine

仪表盘

服务目录

服务治理

全局配置

事务看板

←

服务目录

您可以从应用、微服务和实例的维度查看微服务详细信息。如果微服务较多，您可以通过搜索功能查找目标微服务。[如何维护微服务？](#)

应用列表 微服务列表 实例列表

创建微服务

删除

全部应用(1)

微服务名称

Q

C

<input type="checkbox"/> 微服务名称	所属应用	版本号	实例数	创建时间	操作
<input type="checkbox"/> <a href="#">springboot</a>	default	1	1	2019/02/25 14:26:38 GMT+08:00	<a href="#">删除</a>
<input type="checkbox"/> <a href="#">client-demo</a>	default	1	1	2019/02/25 10:45:06 GMT+08:00	<a href="#">删除</a>
<input type="checkbox"/> <a href="#">server-demo</a>	default	1	1	2019/02/25 09:08:16 GMT+08:00	<a href="#">删除</a>

## □开发CSEGoSDK服务

我们在Day11时已经开发了provider和consumer，此时我们可以直接使用

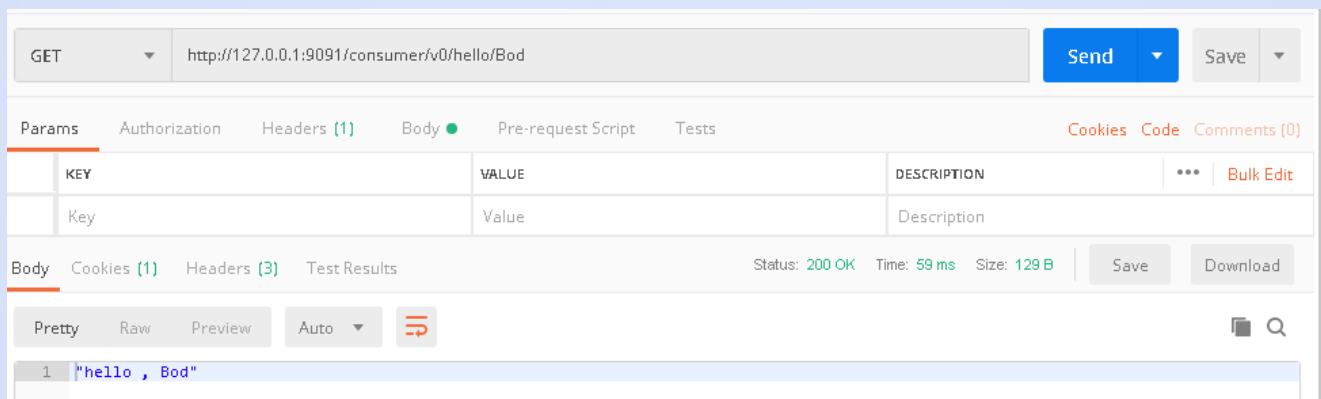
➤ 修改Day11中的consumer访问服务名为springboot

```
func (*Client) Hello(ctx *restful.Context) {  
    //可以使用 cse:// 和 http://作为前缀  
    //req, err := rest.NewRequest(http.MethodGet, fmt.Sprintf("http://server-demo/provider/v0/hello/%s", ctx.ReadPathParameter("name")), nil)  
    req, err := rest.NewRequest(http.MethodGet, fmt.Sprintf("cse://springboot/provider/v0/hello/%s", ctx.ReadPathParameter("name")), nil)  
    if err != nil {
```

同时按照Day11的启动方式将Day11的provider和consumer进行启动

# □访问服务

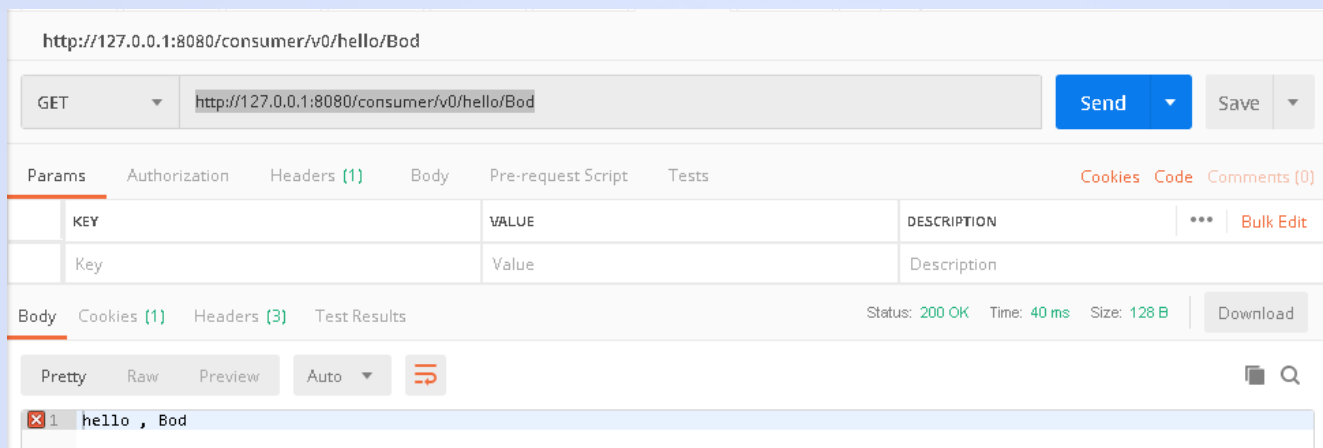
访问springboot服务的consumer接口 , /consumer/v0/{path}。  
调用 <http://127.0.0.1:9091/consumer/v0/hello/Bod> ,反馈结果



# □访问服务

访问Day11服务的consumer接口，/consumer/v0/{path}。

调用 <http://127.0.0.1:8080/consumer/v0/hello/Bod>,反馈结果





## □更多接入mesher-demo参考

- [nodejs接入CSE](#)
- [php接入CSE](#)
- [.Net接入CSE](#)
- [更多](#)

# Thank You





# 21天微服务实战营

华为云DevCloud & ServiceStage服务联合出品



# DAY15 微服务云应用平台介绍

在前面的两周课程中，介绍了大量微服务相关的技术，从本节开始介绍由华为云隆重推出的微服务云应用平台：ServiceStage。

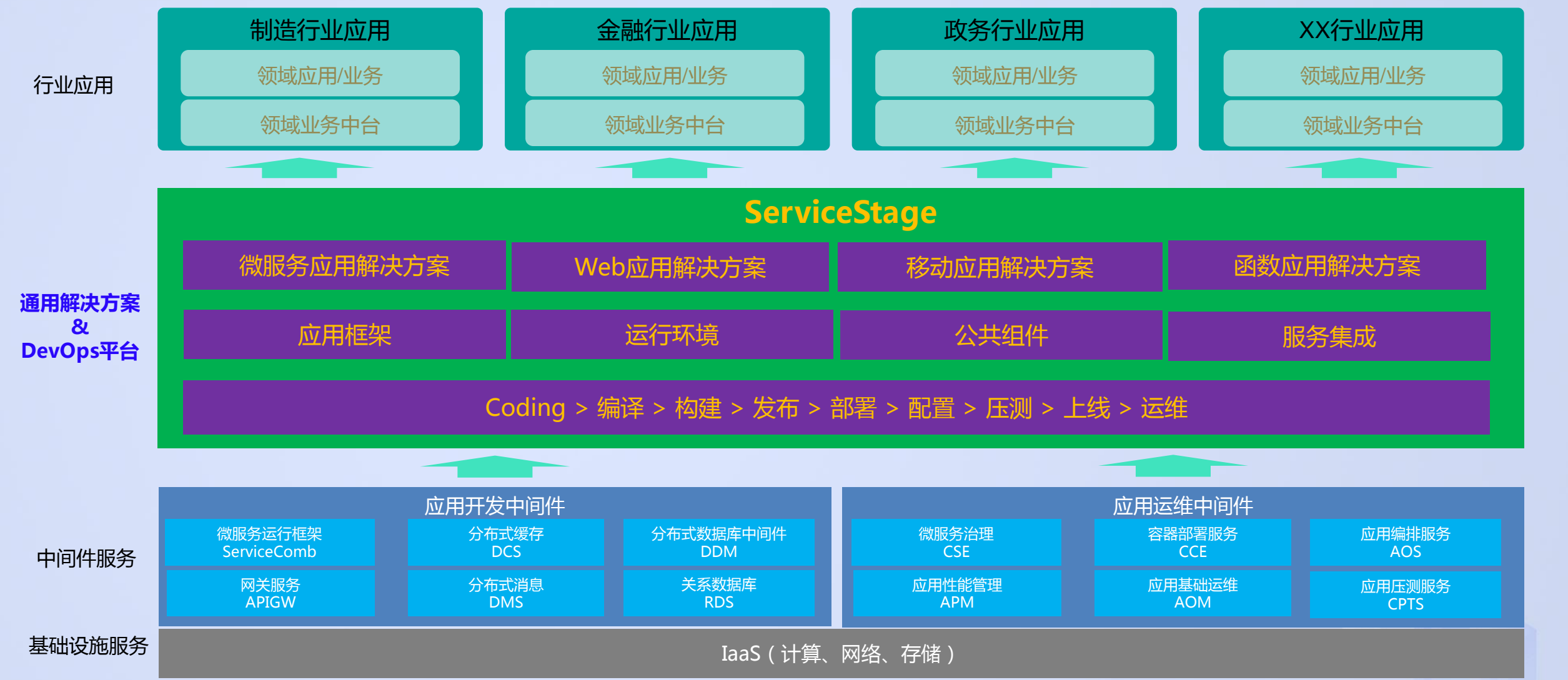
- **ServiceStage是什么**
- **ServiceStage的使用场景**
- **ServiceStage的应用案例**

# ServiceStage是什么

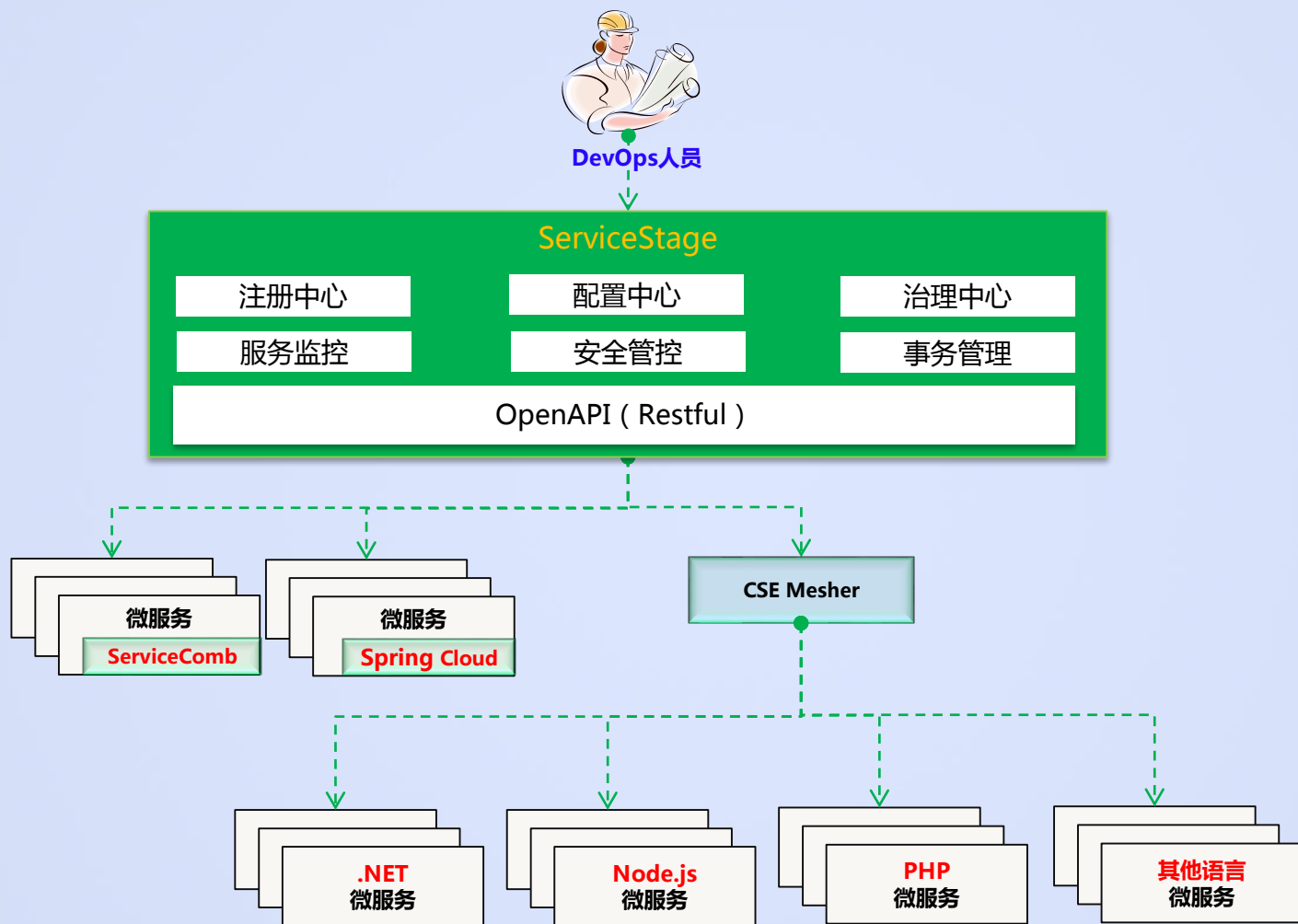
ServiceStage是集华为全面云化转型成功经验和技术创新成果为一体的一站式应用云平台，面向企业提供EI、区块链、微服务、移动和Web类应用开发的全栈解决方案，帮助用户快速创建企业级云原生应用，加速业务创新。



# 企业基于ServiceStage可以快速构建行业应用



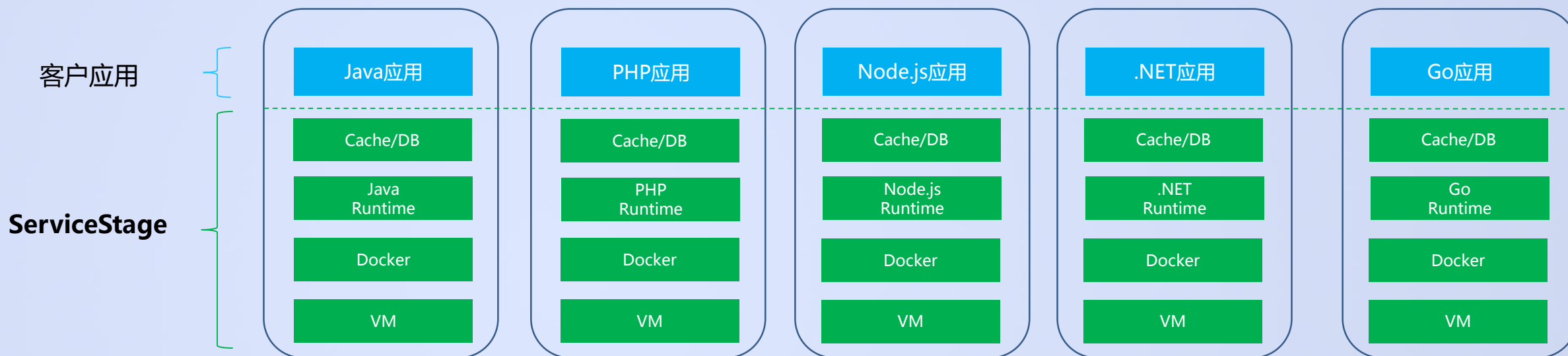
# 场景一：微服务开发和管理



- **基于契约(Open API)的开发模式**：让微服务的开发、测试、文档、协作和管控活动标准化、自动化
- **高性能REST/RPC微服务开发框架**：打包了微服务注册、发现、通信和治理等基础能力，开箱即用
- **一站式微服务治理控制台**：提供微服务负载均衡、限流、降级、熔断、容错、错误注入等治理能力
- **提供ServiceComb、Spring Cloud、Service Mesh商业版**

## 场景二：Web应用快速开发和部署

支持 Java、PHP、Node.js、.NET、Go 等等。开发者只需轻松编写代码即可。



### 源码仓库

★ 代码源来源



Gitee



GitHub



Bitbucket



GitLab



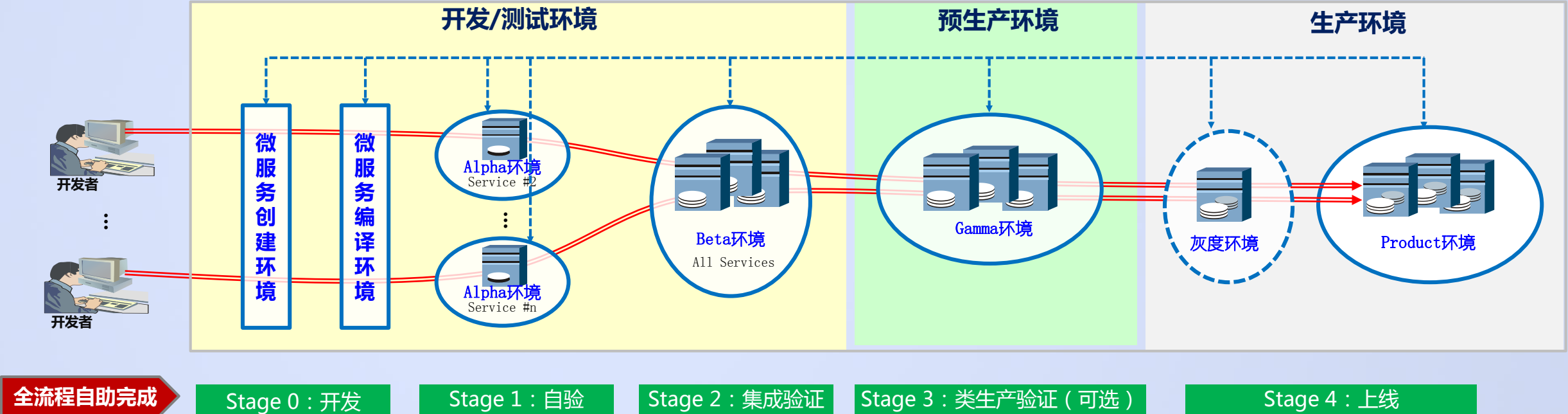
DevCloud

开发者只需使用 **ServiceStage+任意源码仓库**，通过流水线功能实现轻松部署和更新。



# 场景三：持续集成和持续交付

基于ServiceStage流水线全流程“自助式”开发、集成、验证与上线



以“月/季度/半年”为周期 → 以“天/小时”为周期

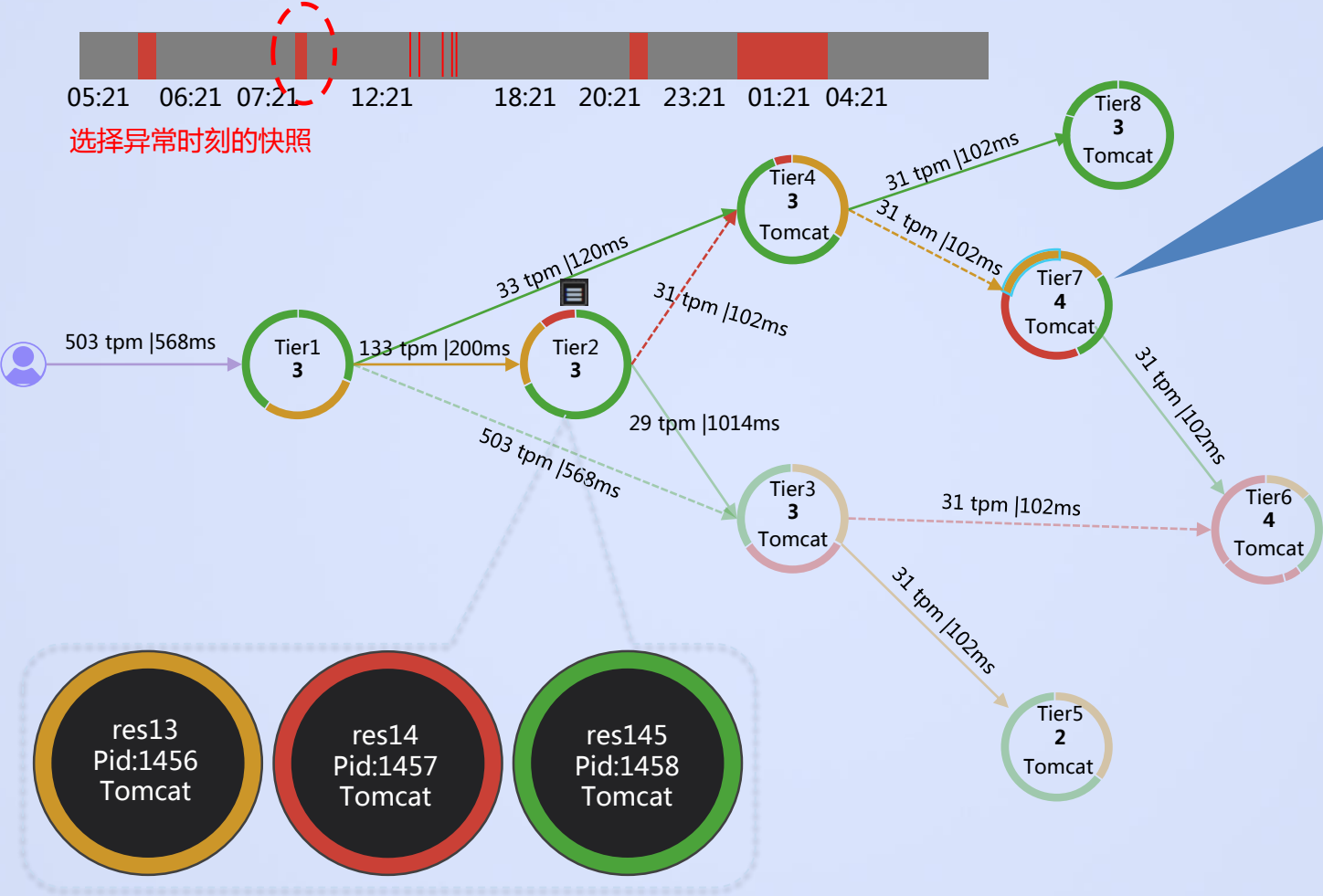
## 场景四：灰度发布

什么是灰度发布：为保障新特性能平稳上线，可以通过灰度发布功能选择少部分用户试用，降低发布风险。

更安全的发布方式，让业务上线不仅仅是快。



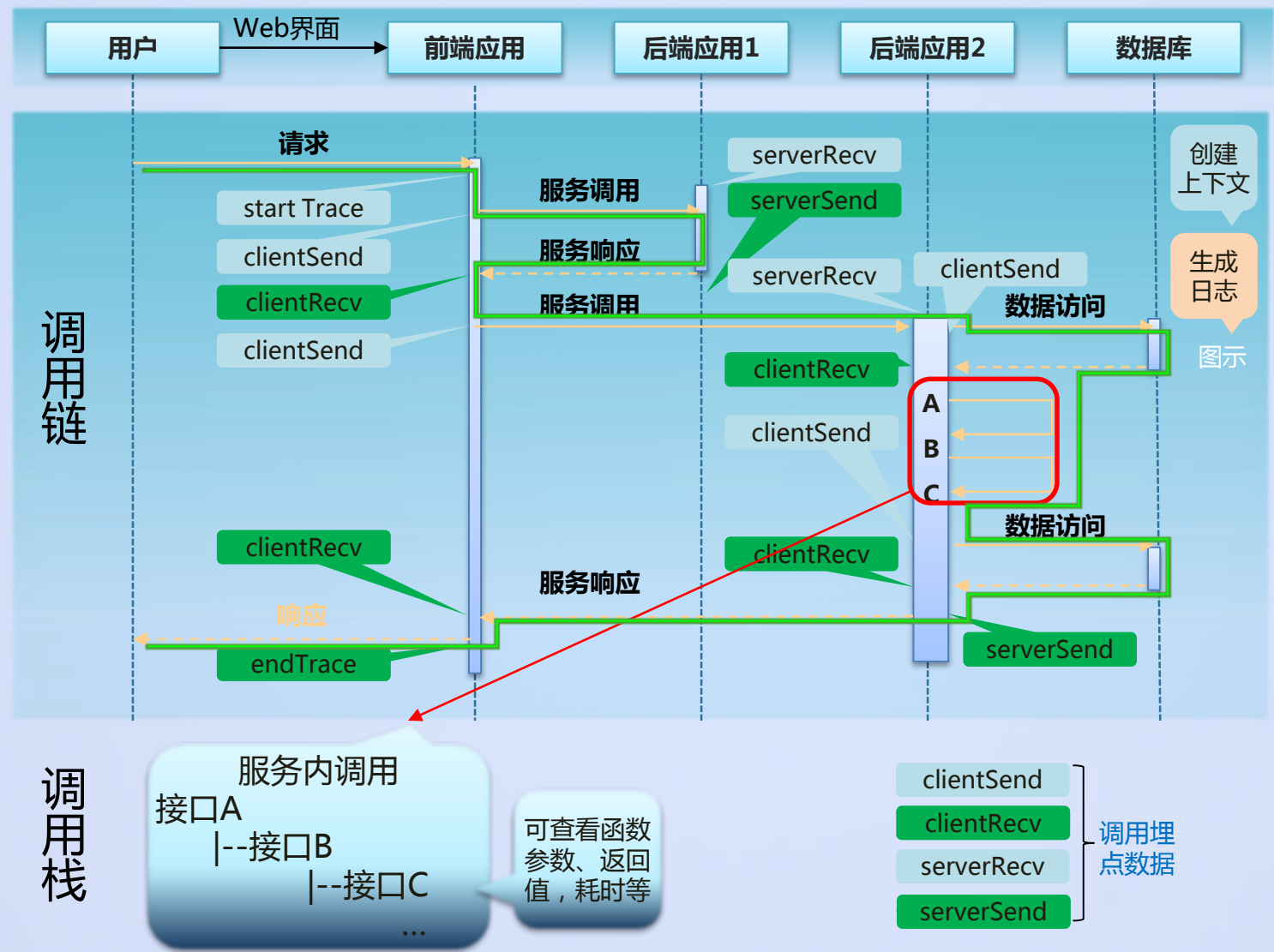
# 场景五：应用性能分析



微服务名称：Tier7  
实例数量：4  
容器类型：Tomcat  
图例：环状图表示微服务，环按照实例数量被拆分4段，每段的颜色表示每实例的状态，红色表示异常，黄色表示警告；

- 应用发现与依赖关系：非侵入采集应用KPI数据，并通过服务间接口自动生成依赖关系。
- 应用KPI汇聚：微服务实例汇聚到应用（数字表示XX个实例），KPI数据自动汇聚到应用。

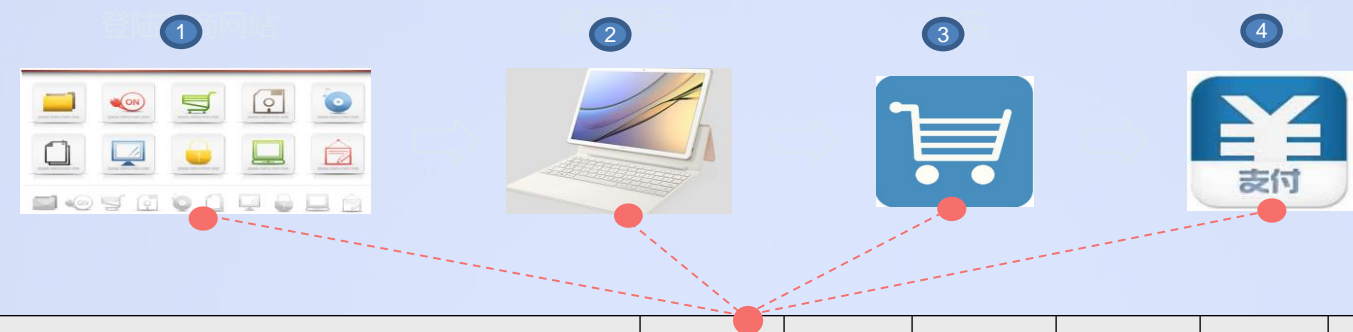
# 场景六：调用链跟踪与监控



支持平台、资源、应用的监控和微服务调用链分析

- **大规模**：支持百万容器监控，秒级查询响应。
- **故障下钻**：通过单击故障节点可自动下钻到故障的微服务实例、也可以关联到失败的调用链和调用栈，查看失败函数的入参和返回值。

# 场景七：事务分析



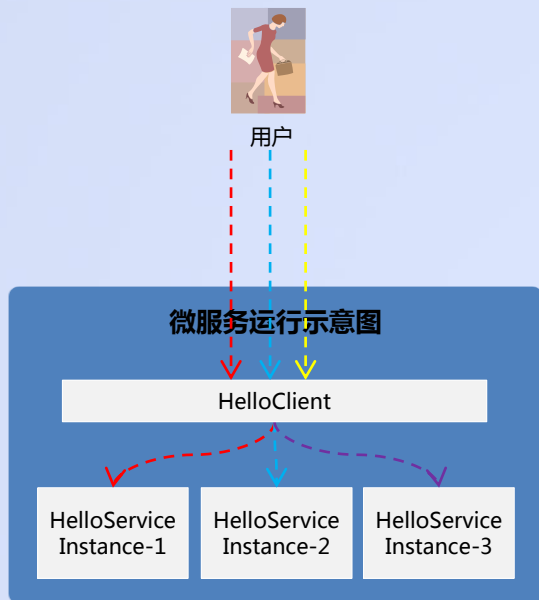
	事务URL	事务名称	吞吐率 (次/分钟)	平时时延	健康指标	错误率	性能分析
1	hwid1.vmall.com/CAS/portal/login.html?validated=true	登陆	52	323ms	正常	0%	<a href="#">调用链分析</a>
2	hwid1.vmall.com/portal/search?id=34211223411	商品搜索	234	721ms	较慢	1%	<a href="#">调用链分析</a>
3	hwid1.vmall.com/portal/buy?id=34211223411	购买	3	1.32s	正常	0%	<a href="#">调用链分析</a>
4	hwid1.vmall.com/portal/pay?id=34211223411	支付	1	2.1s	异常	100%	<a href="#">调用链分析</a>

实时跟踪每条业务交易，快速分析交易的运行状态并提供诊断能力

- **自定义事务**：用户可根据每条URL定义事务名称，方便理解。
- **健康规则配置**：可以对每条事务配置健康规则，如超过1s提示异常；
- **性能追踪**：精确采集异常性能数据，可对比历史基线数据，也能找到应用的异常方法，提升运维效率

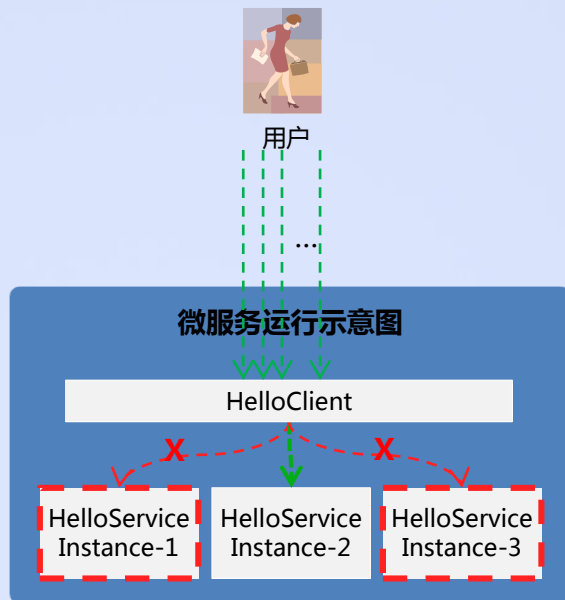
★ 通过对业务监控，用户可以了解最终**消费者行为**，用于**业务发展决策**；其次可以**快速发现业务运营的状态**，对于异常的应用程序快速诊断；

# 场景八：服务治理



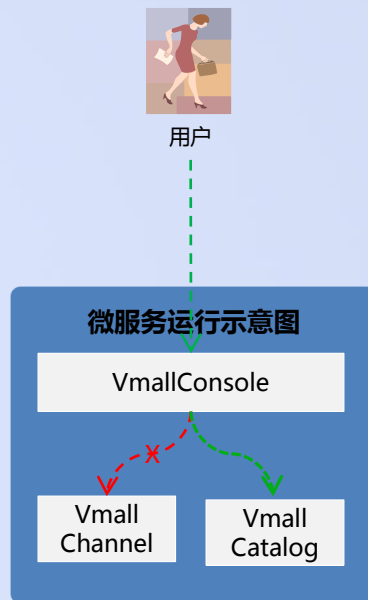
## 自动负载均衡

可选策略：顺序、随机、本地、粘滞、基于实例负载、基于权重)



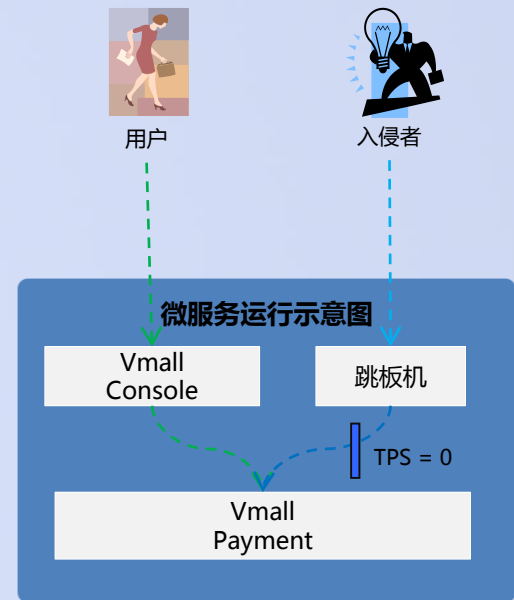
## 自动容错

任意实例Down掉(至少保留一个实例)，不影响正常业务



## 服务降级

关闭非核心业务，保证核心业务可用

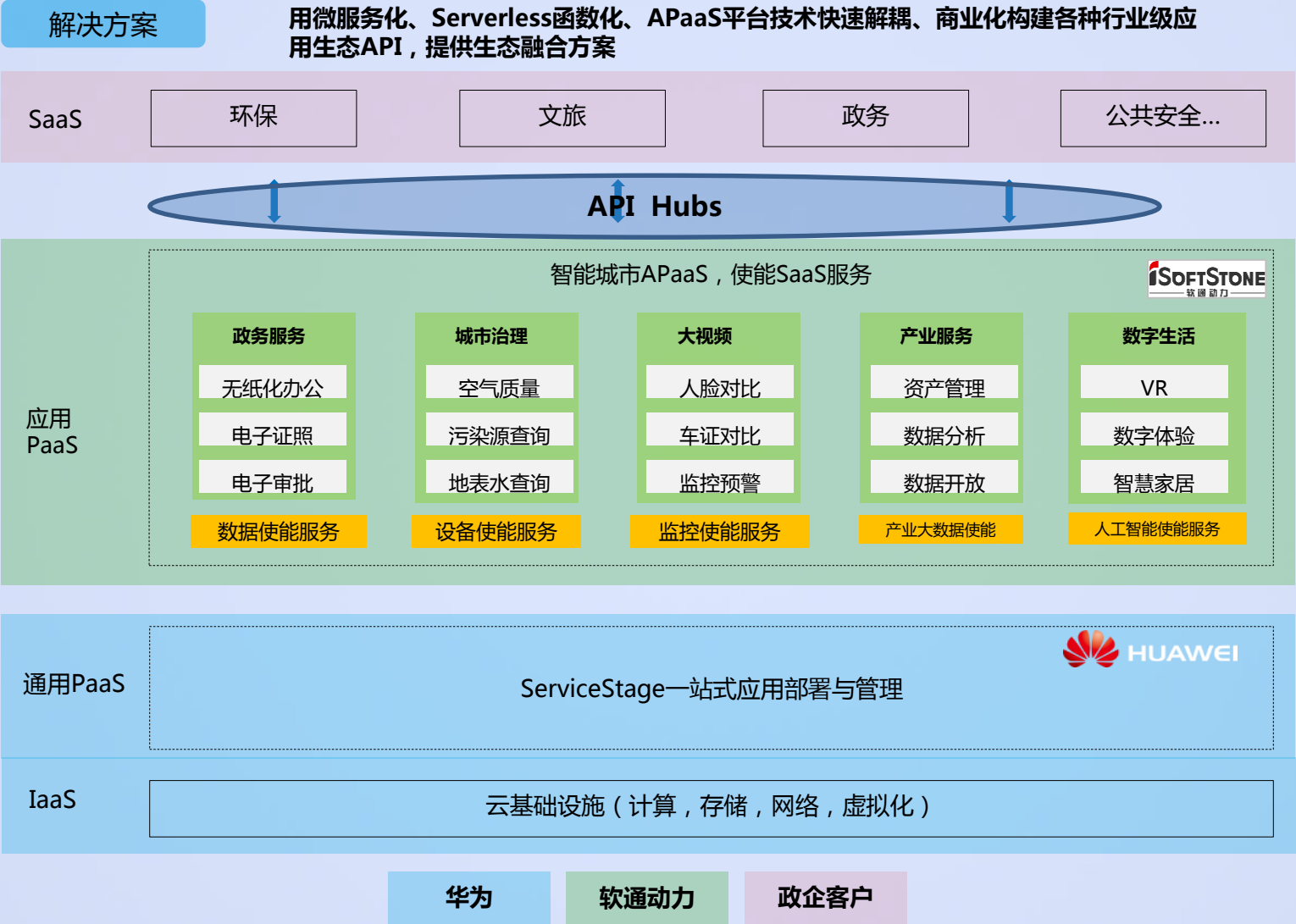


## 服务限流

请求超过处理能力，按策略丢弃，保证已接纳请求正常处理

支持微服务接口级SLA指标(吞吐量、时延、成功率)实时(秒级)监控和治理，保障应用运行不断服

# 应用案例1：华为云加速软通动力行业应用PaaS构建，无需关心基础资源与应用管理



## 面临挑战

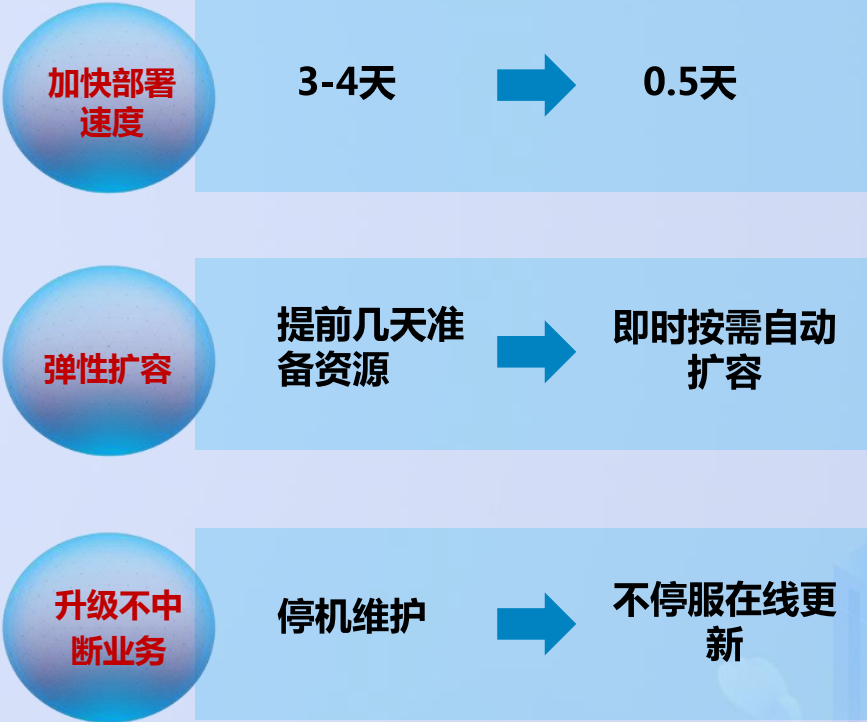
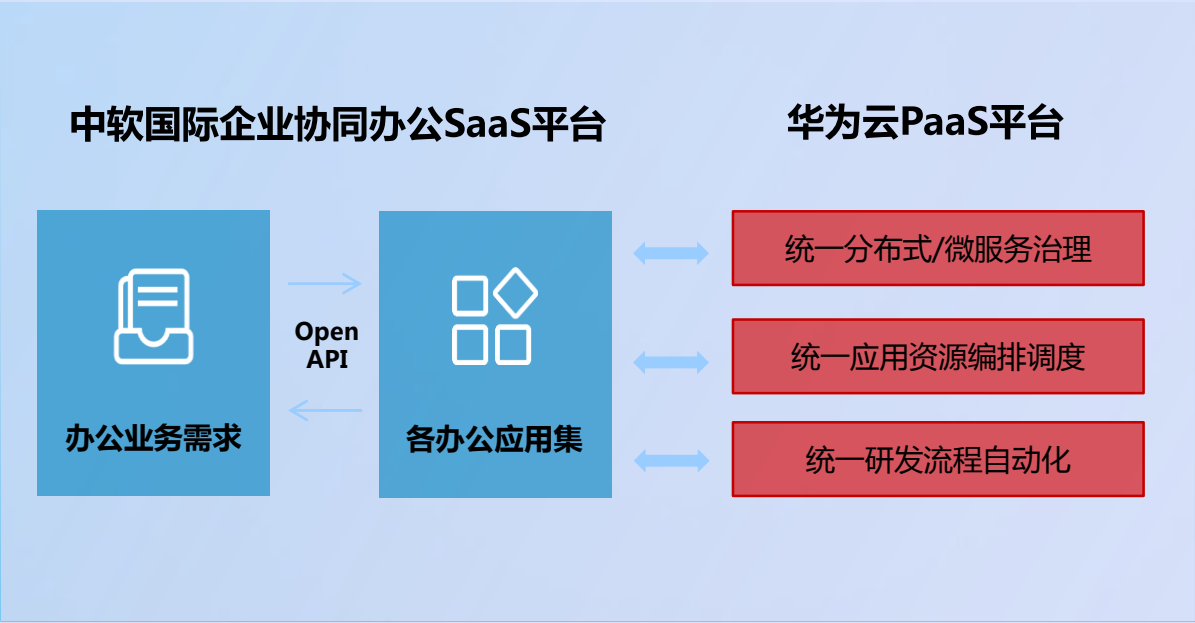
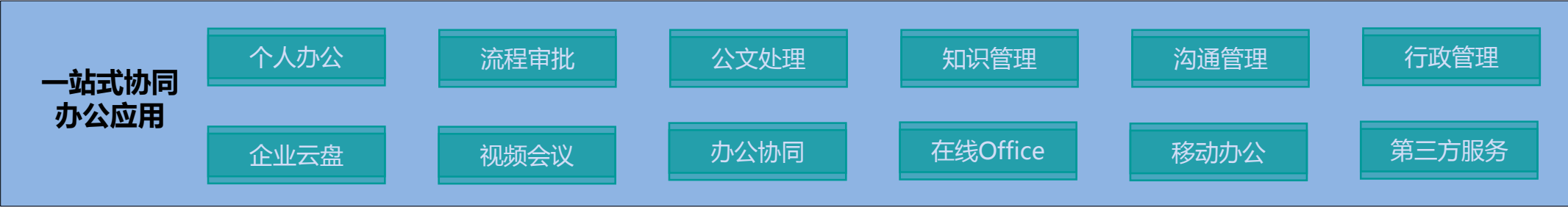
- 快速上云，现有系统改动小
- 多租户部署、隔离和管理比较复杂
- 打造行业级产品但研发周期长
- 需要更多业务合作、融合新业务体

## 客户价值

- 新生态：通过API、编排，融合新应用
- 成本低：函数秒级扩展编程，镜像分钟级发布
- 维护易：APaaS指标监控
- 运营模式多样化：SaaS租户、APaaS、GPaaS、IPaaS等商业化APIKey运营



# 应用案例2：基于ServiceStage助力中软国际构建企业协同办公SaaS平台





# 应用案例3：ServiceStage帮助文思海辉实现楼宇设施管理系统快速微服务化

### 基于RFID技术的楼宇设施管理

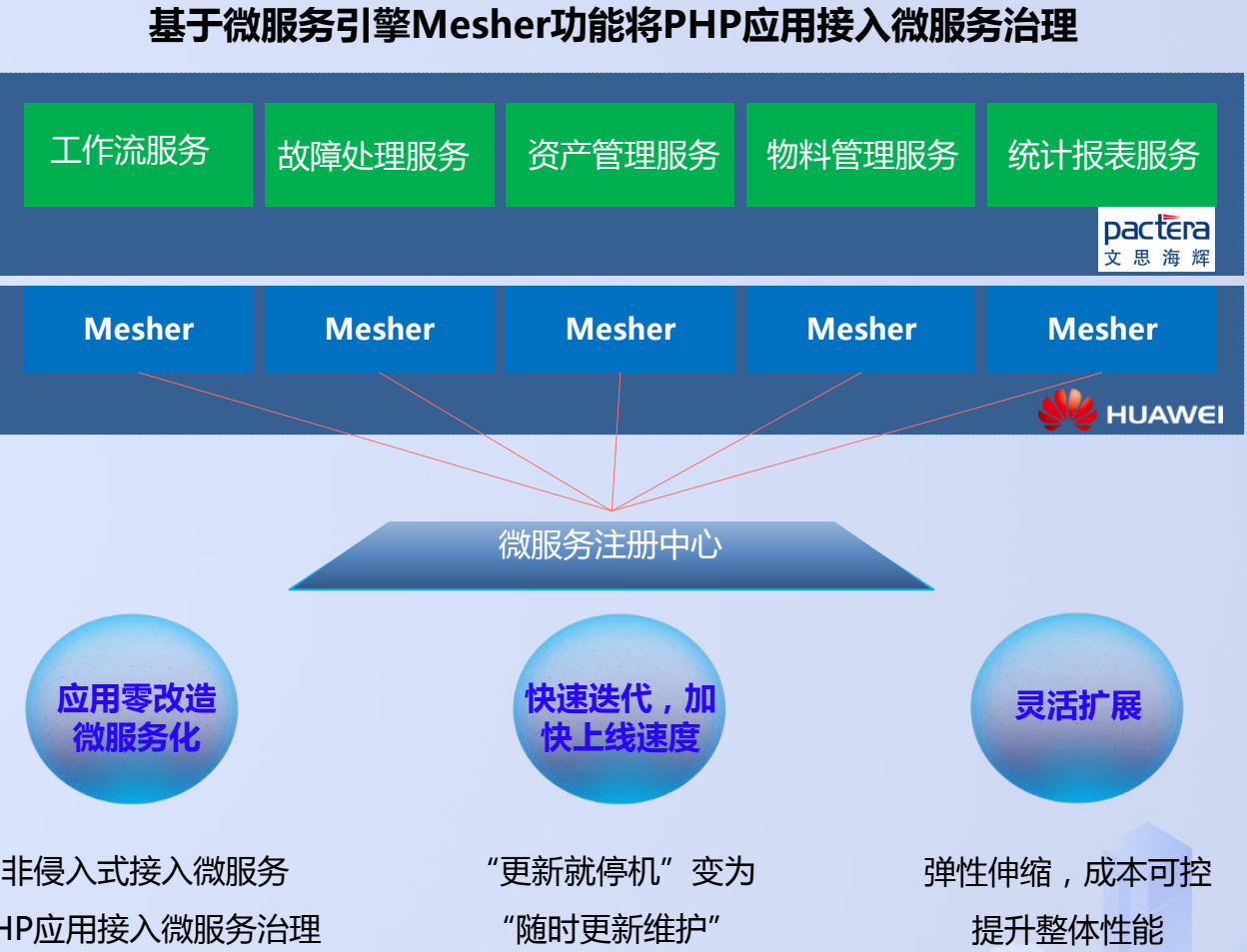


### 遇到的问题

复杂度高  
不易扩展

性能瓶颈  
稳定性差

- 单体应用，功能耦合
- 新增功能影响现有业务，升级业务中断
- 业务增长对性能要求高，增加机器不能解决问题
- 一个业务问题影响整个系统，风险高



# 应用案例4：ServiceStage助力盟拓软件房企系统云化，1天上线1个微服务



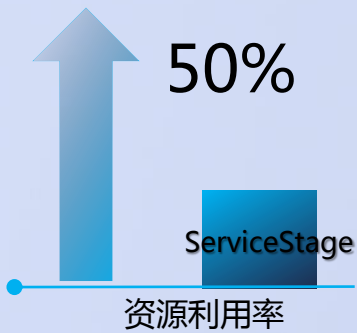
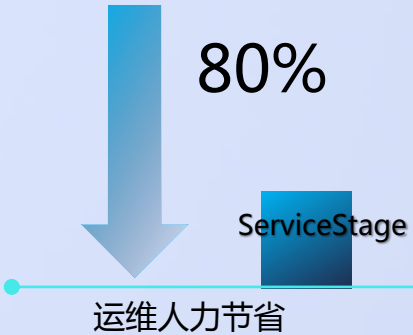
## 容器化应用，秒级弹性伸缩

- 支持4000,000并发用户数
- 支持100tps微服务应用访问

## 微服务自动化运维

## 开放的微服务框架

- ServerComb支持多语言应用
- Mesher支持遗留应用零改造接入 ( SpringCloud/Dubbo )



# Thank You





# 21天微服务实战营

华为云 DevCloud & 微服务产品 联合出品



# DAY16 微服务应用开发之持续交付

本节介绍的内容主要包括：

- DevOps之持续交付
- ServiceStage持续交付解决方案
- ServiceStage持续交付应用案例

# DevOps

## DevOps ( Development & Operations )

- 概念：一组文化、流程与工具整合后的统称
- 目的：更好服务客户，高效参与市场竞争
- 过程：打破孤岛，促进开发和运维之间高度协同
- 实践：频繁小规模更新，微服务架构
- 问题：部署量大幅度增加，成本增加
- 方案：通过CI/CD以安全可靠的方式快速迭代

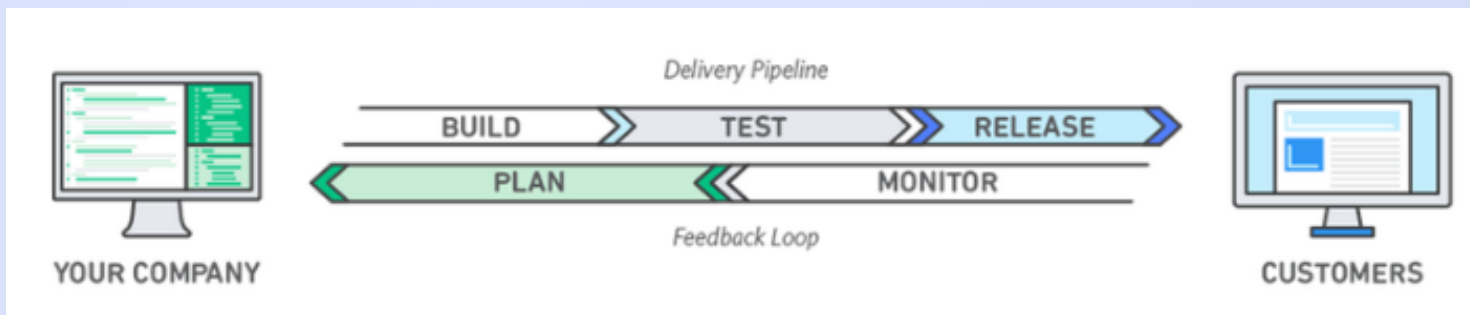


图 DevOps模式

# 持续交付

- CD ( Continuous Delivery , **持续交付** ) 是一种 DevOps 软件开发实践
- 频繁地将软件的新版本，交付给质量团队或者用户，以供评审
- 采用持续交付时，系统会自动构建、测试并准备代码变更，以便将其发布到生产环境中。
- 持续交付并不是指软件每一个改动都要尽快部署到产品环境中，它指的是**任何的代码修改都可以在任何时候实施部署**。
- 持续交付可实现**整个软件发布流程的自动化**。提交的每一个修订都会触发一个自动化流程，即构建、测试并提供更新。部署到实际生产环境的最终决定由开发人员触发。

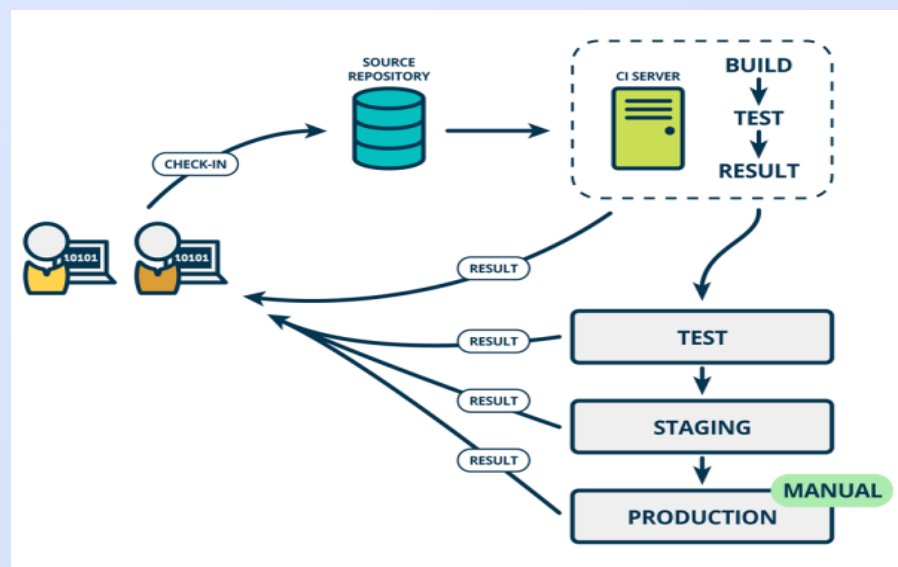


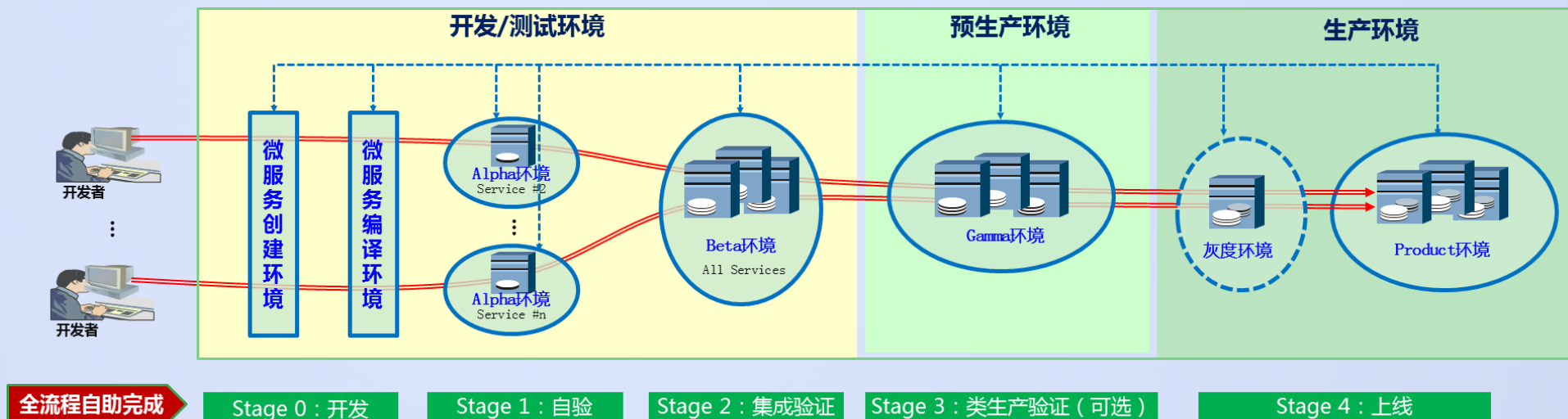
图 持续交付



# ServiceStage持续交付解决方案

基于ServiceStage流水线实现应用全流程“自助式”开发、集成、验证与上线。

- 一键生成持续交付环境  
自动生成应用框架代码、构建、部署及测试环境。
- 多语言应用  
Java、go、node.js、php、python、ruby、.net。
- 多种源码仓库  
DevCloud、GitHub、Gitee、GitLab、Bitbucket。



以“月/季度/半年”为周期 → 以“天/小时”为周期



# 应用案例之云上工程

**ServiceStage云上工程**，快速生成持续交付环境，一键式创建应用框架代码、构建、部署及流水线

- 对接多种源码仓库(DevCloud、GitHub、Gitee、GitLab和Bitbucket等)
- 对接多种编程语言(Java、Go、Node.js、PHP、Python和Ruby等)
- 对接多种热门框架 ( ServiceComb、SpringBoot、Gosimple-Webapp、Node-Express、Python-Django等 )



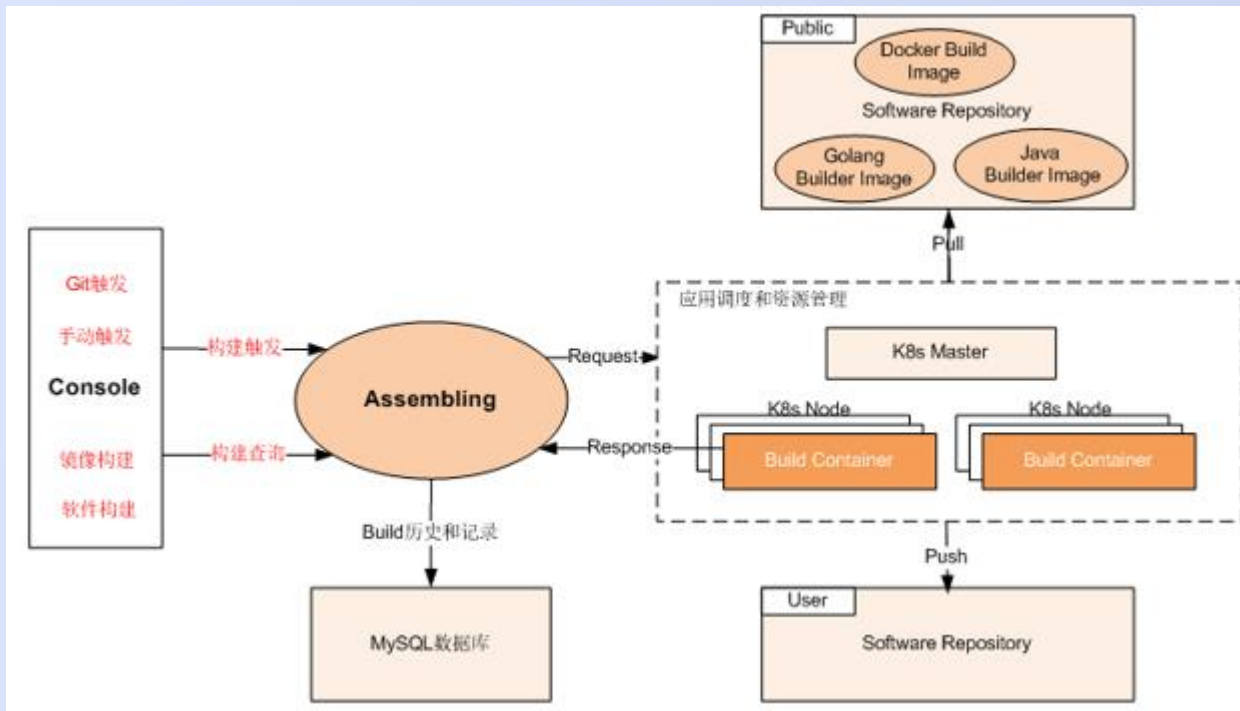
# 应用案例之构建

**ServiceStage构建**，是一个容器化的镜像/软件包构建解决方案

- 可用于编写源代码、运行测试并生成可立即部署的软件包或镜像
- 支持Java、go、node.js、php、python、ruby、.net等
- 无需安装、配置及管理私人构建服务器

**优点：**

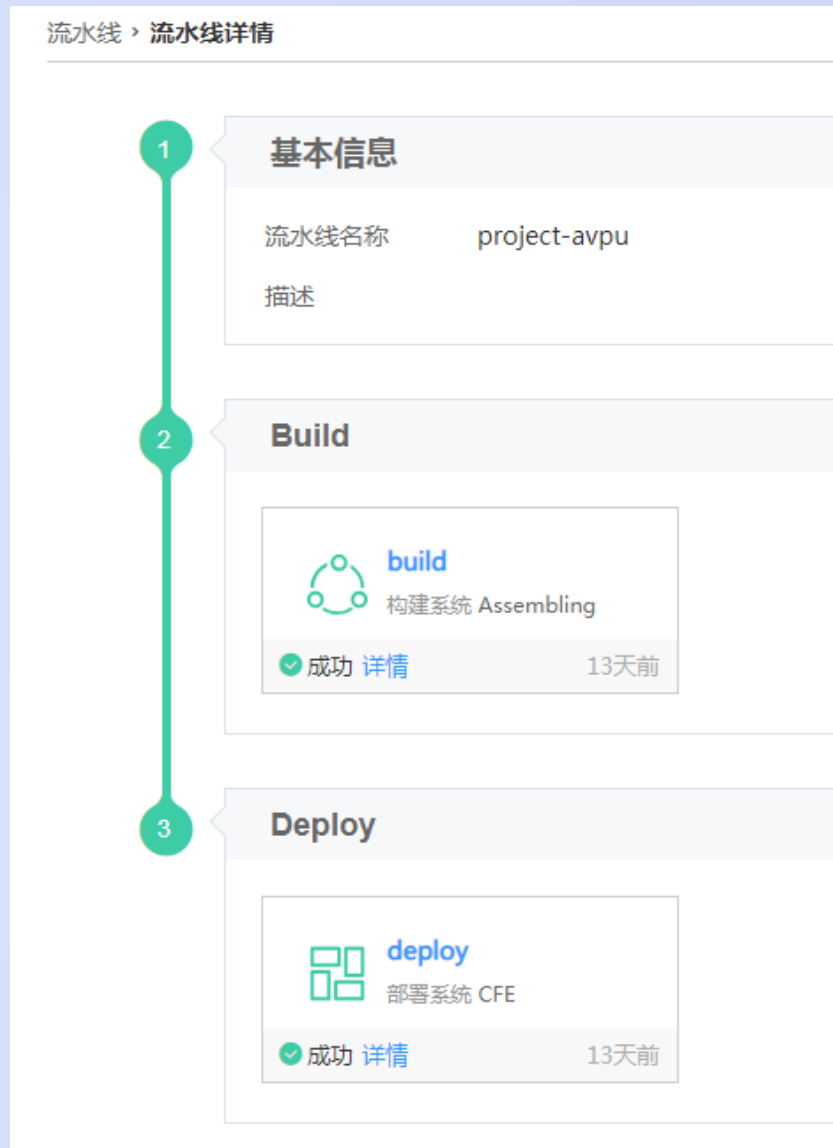
- ✓ 完全托管
- ✓ 预置编译环境（Java/Go/Docker等）
- ✓ 可扩展编译环境
- ✓ 支持标准Docker Registry
- ✓ 默认推送PaaS仓库
- ✓ 支持自定义编译命令



# 应用案例之流水线

**ServiceStage流水线**，实现快速、可靠的应用程序和基础设施更新

- 提供流程和任务灵活定制能力，提供可视化图形用户界面，支持开发者自定义应用发布流程。
- 提供对主流工具的默认支持，包括从源码、构建、部署、测试阶段核心组件
- 提供工具快速接入能力，可定制对接客户已有工具，服务领域开发工具对接
- 支持多租户，通过IAM用户、IAM角色授予用户对流水线的访问权限
- 高性能高并发，无状态多实例轻量级调度流水线，每个实例都可以扩展



# Thank You





# 21天微服务实战营

华为云DevCloud & ServiceStage服务联合出品



# DAY17 微服务应用开发之本地工具

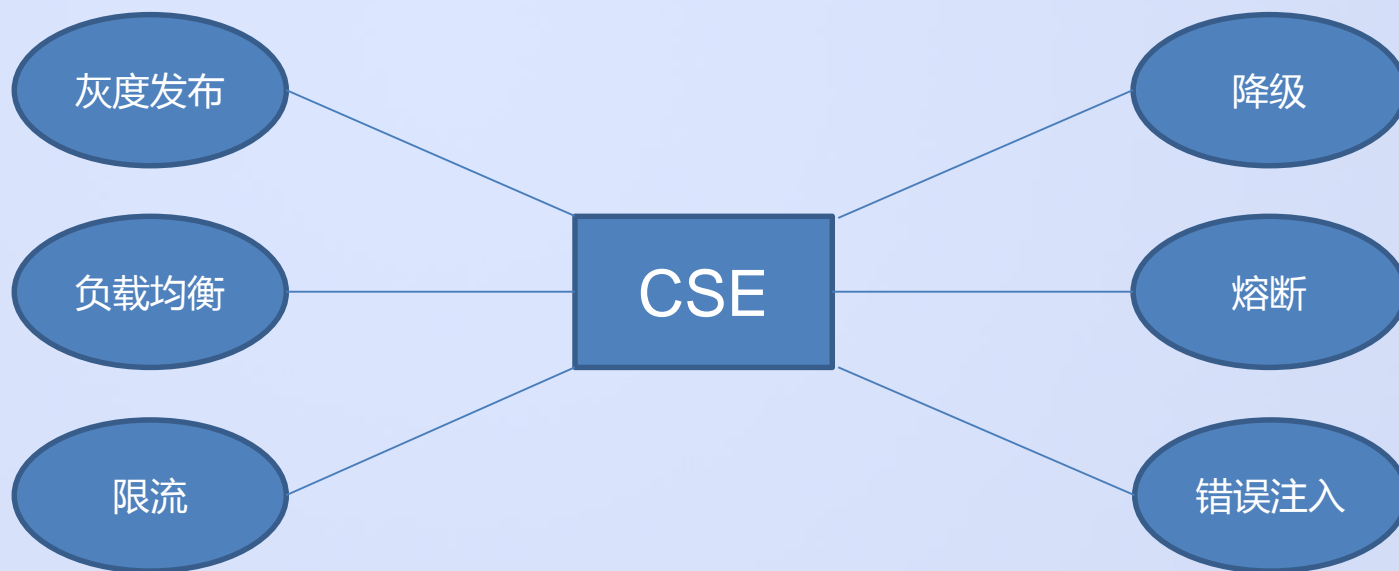
本节介绍的内容主要包括：

- 微服务框架简介
- 本地轻量化服务中心简介
- Mesher简介
- 远程调试工具简介
- 密钥生成工具简介
- 本地轻量化微服务引擎简介
- Eclipse ServiceStage插件简介

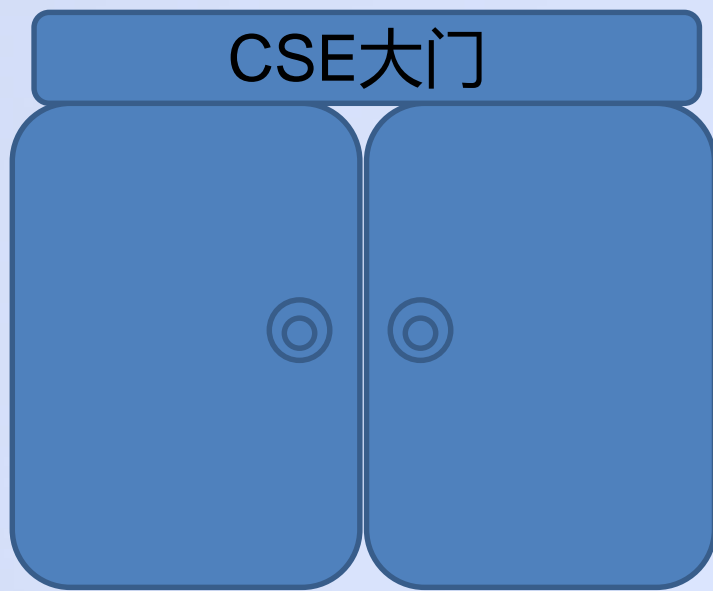
# 微服务框架之CSE Java SDK

CSE Java SDK开发微服务(简称CSE)，可以最大化的简化开发门槛，提升产品上线速度。同时可以获得微服务运行时高可靠性保证、运行时动态治理等一系列开箱即用的能力。

该框架主要拥有的能力

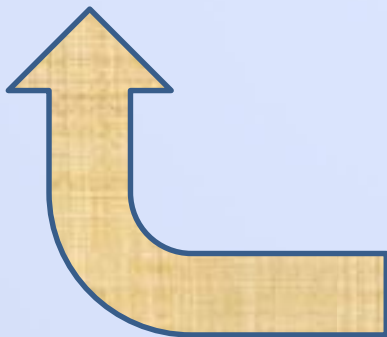


# 微服务框架之CSE Java SDK



CSE支持使用调用链(对服务的调用进行监控),支持使用仪表盘(查看自己的运行相关数据),还支持使用分布式事务TCC、文件上传等。

不仅我们强大的ServiceComb应用可以接入到我们CSE中去,连我们众所周知的Spring Cloud应用也可以方便的接入到CSE提供的基础服务。



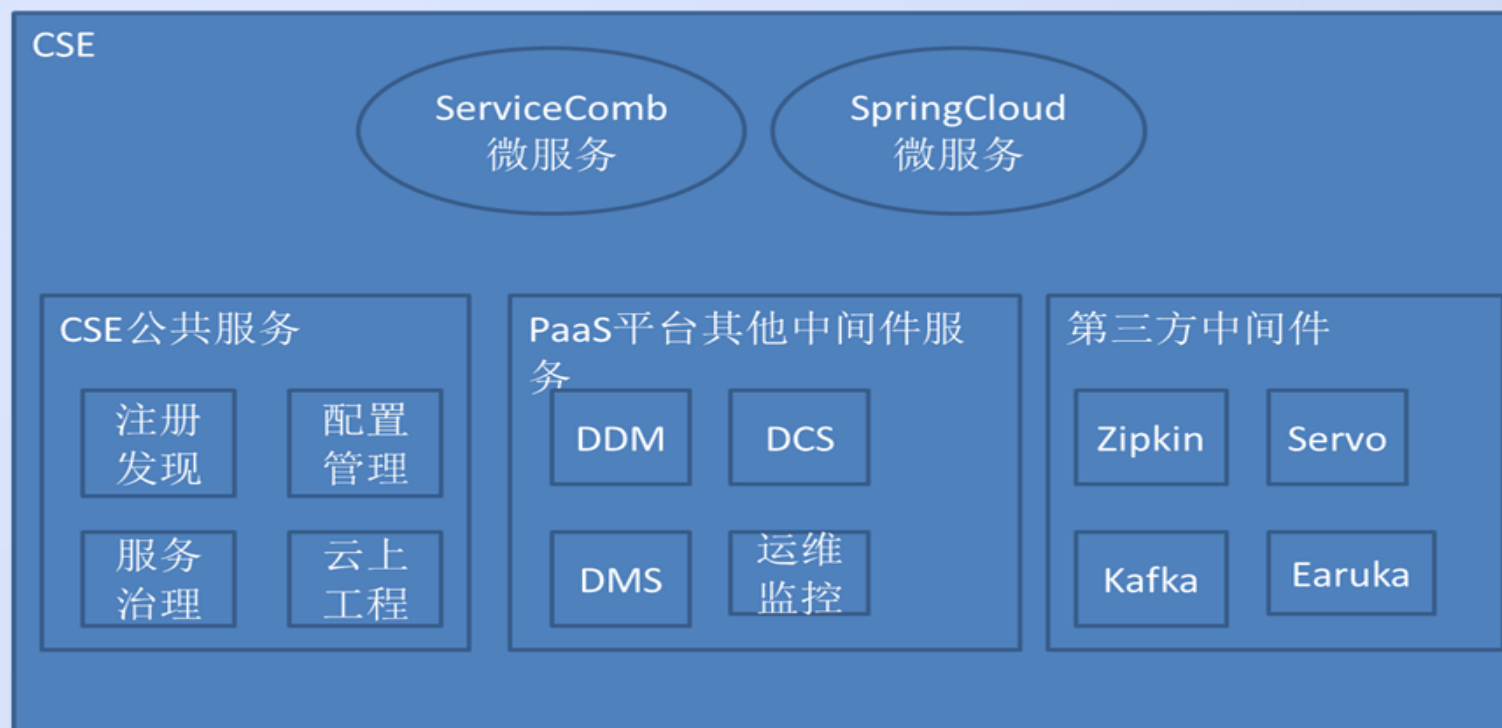


# 微服务框架之CSE Java SDK

接入CSE服务有如下好处：

1. 开发者可以专注于业务系统的开发，把精力从中间件的可靠性评估、集群部署、运维监控等复杂的事情中解放出来。
2. 实现业务快速交付和敏捷开发。利用PaaS平台，根据业务规模，动态的调整资源使用，降低业务风险。

CSE基础服务、PaaS平台服务和第三方服务的关系



# 微服务框架之CSE Go SDK

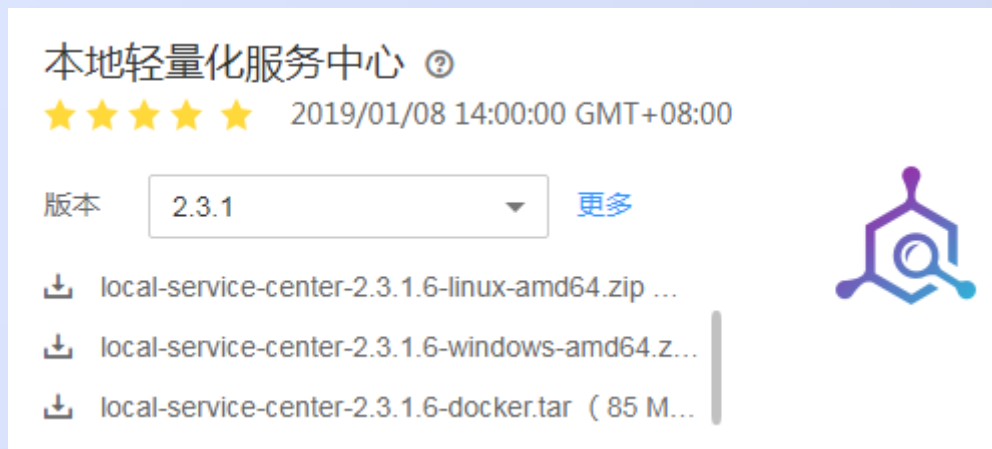
CSE(Cloud Service Engine) Go Chassis是华为推出的产品级微服务开发框架。使用CSE Go Chassis开发微服务，可以最大化的简化开发门槛，提升产品上线速度。同时可以获得微服务运行时高可靠性保证、运行时动态治理等一系列开箱即用的能力。



# 本地轻量化服务中心

LocalServiceCenter是CSE微服务框架的注册中心, 记录了服务和地址的映射关系, 在分布式架构中, 服务会注册到这里, 当服务需要调用其它服务时, 就到这里找到服务的地址, 进行调用。本地轻量化服务中心可用于本地开发调试, 其作用相当于是本地起的一个Eureka。

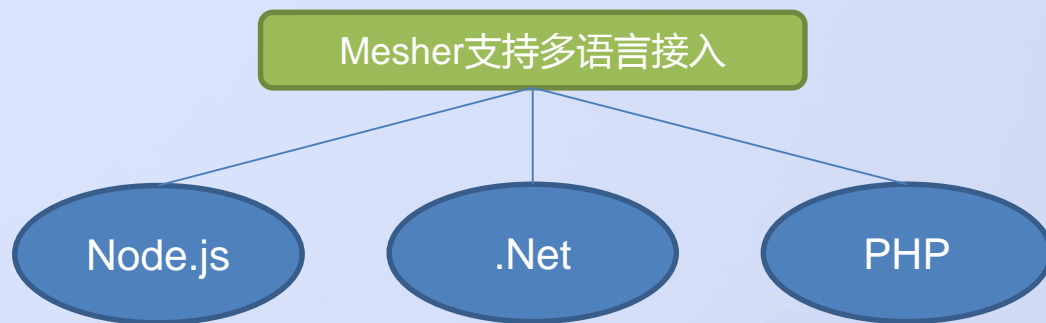
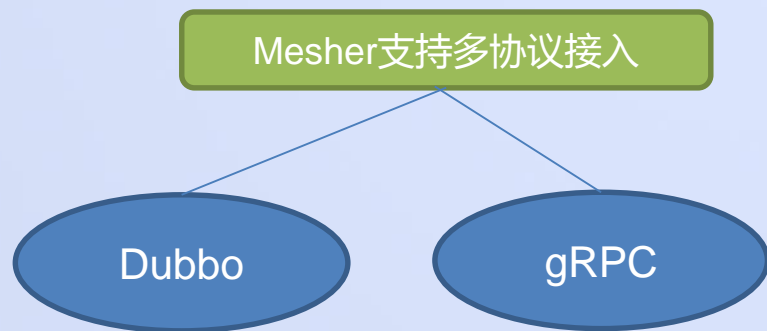
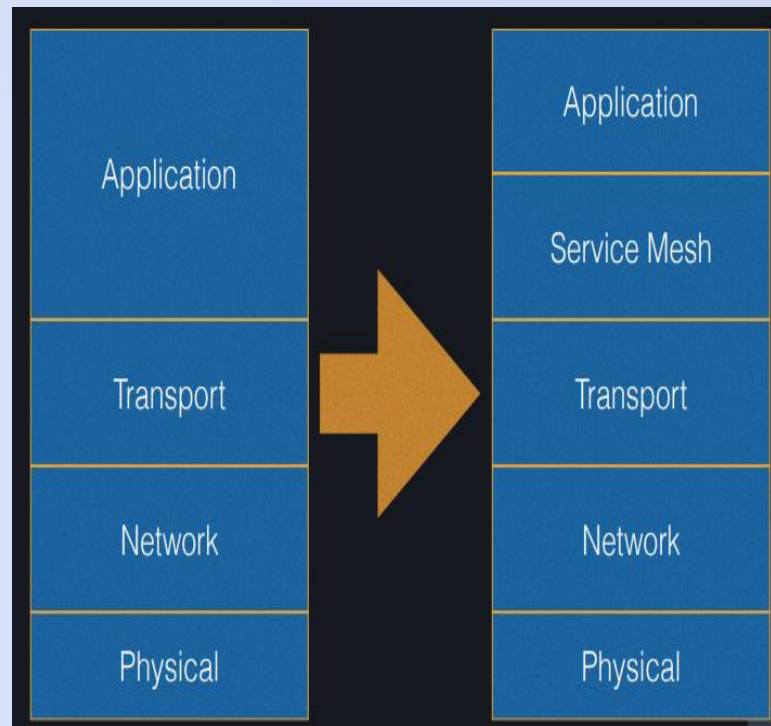
有容器版本, Windows版本和Linux版本, 用户可以根据自己的场景选择合适的版本



# Mesher

Mesher是Service Mesh的一个具体的实现，是一个轻量的代理服务以Sidecar的方式与微服务一起运行。

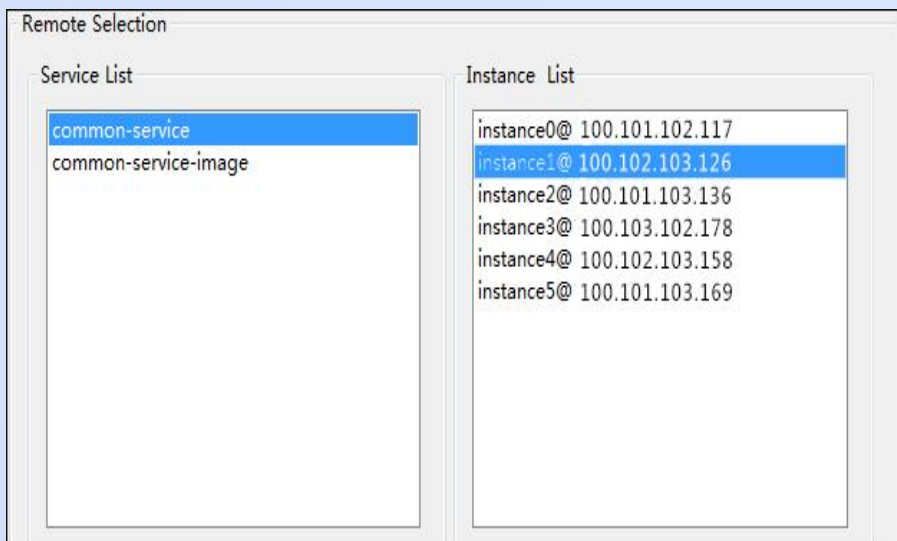
Service Mesh是一个基础设施层，用于处理服务间通信。云原生应用有着复杂的服务拓扑，Service Mesh保证请求可以在这些拓扑中可靠地传输。在实际应用当中，Service Mesh通常是由一系列轻量级的网络代理组成的，它们与应用程序部署在一起，但应用程序不需要知道它们的存在。



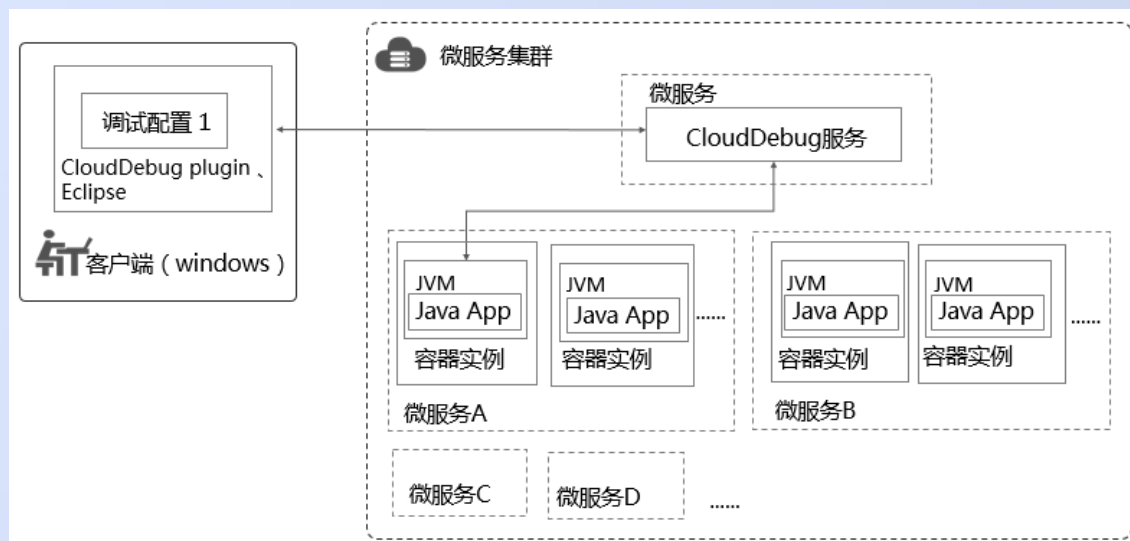
# 远程调试工具

云上调试（CloudDebug）主要用于支撑华为云的租户调试人员对单个集群内微服务实例中的Java程序进行远程调试。（是一个eclipse插件,可以直接在eclipse上操作,对集群的微服务进行查询和调试）

**查询功能：**可对集群内所有的微服务进行查询，查询结果展示在插件UI界面中的“Service List”中。针对某一微服务，查询属于该微服务的所有实例名和实例的集群内IP地址，并将查询结果展示在插件UI界面中的“Instance List”中。



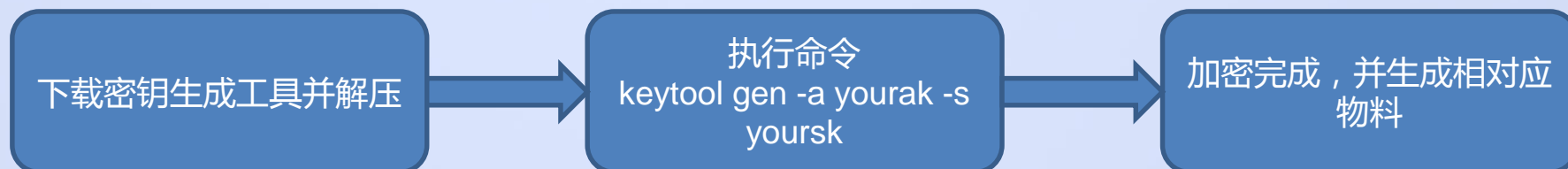
**远程调试功能：**CloudDebug主要设计用于对华为云ServiceStage租户集群内微服务实例中的Java程序进行远程调试，支持所有通用Java调试功能，如程序断点、变量查看、堆栈查看等。



# 密钥生成工具

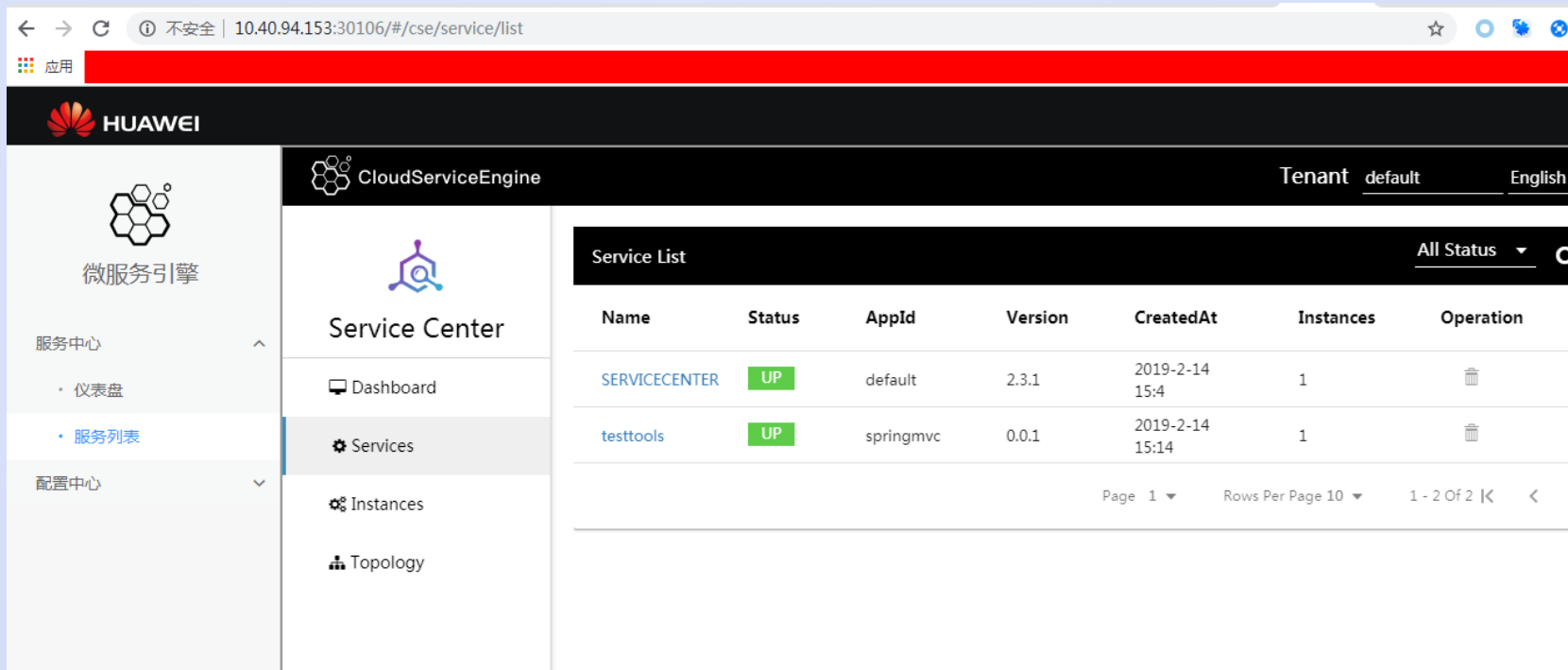
密钥生成工具是一个加密工具包, 基于共享密钥的AES256加解密存储方案, 通过工具生成密钥物料, 然后使用工具利用密钥文件对指定的明文进行加密。例如可以使用这种方法对数据库密码进行加密, 使用的时候再使用CSE SDK接口进行解密。

在ServiceStage场景中, 我们经常会用这个密钥生成工具对我们AK/SK进行加密。



# 本地轻量化微服务引擎

本地轻量化微服务引擎是集成了本地轻量化服务中心和配置中心及console界面,下载该zip包,解压并运行,我们可以直接在本地图微服务,将微服务注册在本地的注册中心及配置中心,看到服务的注册情况及运行情况。  
下图就是本地的console界面：



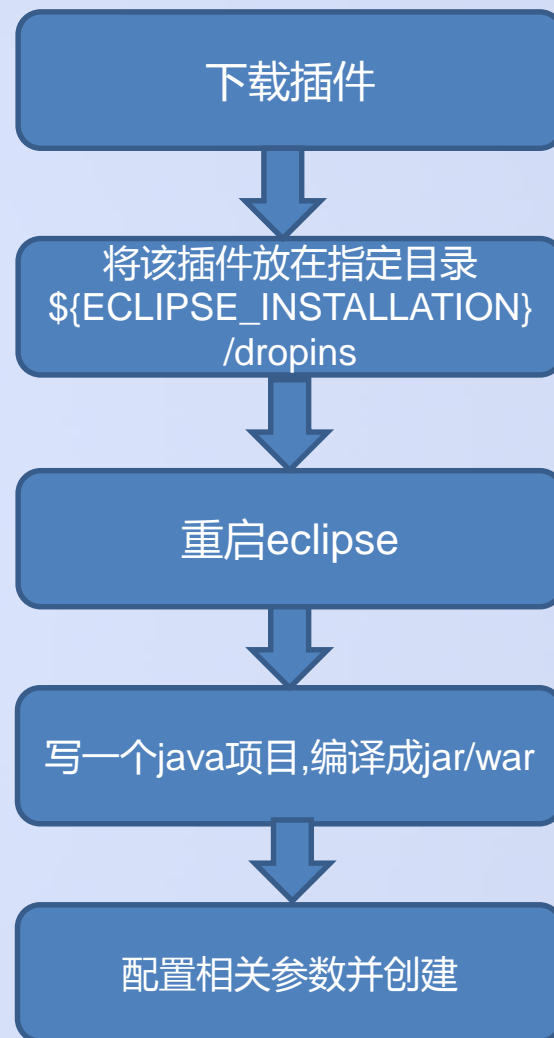
The screenshot displays the Huawei CloudServiceEngine console interface. The top navigation bar includes the Huawei logo, the product name 'CloudServiceEngine', and options for 'Tenant' (default) and 'English'. The left sidebar contains a 'Service Center' menu with sub-items like '仪表盘' (Dashboard) and '服务列表' (Service List). The main content area shows a 'Service List' table with columns for Name, Status, AppId, Version, CreatedAt, Instances, and Operation. Two services are listed: 'SERVICECENTER' and 'testtools', both with a status of 'UP'.

Name	Status	AppId	Version	CreatedAt	Instances	Operation
SERVICECENTER	UP	default	2.3.1	2019-2-14 15:4	1	
testtools	UP	springmvc	0.0.1	2019-2-14 15:14	1	

Page 1 Rows Per Page 10 1 - 2 Of 2

# Eclipse ServiceStage插件

Eclipse ServiceStage插件使应用开发者能轻易实现在本地eclipse上与华为云微服务云应用平台的集成,能够在eclipse上对servicestage的应用进行配置、创建、更新及应用状态的查询。该插件暂只支持Tomcat和Node.js应用.





# Thank You





# 21天微服务实战营

华为云DevCloud & ServiceStage服务联合出品



# DAY18 微服务应用实战之快速上线

本节介绍的内容主要包括：

- 一键式部署
- 模板生成微服务
- 多代码仓库支持
- 多语言、多运行环境支持
- 编译构建
- 多部署引擎支持
- 中间件集成

# 一键式部署

ServiceStage作为华为云的应用管理平台，为微服务上线提供了端到端的流程



# 模板生成微服务

对于微服务的新创建场景，ServiceStage提供了主流框架的模板，生成微服务工程骨架，用户只需聚集业务实现



CSE-Java (SpringMVC)

2.3.35

基于CSE框架，支持SpringMVC注解，...



CSE-Java (JAX-RS)

2.3.35

基于CSE框架，支持JAX-RS注解，使用...



CSE-Java (POJO)

2.3.35

基于CSE框架，支持接口和接口实现的...



SpringBoot-WebApp

SpringBoot-1.5.15.RELEASE

基于Spring Boot的Web应用

后续上线



SpringBoot-WebService

SpringBoot-1.5.15.RELEASE

基于Spring Boot的Web Service

后续上线



SpringCloud-Consumer-Micro...

SpringBoot-1.5.10.RELEASE

微服务，服务消费者

后续上线



SpringCloud-Provider-Micros...

SpringBoot-1.5.10.RELEASE

微服务，服务提供者

后续上线



SpringBoot-Webapp-Tomcat

SpringBoot-1.5.3.RELEASE

Web应用，运行于独立部署的Web服务...



SpringBoot-WebService-Tom...

SpringBoot-1.5.3.RELEASE

Web Service，运行于独立部署的Web...



Laravel

laravel-v5.6.28

简洁、优雅，为WEB艺术家创造的PHP...



Slim

slim-3.10.0

快速、轻量的微型PHP框架



Express

express-4.16.0

高度包容、快速而极简的 Node.js Web ...



Koa

koa-2.5.2

基于 Node.js 平台的下一代 web 开发...

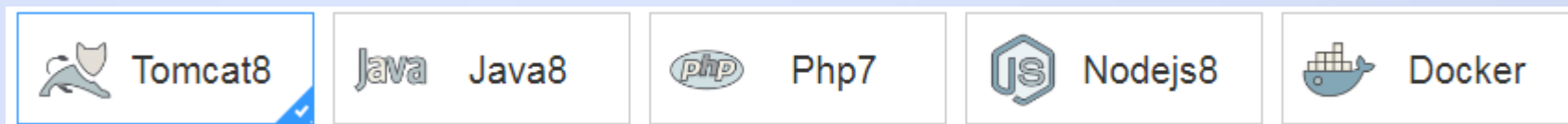
# 多代码仓库支持

ServiceStage支持直接从代码部署微服务，对接主流代码仓库github，gitlab等通过OAuth授权访问



# 多语言、多运行环境支持

支持多种主流开发语言：java, php, nodejs, 同时提供docker镜像以支持更灵活的定制场景



# 编译构建

- 1、自动根据代码语言进行编译环境准备，集成 maven，gradle，ant 等构建工具。
- 2、自适应的编译命令。如对于 java，指定编译命令 > build.sh > mvn > gradle > ant
- 3、docker 镜像构建。如果没提供 Dockerfile，自动生成

```
[ -n "${BUILD_COMMANDS}" ] && {
    BuildWithCommands && PostBuild
    return $?
}
[ -f build.sh ] && {
    echo '> build with build.sh ...'
    bash build.sh && PostBuild
    return $?
}
[ -f pom.xml ] && {
    echo '> build with maven...'
    mvn -q -e clean package && PostBuild
    return $?
}
[ -f build.gradle ] && {
    echo '> build with gradle...'
    gradle build --info assemble && PostBuild
    return $?
}
[ -f build.xml ] && {
    echo '> build with ant...'
    ant && PostBuild
    return $?
}
```

```
function genDockerfile() {
    cp -f ${current_path}/.dockerignore ${WORKSPACE_HOME}
    cat > ${WORKSPACE_HOME}/Dockerfile <<EOF
# Copyright(C) 2018 Huawei Technologies Co., Ltd. All Rights Reserved
FROM ${BASE_IMAGE_NAME}
WORKDIR /var/${SERVICE_NAME}
COPY ./ /var/${SERVICE_NAME}/
$(GetDockerCMD 'java')
EOF
}

dockerfile=$(find ${WORKSPACE_HOME} -type f -name "Dockerfile")
[ -z "${dockerfile}" ] && {
    echo '> generate Dockerfile...'
    genDockerfile || exit 1
}
```



# 多部署引擎支持

ServiceStage集成CCE，CCI，虚拟机3个部署引擎，封装了底层引擎的复杂，聚集业务应用本身



# 中间件集成

集成RDS，DCS，微服务引擎等中间件，提供一站式使用体验

★ 负载均衡

新建负载均衡 **elb-h3u4** [更换配置](#)

应用性能管理

[应用性能管理服务](#) 可协助您快速进行应用问题定  
Java 探针  通过字节码增强技术进行Java

数据库

☒ 分布式会话

新建分布式缓存服务 **dcs-umd8** [更换配置](#)

☒ 关系型数据库

新建关系型数据库 **rds-4jh3** [更换配置](#)

★ 内网安全组

新建安全组 **casapp-x46mza**

# Thank You





# 21天微服务实战营

华为云 DevCloud & 微服务产品 联合出品



# DAY19 微服务应用实战之服务治理

本节介绍的内容主要包括：

- 治理能力
- 负载均衡
- 限流
- 降级
- 灰度发布

# 治理能力

基于ServiceComb框架，对微服务提供了负载均衡，限流，降级，容错，熔断，错误注入，灰度发布等治理能力。

# 负载均衡

基于Ribbon的负载均衡方案，  
支持随机、顺序、基于响应时间的权值等多种负载均衡路由策略

# 限流

为避免个别接入流量的峰涌导致系统的崩溃，可以配置限流策略

用户在provider端使用限流策略，可以限制指定微服务向其发送请求的频率，达到限制每秒钟最大请求数量的效果



# 降级

降级策略是当服务请求异常时，微服务所采用的异常处理策略。

降级策略有三个相关的技术概念：“隔离”、“熔断”、“容错”：

“隔离”是一种异常检测机制，常用的检测方法是请求超时、流量过大等。一般的设置参数包括超时时间、同时并发请求个数等。

“熔断”是一种异常反应机制，“熔断”依赖于“隔离”。熔断通常基于错误率来实现。一般的设置参数包括统计请求的个数、错误率等。

“容错”是一种异常处理机制，“容错”依赖于“熔断”。熔断以后，会调用“容错”的方法。一般的设置参数包括调用容错方法的次数等。

把这些概念联系起来：当“隔离”措施检测到N次请求中共有M次错误的时候，“熔断”不再发送后续请求，调用“容错”处理函数。这个技术上的定义，是和Netflix Hystrix一致的，通过这个定义，非常容易理解它提供的配置项，参考：

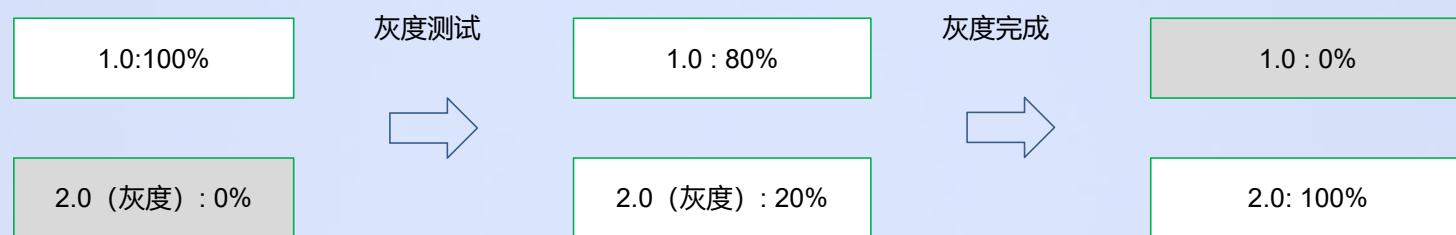
<https://github.com/Netflix/Hystrix/wiki/Configuration>。

当前ServiceComb提供两种容错方式，分别为返回null值和抛出异常。

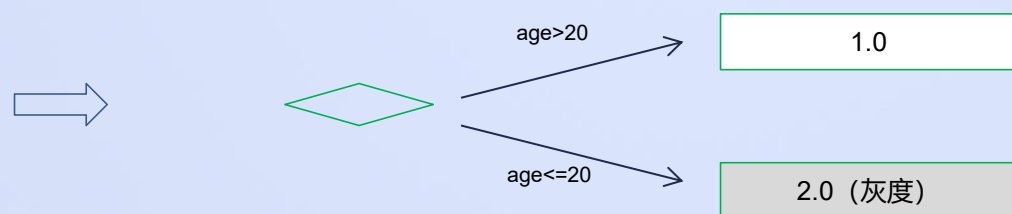
# 灰度发布

ServiceComb微服务支持两种灰度策略

## 1、流量权重：



## 2、自定义参数：根据接口参数进行灰度导流



# Thank You





# 21天微服务实战营

华为云DevCloud & ServiceStage服务联合出品



# DAY20 微服务应用运维之应用监控

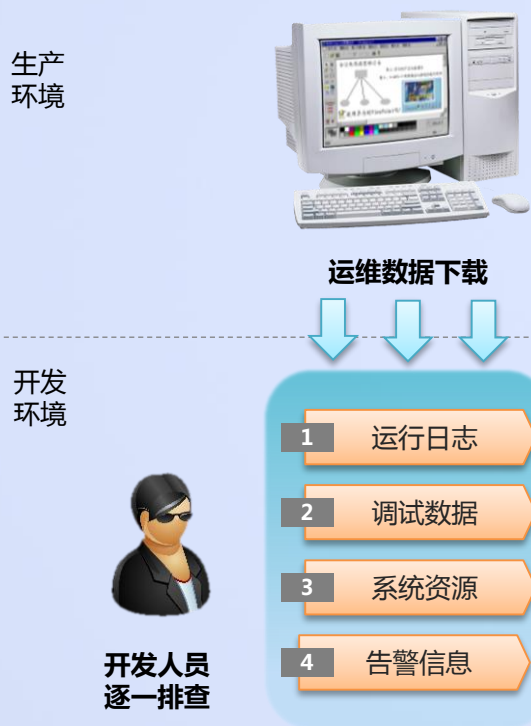
本节介绍的内容主要包括：

- 云化场景下分布式应用的运维挑战
- 云化场景下应用的统一运维监控
- 全链路监控：覆盖从Browser与Mobile端侧到数据中心全链路
- 应用拓扑分析：应用关系与异常一目了然、故障下钻
- 业务会话监控：监控每笔交易的KPI数据，提升用户体验
- 非侵入式数据采集：一键式采集部署，用户无感知

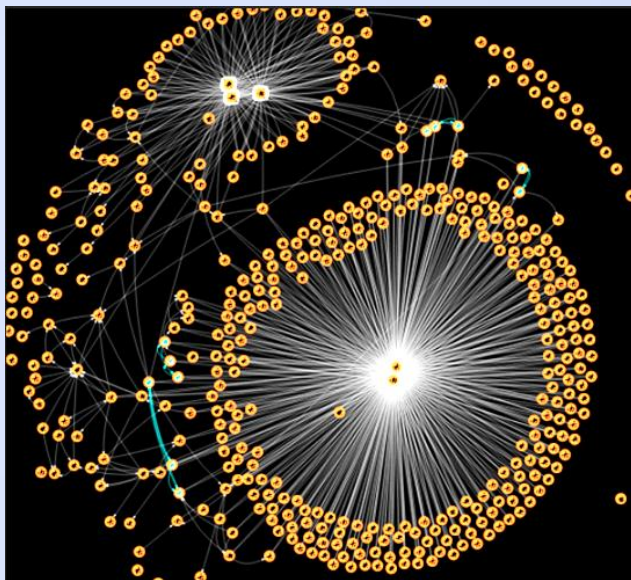
# 云化场景下分布式应用的运维挑战

相比传统的开发运维模式，云化场景下的分布式微服务应用关系更为复杂，随着应用复杂度的不断提升、用户数量的不断增加，海量业务下如何保障应用正常、如何快速完成问题定位、如何快速找到性能瓶颈，已经成为应用运维的巨大挑战。

## 传统运维模式



## 大型分布式应用关系错综复杂



1. 微服务依赖关系能否可视化？
2. 最终用户体验如何？
3. 问题如何快速追踪？
4. 散落的日志无法关联分析？



# 云化场景下应用的统一运维监控

统一运维监控管理：资源、应用、业务一站式监控与分析。



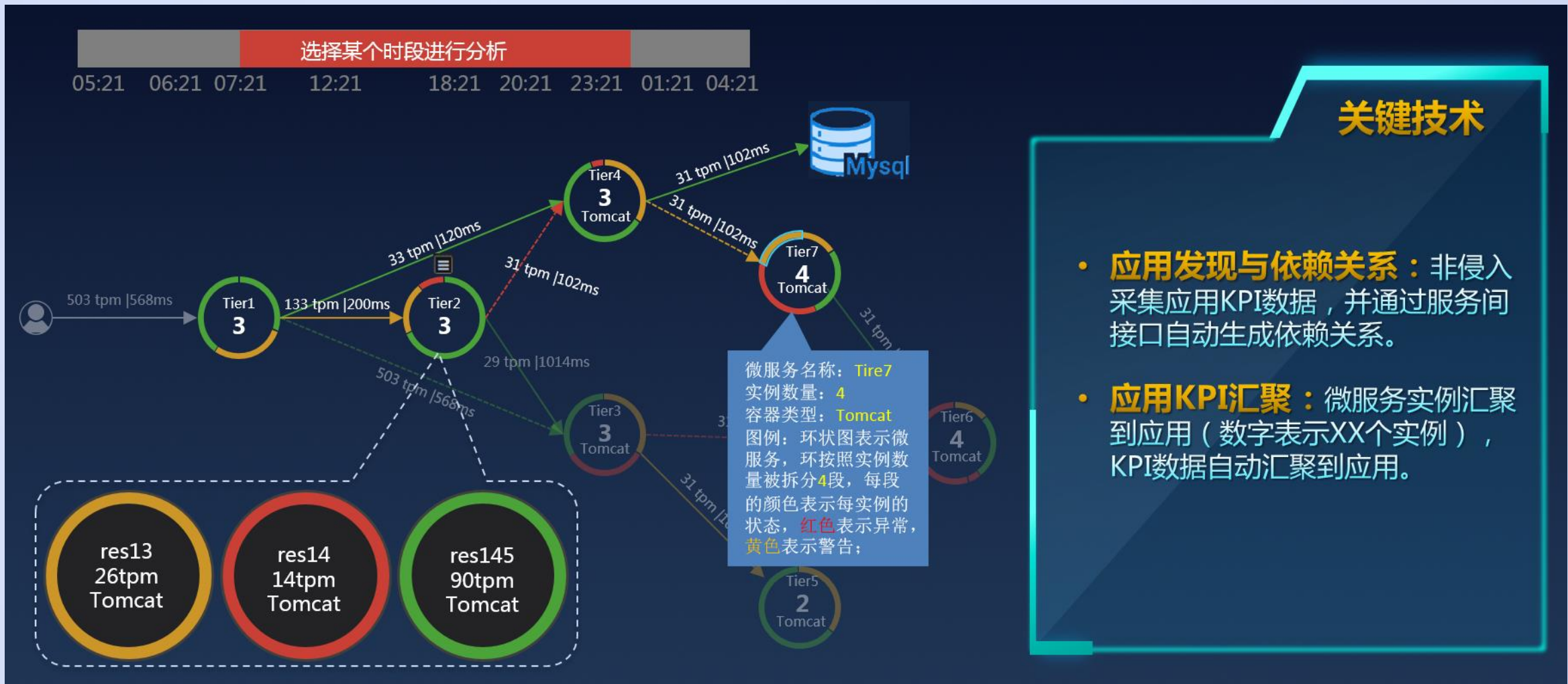
立体运维全景图

# 全链路监控：覆盖从Browser与Mobile端侧到数据中心全链路





# 应用拓扑分析：应用关系与异常一目了然、故障下钻



# 业务会话监控：监控每笔交易的KPI数据，提升用户体验

## 关键技术

实时跟踪每条业务交易，快速分析交易的运行状态并提供诊断能力

- **自定义事务**：用户可根据每条URL定义事务名称，方便理解。
- **健康规则配置**：可以对每条事务配置健康规则，如超过1s提示异常；
- **性能追踪**：精确采集异常性能数据，可对比历史基线数据，也能找到应用的异常方法，提升运维效率。

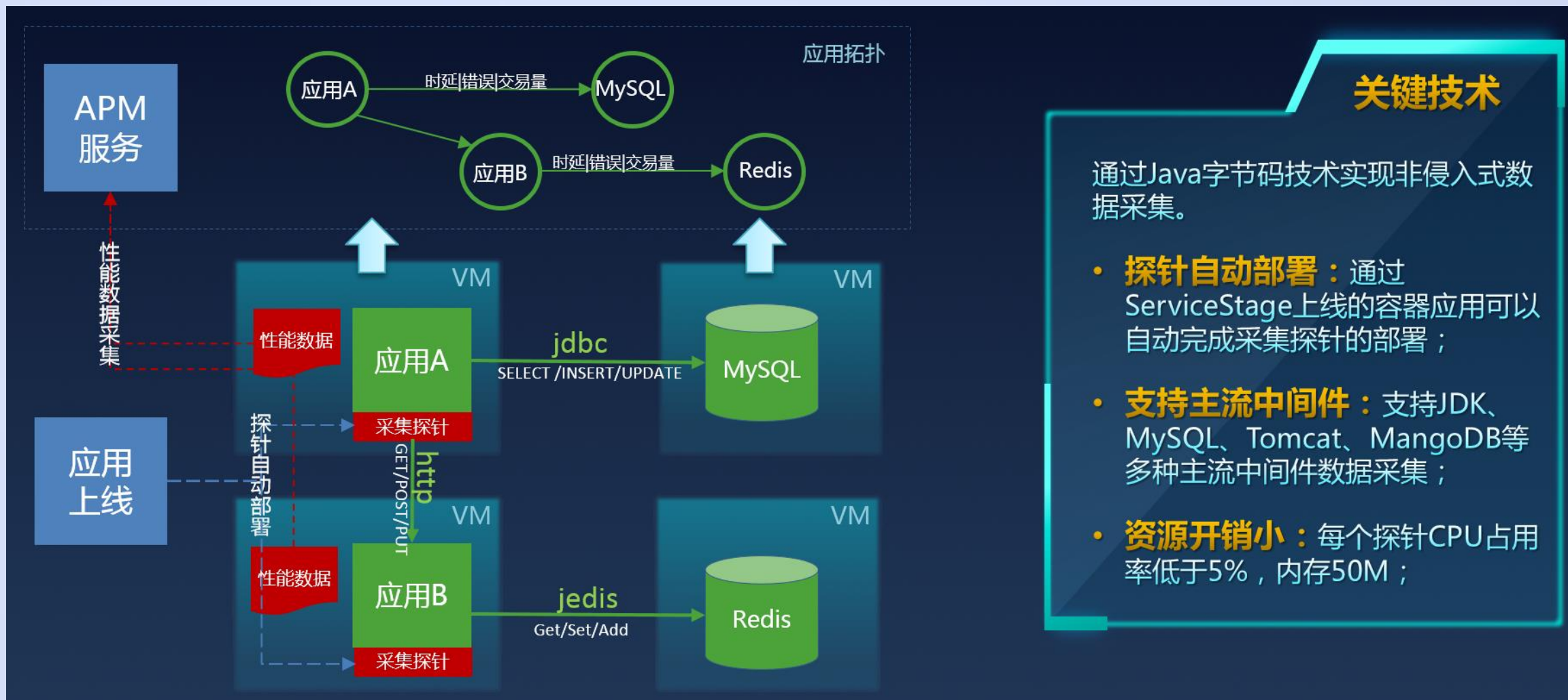


事务URL	事务名称	吞吐率 (次/分钟)	平均时延	健康指标	错误率	性能分析
1 vmall.com/CAS/portal/login.html?validated	登陆	52	323ms	正常	0%	<a href="#">调用链分析</a>
2 vmall.com/portal/search?id=34211223411	商品搜索	234	721ms	较慢	1%	<a href="#">调用链分析</a>
3 vmall.com/portal/buy?id=34211223411	购买	3	1.32s	正常	0%	<a href="#">调用链分析</a>
4 vmall.com/portal/pay?id=34211223411	支付	1	2.1s	异常	100%	<a href="#">调用链分析</a>

★ 通过对业务监控，用户可以了解最终**消费者行为**，用于**业务发展决策**；其次可以**快速发现业务运营的状态**，对于异常的应用程序快速诊断；



# 非侵入式数据采集：一键式采集部署，用户无感知



# Thank You





# 21天微服务实战营

华为云DevCloud & ServiceStage服务联合出品



# DAY21 微服务应用运维之调用追踪

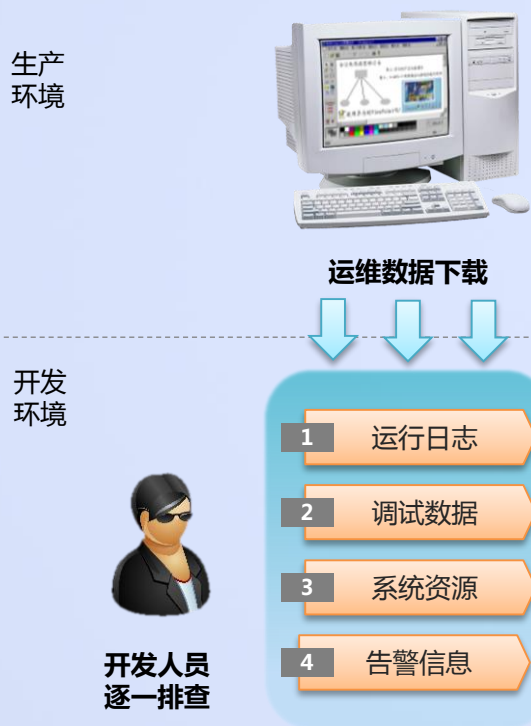
本节介绍的内容主要包括：

- 云化场景下分布式应用的运维挑战
- 云化场景下应用的调用链跟踪
- 调用追踪之性能瓶颈定界
- 调用追踪之故障辅助定位

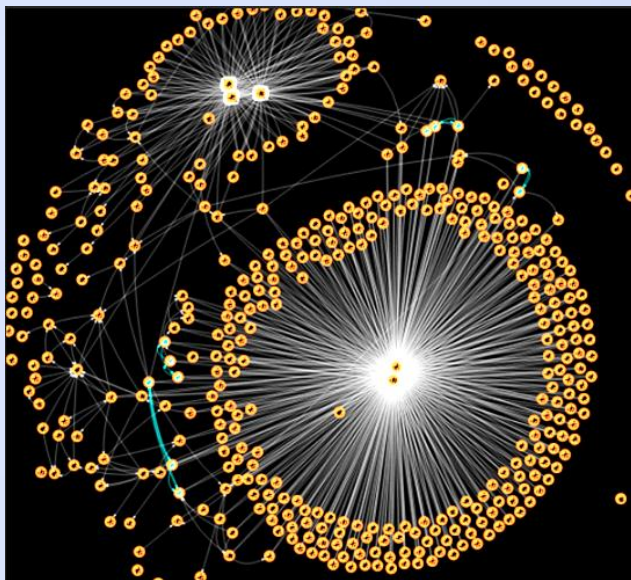
# 云化场景下分布式应用的运维挑战

相比传统的开发运维模式，云化场景下的分布式微服务应用关系更为复杂，随着应用复杂度的不断提升、用户数量的不断增加，海量业务下如何保障应用正常、如何快速完成问题定位、如何快速找到性能瓶颈，已经成为应用运维的巨大挑战。

## 传统运维模式



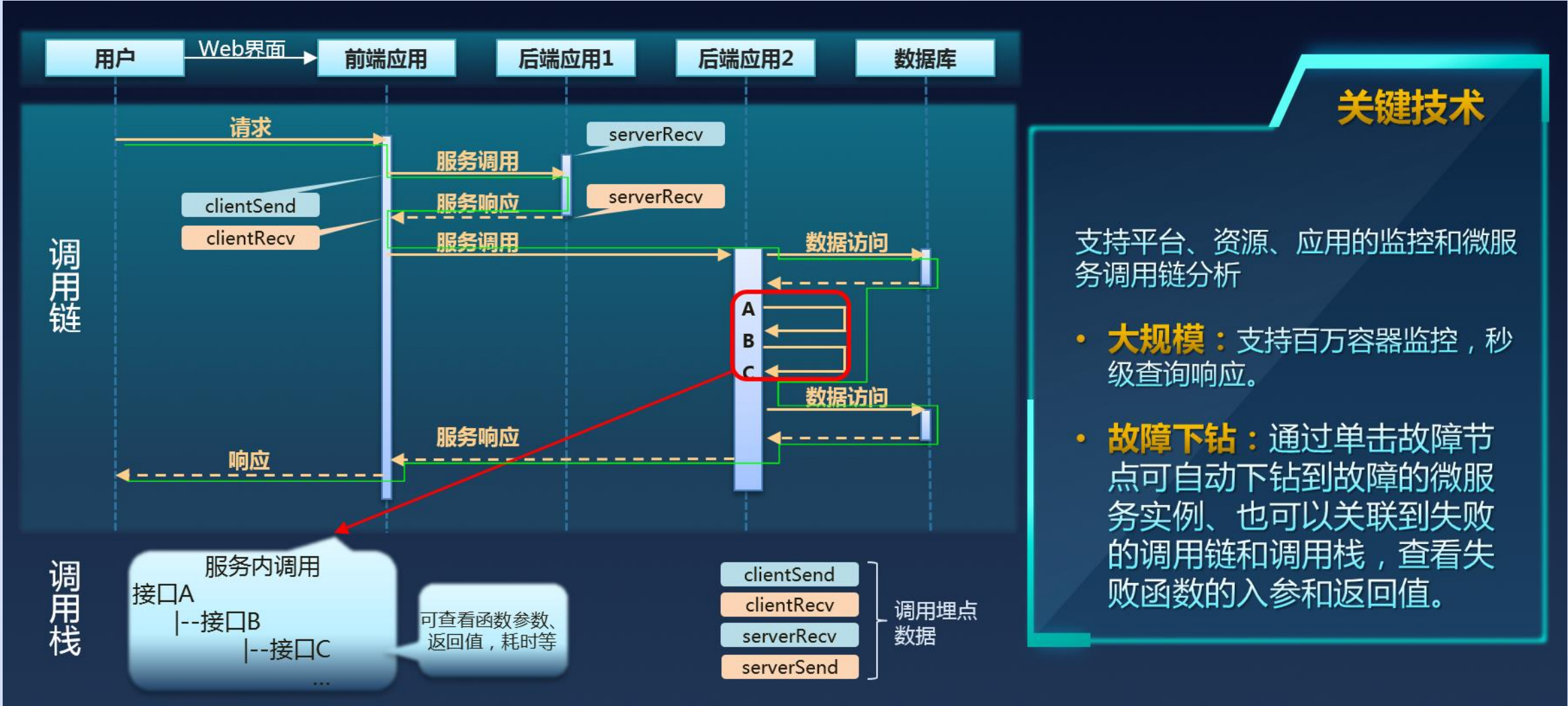
## 大型分布式应用关系错综复杂



1. 微服务依赖关系能否可视化？
2. 最终用户体验如何？
3. 问题如何快速追踪？
4. 散落的日志无法关联分析？



# 云化场景下应用的调用链跟踪

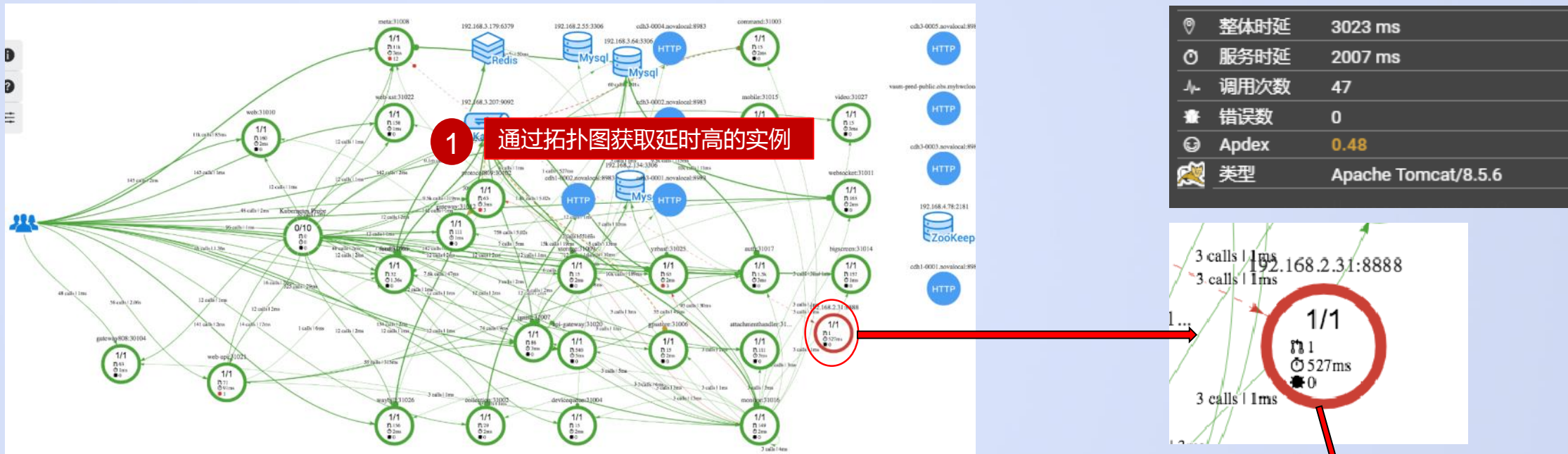




# 调用追踪之性能瓶颈定界

## 场景：找出系统的性能瓶颈点

问题背景：客户系统出现性能问题，某个接口调用耗时过长。使用调用追踪进行性能分析。  
问题价值点：客户表示之前有一处代码使用了递归逻辑，通过调用追踪发现时延800ms，优化代码后下降到20ms，认为确实可以发挥很大作用。



## 2 下钻到对应的调用链寻找耗时最高的调用链信息进行优化

sun.net.www.protocol.http.httpURLConnection.getin...	sun.net.www.protocol.http.httpURLConnection.getInputStream	1	扩展信息
org.apache.catalina.core.StandardHostValve.in...	/persistence/user	0	扩展信息
org.springframework.web.servlet.frameworko...		2	扩展信息
com.huawei.cloud.api.persistence.esc...		0	扩展信息
com.mysql.jdbc.NonRegisteringDriver.co...		423	扩展信息
com.mysql.jdbc.PreparedStatement.ex...	SELECT * FROM...	1	扩展信息

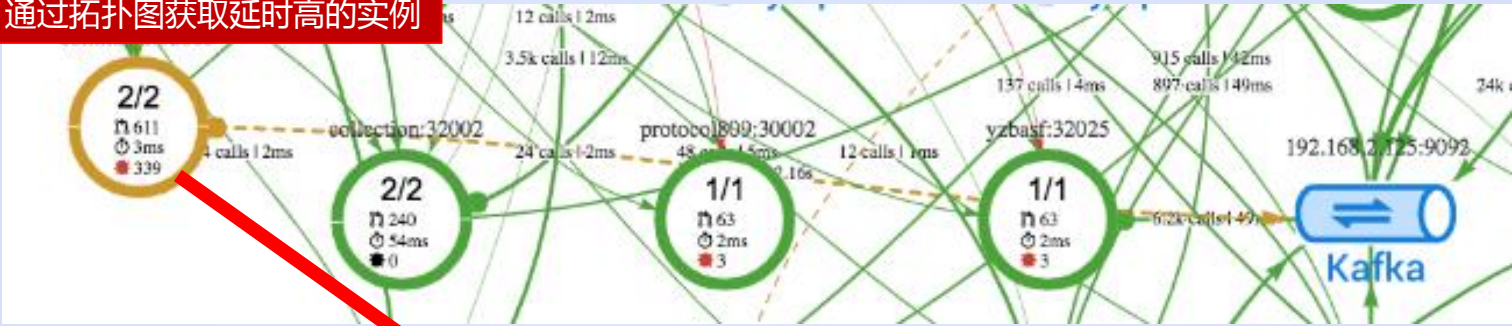
# 调用追踪之故障辅助定位

## 场景：定位系统异常

问题背景：客户微服务有30个，系统某次调用出现问题，对于这类复杂的系统，需要借助调用追踪功能进行定位。

问题价值点：对于复杂的系统，通过调用追踪可以迅速找到出现问题的实例，避免客户运维逐一排查。大大节省客户运维时间，提高运维效率。

### 1 通过拓扑图获取延时高的实例



### 2 下钻到出现错误的调用链，查看调用链详情进行定位

调用链 > 调用关系

服务数: 1 调用深度: 5 总Span: 5

服务	方法
storage	org.apache.talinala.cc
storage	org.springframework
storage	com.yudao.veh
storage	org.apache
storage	org.apache
storage	org.springframework.w

扩展信息

类型	描述
TX-TYPE	POST_/alarm/handle
clusterId	54a92635-8f10-11e8-ac7e-0255...
exception.class	java.net.UnknownHostException
exception.msg	cdh3-0003.novalocal: Name or s...
http.internal.display	cdh3-0003.novalocal:8983
monitorGroup	prod
namespace	prod
result	1
root	false
serviceType	HTTP CLIENT 4 INTERNAL

时间线(ms)

时间线(ms)	操作
3	扩展信息 查看
4	扩展信息 查看
1	扩展信息 查看
1	扩展信息 查看
7	扩展信息 查看
0	扩展信息 查看
2	扩展信息 查看

# Thank You

