

微服务架构进化论



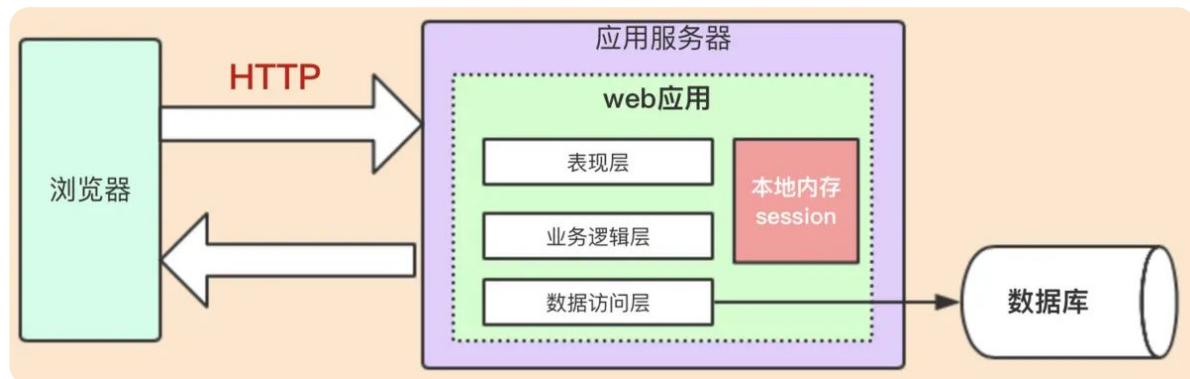
Spring Cloud

微服务架构是一种架构模式或者说是一种架构风格。



单体应用阶段 (夫妻摊位)

在互联网发展的初期，用户数量少，一般网站的流量也很少，但硬件成本较高。因此，一般的企业会将所有的功能都集成在一起开发一个单体应用，然后将该单体应用部署到一台服务器上即可满足业务需求。



生活中的单体应用

小夫妻俩刚结婚，手里资金有限，就想着开一个路边烧烤摊。丈夫负责烤串做菜、妻子负责服务收银及上菜。这是一个典型的路边烧烤摊的经营模式。

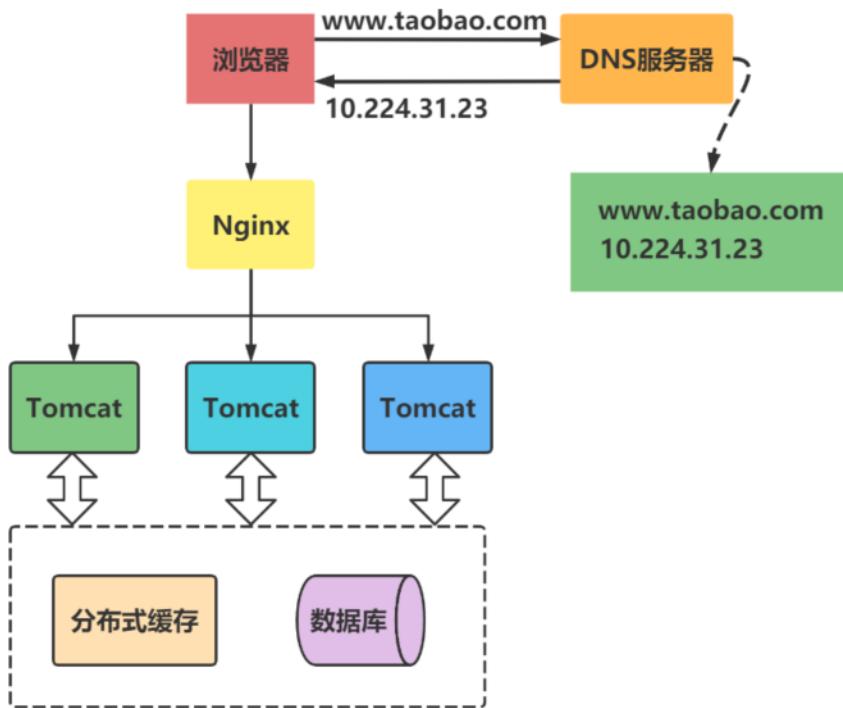
单体应用的特点：

- 能够接纳的请求数量时有限的，因为服务器的内存、CPU配置是有限的。
- 展现层、控制层、持久层全都在一个应用里面，调用方便、快速。单个请求的响应结果超快。
- 开发简单、上手快、三五个人团队好管好用。

垂直应用阶段（门面饭店）

随着小夫妻俩经营有方、待客有道，开始有人愿意为了吃他们做的烧烤排队了。夫妻俩一想，我们这俩人也干不过来啊，怎么办？招人吧、扩大规模吧。

- 招什么人？当然是厨师啊、端菜收银的妻子自己还能干得过来，主要是丈夫的活挺不住了。那就招厨师。
- 不能让人站着吃吧？租一个附近的门市、添置更多的桌椅板凳。



问题：

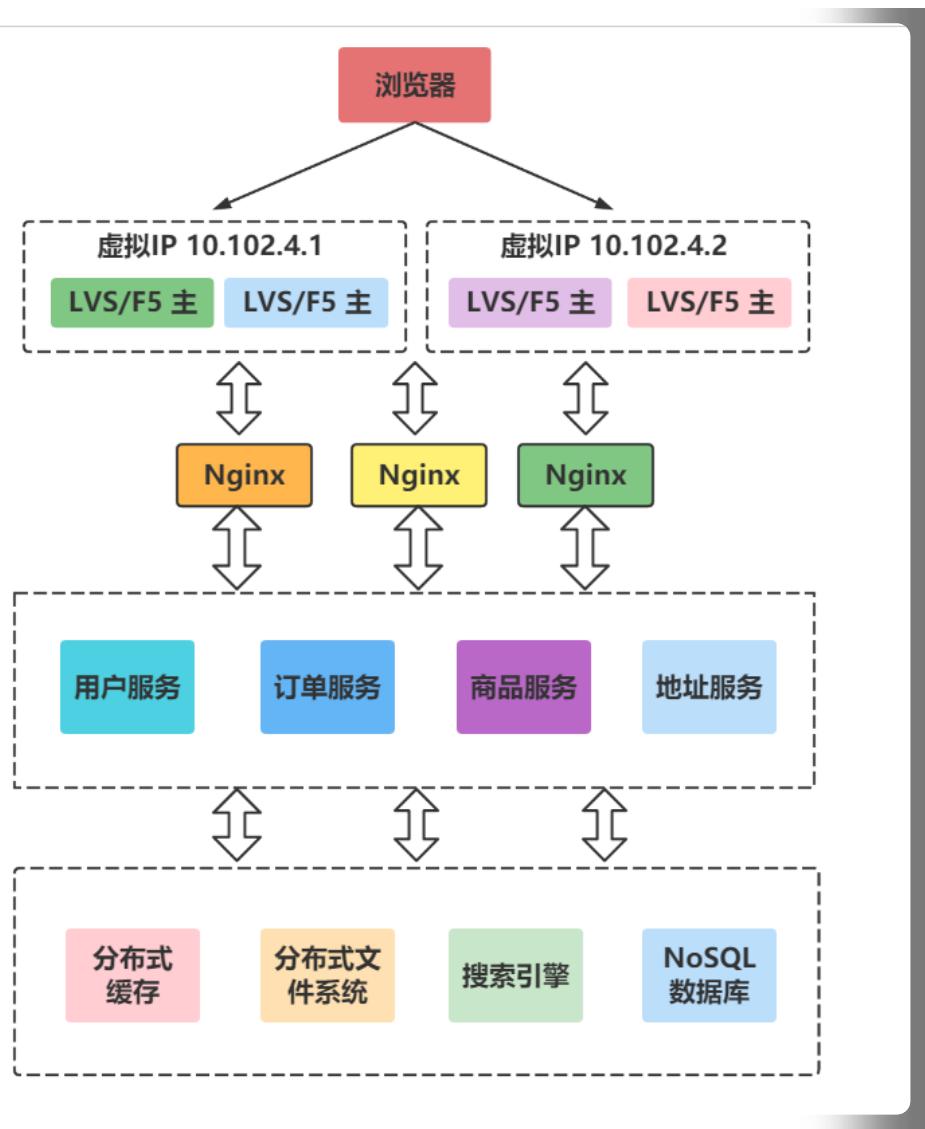
在处理并发请求的能力和容量上增强了，但是在单个请求的处理速度上下降了。

分布式系统阶段（酒店）

为了解决上一阶段遇到的问题：单个请求的处理速度下降。也就是饭店针对单个订单做菜响应速度下降了，但是由于饭店的菜确实好吃、菜品精良，客流量又持续的增高。该店又再次面临扩容的问题。

- 为了解决客流量持续增高，夫妻又招聘了4位厨师

- 为了解决单个订单处理速度下降的问题，将厨师分为两组，一组专门做烧烤，一组专门做饭菜。专业的人做专业的事情，注意力越集中，办事越熟练、效率越高。³



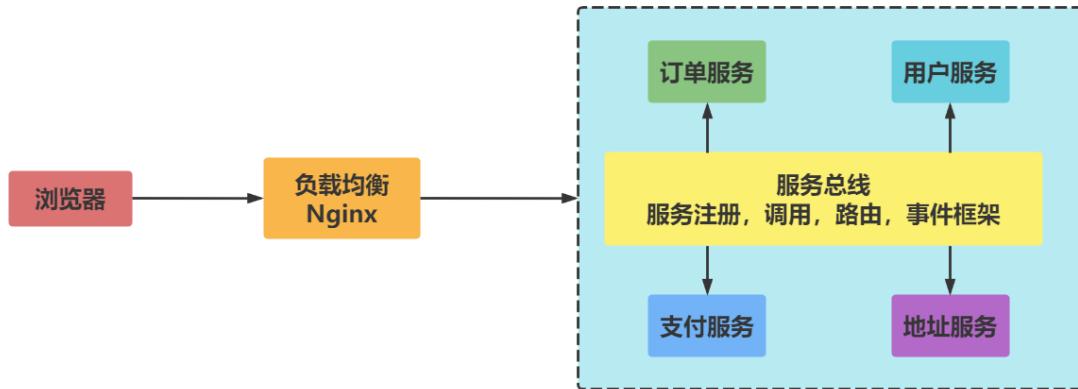
服务治理阶段（大酒店）

新的问题又出现了，有的顾客既点烧烤又点饭菜。导致后端两组厨师之间沟通不畅，怎么组合套餐推送给前台？厨师之间怎么调用、怎么沟通啊？谁是头？谁是大脑？谁记得A厨师的烧烤和B厨师的饭菜是一桌的？



**针对分布式系统存在
的问题，可以通
过治理基础服务来
解决。**

随着服务数量的不断增加，服务中的资源浪费和调度问题日益突出。此时需要增加一个调度中心来治理服务。调度中心可基于访问压力来实时管理集群的容量，从而提高集群的利用率。



注意：

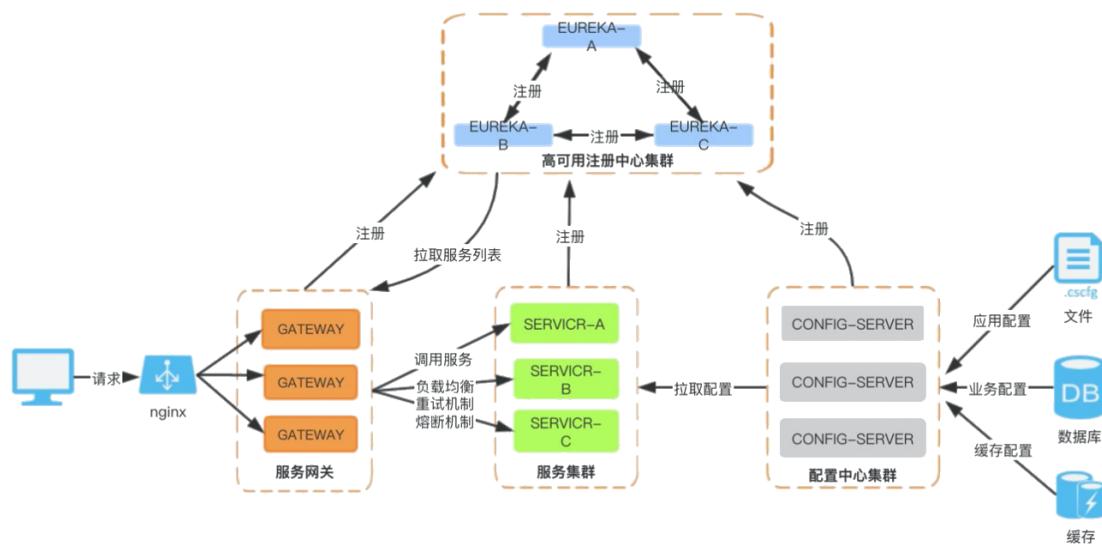
在服务治理(SOA) 架构中，需要一个企业服务总线(ESB) 将基于不同协议的服务节点连接起来，它的工作是转换、解释消息和路由。说白了就是丈夫做菜品的配置管理、做订单的服务注册。丈夫负责主动观察问询各工种的工作状态并记录，妻子主动向丈夫问询后端厨师的状态，并根据丈夫的反馈分配订单。

微服务阶段（五星大酒店）

饭店的规模越来越大了、岗位分工也越来越细了。真的成了超级大饭店了，怎么管？

什么是微服务：

将系统的业务功能划分为极小的独立微服务，每个微服务只关注于完成某个小的任务。系统中的单个微服务可以被独立部署和扩展，且各个微服务之间是高内聚、松耦合的。微服务之间采用轻量化通信机制暴露接来实现通信。



解释：

- 服务网关：前台。所有的顾客进来，由前台统一接待。比如：Spring Cloud Gateway。
- 熔断机制：菜品限量，法式菜品、意大利菜品、日本料理。什么时间可以吃得到、可提供多少人份？这些服务都是有限制的。
- 工作效率监督：工作流程中每个岗位做了什么工作、用了多长时间。哪个环节出现问题、哪个岗位需要调整。比如：Sleuth、日志监控ELK等。
- 配置中心：菜单，川菜，东北菜，杭帮菜，烩菜。
- 服务集群：厨师微服务集群包含，川菜厨师微服务，杭帮菜厨师微服务等。
- 高可用注册中心：大堂经理，负责那些人上班了，他在哪里干的什么工作。

实时效果反馈

1. 分布式系统架构存在问题通过____解决。

A 服务性能

B 服务治理

C 高可用

D 以上都

2. 下列描述微服务架构正确的是____。

A 代码部署在多台服务器上并作为一个整体提供一类服务

B 所有的代码都放在一个项目中

C 多个相同的子系统在不同的服务器上

D 将一套系统拆分成不同子系统部署在不同服务器上

答案

1=>B 2=>D

微服务的拆分规范和原则



**微服务拆分没有一个
绝对正确的方案**



压力模型拆分

压力模型简单来说就是用户访问量，我们要识别出某些超高并发量的业务，尽可能把这部分业务独立拆分出来。

压力模型拆解为三个维度：

- 高频高并发场景

Apple iPhone 13 Pro Max (A2644) 256GB 远峰蓝色 支持移动联通电信5G 双卡双待手机
自适应高刷新率，画面更流畅、响应更灵敏，电影效果模式随手拍大片！选购[快充套装]加99元得20W快充头！更多优惠！[查看>](#)

京 东 价 ¥ 9799.00 降价通知
促 销 赠品 ×1 纸箱 ×1 盒装 ×1 (赠完即止)

累 5

增值业务 高价回收，极速到账)
配 送 至 山西太原市古交市岔口乡 有货 支持 可配送港澳台 | 本地仓 | 99元免基础运费 免举证退换货 ✓
由 京东 发货，并提供售后服务。

重 量 0.4kg

选择颜色 远峰蓝色 石墨色 银色 金色
选择版本 128GB 256GB 512GB 1TB
购买方式 公开版 【值享焕新版】 【1年期官方AppleCare+版】 快充套装 AirPods套餐
套 装 优惠套装1 优惠套装2 优惠套装3 优惠套装4 优惠套装5 优惠套装6
原厂服务 1年AC+ ¥848.00
增值保障 2年碎屏保修 ¥399.00 2年全保+ ¥449.00 2年电池换新 ¥69.00
京东服务 黑科技充电宝 ¥129.00

关注 分享 对比 举报

原因：

比如商品详情页，它既是一个高频场景（时时刻刻都会发生），同时也是高并发的场景（QPS - Query per seconds极高）

低频突发流量场景

京东秒杀
16:00 点场 距结束
01 : 15 : 59

中兴 V2022 4G墨云灰 4GB...
¥598.00

一加 9RT 5G 120Hz 高刷好...
¥3269.00

ZTE中兴Axon30吴京代言 ...
¥2998.00

四季沐歌 (MICOE) 航+飞...
¥5399.00

手机配件好物秒杀
荣耀60 SE低至2199
品类秒杀 >

原因：

秒杀场景它并不是高频场景（偶尔发生），但是它会产生突发流量。再跟大家举一个例子，那就是“商品发布”，对新零售业务来说，当开设一个线下大型卖场以后，需要将所有库存商品一

键上架，这里的商品总数是个非常庞大的数字（几十万+），瞬间就可以打出很高的QPS

低频流量场景

The screenshot shows a user management interface with a sidebar for organizational structures. The tree view includes '若依科技' at the root, followed by '深圳总公司' and '长沙分公司'. Under '深圳总公司', there are departments: '研发部门', '市场部门', '测试部门', '财务部门', and '运营'. Under '长沙分公司', there are '市场部门' and '财务部门'. The main panel displays a list of users with columns: '用户ID', '登录名称', '用户名', '部门', '手机', '用户状态', and '创建时间'. There are two entries: one for 'admin' in the '研发部门' and another for 'ry' in the '测试部门'. At the top, there are search and filter fields for '登录名称', '手机号码', and '用户状态', along with buttons for '搜索' and '重置'. Below the table, there are buttons for '+新增', '修改', '删除', '导入', and '导出'. At the bottom, there is a pagination bar showing '显示第 1 到第 2 条记录, 共共 2 条记录 每页显示 10 条记录'.

原因：

后台运营团队的服务接口，比如商品图文编辑，添加新的优惠计算规则，上架新商品。它发生的频率比较低，而且也不会造成很高的并发量。

业务模型拆分

业务模型拆分的维度有很多，我们在实际项目中应该综合各个不同维度做考量。我这里主要从主链路、领域模型和用户群体三个维度。

主链路拆分

在电商领域“主链路”是一个很重要的业务链条，它是指用户完成下单场景所必须经过的场景。按照我们平时买买买的剁手经验，可以识别出很多核心主链路，比如商品搜索->商品详情页->购物车模块->订单结算->支付业务，这是就是一条最简单的主链路。如果这是一场战斗的话，那么主链路就是这场战斗的正面战场，我们必须力保主链路不失守。

核心主链路拆分，有以下几个目的：

- ① 异常容错：为主链路建立层次化的降级策略（多级降级），以及合理的熔断策略。
- ② 调配资源：主链路通常来讲都是高频场景，自然需要更多的计算资源，最主要的体现就是集群里分配的虚机数量多。

- ③ 服务隔离：主链路是主打输出的C位，把主链路与其他打辅助的业务服务隔离开来，避免边缘服务的异常情况影响到主链路。

领域模型拆分

领域驱动设计DDD（Domain-Driven Design 领域驱动设计）不是一个新概念，但老外们有个毛病，做什么事情特别喜欢提炼方法论，本来一个非常简单的概念，愣是被吹到神乎其神高深莫测。

用户群体拆分

根据用户群体做拆分，我们首先要了解自己的系统业务里有哪些用户，比如说电商领域，我们有2C的小卖家，也有2B的大客户，在集团内部有运营、采购、还有客服小二等等。对每个不同的用户群体来说，即便是相同的业务领域，也有该群体其独有的业务场景。

用户群体相当于一个二级域，我们建议先根据主链路和领域模型做一级域的拆分，再结合具体的业务分析，看是否需要在用户领域方向上做更细粒度的拆分。

实时效果反馈

1. 下列属于拆分模型的是__。

- A 业务模型拆分
- B 压力模型拆分
- C 领域模型拆分
- D 以上都正确

答案

1=>B

为什么选择Spring Cloud



让人人享有高品质教育



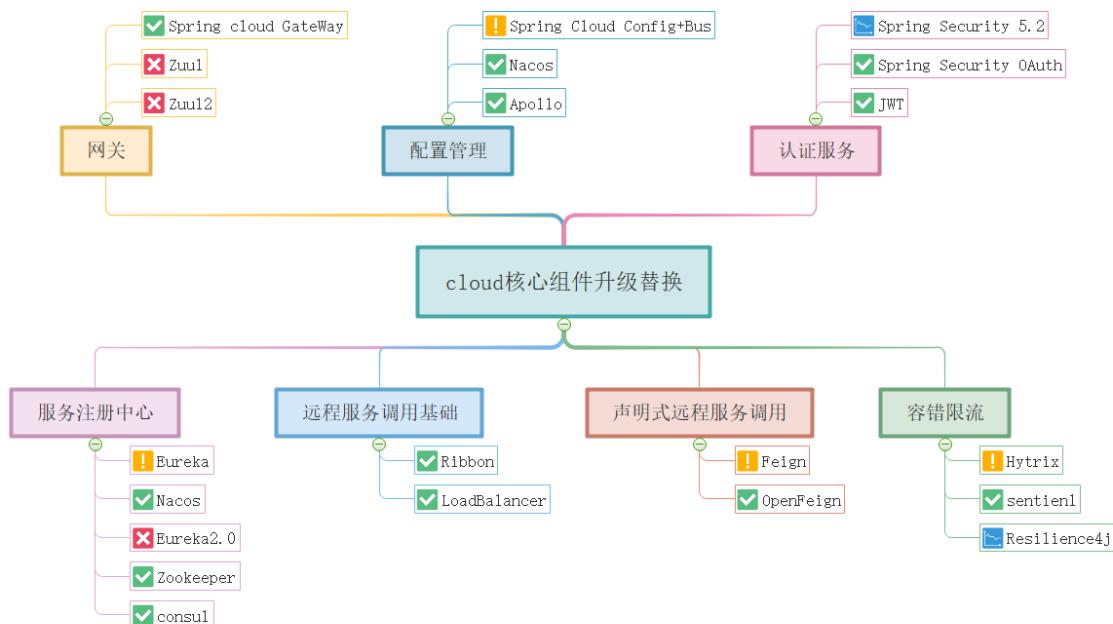
有 Spring Cloud 的地方
就有江湖，我们就来看一看
在这个江湖中都有哪些独霸
一方的门派！

Spring Cloud与Netflix

Netflix是一家做视频网站的公司，之所以要说一下这个公司是因为 Spring Cloud在发展之初，Netflix做了很大的贡献。包括服务注册中心Eureka、服务调用Ribbon、Feign，服务容错限流Hystrix、服务网关Zuul等众多组件都是Netflix贡献给Spring Cloud社区的。

什么是SpringCloud

Spring Cloud是一个基于Spring Boot实现的微服务架构开发工具。它为微服务架构中涉及的配置管理、服务治理、断路器、智能路由、控制总线、分布式会话和集群状态管理等操作提供了一种简单的开发方式。



核心事件追踪

- 2018年6月底，Eureka 2.0 开源工作宣告停止，继续使用风险自负。
- 2018年11月底，Hystrix 宣布不再在开源版本上提供新功能。
- 2018年12月，Spring官方宣布Netflix的相关项目进入维护模式。

从此，Spring Cloud逐渐告别Netflix时代。

- 2018年10月31日，Spring Cloud Alibaba正式入驻了Spring Cloud官方孵化器，并在maven中央库发布了第一个版本。

服务注册中心选型

- **Eureka**: Spring Cloud与Netflix的大儿子，出生的时候家里条件一般，长大后素质有限。
- **Nacos**: 后起之秀，曾经Spring Cloud眼中“别人家的孩子”，已经纳入收养范围（Spring Cloud Alibaba孵化项目）。
- **Apache Zookeeper**: 关系户，与hadoop关系比较好
- **etcd**: 关系户，与kubernetes关系比较好
- **Consul**: 关系户，曾经与docker关系比较好

注意：

如果你的应用已经使用到了Hadoop、Kubernetes、Docker，在Spring Cloud实施过程中可以考虑使用其关系户组件，避免搭建两套注册中心，节省资源。

分布式配置管理

目前可选的分布式配置管理中心，有阿里的Nacos、携程的Apollo、和Spring Cloud Config。

服务网关

服务网关这块就不多说了，没有任何悬念，Spring Cloud Gateway 在各方面都碾压Zuul，Zuul2也基本上是胎死腹中。

熔断限流

Hystrix

2018年12月，Spring官方宣布Netflix的相关项目进入维护模式。不再开发新的功能，但是Hystrix整体上还是比较稳定的，对于老用户不必更换，影响也不大。

resilience4j

Hystrix停更之后，Netflix官方推荐使用resilience4j，它是一个轻量、易用、可组装的高可用框架，支持熔断、高频控制、隔离、限流、限时、重试等多种高可用机制。

Sentinel (重点)

Sentinel是阿里中间件团队开源的，面向分布式服务架构的轻量级高可用流量控制组件，主要以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度来帮助用户保护服务的稳定性。

实时效果反馈

1. Spring Cloud是一个基于____实现的微服务架构开发工具。

- A Spring Boot
- B Spring MVC
- C Spring Data
- D Spring Batch

答案

1=>A

Spring Cloud版本选择

Documentation

Each Spring project has its own; it explains in great details how you can use [project features](#) and what you can achieve with them.

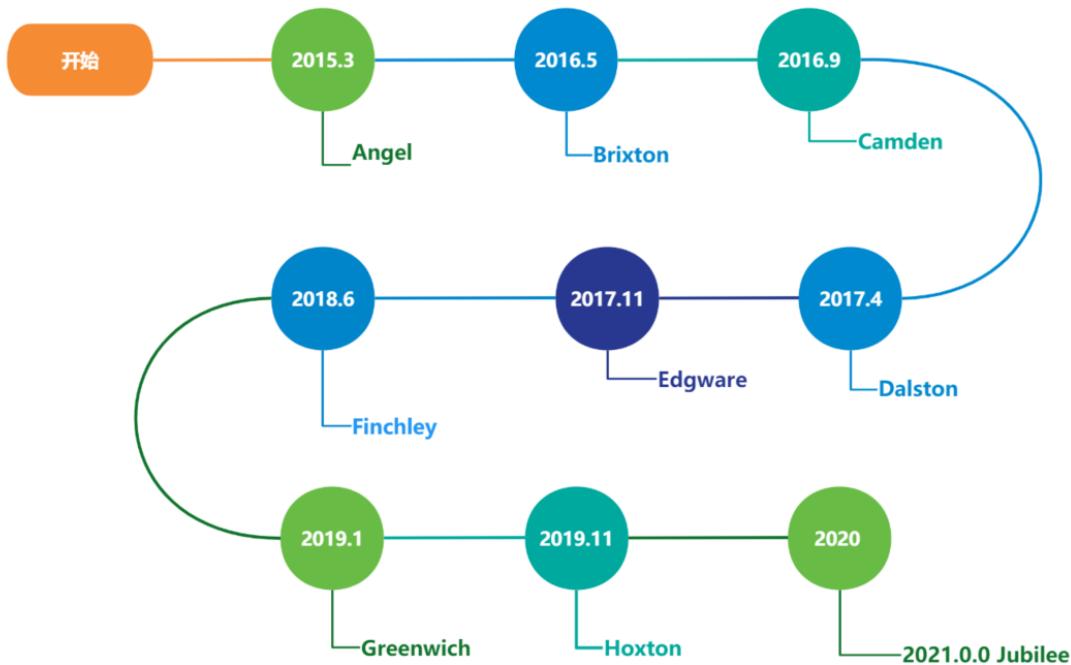
2021.0.0	CURRENT GA	Reference Doc.
2022.0.0-M1	PRE	Reference Doc.
2021.0.1-SNAPSHOT	SNAPSHOT	Reference Doc.
2020.0.5-SNAPSHOT	SNAPSHOT	Reference Doc.
2020.0.4	GA	Reference Doc.
Hoxton.SR12	GA	Reference Doc.
Hoxton.BUILD-SNAPSHOT	SNAPSHOT	Reference Doc.



版本太多了，
我真的哭了

SpringCloud版本号由来

SpringCloud的版本号是根据英国伦敦地铁站的名字进行命名的，由地铁站名称字母A-Z依次类推表示发布迭代版本。



SpringCloud和SpringBoot版本对应关系

英文	中文	终结版本	boot大版本	boot代表	说明
Angel	安吉尔	SR6	1.2.X	1.2.8	GA
Brixton	布里克斯顿	SR7	1.3.X	1.3.8	GA
Camden	卡梅登	SR7	1.4.X	1.4.2	GA
Dalston	达斯顿	SR5	1.5.X	*	GA
Edgware	艾奇韦尔	SR5	1.5.X	1.5.19	GA
Finchley	芬奇利	SR2	2.0.X	2.0.8	GA
Greenwich	格林威治	RC2	2.1.X	2.1.2	GA
hoxton	霍克斯顿	RC2	2.2.X	2.2.6	GA

注意事项：

其实SpringBoot与SpringCloud需要版本对应，否则可能会造成很多意料之外的错误，比如eureka注册了结果找不到服务类啊，比如某些jar导入不进来啊，等等这些错误。

版本说明

名字	描述
SNAPSHOT	快照版，可以稳定使用，且仍在继续改进版本。
PRE	预览版,内部测试版.主要是给开发人员和测试人员测试和找BUG用的，不建议使用；
RC	发行候选版本，基本不再加入新的功能，主要修复bug。
SR	修正版或更新版
GA	正式发布的版本

从 Spring Cloud 2020.0.0-M1 开始，Spring Cloud 废除了这种英国伦敦地铁站的命名方式，而使用了全新的 "日历化" 版本命名方式。

实时效果反馈

1. SpringCloud版本选择尽量使用最新_____版。

A RC

B PRE

C SR

D GA

2. SpringCloud版本中GA版代表_____。

A 预览版

B 发行候选版本

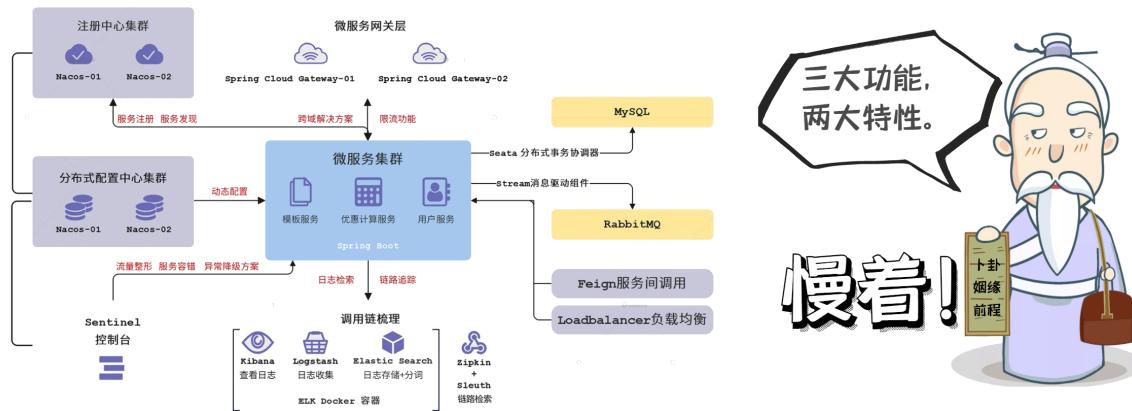
C 修正版或更新版

D 正式发布的版本

答案

1=>D 2=>D

如何学习微服务Spring Cloud



简单来说，就是“三大功能，两大特性”。

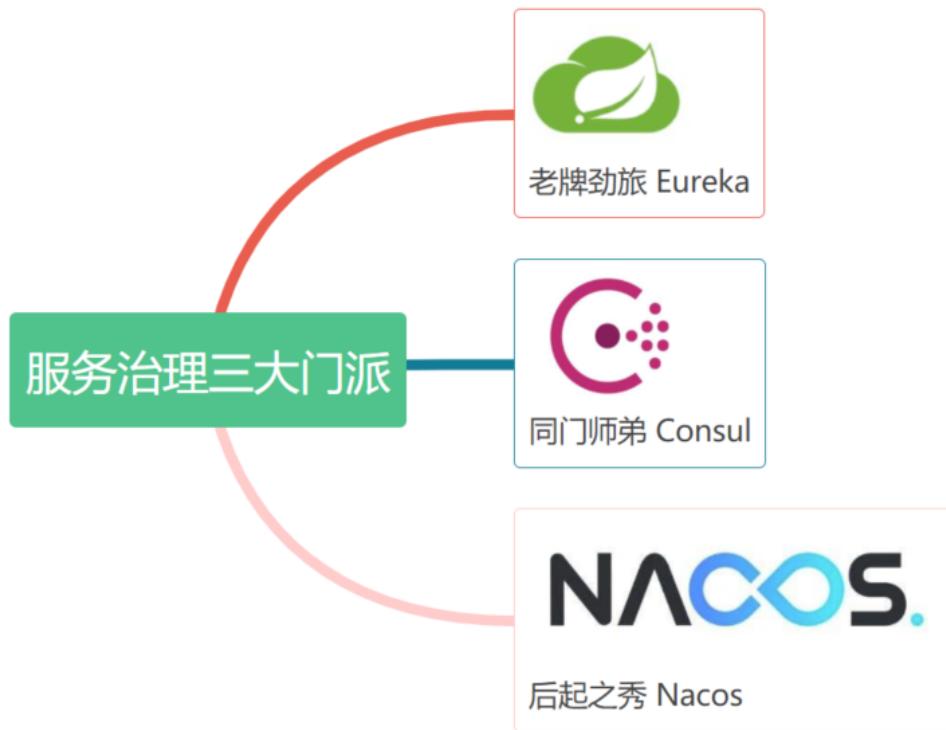
三大功能	高可用性		高可扩展性	
	服务间通信	服务容错 异常排查	分布式能力建设	分布式的建设
	服务治理 负载均衡 服务间调用	流量整形 降级熔断 调用链追踪	微服务网关 分布式事务 消息驱动	配置管理

三大功能是指微服务核心组件的功能维度，由浅入深层次递进；而两大特性是构建在每个服务组件之上的高可用性和高可扩展性。别看微服务框架组件多，其实你完全可以按照这三大功能模块，给它们有简入难对号入座。

注意：

- **服务间通信**：包括服务治理、负载均衡、服务间调用；
- **服务容错和异常排查**：包括流量整形、降级熔断、调用链追踪；
- **分布式能力建设**：包括微服务网关、分布式事务、消息驱动、分布式配置中心。

从哪里入手



从微服务组件的功能维度来讲，服务间通信是最基础的功能特性，这个功能模块是最适合作为初学者学习微服务技术的切入点。

实时效果反馈

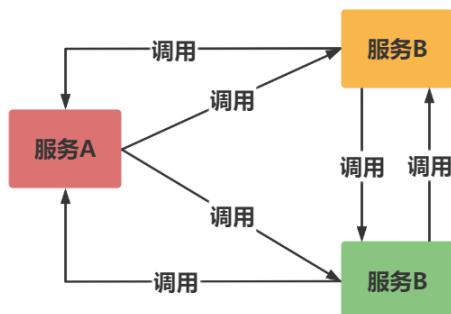
1. 学习微服务从哪里入手__。

- A 服务间通信
- B 服务容错和异常排查
- C 分布式能力建设
- D 以上都是错误

答案

1=>A

服务注册发现_什么是服务治理

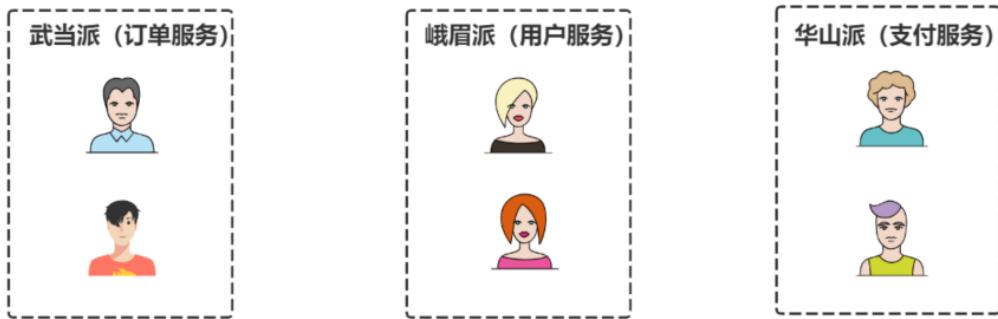


服务治理是通向微服务架构的第一关。

为什么需要服务治理

在没有进行服务治理前,服务之间的通信是通过服务间直接相互调用来实现的。

微服务集群 (武林门派)



过程：

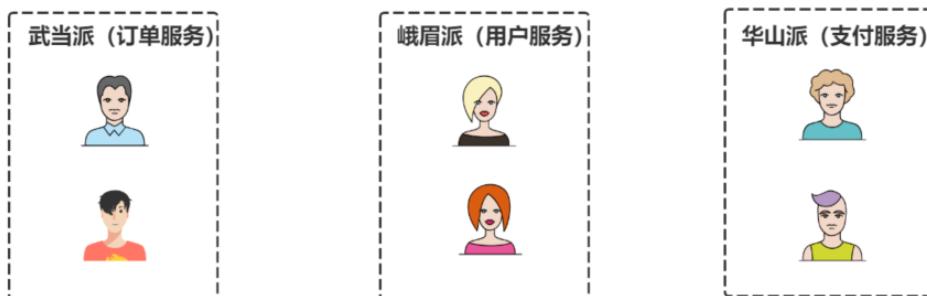
武当派直接调用峨眉派和华山派，同样，华山派直接调用武当派和峨眉派。如果系统不复杂，这样调用没什么问题。但在复杂的微服务系统中，采用这样的调用方法就会产生问题。

微服务系统中服务众多，这样会导致服务间的相互调用非常不便，因为要记住提供服务的IP地址、名称、端口号等。这时就需要中间代理，通过中间代理来完成调用。

服务治理的解决方案



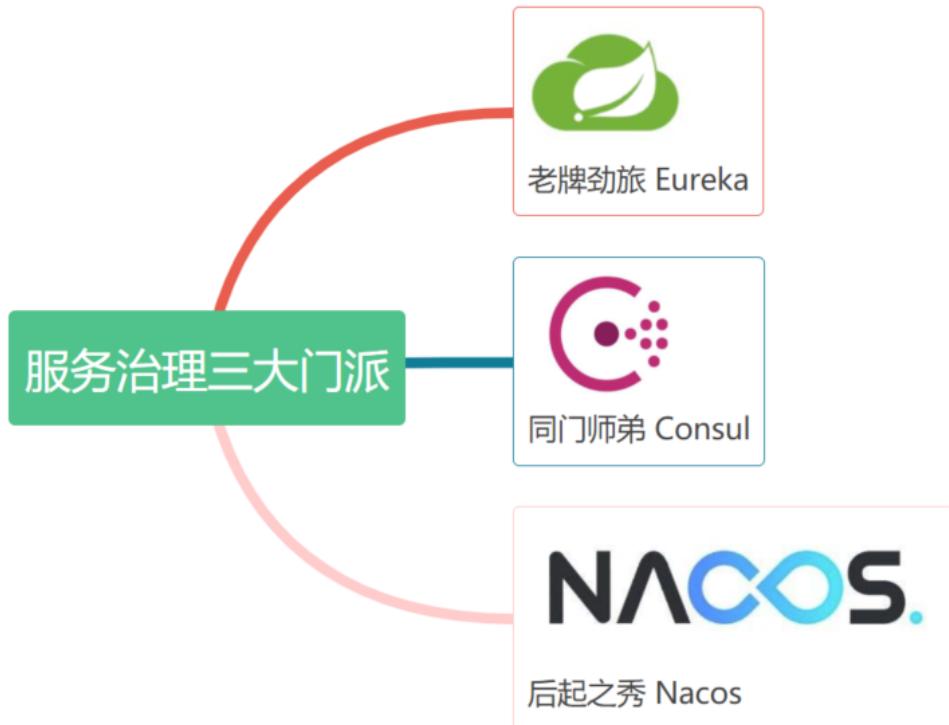
微服务集群 (武林门派)



服务治理责任:

- 你是谁: 服务注册 - 服务提供方自报家门
- 你来自哪里: 服务发现 - 服务消费者拉取注册数据
- 你好吗: 心跳检测, 服务续约和服务剔除 一套由服务提供方和注册中心配合完成的去伪存真的过程
- 当你死的时候: 服务下线 - 服务提供方发起主动下线

服务治理技术选型



注意：

在架构选型的时候，我们需要注意一下切记不能为了新而新，忽略了对于当前业务的支持，虽然Eureka2.0不开源了，但是谁知道以后会不会变化，而且1.0也是可以正常使用的，也有一些贡献者在维护这个项目，所以我们没有必要过多的担心这个问题，要针对业务看下该技术框架是否支持在做考虑。

实时效果反馈

1. 服务治理解决__问题。

- A 服务性能
- B 单点故障问题
- C 分布式服务调用问题
- D 兼容

答案

1=>C

服务注册发现_Eureka概述



SpringCloud封装了Netflix公司开发的Eureka模块来实现服务治理。



Spring Cloud Eureka 是Netflix 开发的注册发现组件，本身是一个基于 REST 的服务。提供注册与发现，同时还提供了负载均衡、故障转移等能力。

Eureka3个角色

- 服务中心
- 服务提供者
- 服务消费者。



注意：

- **Eureka Server**: 服务器端。它提供服务的注册和发现功能，即实现服务的治理。
- **Service Provider**: 服务提供者。它将自身服务注册到Eureka Server中，以便“服务消费者”能够通过服务器端提供的服务清单（注册服务列表）来调用它。

- **Service Consumer**: 服务消费者。它从 Eureka 获取“已注册的服务列表”，从而消费服务。

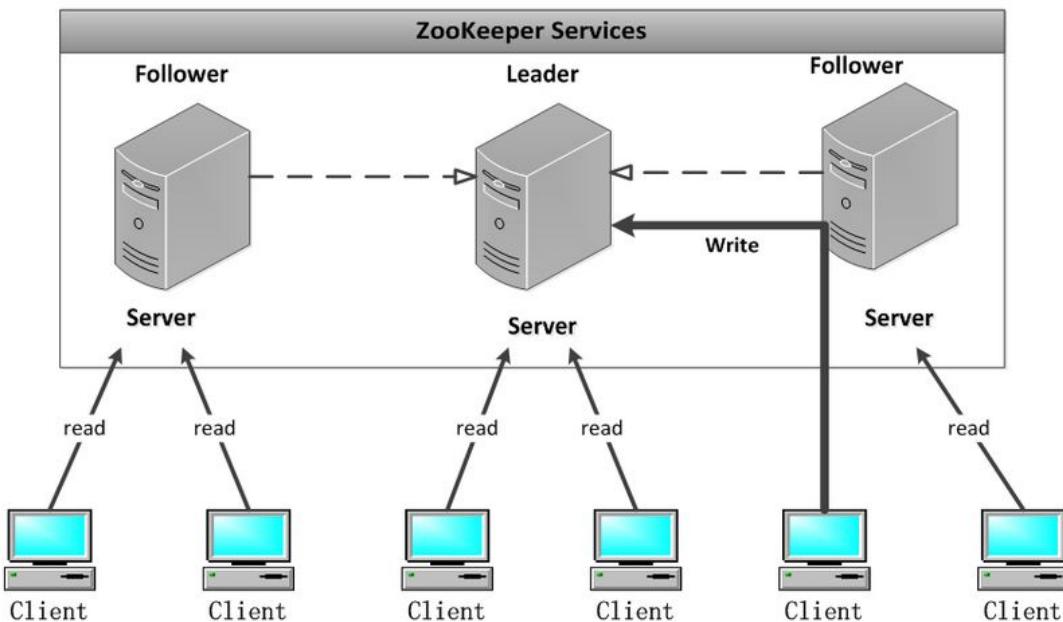
比Zookeeper好在哪里呢

ZooKeeper

知识点要来了



当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。



注意：

Zookeeper会出现这样一种情况，当Master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30~120s，且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。

结论

Eureka看明白了这一点，因此在设计时就优先保证可用性。

实时效果反馈

1. Eureka 是Netflix 开发的__组件。

A 注册发现组件

B 熔断

C 网关

D 以上都是错误

2. Zookeeper服务治理问题是__。

A 选举leader的时间太长

B 安全

C 单点故障问题

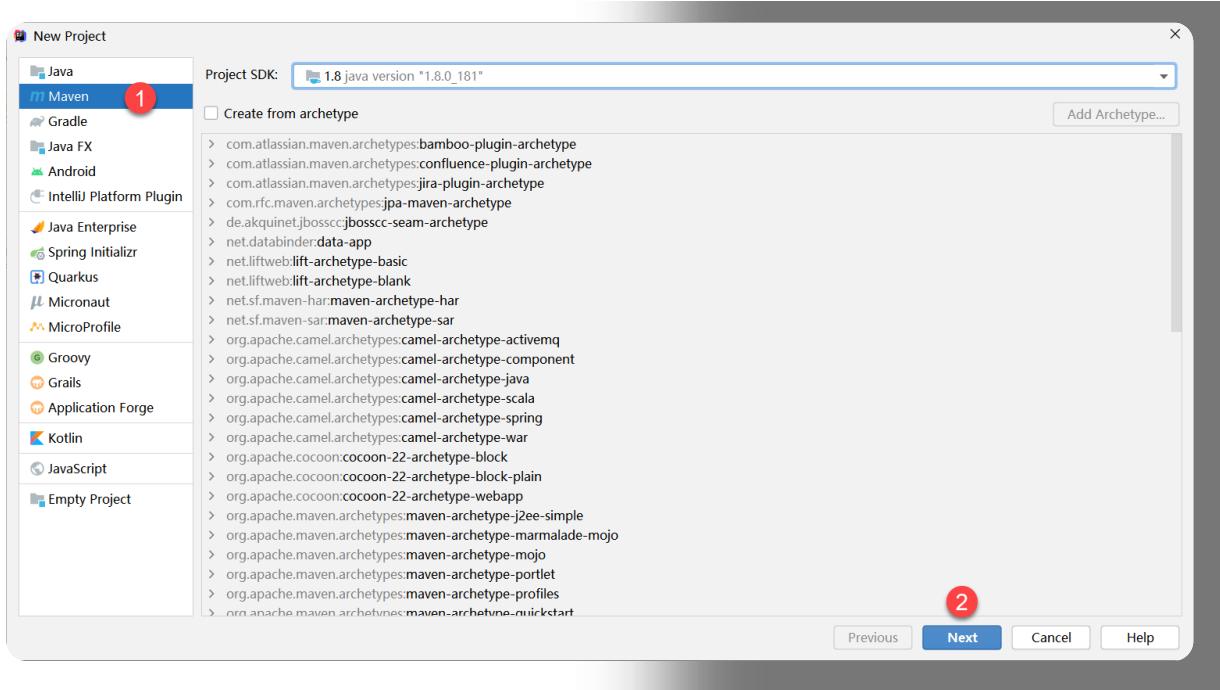
D 以上都是错误

答案

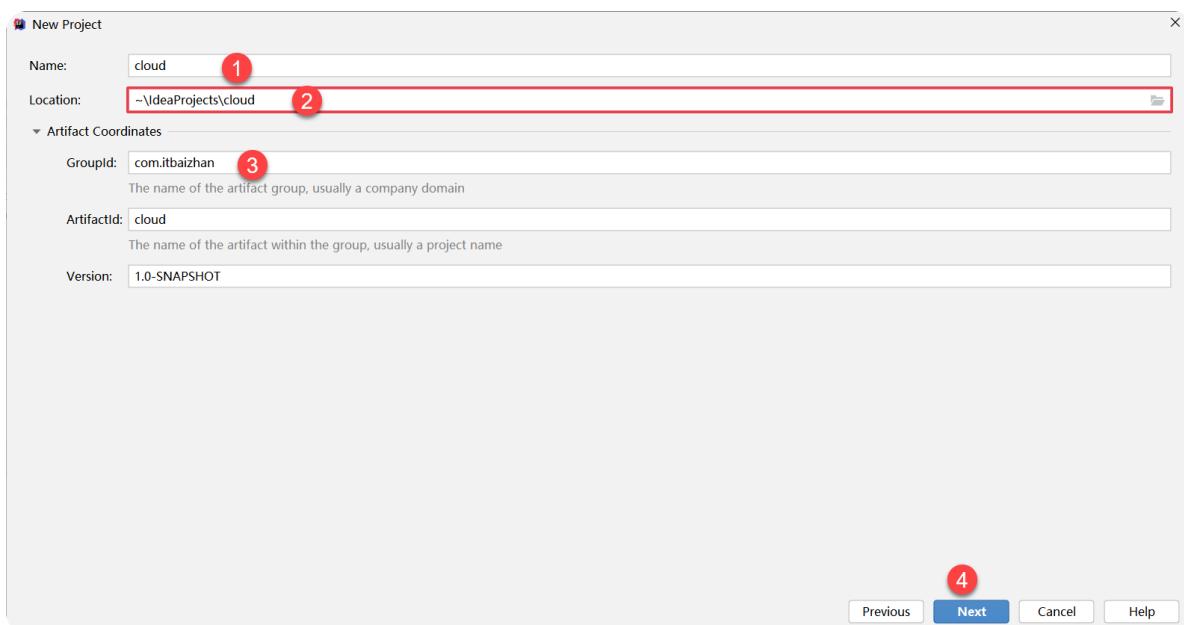
1=>A 2=>A

服务注册发现_微服务聚合父工程构建

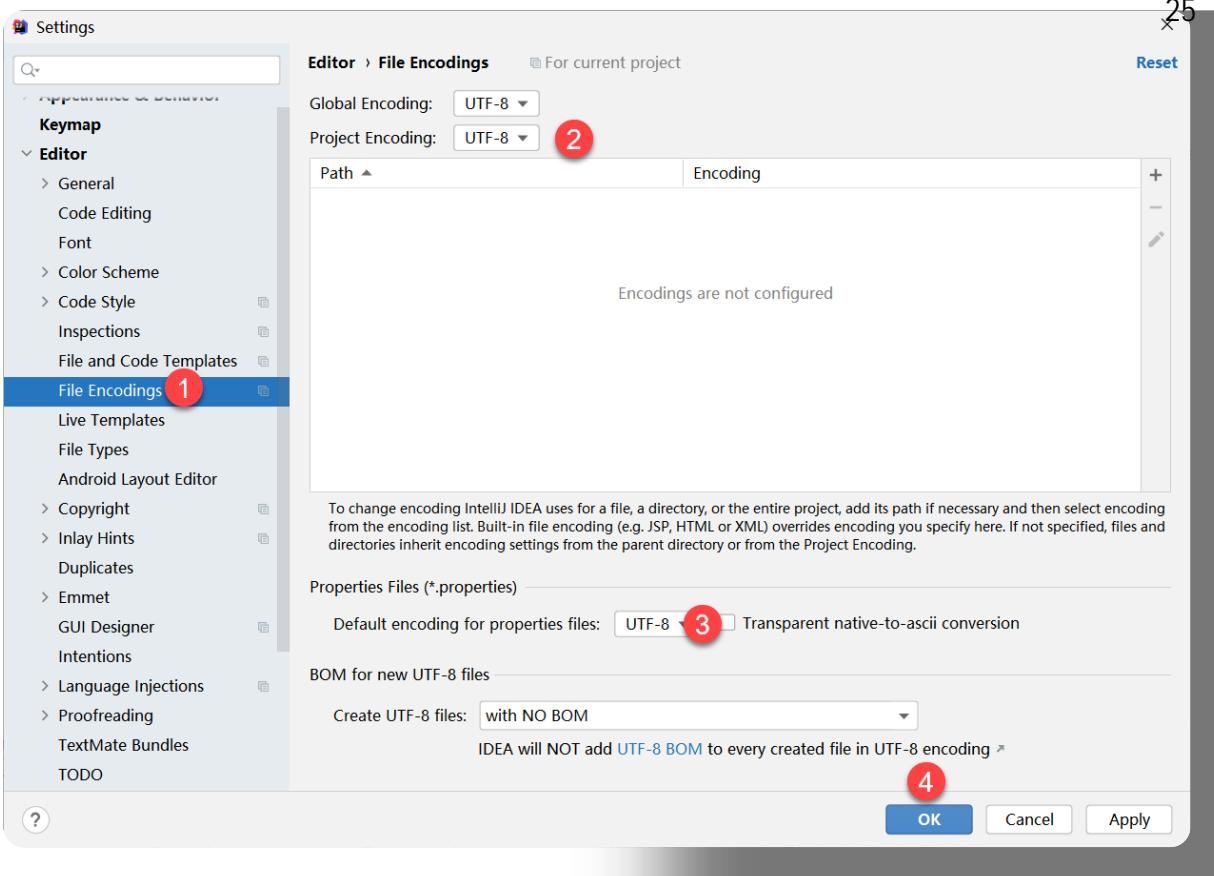
New Project



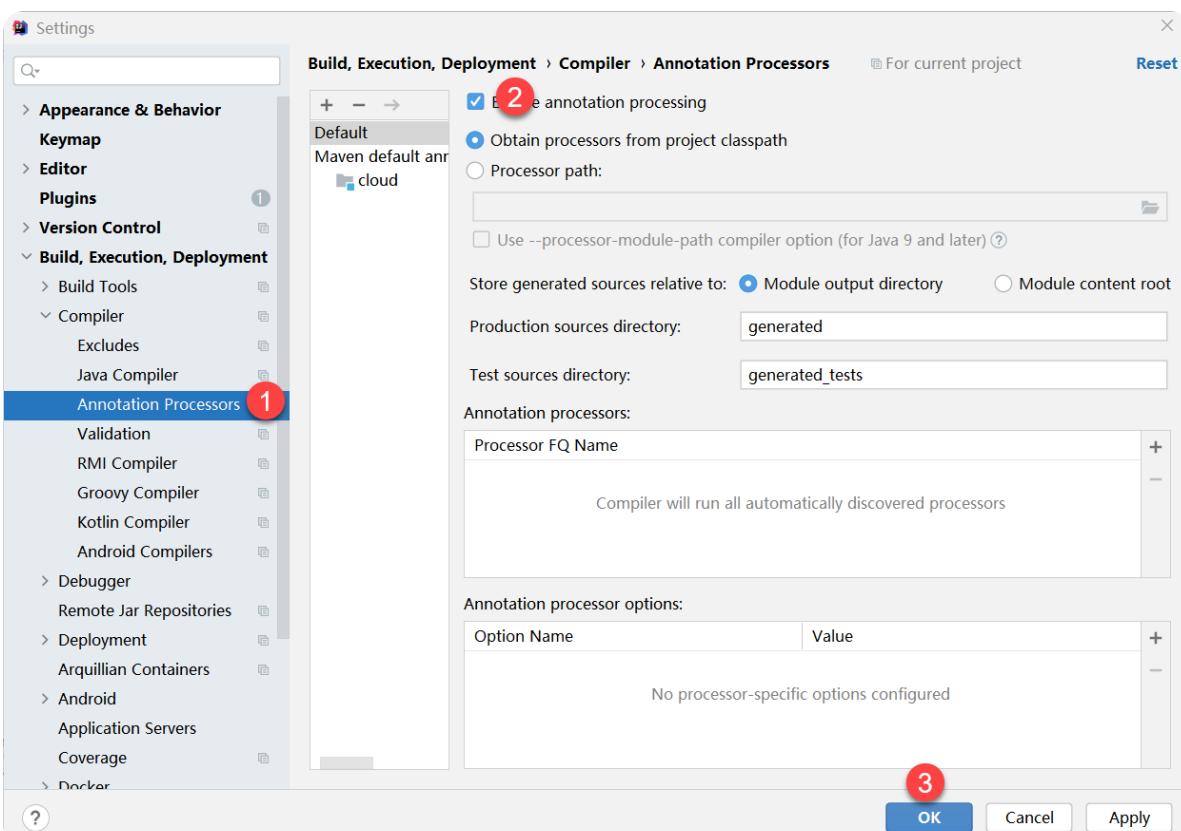
聚合总工程名称



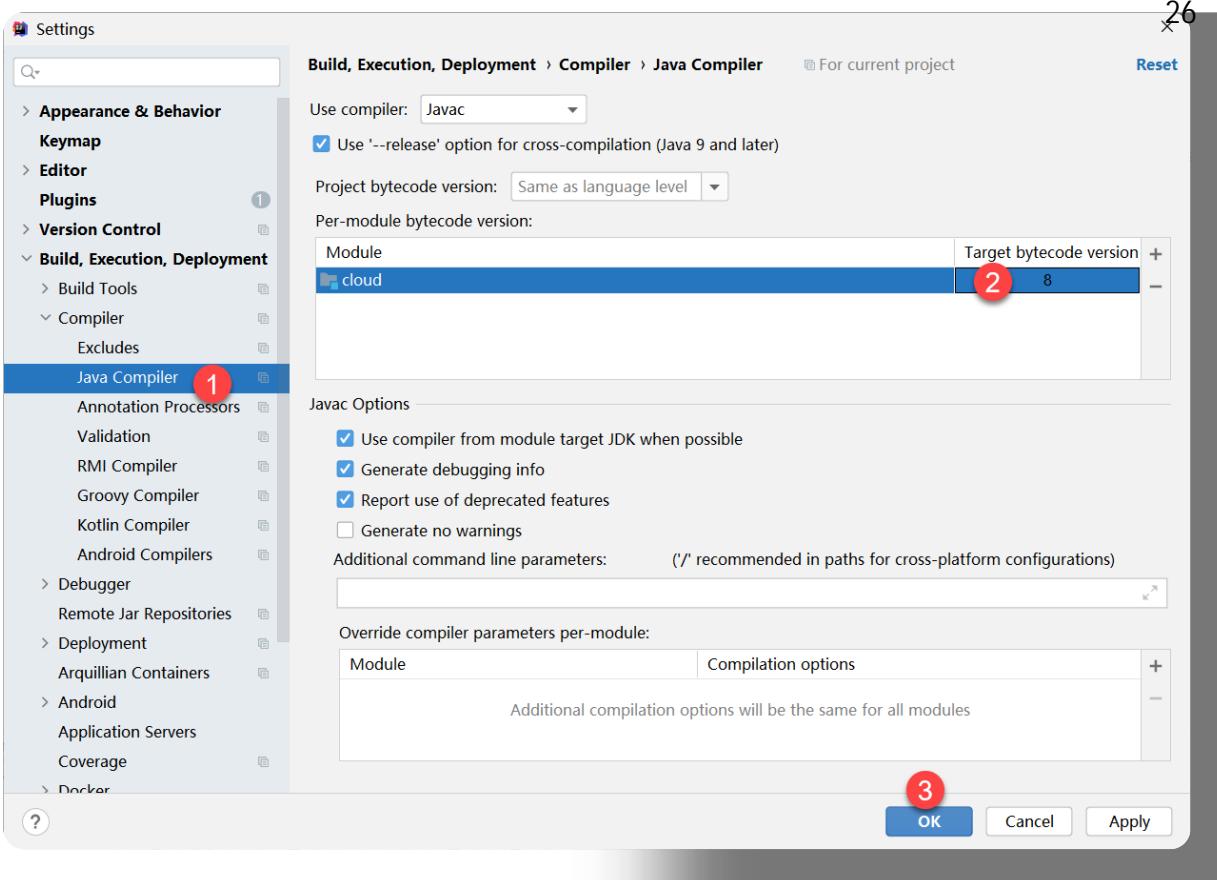
字符编码



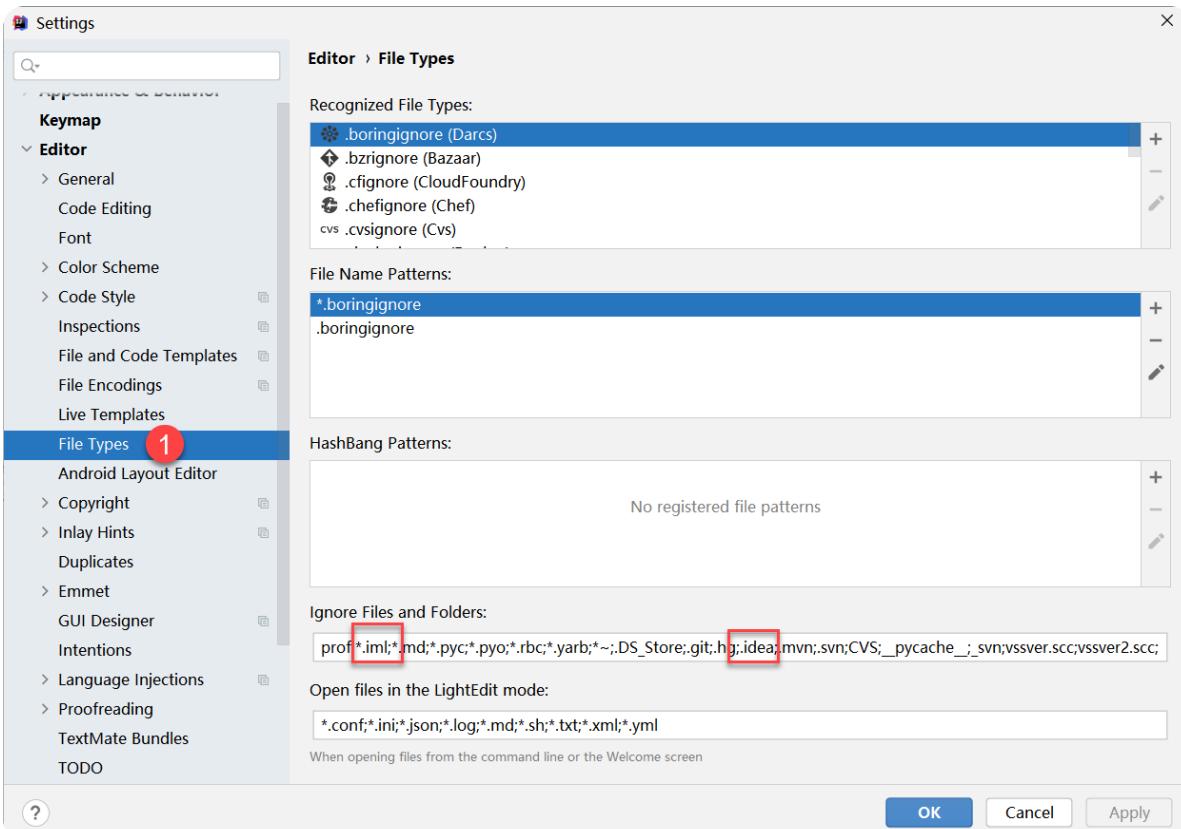
注解生效激活



Java编译版本选择



File Type过滤



父工程POM

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

```
2 <project  
3   xmlns="http://maven.apache.org/POM/4.0.0"  
4  
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-  
6     instance"  
7  
8   xsi:schemaLocation="http://maven.apache.org/  
9     POM/4.0.0 http://maven.apache.org/xsd/maven-  
10    4.0.0.xsd">  
11     <modelVersion>4.0.0</modelVersion>  
12  
13     <groupId>com.itbaizhan</groupId>  
14     <artifactId>cloud</artifactId>  
15     <version>1.0-SNAPSHOT</version>  
16     <packaging>pom</packaging>  
17  
18     <!-- 统一管理jar包版本 -->  
19     <properties>  
20       <project.build.sourceEncoding>UTF-  
21         8</project.build.sourceEncoding>  
22  
23       <maven.compiler.source>1.8</maven.compiler.s  
24         ource>  
25  
26       <maven.compiler.target>1.8</maven.compiler.t  
27         arget>  
28         <spring-  
29           cloud.version>2021.0.0</spring-  
30             cloud.version>  
31             <spring-boot.version>2.6.3</spring-  
32               boot.version>  
33             </properties>
```

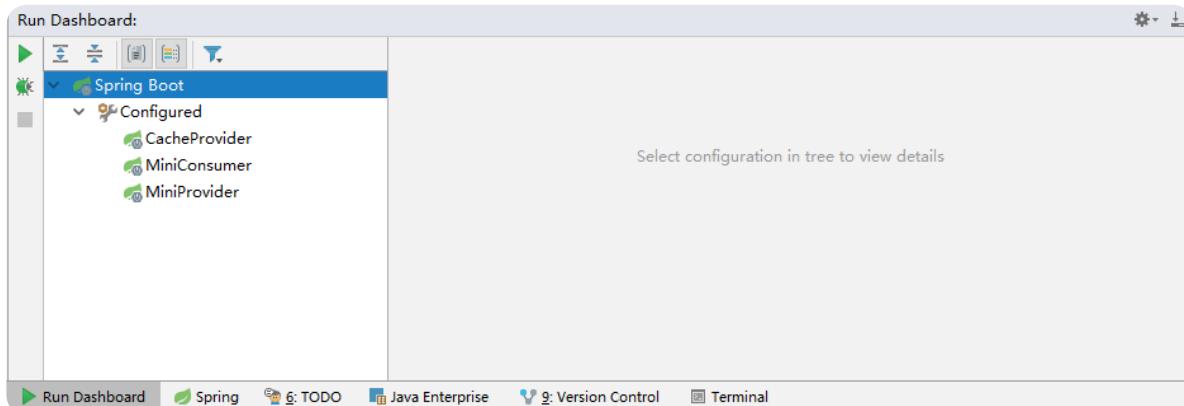
```
20
21      <!-- 子模块继承之后，提供作用：锁定版本+子
22      module不用写groupId和version -->
23      <dependencyManagement>
24          <dependencies>
25              <!--spring boot 2.6.3-->
26              <dependency>
27                  <groupId>org.springframework.boot</groupId>
28                      <artifactId>spring-boot-
29                      starter-parent</artifactId>
30                          <version>${spring-
31                          boot.version}</version>
32                          <type>pom</type>
33                          <scope>import</scope>
34                      </dependency>
35              <!--spring cloud 2021.0.0-->
36              <dependency>
37                  <groupId>org.springframework.cloud</groupId>
38                      <artifactId>spring-cloud-
39                      dependencies</artifactId>
40                          <version>${spring-
41                          cloud.version}</version>
42                          <type>pom</type>
43                          <scope>import</scope>
44                      </dependency>
45                  </dependencies>
46          </dependencyManagement>
47
48      </project>
```

IDEA开启Dashboard

普通的Run面板



Run Dashboard面板



修改配置文件

在.idea/workspace.xml 文件中找到

```

1 <component name="RunDashboard">
2   <option name="ruleStates">
3     <list>
4       <RuleState>
5         <option name="name"
value="ConfigurationTypeDashboardGroupingRule" />
6       </RuleState>
7       <RuleState>
8         <option name="name"
value="StatusDashboardGroupingRule" />
9       </RuleState>
10      </list>
11    </option>
12  </component>

```

添加配置

```

1 <component name="RunDashboard">
2   <option name="ruleStates">
3     <list>
4       <RuleState>
5         <option name="name"
value="ConfigurationTypeDashboardGroupingRule" />
6       </RuleState>
7       <RuleState>
8         <option name="name"
value="StatusDashboardGroupingRule" />
9       </RuleState>
10      </list>
11    </option>
12    <option name="configurationTypes">

```

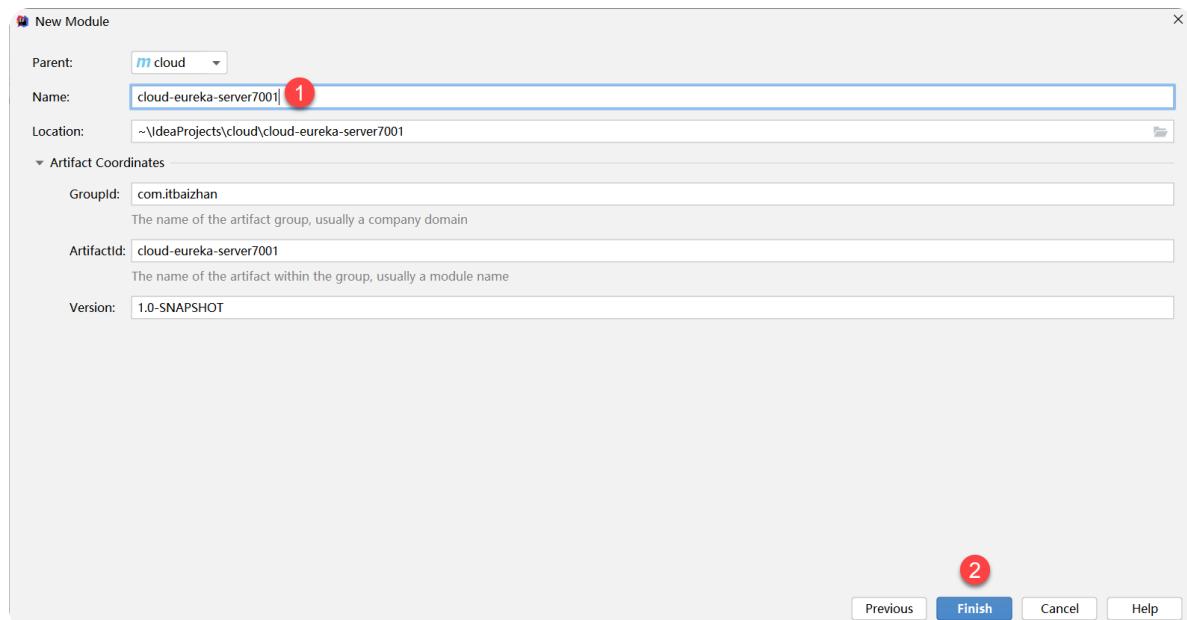
```

13 <set>
14     <option
15         value="SpringBootApplicationConfigurationTyp
16             e" />
17     </set>
18 </option>
19 </component>

```

服务注册发现_搭建单机Eureka注册中心

创建cloud-eureka-server7001模块



pom添加依赖

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project
3     xmlns="http://maven.apache.org/POM/4.0.0"
4
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-
6     instance"

```

```
4      xsi:schemaLocation="http://maven.apache.org/
POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
5      <parent>
6          <artifactId>cloud</artifactId>
7          <groupId>com.itbaizhan</groupId>
8          <version>1.0-SNAPSHOT</version>
9      </parent>
10     <modelVersion>4.0.0</modelVersion>
11
12     <artifactId>cloud-eureka-
server7001</artifactId>
13
14     <dependencies>
15         <!-- 服务注册发现Eureka-->
16         <dependency>
17             <groupId>org.springframework.cloud</groupId>
18             <artifactId>spring-cloud-
starter-netflix-eureka-server</artifactId>
19             </dependency>
20             <dependency>
21                 <groupId>org.projectlombok</groupId>
22                     <artifactId>lombok</artifactId>
23                     </dependency>
24             <dependency>
25                 <groupId>org.springframework.boot</groupId>
```

```

26 <artifactId>spring-boot-starter-
27   test</artifactId>
28     <scope>test</scope>
29   </dependency>
30 </dependencies>
31 </project>

```

写yml文件

```

1 server:
2   port: 7001
3 eureka:
4   instance:
5     # eureka服务端的实例名字
6     hostname: localhost
7   client:
8     # 表示是否将自己注册到Eureka Server
9     register-with-eureka: false
10    # 表示是否从Eureka Server获取注册的服务信息
11    fetch-registry: false
12    # 设置与 Eureka server交互的地址查询服务和注
13    # 册服务都需要依赖这个地址
14    service-url:
      defaultZone:
        http://${eureka.instance.hostname}:${server.
port}/eureka/

```

主启动类

```

1 /**
2  * 主启动类
3 */
4 @Slf4j
5 @SpringBootApplication
6 @EnableEurekaServer
7 public class EurekaMain7001 {
8     public static void main(String[] args) {
9
10        SpringApplication.run(EurekaMain7001.class,
11        args);
12        log.info("***** Eureka 服
13        务启动成功 端口 7001 *****");
14    }
15 }

```

测试

访问浏览器localhost:7001

The screenshot shows the Spring Eureka dashboard. At the top, it says "spring Eureka" and "HOME LAST 1000 SINCE STARTUP".

System Status

Environment	N/A	Current time	2022-02-09T17:56:06 +0800
Data center	N/A	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	337mb
num-of-cpus	12
current-memory-usage	178mb (52%)

服务注册发现_解读Eureka注册中心UI界面

System Status

Environment	N/A	Current time	02-20T17:02:53 +0800
Data center	N/A	Uptime	03:23
		Lease expiration enabled	false
		Renews threshold	3
		Renews (last min)	2

参数：

- Environment: 环境，默认为test，该参数在实际使用过程中，可以不用更改
- Data center: 数据中心，使用的是默认的是“MyOwn”
- Current time: 当前的系统时间
- Uptime: 已经运行了多少时间
- Lease expiration enabled: 是否启用租约过期，自我保护机制关闭时，该值默认是true，自我保护机制开启之后为false。
- Renews threshold: 每分钟最少续约数，Eureka Server 期望每分钟收到客户端实例续约的总数。
- Renews (last min): 最后一分钟的续约数量（不含当前，1分钟更新一次），Eureka Server 最后 1 分钟收到客户端实例续约的总数。

DS Replicas

DS Replicas

eureka7002.com

参数：

这个下面的信息是这个Eureka Server相邻节点，互为一个集群。注册到这个服务上的实例信息

Instances currently registered with Eureka

注册到Eurka服务上的实例信息。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CLOUD-PAYMENT-PROVIDER	n/a (1)	(1)	UP (1) - payment8003

参数：

- Application: 服务名称。配置的spring.application.name属性
- AMIs: n/a (1)，字符串n/a+实例的数量，我不了解
- Availability Zones: 实例的数量
- Status: 实例的状态 + eureka.instance.instance-id的值。

实例的状态分为UP、DOWN、STARTING、OUT_OF_SERVICE、UNKNOWN.

- UP：服务正常运行，特殊情况当进入自我保护模式，所有的服务依然是UP状态，所以需要做好熔断重试等容错机制应对灾难性网络出错情况
- OUT_OF_SERVICE : 不再提供服务，其他的Eureka Client将调用不到该服务，一般有人为的调用接口设置的，如：强制下线。
- UNKNOWN: 未知状态
- STARTING : 表示服务正在启动中
- DOWN: 表示服务已经宕机，无法继续提供服务

General Info

General Info	
Name	Value
total-avail-memory	352mb
num-of-cpus	12
current-memory-usage	206mb (58%)
server-upptime	03:23
registered-replicas	http://eureka7002.com:7002/eureka/
unavailable-replicas	http://eureka7002.com:7002/eureka/ ,
available-replicas	

参数：

- total-avail-memory : 总共可用的内存
- environment : 环境名称，默认test
- num-of-cpus : CPU的个数
- current-memory-usage : 当前已经使用内存的百分比
- server-upptime : 服务启动时间
- registered-replicas : 相邻集群复制节点
- unavailable-replicas : 不可用的集群复制节点，如何确定不可用？主要是server1 向server2和server3发送接口查询自身的注册信息。
- available-replicas : 可用的相邻集群复制节点

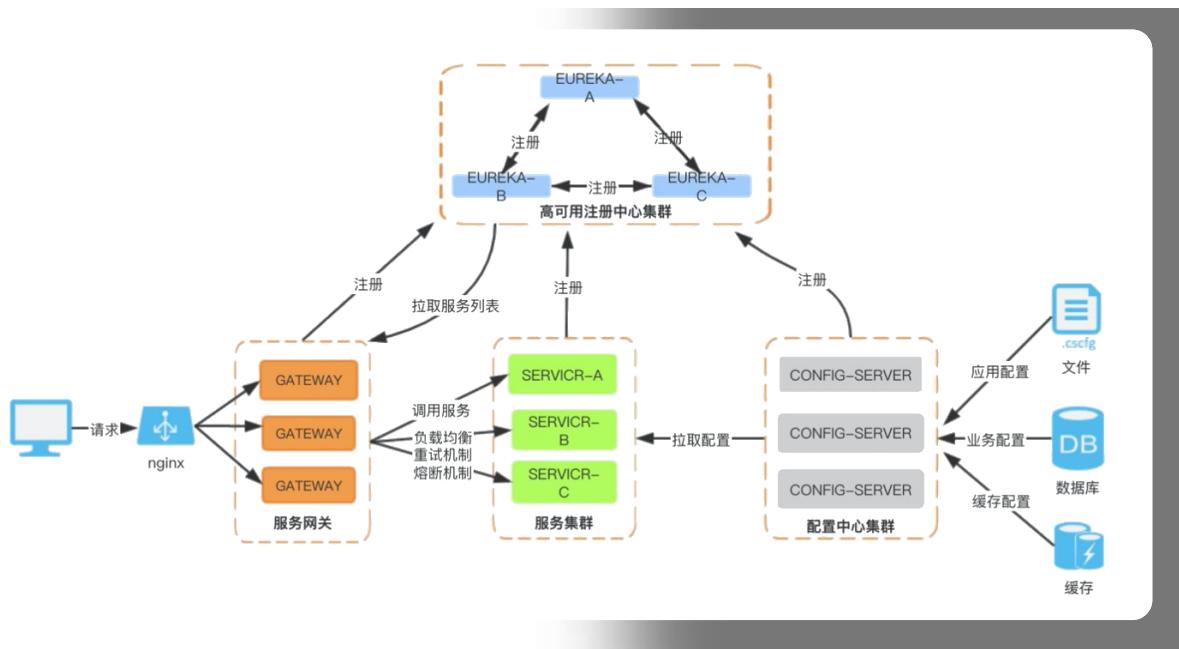
Instance Info

Instance Info	
Name	Value
ipAddr	192.168.1.8
status	UP

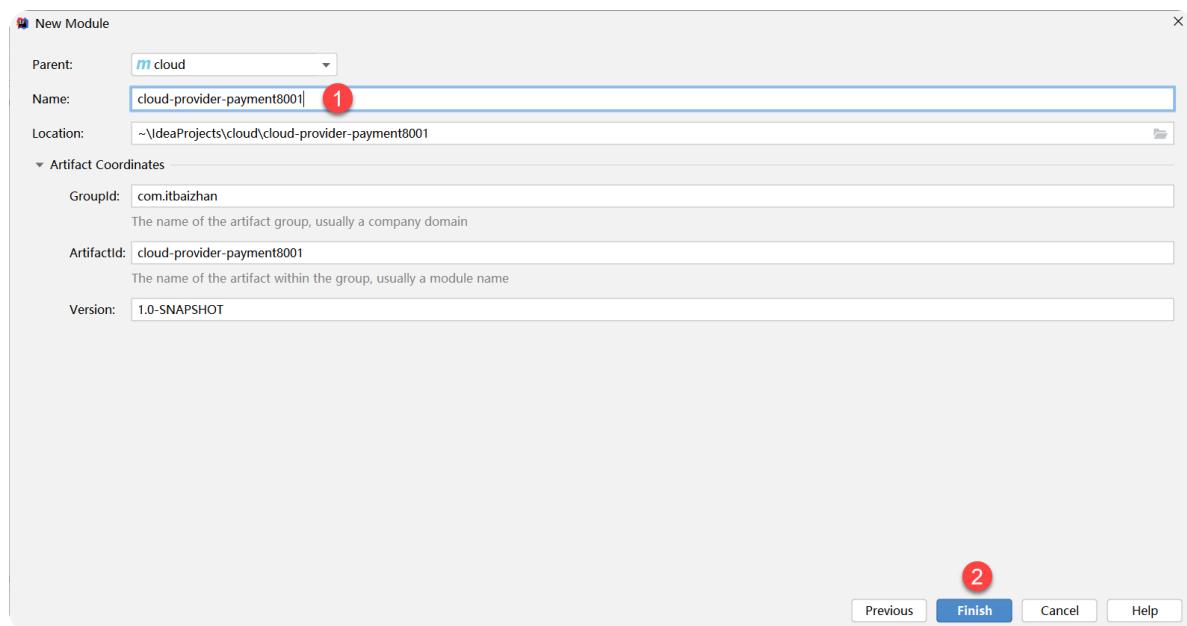
参数：

- ipAddr: eureka服务端IP
- status: eureka服务端状态

服务注册发现_创建服务提供者



创建cloud-provider-payment8001模块



pom文件添加依赖

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project
3   xmlns="http://maven.apache.org/POM/4.0.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
5     instance"

```

```
4      xsi:schemaLocation="http://maven.apache.org/
POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
5      <parent>
6          <artifactId>cloud</artifactId>
7          <groupId>com.itbaizhan</groupId>
8          <version>1.0-SNAPSHOT</version>
9      </parent>
10     <modelVersion>4.0.0</modelVersion>
11
12     <artifactId>cloud-provider-
payment8001</artifactId>
13
14     <dependencies>
15         <!-- 引入Eureka client依赖 -->
16         <dependency>
17             <groupId>org.springframework.cloud</groupId>
18             <artifactId>spring-cloud-
starter-netflix-eureka-client</artifactId>
19             </dependency>
20             <dependency>
21                 <groupId>org.springframework.boot</groupId>
22                     <artifactId>spring-boot-starter-
web</artifactId>
23                     </dependency>
24             <dependency>
25                 <groupId>org.projectlombok</groupId>
```

```
26 <artifactId>lombok</artifactId>
27 <version>1.18.22</version>
28 </dependency>
29 </dependencies>
30
31 </project>
```

写yml文件

```
1 server:
2   port: 8001
3 eureka:
4   client:
5     service-url:
6       # Eureka server 地址
7     defaultZone:
8       http://localhost:7001/eureka/
```

编写主启动类

```

1 /**
2  * 主启动类
3 */
4 @EnableEurekaClient
5 @SpringBootApplication
6 @Slf4j
7 public class PaymentMain8001 {
8     public static void main(String[] args) {
9
10        SpringApplication.run(PaymentMain8001.class
11        ,args);
12        log.info("***** 服务提供者启动成功
13        *****");
14    }
15 }
```

测试

先启动EurekaServer服务，访问<http://localhost:7001>

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
UNKNOWN	n/a (1)	(1)	UP (1) - DESKTOP-637RQ4D

General Info

Name	Value
total-avail-memory	343mb
num-of-cpus	12
current-memory-usage	75mb (21%)
server-upptime	00:14

注意：

application名字未定义。

修改yml文件

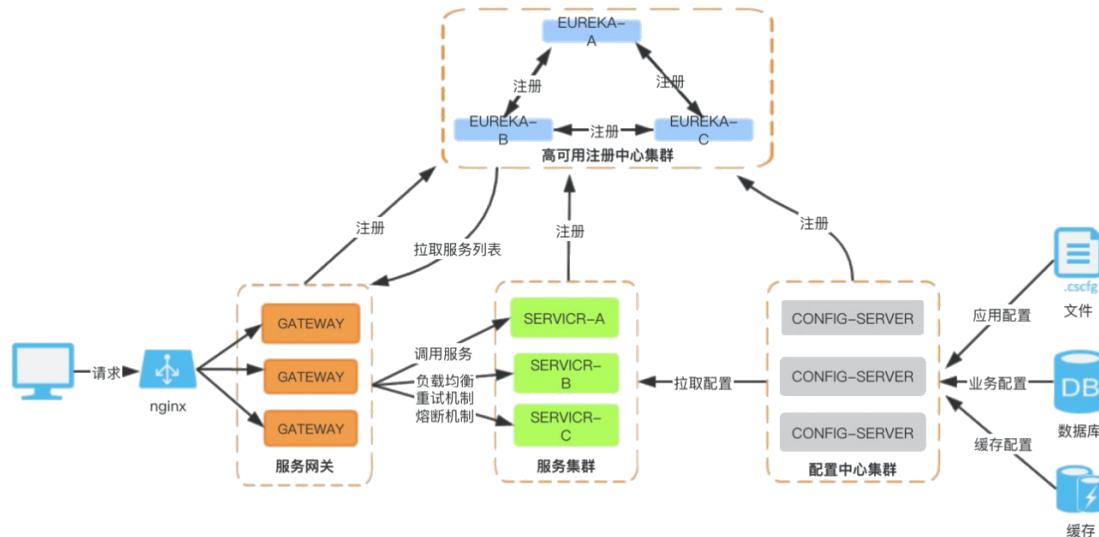
```

1 spring:
2   application:
3     # 设置应用名词
4     name: cloud-payment-provider

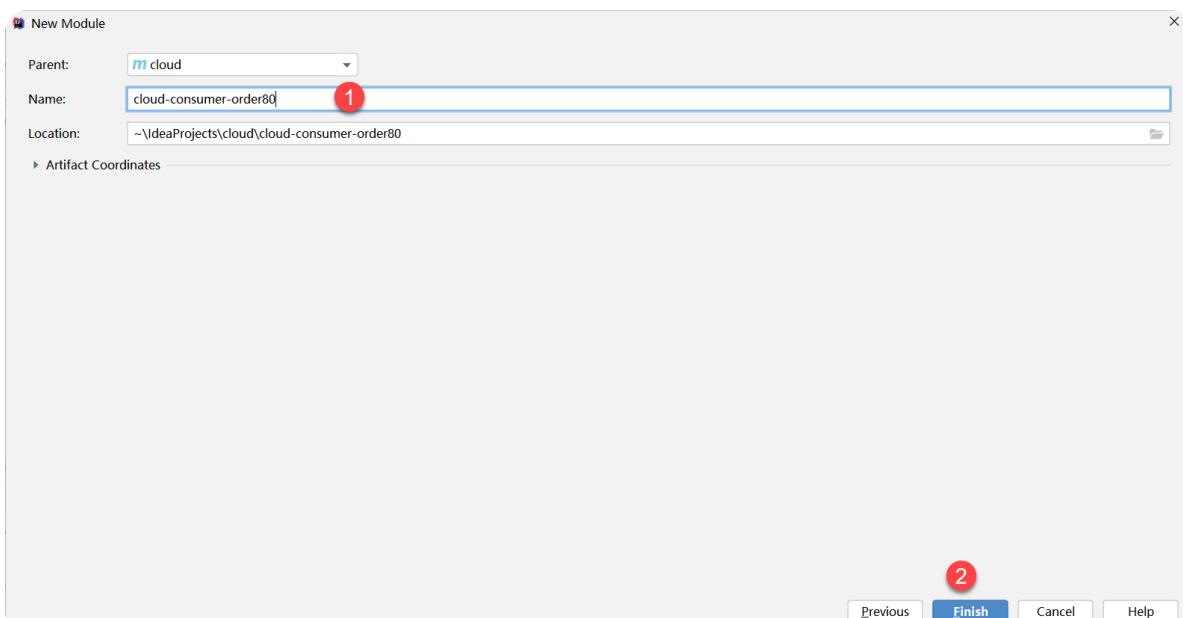
```



服务注册发现_创建服务消费者



创建cloud-consumer-order80模块



pom文件添加依赖

```
1 <dependencies>
2     
3         <dependency>
4
5             <groupId>org.springframework.cloud</groupId>
6
7                 <artifactId>spring-cloud-
8                     starter-netflix-eureka-client</artifactId>
9
10                </dependency>
11                <dependency>
12
13                    <groupId>org.springframework.boot</groupId>
14                        <artifactId>spring-boot-starter-
15                            web</artifactId>
16
17                </dependency>
18                <dependency>
19
20                    <groupId>org.projectlombok</groupId>
21                        <artifactId>lombok</artifactId>
22                        <version>1.18.22</version>
23
24                </dependency>
25            </dependencies>
```

写yml文件

```

1 eureka:
2   client:
3     # Eureka Server地址
4     service-url:
5       defaultZone:
6       http://localhost:7001/eureka
7   spring:
8     application:
9       # 设置应用名字
10      name: cloud-order-consumer
11
12    server:
13      port: 80

```

编写主启动类

```

1 /**
2  * 主启动类
3 */
4 @SpringBootApplication
5 @EnableEurekaClient
6 @Slf4j
7 public class OrderMain80 {
8   public static void main(String[] args) {
9
10     SpringApplication.run(OrderMain80.class,args);
11     log.info("***** 订单服务消费者启动成功 *****");
12   }
13 }

```

测试

先启动EurekaServer服务，访问<http://localhost:7001>

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-ORDER-CONSUMER	n/a (1)	(1)	UP (1) - DESKTOP-637RQ4D.cloud-order-consumer:80
CLOUD-PAYMENT-PROVIDER	n/a (1)	(1)	UP (1) - DESKTOP-637RQ4D.cloud-payment-provider

服务注册发现_服务自保和服务剔除机制

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

Instances currently registered with Eureka

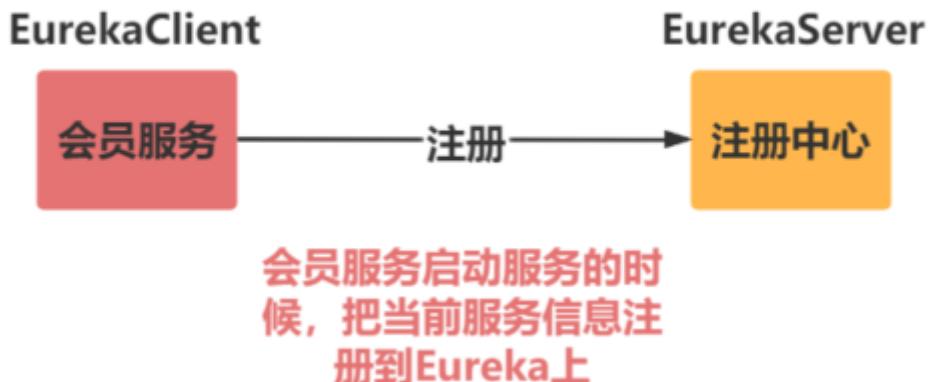
Application	AMIs	Availability Zones	Status
CLOUD-ORDER-CONSUMER	n/a (1)	(1)	UP (1) - DESKTOP-637RQ4D.cloud-order-consumer:80
CLOUD-PAYMENT-PROVIDER	n/a (1)	(1)	UP (1) - DESKTOP-637RQ4D.cloud-payment-provider



自保机制说白了就是好死不如赖活着。

服务剔除，服务自保，这两套功法一邪一正，俨然就是失传多年的上乘心法的上卷和下卷。但是往往你施展了服务剔除便无法施展服务自保，而施展了服务自保，便无法施展服务剔除。也就是说，注册中心在同一时刻，只能施展一种心法，不可两种同时施展。

服务剔除



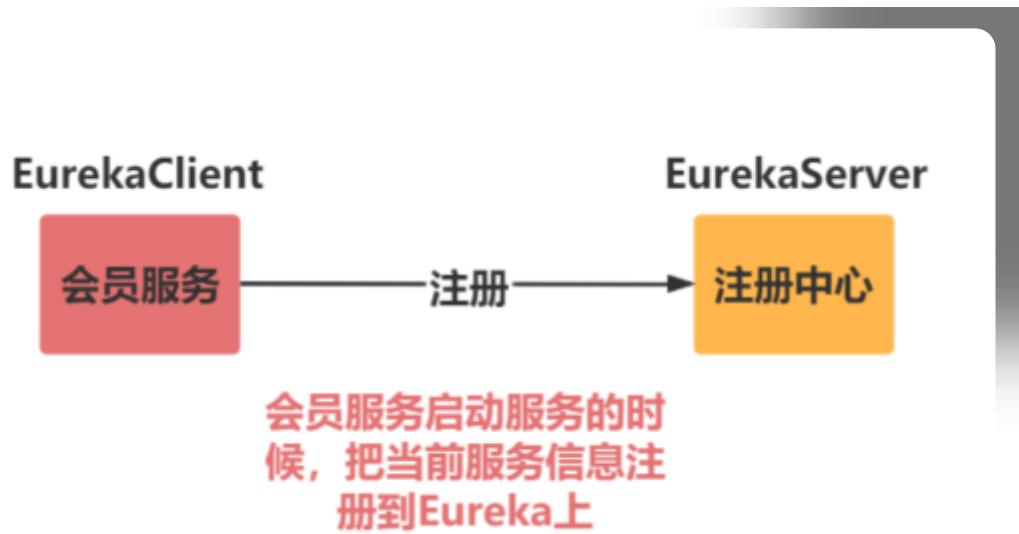
注意：

服务剔除把服务节点果断剔除，即使你的续约请求晚了一步也毫不留情，招式凌厉，重在当断则断，忍痛割爱。

心法总决简明扼要：

欲练此功，必先自宫

服务自保



注意：

服务自保把当前所有节点保留，一个都不能少，绝不放弃任何队友。心法的指导思想是，即便主动删除，也许并不能解决问题，且放之任之，以不变应万变。

心法总决引人深思：

宫了以后，未必成功

如果不宫，或可成功

心法总纲

在实际应用里，并不是所有无心跳的服务都不可用，也许因为短暂的网络抖动等原因，导致服务节点与注册中心之间续约不上，但服务节点之间的调用还是属于可用状态，这时如果强行剔除服务节点，可能会造成大范围的业务停滞。

Eureka服务自保的触发机关

自动开关

The screenshot shows the Spring Eureka dashboard with the following details:

System Status

Environment	N/A	Current time	02-09T18:30:38 +0800
Data center	N/A	Uptime	0:10
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	2

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-CONSUMER	n/a (1)	{1}	UP (1) - DESKTOP-637RQ4D:cloud-order-consumer:80
CLOUD-PAYMENT-PROVIDER	n/a (1)	{1}	UP (1) - DESKTOP-637RQ4D:cloud-payment-provider

注意：

服务自保模式往往是为了应对短暂的网络环境问题，在理想情况下服务节点的续约成功率应该接近100%，如果突然发生网络问题，比如一部分机房无法连接到注册中心，这时候续约成功率有可能大幅降低。但考虑到Eureka采用客户端的服务发现模式，客户端手里有所有节点的地址，如果服务节点只是因为网络原因无法续约但其自身服务是可用的，那么客户端仍然可以成功发起调用请求。这样就避免了被服务剔除给错杀。

手动开关

这是服务自保的总闸，以下配置将强制关闭服务自保，即便上面的自动开关被触发，也不能开启自保功能。

- 1 # 参数来关闭保护机制，以确保注册中心可以将不可用的实例正确剔除，默认为true。
- 2 `eureka.server.enable-self-preservation=false;`

实时效果反馈

1. Eureka技术中服务自保主要应对__问题。

- A 网络环境
- B 单点故障

C 兼容

D 安全

2. Eureka技术中如何手动关闭服务自保的总闸____。

A `eureka.server.enable-self=true;`

B `eureka.server.enable-self=false;`

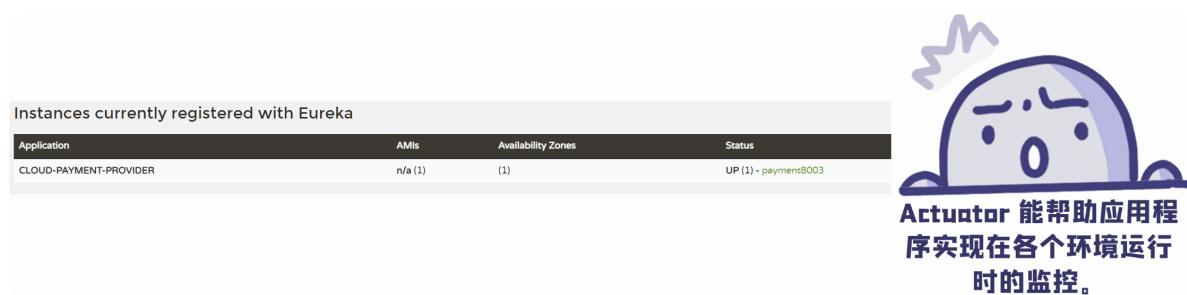
C `eureka.server.enable-self-preservation=true;`

D `eureka.server.enable-self-preservation=false;`

答案

1=>A 2=>D

服务注册发现_actuator微服务信息完善



The screenshot shows the Eureka dashboard with the heading "Instances currently registered with Eureka". A table lists one instance: "CLOUD-PAYMENT-PROVIDER" with 1 AMI and 1 Availability Zone, status UP (1) - payment:8003. To the right, a purple cartoon character with a thinking bubble says: "Actuator 能帮助应用程序实现在各个环境运行时的监控。"

SpringCloud体系里的，服务实体向eureka注册时，注册名默认是IP名:应用名:应用端口名。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-CONSUMER	n/a (1)	(1)	UP (1) - DESKTOP-637RQ4D:cloud-order-consumer:80
CLOUD-PAYMENT-PROVIDER	n/a (1)	(1)	UP (1) - DESKTOP-637RQ4D:cloud-payment-provider:8001

问题：

自定义服务在Eureka上的实例名怎么弄呢

在服务提供者pom中配置Actuator依赖

```

1 <!-- actuator监控信息完善 -->
2 <dependency>
3
4   <groupId>org.springframework.boot</groupId>
5     <artifactId>spring-boot-starter-
6       actuator</artifactId>
7   </dependency>

```

在服务提供者生产者application.yml中加入

```

1 eureka:
2   instance:
3     #根据需要自己起名字
4     instance-id: springcloud-dept-8001

```

测试

DS Replicas			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-PAYMENT-PROVIDER	n/a (1)	(1)	UP (1) - cloud-payment-provider

实时效果反馈

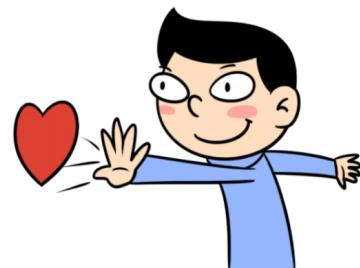
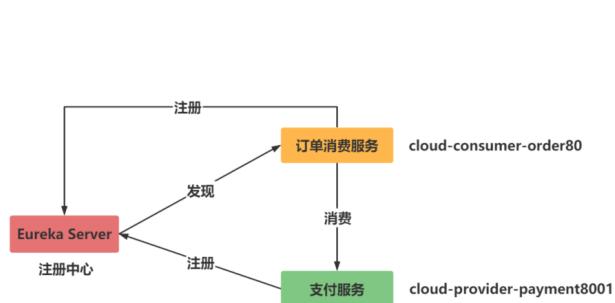
1.下列自定义服务在Eureka上的实例名正确的是_____。

- A eureka.instance.instance
- B eureka.instance.id
- C eureka.instance.instance-id
- D 以上都是错误

答案

1=>C

服务注册发现_服务发现Discovery



注册进入Eureka里面的微服务，可以通过服务发现来获得该服务的信息。

修改payment8001的Controller

```
1 /**
2  * 支付控制层
3 */
4 @Slf4j
5 @RestController
6 public class PaymentController {
7
8     @Autowired
9     private DiscoveryClient discoveryClient;
10
11    @GetMapping("/payment/discovery")
12    public Object discovery(){
13        // 获取所有微服务信息
14        List<String> services =
discoveryClient.getServices();
15        for (String service : services) {
```

```

16         log.info("server:={}", service);
17     }
18     return this.discoveryClient;
19 }
20
21 }
```

RestTemplate介绍

RestTemplate 是从 Spring3.0 开始支持的一个 HTTP 请求工具，它提供了常见的REST请求方案的模版，例如 GET 请求、POST 请求、PUT 请求、DELETE 请求以及一些通用的请求执行方法 exchange 以及 execute。

在配置类中的restTemplate添加@LoadBalanced注解

这个注解会 给这个组件 有负载均衡的功能

```

1 @Configuration
2 public class CloudConfig {
3     @LoadBalanced
4     @Bean
5     public RestTemplate restTemplate(){
6         return new RestTemplate();
7     }
8 }
```

修改payment8001工程controller

```

1 @Slf4j
2 @RestController
3 @RequestMapping("/payment")
4 public class PaymentController {
5
6     @GetMapping("/index")
7     public String index(){
8         return "payment + success";
9     }
10 }
```

编写order80工程Controller

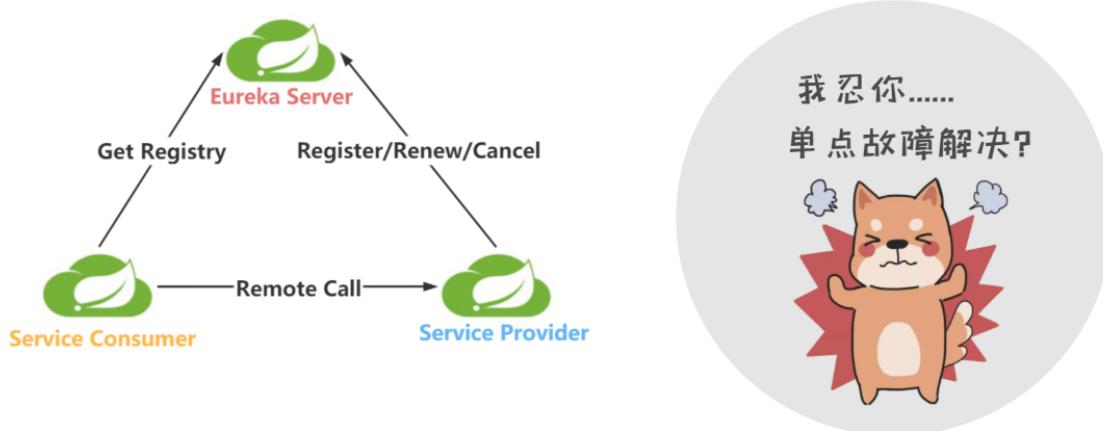
```

1 @RestController
2 @RequestMapping("/order")
3 public class OrderController {
4
5     // HTTP 请求工具
6     @Autowired
7     private RestTemplate restTemplate;
8
9     /**
10      * 测试服务发现接口
11      * @return
12      */
13     @GetMapping("/index")
14     public String index(){
15         //1.远程调用方法的主机
16         //String host = "http://localhost:1000";
17         //将远程微服务调用地址从"IP地址+端口号改
18         //成"微服务名称"
19         String host = "http://cloud-payment-
20         provider";
```

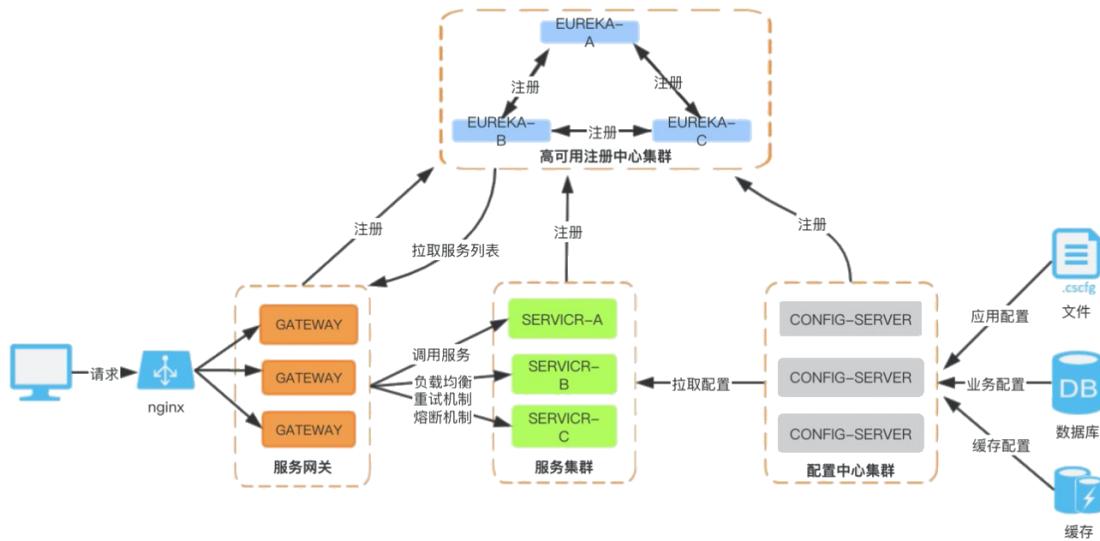
```

19 // 2. 远程调用方法具体URL地址
20     String url = "/payment/index";
21     // 3. 发起远程调用
22     //getForObject: 返回响应体中数据转化成的
23     //对象，可以理解为json
24     //getForEntity: 返回的是ResponseEntity
25     //的对象包含了一些重要的信息
26     String forObject =
27         restTemplate.getForObject(host + url,
28             String.class);
29     return forObject;
30 }
31 }
```

服务注册发现_高可用Eureka注册中心



在微服务架构这样的分布式环境中，我们需要充分考虑发生故障的情况，所以在生产环境中必须对各个组件进行高可用部署，对于微服务如此，对于服务注册中心也一样。



问题：

Spring-Cloud为基础的微服务架构，所有的微服务都需要注册到注册中心，如果这个注册中心阻塞或者崩了，那么整个系统都无法继续正常提供服务，所以，这里就需要对注册中心搭建，高可用（HA）集群。

Eureka Server的设计一开始就考虑了高可用问题，在Eureka的服务治理设计中，所有节点即是服务提供方，也是服务消费方，服务注册中心也不例外。是否还记得在单节点的配置中，我们设置过下面这两个参数，让服务注册中心不注册自己：

- 1 `eureka.client.register-with-eureka=false`
- 2 `eureka.client.fetch-registry=false`

实时效果反馈

1. 单机Eureka注册中心服务会出现____问题。

- A 网络环境
- B 单点故障
- C 兼容

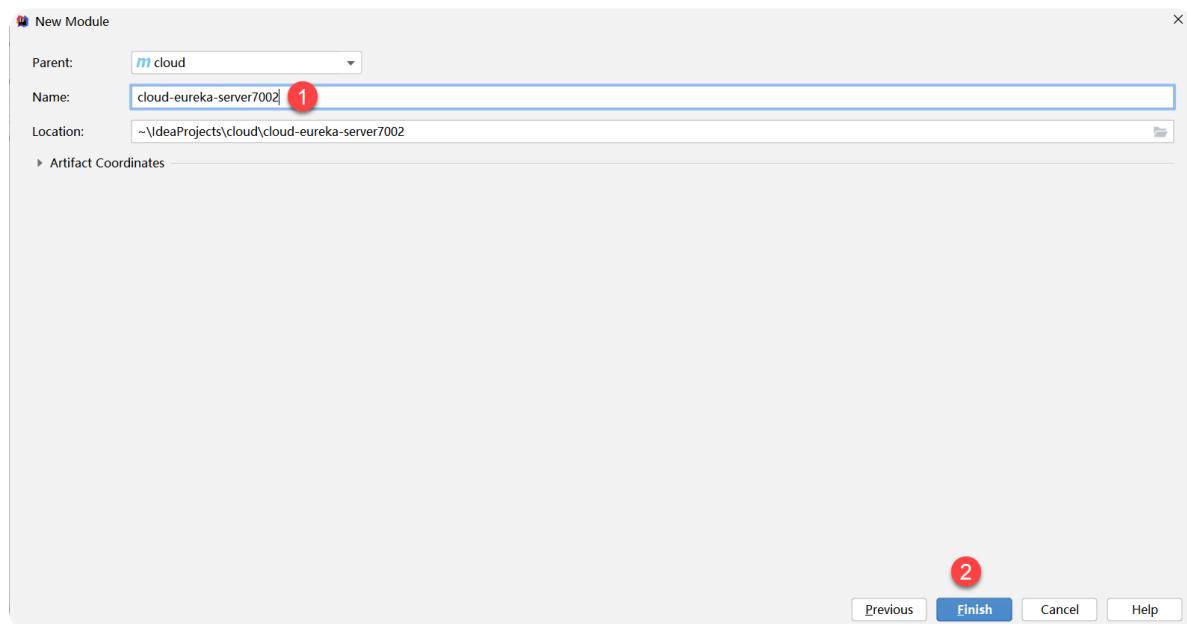
D 安全

答案

1=>B

服务注册发现_高可用Eureka注册中心搭建

构建cloud-eureka-server7002工程



修改Pom

```

1 <dependencies>
2   <!-- 服务注册发现Eureka-->
3     <dependency>
4
5       <groupId>org.springframework.cloud</groupId>
6
7         <artifactId>spring-cloud-
8           starter-netflix-eureka-server</artifactId>
9         </dependency>
10        <dependency>
11
12          <groupId>org.springframework.cloud</groupId>
13
14            <artifactId>spring-cloud-
15              starter-netflix-eureka-client</artifactId>
16            </dependency>
17
18          <dependency>
19
20            <groupId>org.springframework.cloud</groupId>
21
22              <artifactId>spring-cloud-
23                starter-netflix-eureka-discovery</artifactId>
24              </dependency>
25
26        </dependencies>
27      
```

```

8 <groupId>org.projectlombok</groupId>
9   <artifactId>lombok</artifactId>
10  </dependency>
11  <dependency>
12
13    <groupId>org.springframework.boot</groupId>
14      <artifactId>spring-boot-starter-
15 test</artifactId>
16      <scope>test</scope>
17    </dependency>
18  </dependencies>

```

修改映射配置

找到C:\Windows\System32\drivers\etc\hosts

```

1 #添加如下配置
2 127.0.0.1 eureka7001.com
3 127.0.0.1 eureka7002.com

```

修改7001YML文件

```

1 #修改7001主机yml文件
2 server:
3   port: 7001
4 eureka:
5   instance:
6     # eureka服务端的实例名字
7     hostname: eureka7001.com
8 client:
9   #表示是否将自己注册到Eureka Server
10  register-with-eureka: false
11  # 表示是否从Eureka Server获取注册的服务信息

```

```

12   fetch-registry: false
13   # 设置与 Eureka server交互的地址查询服务和注
14   #册服务都需要依赖这个地址
15   service-url:
16     defaultZone:
17       http://eureka7002.com:7002/eureka/

```

修改7002YML文件

```

1 #修改7001主机yml文件
2 server:
3   port: 7002
4 eureka:
5   instance:
6     # eureka服务端的实例名字
7     hostname: eureka7002.com
8   client:
9     #表示是否将自己注册到Eureka Server
10    register-with-eureka: false
11    # 表示是否从Eureka Server获取注册的服务信息
12    fetch-registry: false
13    # 设置与 Eureka server交互的地址查询服务和注
14    #册服务都需要依赖这个地址
15    service-url:
16      defaultZone:
17        http://eureka7001.com:7001/eureka/

```

编写主启动类

```

1 /**
2  * 主启动类
3 */
4 @Slf4j
5 @SpringBootApplication
6 @EnableEurekaServer
7 public class EurekaMain7002 {
8     public static void main(String[] args) {
9
10         SpringApplication.run(EurekaMain7002.class,
11         args);
12         log.info("***** Eureka 服务
13         启动成功 *****");
14     }
15 }

```

将支付微服务8001发布到Eureka集群上

```

1 eureka:
2   client:
3     service-url:
4       defaultZone:
5         http://eureka7001.com:7001/eureka,http://eure
6         ka7002.com:7002/eureka

```

将订单微服务80发布到Eureka集群上

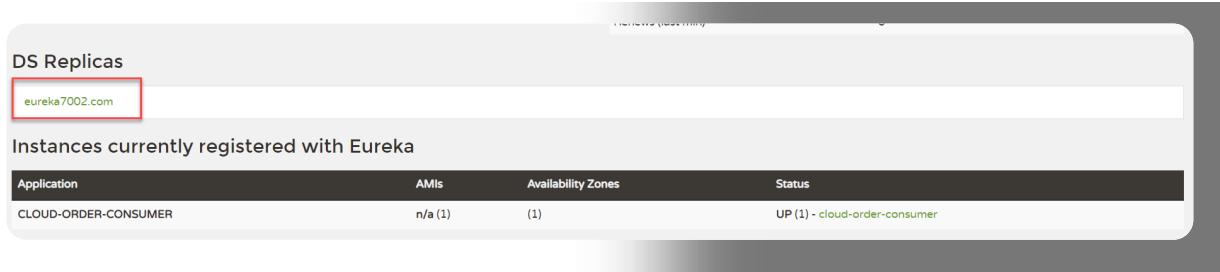
```

1 eureka:
2   client:
3     service-url:
4       defaultZone:
5         http://eureka7001.com:7001/eureka,http://eure
6         ka7002.com:7002/eureka

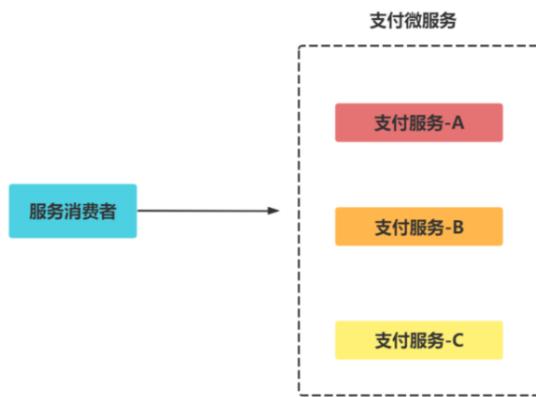
```

测试

- ① 先启动EurekaServer集群
- ② 在启动服务提供者provider服务
- ③ 在启动消费者服务



客户端负载均衡_什么是负载均衡



负载均衡是我们处理高并发、缓解网络压力和进行服务器扩容的重要手段之一。

为什么需要负载均衡

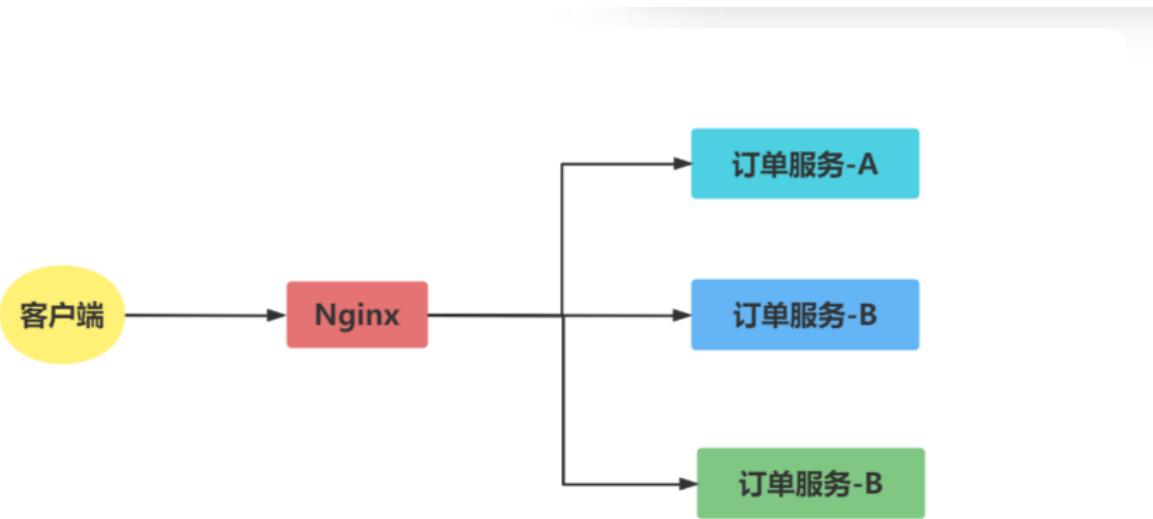
俗话说在生产队薅羊毛不能逮着一只羊薅，在微服务领域也是这个道理。面对一个庞大的微服务集群，如果你每次发起服务调用都只盯着那一两台服务器，在大用户访问量的情况下，这几台被薅羊毛的服务器一定会不堪重负。

负载均衡要干什么事情

负载均衡有两大门派，**服务端负载均衡**和**客户端负载均衡**。我们先来聊聊这两个不同门派的使用场景，再来看看本节课的主角Spring Cloud Loadbalancer 属于哪门哪派。

服务端负载均衡

在服务集群内设置一个中心化负载均衡器，比如Nginx。发起服务间调用的时候，服务请求并不直接发向目标服务器，而是发给这个全局负载均衡器，它再根据配置的负载均衡策略将请求转发到目标服务。



优点：

- 服务端负载均衡应用范围非常广，它不依赖于服务发现技术，客户端并不需要拉取完整的服务列表；同时，发起服务调用的客户端也不用操心该使用什么负载均衡策略。

劣势：

- 网络消耗
- 复杂度和故障率提升

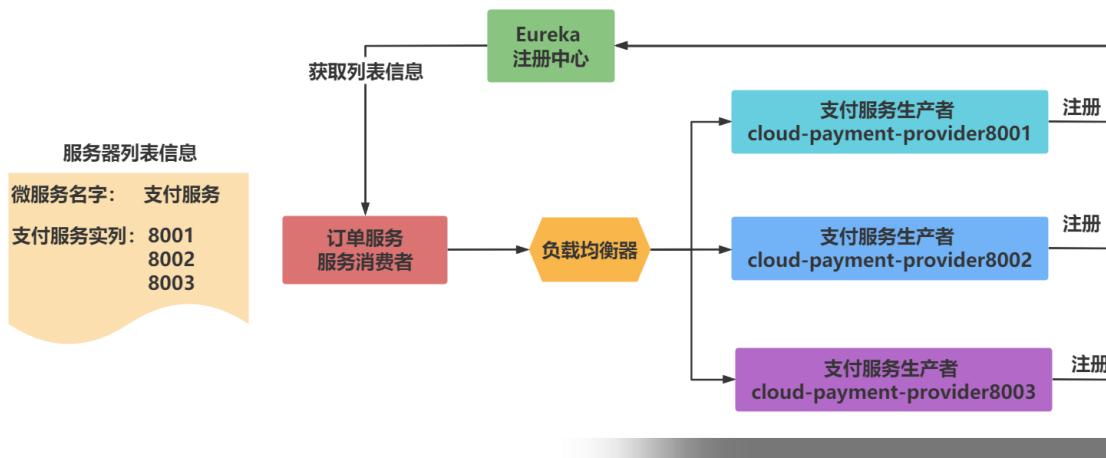
Spring Cloud Loadbalancer 可以很好地弥补上面的劣势，那么它是如何做到的呢？

客户端负载均衡

Spring Cloud Loadbalancer 采用了客户端负载均衡技术，每个发起服务调用的客户端都存有完整的目标服务地址列表，根据配置的负载均衡策略，由客户端自己决定向哪台服务器发起调用。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CLOUD-PAYMENT-PROVIDER	n/a (1)	(1)	UP (1) - cloud-payment-provider8002 cloud-payment-provider8001



优势：

- 网络开销小
- 配置灵活

劣势：

需要满足一个前置条件，发起服务调用的客户端需要获取所有目标服务的地址，这样它才能使用负载均衡规则选取要调用的服务。也就是说，客户端负载均衡技术往往需要依赖服务发现技术来获取服务列表。

负载均衡需要解决两个最基本的问题：

第一个是从哪里选服务实例

在Spring Cloud的Eureka微服务系统中，维护微服务实例清单的是Eureka服务治理中心，而具体微服务实例会执行服务获取，获得微服务实例清单，缓存到本地，同时，还会按照一个时间间隔更新这份实例清单（因为实例清单也是在不断维护和变化的）。

第二个是如何选择服务实例

通过过负载均衡的策略从服务实例清单列表中选择具体实例。

注意：

Eureka和Loadbalancer自然而然地到了一起，一个通过服务发现获取服务列表，

另一个使用负载均衡规则选出目标服务器，然后过着没羞没躁的生活。

什么是Spring Cloud Ribbon

Spring Cloud Ribbon是NetFlix发布的负载均衡器，它有助于Http和Tcp的客户端行为。可以根据负载均衡算法（轮询、随机或自定义）自动帮助消费者请求，默认就是轮询。

问题：

- 状态 - 停更进维
- 替代方案 - Spring Cloud Loadbalancer

什么是Spring Cloud LoadBalancer

但是由于Ribbon已经进入维护模式，并且Ribbon 2并不与Ribbon 1相互兼容，所以Spring Cloud全家桶在Spring Cloud Commons项目中，添加了Spring cloud Loadbalancer作为新的负载均衡器，并且做了向前兼容，就算你的项目中继续用 Spring Cloud Netflix 套装（包括Ribbon，Eureka，Zuul，Hystrix等等）让你的项目中有这些依赖，你也可以通过简单的配置，把Ribbon替换成Spring Cloud LoadBalancer。

实时效果反馈

1. Spring Cloud Ribbon是NetFlix发布的_____。

- A 网络环境
- B 单点故障
- C 负载均衡器
- D 安全

2. Spring Cloud Ribbon替代方案是_____。

- A Spring Cloud LoadBalancer

B Spring Cloud Eureka

C Spring Cloud OpenFeign

D Spring Cloud Gateway

答案

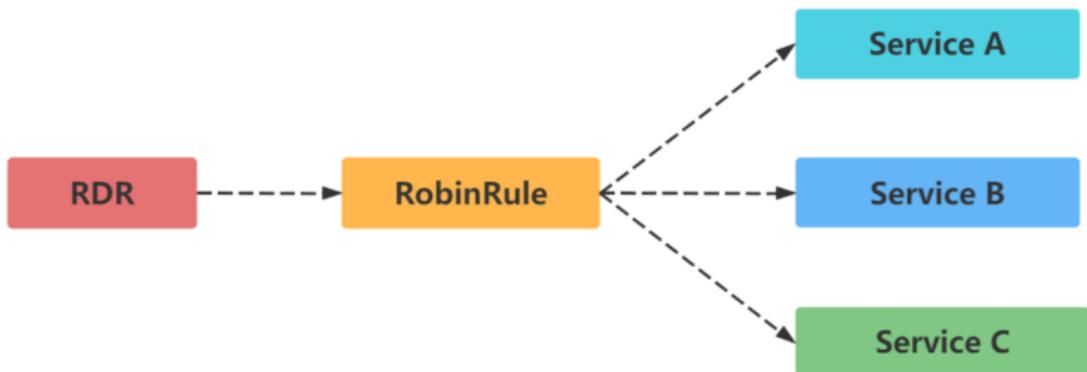
1=>C 2=>A

客户端负载均衡_负载均衡策略



以前的Ribbon有多种负载均衡策略

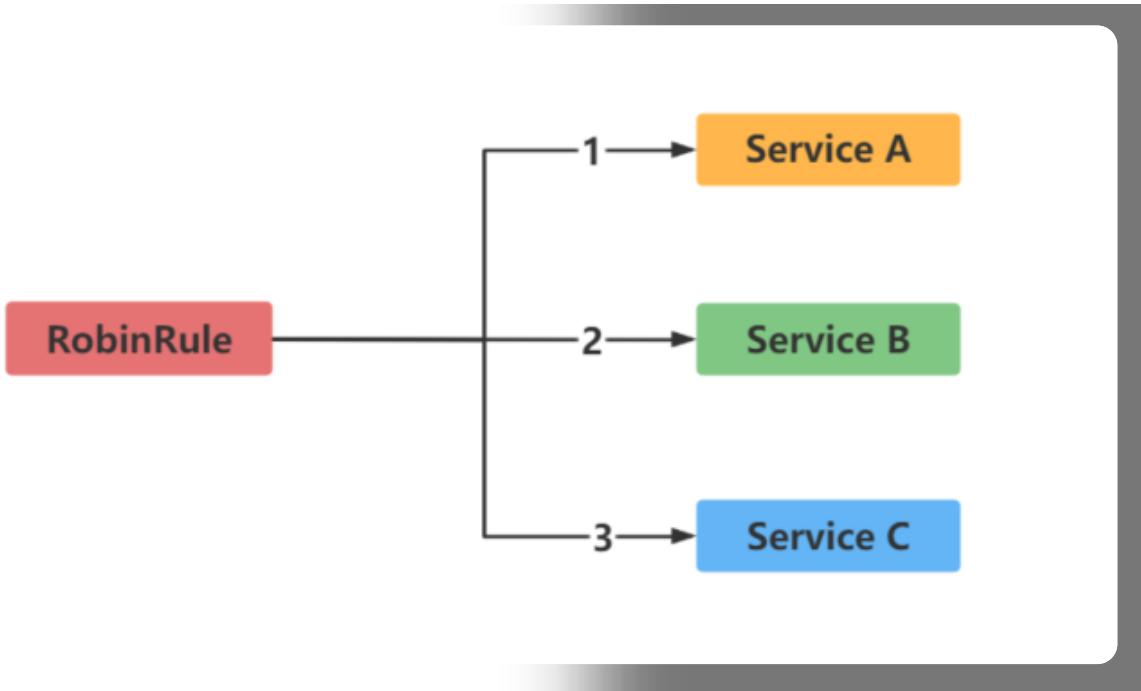
RandomRule - 随性而为



解释:

随机

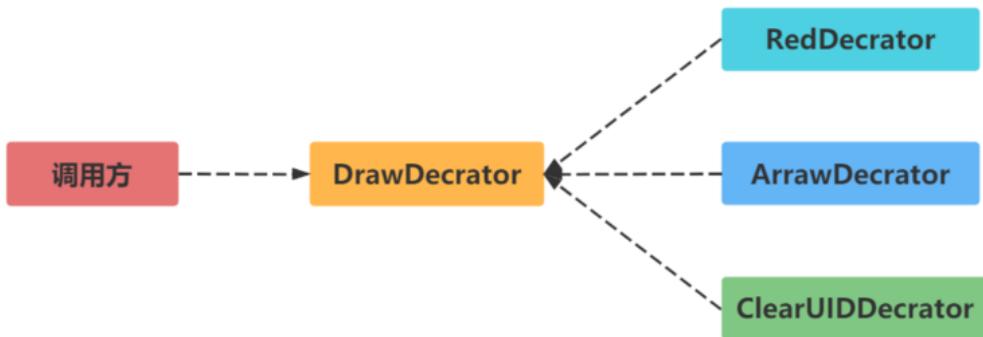
RoundRobinRule - 按部就班



解释:

轮询

RetryRule - 卷土重来

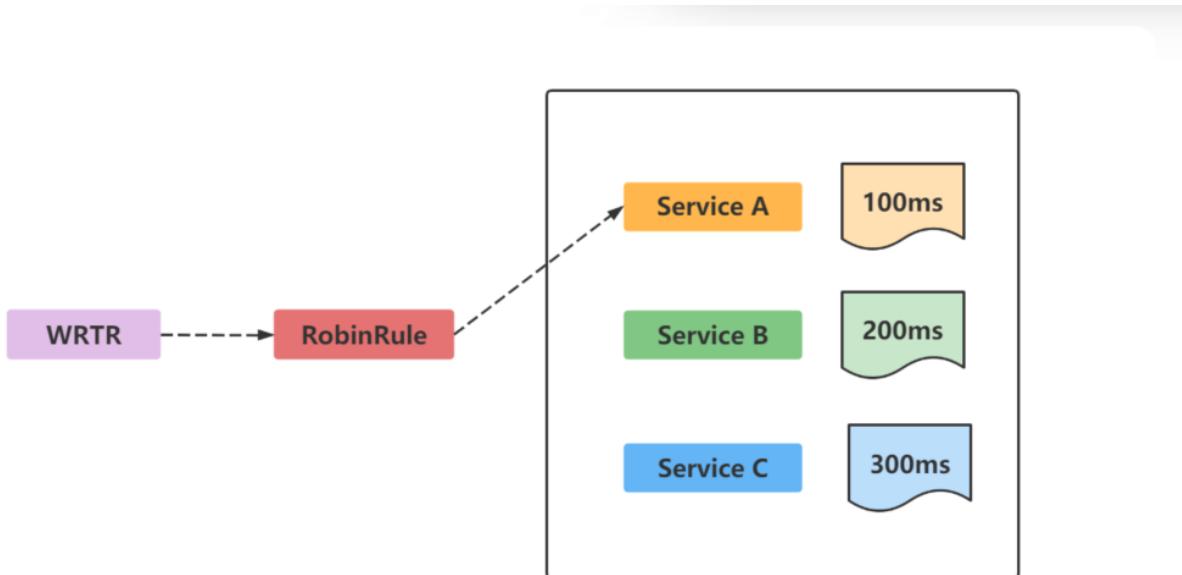


解释:

先按照RoundRobinRule的策略获取服务，如果获取服务失败则在指定时间内会进行重试。

WeightedResponseTimeRule - 能者多劳

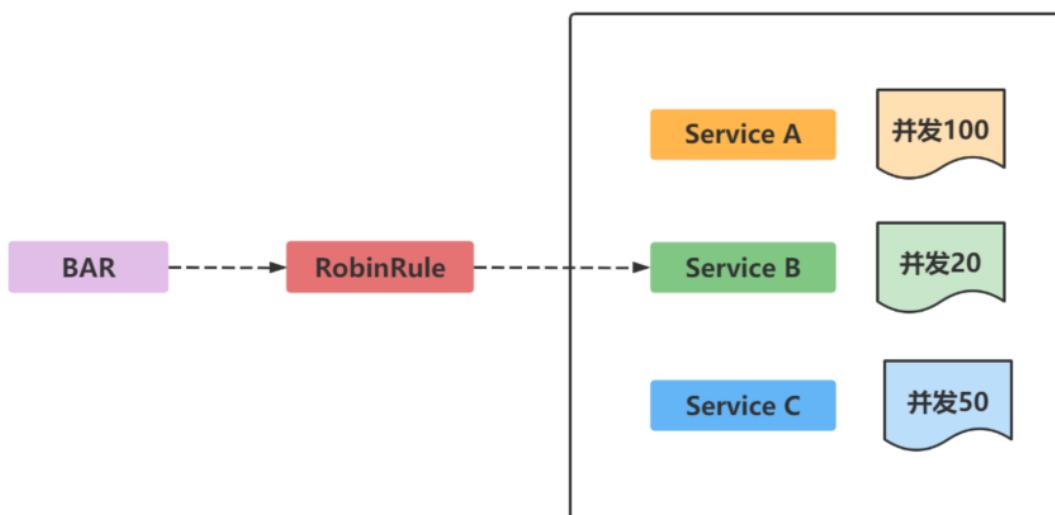
这个Rule继承自RoundRobinRule，他会根据服务节点的响应时间计算权重，响应时间越长权重就越低，响应越快则权重越高，权重的高低决定了机器被选中概率的高低。也就是说，响应时间越小的机器，被选中的概率越大。



解释：

对RoundRobinRule的扩展，响应速度越快的实例选择权重越大，越容易被选择

BestAvailableRule - 让最闲的人来

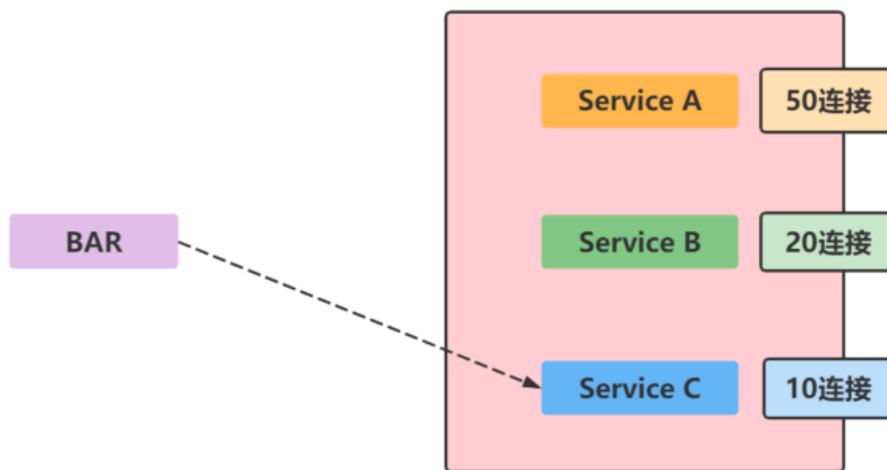


解释：

应该说这个Rule有点智能的味道了，在过滤掉故障服务以后，它会基于过去30分钟的统计结果选取当前并发量最小的服务节点，也就是最“闲”的节点作为目标地址。如果统计结果尚未生成，则采用轮询的方式选定节点。

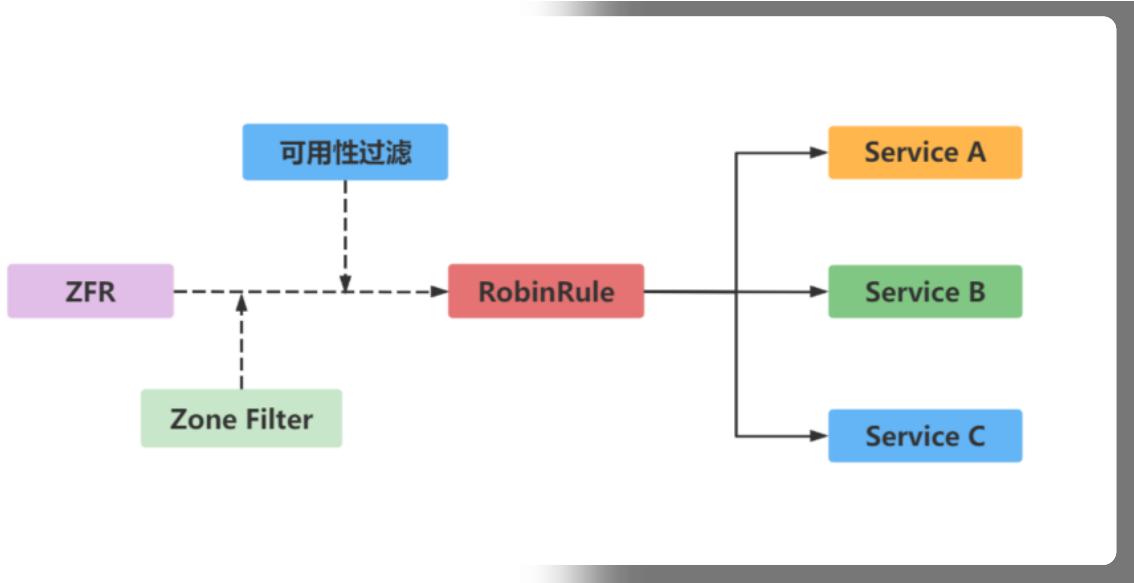
AvailabilityFilteringRule - 我是有底线的

这个规则底层依赖RandomRobinRule来选取节点，但并非来者不拒，它也是有一些底线的，必须要满足它的最低要求的节点才会被选中。如果节点满足了要求，无论其响应时间或者当前并发量是什么，都会被选中。



解释：

每次AvailabilityFilteringRule（简称AFR）都会请求RobinRule挑选一个节点，然后对这个节点做以下两步检查：是否处于不可用，节点当前的active请求连接数超过阈值，超过了则表示节点目前太忙，不适合接客如果被选中的server不幸挂掉了检查，那么AFR会自动重试（次数最多10次），让RobinRule重新选择一个服务节点。



解释:

默认规则，复合判断server所在区域的性能和server的可用性选择服务器

但LoadBalancer只提供了两种负载均衡器

- RandomLoadBalancer 随机
- RoundRobinLoadBalancer 轮询

注意:

不指定的时候默认用的是轮询

实时效果反馈

1.下列是轮询负载均衡策略的是_____。

- A RandomLoadBalancer
- B RoundRobinLoadBalancer
- C ZoneAvoidanceRule
- D AvailabilityFilteringRule

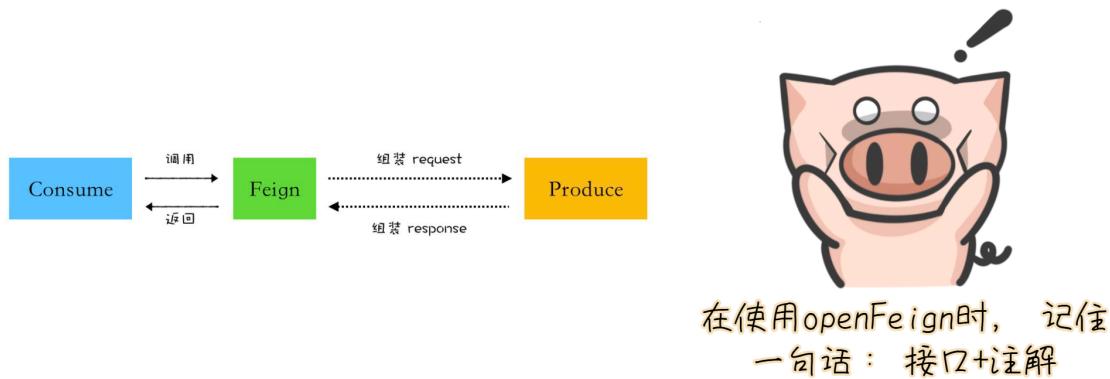
2.下列是随机负载均衡策略的是_____。

- A RandomLoadBalancer
- B RoundRobinLoadBalancer
- C ZoneAvoidanceRule
- D AvailabilityFilteringRule

答案

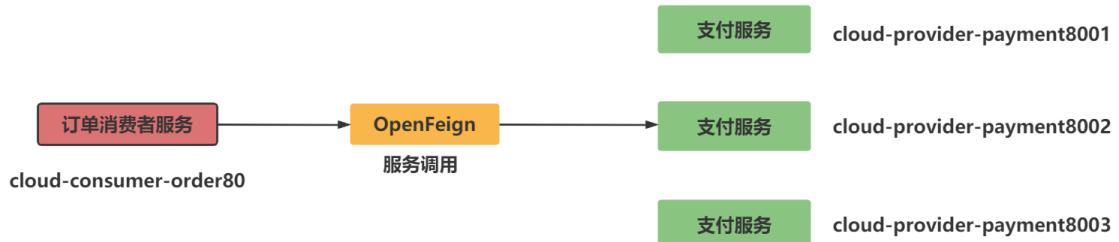
1=>B 2=>A

服务接口调用_OpenFeign概述



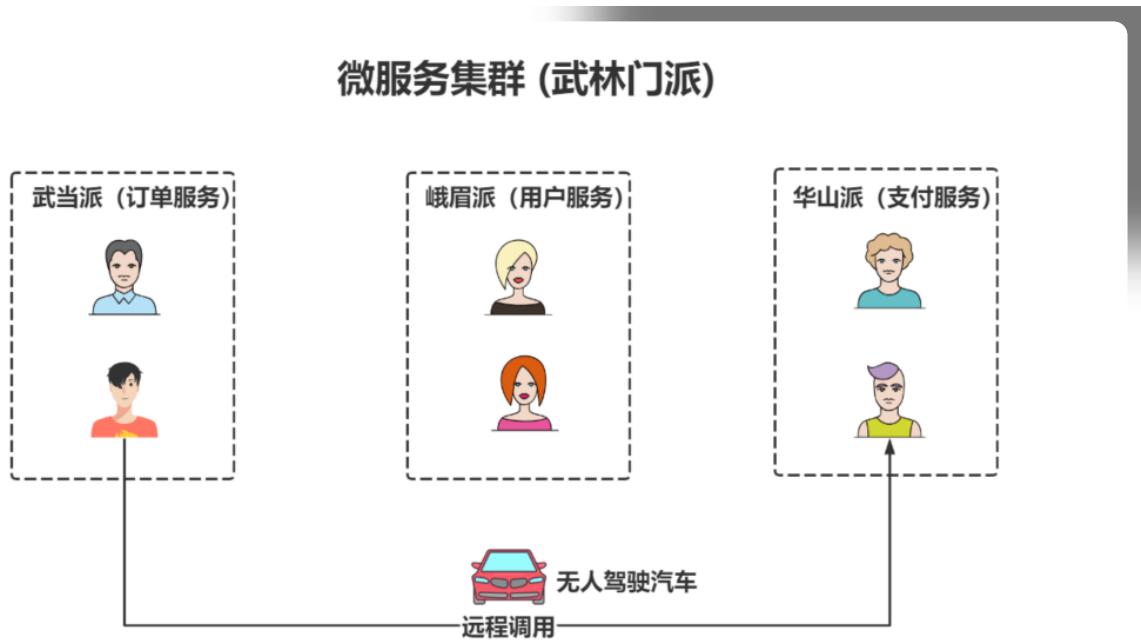
OpenFeign是什么

Spring Cloud OpenFeign用于Spring Boot应用程序的声明式REST客户端。



OpenFeign能干嘛

Feign旨在使编写Java Http客户端变得更容易。前面在使用RestTemplate时，利用RestTemplate对http请求的封装处理，形成了一套模版化的调用方法。



OpenFeign和Feign两者区别

Feign是一个声明式WebService客户端。使用Feign能让编写WebService客户端更加简单。它的使用方法是定义一个服务接口然后在上面添加注解。Feign也支持可拔插式的编码器和解码器。Spring Cloud对Feign进行了封装，使其支持了Spring MVC标准注解和HttpMessageConverters。

Feign	OpenFeign
Feign是Spring Cloud组件中的一个轻量级RESTful的HTTP服务客户端Feign内置了Ribbon，用来做客户端负载均衡，去调用服务注册中心的服务。	OpenFeign是Spring Cloud在Feign的基础上支持了SpringMVC的注解，如@RequesMapping等等。OpenFeign的@Feignclient可以解析SpringMVc的@RequestMapping注解下的接口，并通过动态代理的方式产生实现类，实现类中做负载均衡并调用其他服务。
Spring-cloud-starter-feign	spring-cloud-starter-openfeign

注意：

接口+注解。

实时效果反馈

1. Spring Cloud OpenFeign主要作用是____。

- A 负载均衡
- B 解决单点故障
- C 服务调用
- D 注册中心

2. Spring Cloud OpenFeign支持了____标准注解。

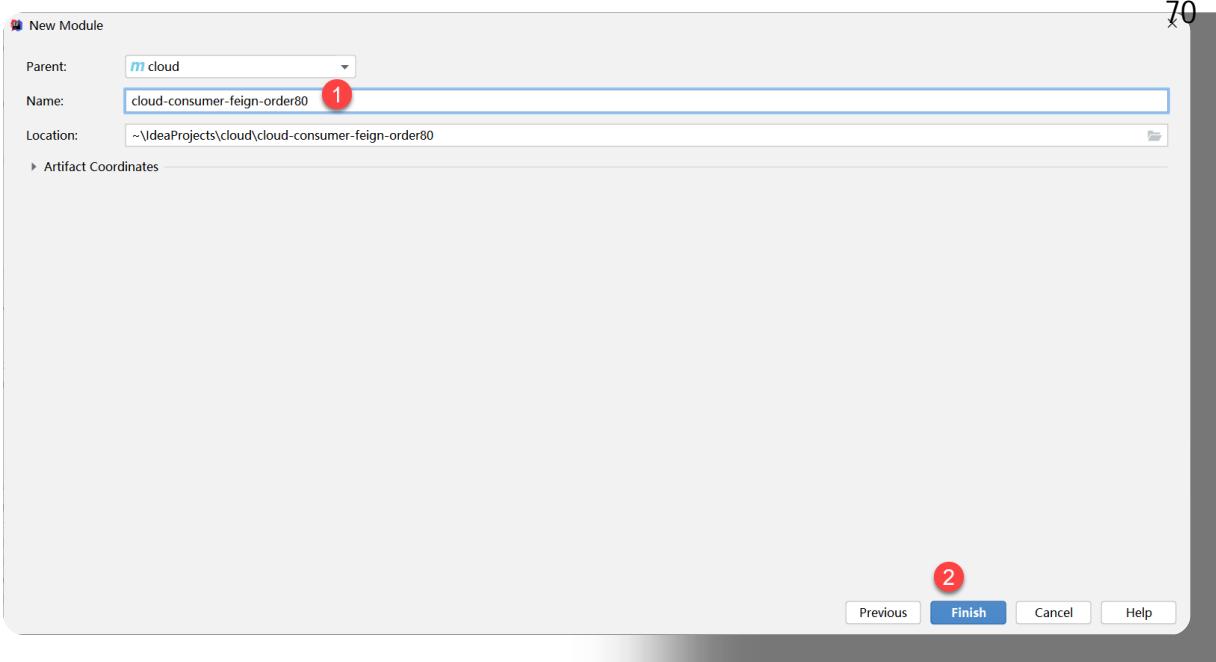
- A Spring MVC
- B Spring Data
- C Spring Batch
- D 以上都错误

答案

1=>A 2=>A

服务接口调用OpenFeign_入门案例

构建cloud-consumer-feign-order80工程



修改POM文件

```
1 <!-- 引入OpenFeign依赖 -->
2 <dependency>
3
4     <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-
6             openfeign</artifactId>
7     </dependency>
```

编写YML文件

```
1 eureka:
2     client:
3         # 表示是否将自己注册到Eureka Server
4         register-with-eureka: true
5         # 表示是否从Eureka Server获取注册的服务信息
6         fetch-registry: true
7         # Eureka Server地址
8         service-url:
9             defaultZone:
10                http://eureka7001.com:7001/eureka,http://eur
11                eka7002.com:7002/eureka
```

```

10 instance:
11     instance-id: cloud-openfeign-order-
12 consumer
13     prefer-ip-address: true
14 spring:
15     application:
16         # 设置应用名词
17         name: cloud-openfeign-order-consumer
18 server:
19     port: 80

```

编写主启动类

```

1 /**
2 * 主启动类
3 */
4 @Slf4j
5 @SpringBootApplication
6 #
7 @EnableFeignClients
8 public class OrderFeignMain80 {
9     public static void main(String[] args) {
10
11         SpringApplication.run(OrderFeignMain80.class,
12             args);
13         log.info("*****\nOrderFeignMain80 服务启动成功\n*****");
14     }
15 }

```

编写业务逻辑接口PaymentFeignService

```

1  /**
2   * 支付远程调用Feign接口
3   */
4 @Component
5 @FeignClient(value = "cloud-payment-
provider")
6 public interface PaymentFeignService {
7
8     @GetMapping("/payment/index")
9     String index();
10
11 }

```

编写控制层Controller

```

1 /**
2  * 订单控制层
3  */
4 @RestController
5 @RequestMapping("/order")
6 public class OrderController {
7
8     @Autowired
9     private PaymentFeignService
paymentFeignService;
10
11 /**
12  * 测试OpenFeign接口调用
13  * @return
14  */
15     @GetMapping("/index")

```

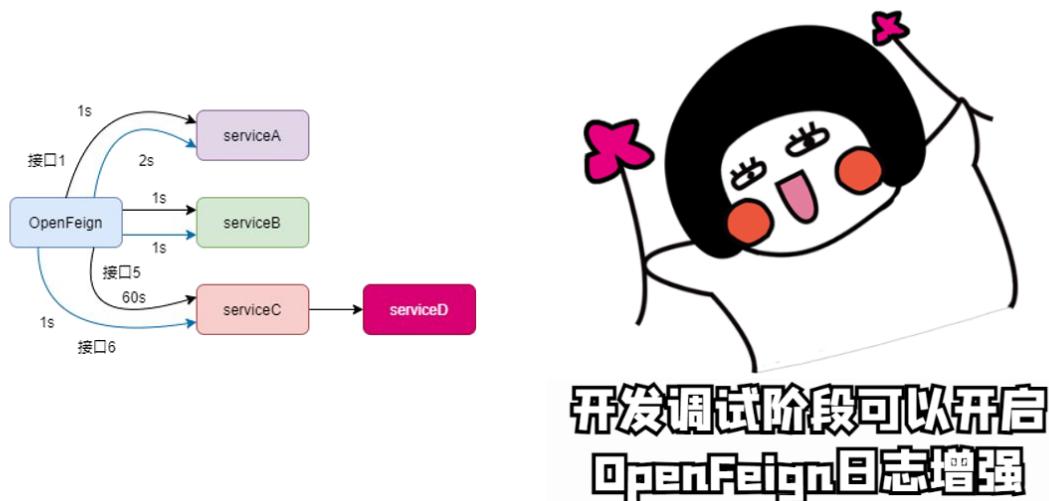
```

16     public String get(){
17         return paymentFeignService.index();
18     }
19 }
```

测试

- ① 先启动2个Eureka集群7001/7002
- ② 在启动服务提供者payment8001
- ③ 在启动服务消费者cloud-consumer-feign-order
- ④ 浏览器访问<http://localhost/order/index>

服务接口调用OpenFeign_日志增强



OpenFeign虽然提供了日志增强功能，但是默认是不显示任何日志的，不过开发者在调试阶段可以自己配置日志的级别。

OpenFeign的日志级别如下：

- **NONE:** 默认的，不显示任何日志；
- **BASIC:** 仅记录请求方法、URL、响应状态码及执行时间；
- **HEADERS:** 除了BASIC中定义的信息之外，还有请求和响应的头信息；
- **FULL:** 除了HEADERS中定义的信息之外，还有请求和响应的正文及元数据。

配置类中配置日志级别

```

1 @Configuration
2 public class OpenFeignConfig{
3
4     /**
5      * 日志级别定义
6      */
7     @Bean
8     Logger.Level feignLoggerLevel(){
9         return Logger.Level.FULL;
10    }
11 }
```

注意：

这里的logger是feign包里的。

yaml文件中设置接口日志级别

```

1 logging:
2   level:
3     com.itbaizhan.service: debug
```

注意：

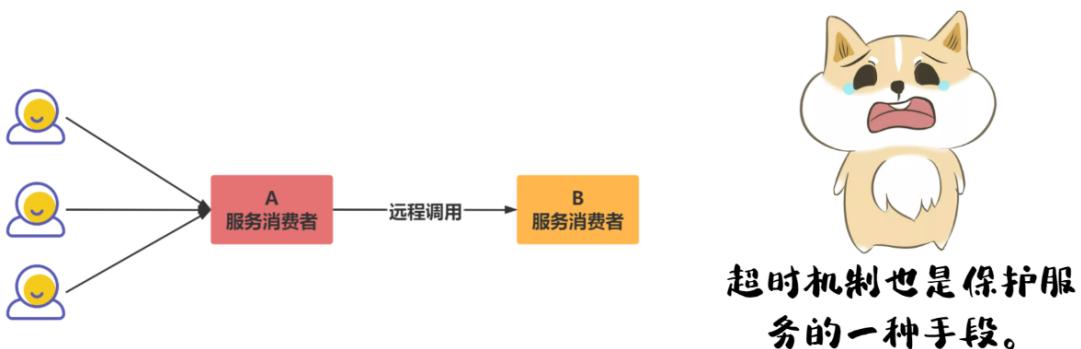
这里的 `com.itbaizhan.service` 是openFeign接口所在的包名，当然你也可以配置一个特定的openFeign接口。

测试

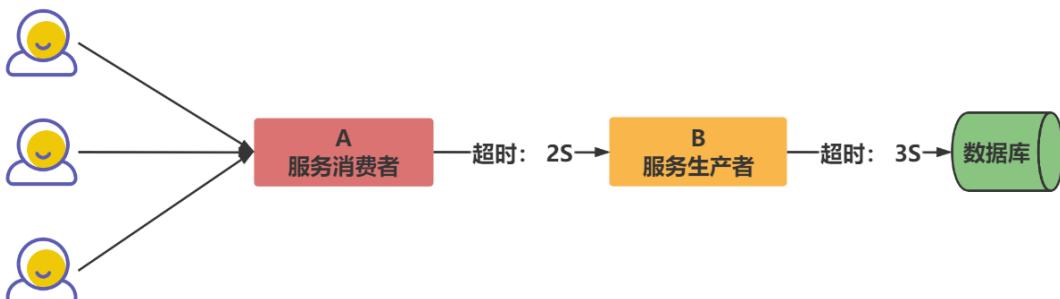
请求<http://localhost/order/index>

```
[PaymentFeignService#index] ---> GET http://CLOUD-PAYMENT-PROVIDER/payment/index HTTP/1.1
[PaymentFeignService#index] ---> END HTTP (0-byte body)
[PaymentFeignService#index] <--- HTTP/1.1 200 (85ms)
[PaymentFeignService#index] connection: keep-alive
[PaymentFeignService#index] content-length: 15
[PaymentFeignService#index] content-type: text/plain; charset=UTF-8 <1 internal call>
[PaymentFeignService#index] keep-alive: timeout=60
[PaymentFeignService#index]
[PaymentFeignService#index] payment success
[PaymentFeignService#index] <--- END HTTP (15-byte body)
```

服务接口调用OpenFeign_超时机制



超时机制



问题:

- 服务消费者在调用服务提供者的时候发生了阻塞、等待的情形，这个时候，服务消费者会一直等待下去。
- 在某个峰值时刻，大量的请求都在同时请求服务消费者，会造成线程的大量堆积，势必会造成雪崩。
- 利用超时机制来解决这个问题，设置一个超时时间，在这个时间段内，无法完成服务访问，则自动断开连接。

配置超时时间

```
1 # 默认超时时间
feign:
  client:
    config:
      default:          # 连接超时时间
      connectTimeout: 2000          # 读取超时时间
      readTimeout: 2000
```

服务提供方8001故意写超时程序

```
1 /**
 * 测试超时机制
 */
@GetMapping("timeout")
public String paymentFeignTimeOut(){
  TimeUnit.SECONDS.sleep(5);
  try {
    (InterruptedException e) {
      e.printStackTrace();
    }
  } catch
  "payment success"; }
```

服务消费方80添加超时方法PaymentFeignService

```
1 @FeignClient("CLOUD-PAYMENT-PROVIDER") public
interface PaymentFeignService {
  @GetMapping("/payment/index")   String
index();
  @GetMapping("/payment/timeout") String
timeout();}
```

服务消费方80添加超时方法OrderController

```
1 /**
 * 测试超时机制
 */
@GetMapping("timeout")
public String timeout(){
  return
paymentFeignService.timeout(); }
```

实时效果反馈

1. OpenFeign技术中给服务设置超时时间解决__问题。

- A 安全性
- B 速度
- C 响应慢
- D 服务雪崩

2. OpenFeign技术中如何设置服务的读取超时时间__。

- A `feign.client.config.default.readTimeout`
- B `feign.client.config.connectTimeout`
- C `feign.client.config.default.read`
- D `feign.client.config.default.Timeout`

答案

1=>D 2=>A

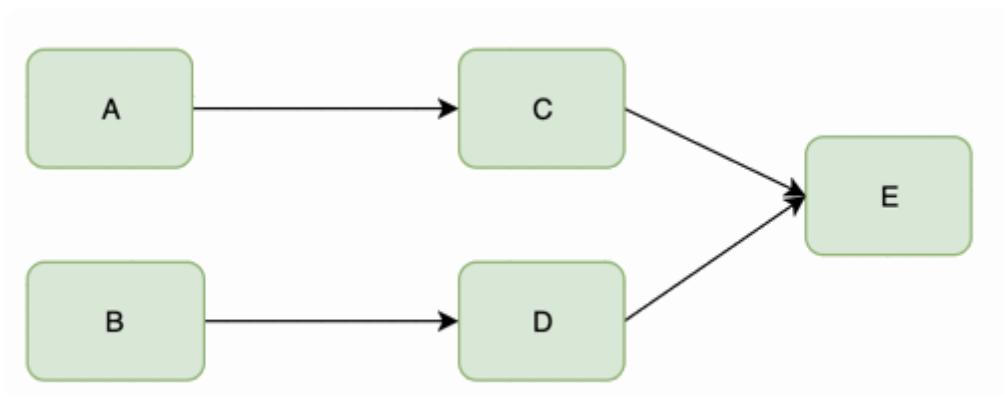
服务断路器_什么是灾难性雪崩效应



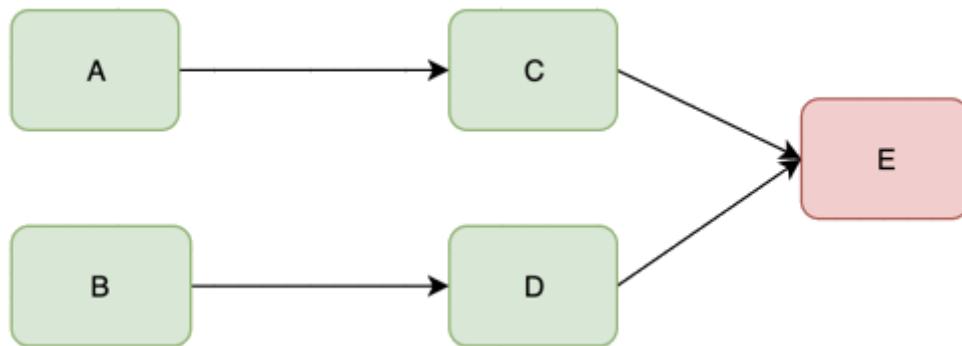
服务雪崩效应太可怕了

什么是灾难性雪崩效应

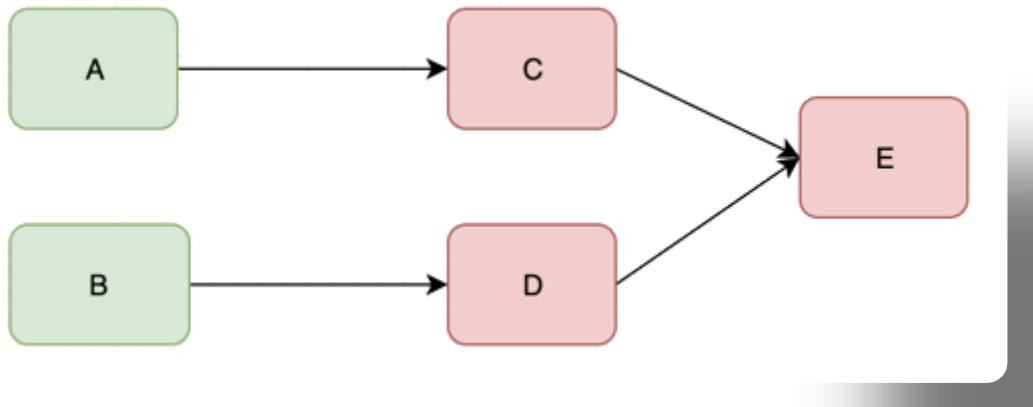
假设我们有两个访问量比较大的服务A和B，这两个服务分别依赖C和D，C和D服务都依赖E服务。



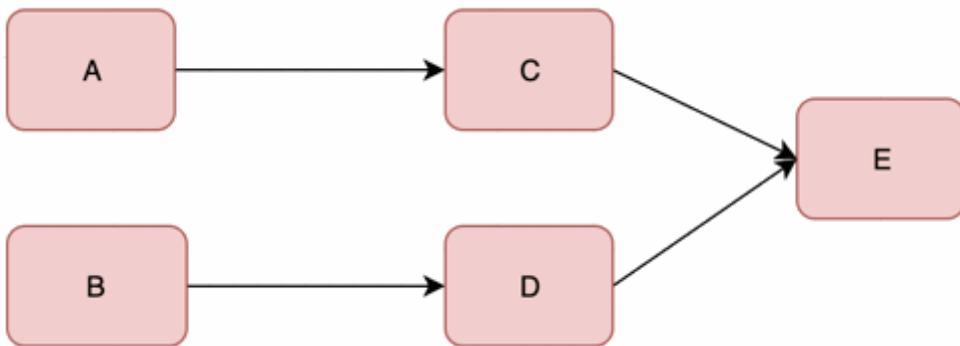
A和B不断的调用C,D处理客户请求和返回需要的数据。当E服务不能供服务的时候，C和D的超时和重试机制会被执行



由于新的调用不断的产生，会导致C和D对E服务的调用大量的积压，产生大量的调用等待和重试调用，慢慢会耗尽C和D的资源比如内存或CPU，然后也down掉。



A和B服务会重复C和D的操作，资源耗尽，然后down掉，最终整个服务都不可访问。



结论：

服务与服务之间的依赖性，故障会传播，造成连锁反应，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。



造成雪崩原因是什么

- ① 服务提供者不可用（硬件故障、程序bug、缓存击穿、用户大量请求）
- ② 重试加大流量（用户重试，代码逻辑重试）
- ③ 服务调用者不可用（同步等待造成的资源耗尽）

注意：

在高并发访问下，系统所依赖的服务的稳定性对系统的影响非常大，依赖有很多不可控的因素，比如网络连接变慢，资源突然繁忙，暂时不可用，服务脱机等。我们要构建稳定、可靠的分布式系统，就必须要有一套容错方法。

实时效果反馈

1.造成灾难性服务雪崩的原因_____。

- A 服务提供者不可用
- B 服务调用者不可用
- C 重试加大流量
- D 以上都正确

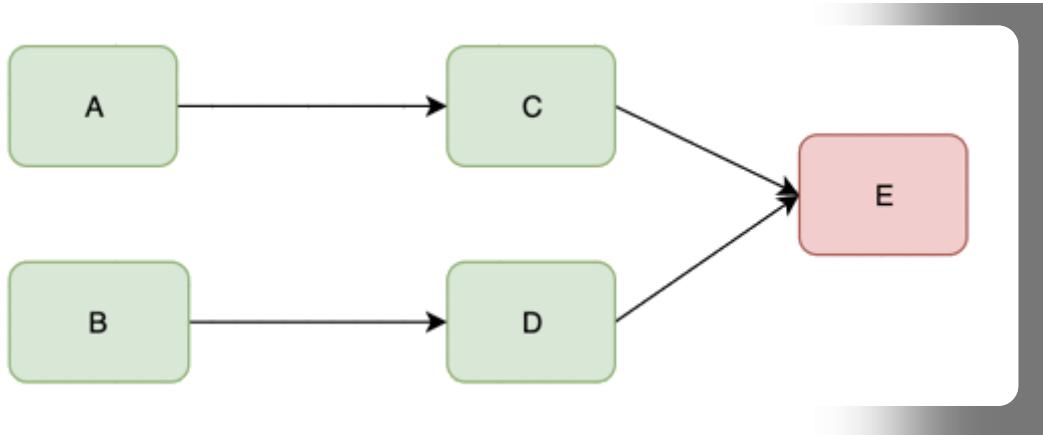
答案

1=>A

服务断路器_服务雪崩解决方案之服务熔断

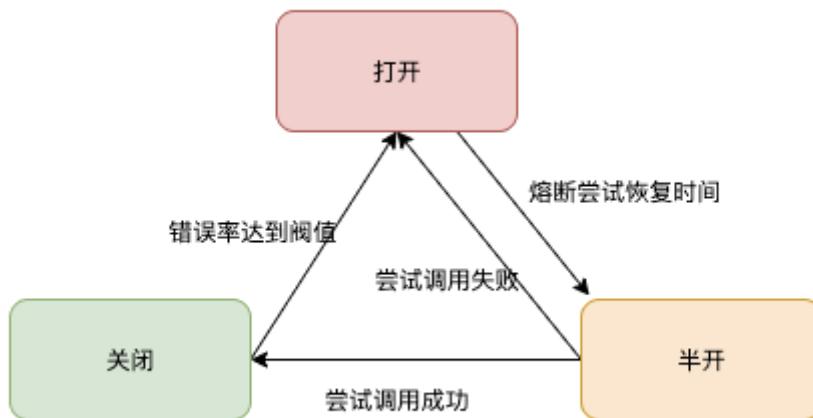


保险丝：电路中正确安置保险丝，保险丝就会在电流异常升高到一定的高度和热度的时候，自身熔断切断电流，保护了电路安全运行。



什么是熔断

熔断就跟保险丝一样，当一个服务请求并发特别大，服务器已经招架不住了，调用错误率飙升，当错误率达到一定阈值后，就将这个服务熔断了。熔断之后，后续的请求就不会再请求服务器了，以减缓服务器的压力。



注意：

当失败率（如因网络故障/超时造成的失败率高）达到阀值自动触发降级，熔断器触发的快速失败会进行快速恢复。

实时效果反馈

1. 服务熔断就像生活中的_____。

- A 锤子
- B 钳子
- C 保险丝
- D 绳子

2. 服务熔断机制中当_____达到一定阈值，就将这个服务熔断。

- A 错误率
- B 调用次数
- C 执行次数
- D 以上都错误

答案

1=>C 2=>C

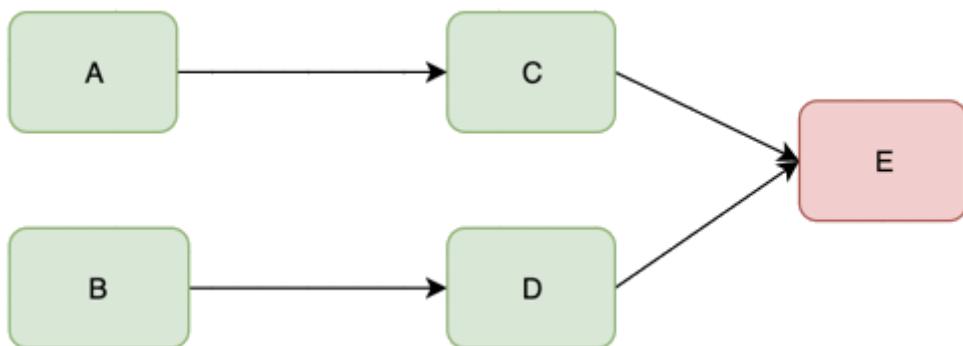
服务断路器_服务雪崩解决方案之服务降级



不让客户端等待并立即返回一个友好的提示



什么是服务降级



两种场景:

- 当下游的服务因为某种原因**响应过慢**, 下游服务主动停掉一些不太重要的业务, 释放出服务器资源, 增加响应速度!
- 当下游的服务因为某种原因**不可用**, 上游主动调用本地的一些降级逻辑, 避免卡顿, 迅速返回给用户!

服务降级 fallback

概念: 服务器繁忙, 请稍后重试, 不让客户端等待并立即返回一个友好的提示。

出现服务降级的情况:

- 程序运行异常
- 超时
- 服务熔断触发服务降级
- 线程池/信号量打满也会导致服务降级

实时效果反馈

1. 服务降级主要解决是__问题。

- A 服务重试
- B 单点故障
- C 服务调用
- D 服务雪崩

2. 下列不属于服务降级情况的是__。

- A 超时
- B 服务运行异常
- C 触发了服务熔断
- D 重试机制

答案

1=>D 2=>D

服务断路器_服务雪崩解决方案之服务隔离



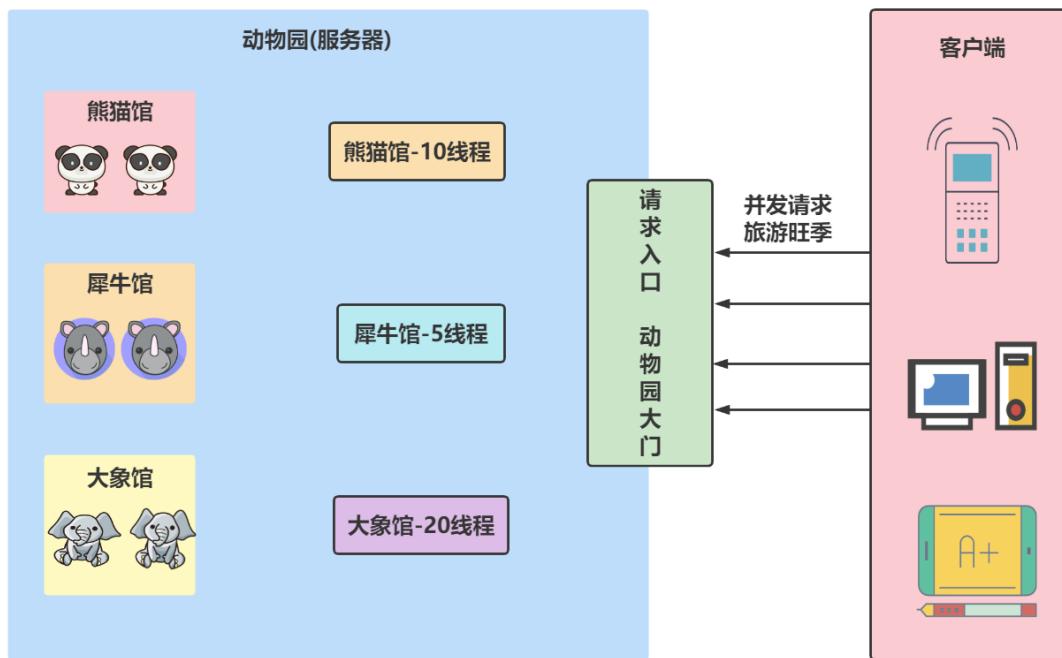
那显而易见，做服务隔离的目的就是避免服务之间相互影响。毕竟谁也不能说自己的微服务百分百可用，如果不做隔离，一旦一个服务出现了问题，整个系统的稳定性都会受到影响！因此，做服务隔离是很有必要的。

什么是线程池隔离

将用户请求线程和服务执行线程分割开来，同时约定了每个服务最多可用线程数。



使用线程池隔离后

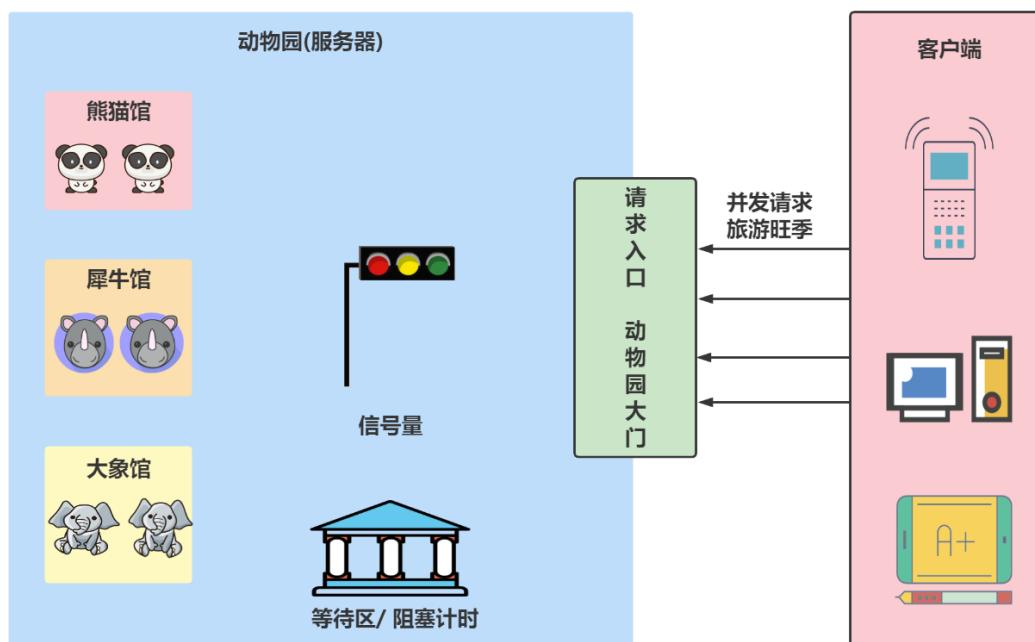


解释:

动物园有了新规矩-线程隔离，就是说每个服务单独设置一个小房间（独立线程池），把大厅区域和服务区域隔离开来，每个服务房间也有接待数量限制，比如我设置了熊猫馆最多接纳10人，犀牛管最多5人，大象馆20人。这样，即便来了20个人想你熊猫，我们也只能接待10人，剩下的10个人就会收到Thread Pool Rejects。如此一来，也不会耽搁动物园为用户提供其他服务。

什么是信号量隔离

小时候我们就知道“红灯停，绿灯行”，跟着交通信号的指示过马路。信号量也是这么一种放行、禁行的开关作用。它和线程池技术一样，控制了服务可以被同时访问的并发数量。



线程池隔离和信号量隔离区别

隔离方式	是否支持超时	是否支持熔断	隔离原理	是否是异步调用	资源消耗
线程池隔离	支持，可直接返回	支持，当线程池到达maxSize后，再请求会触发fallback接口进行熔断	每个服务单独用线程池	可以是异步，也可以是同步。看调用的方法	大，大量线程的上下文切换，容易造成机器负载高
信号量隔离	不支持，如果阻塞，只能通过调用协议（如：socket超时才能返回）	支持，当信号量达到maxConcurrentRequests后。再请求会触发fallback	通过信号量的计数器	同步调用，不支持异步	小，只是个计数器

实时效果反馈

1. 服务隔离主要目的是_____。

- A 解决负载均衡问题
- B 解决单点故障
- C 解决服务调用
- D 避免服务之间相互影响

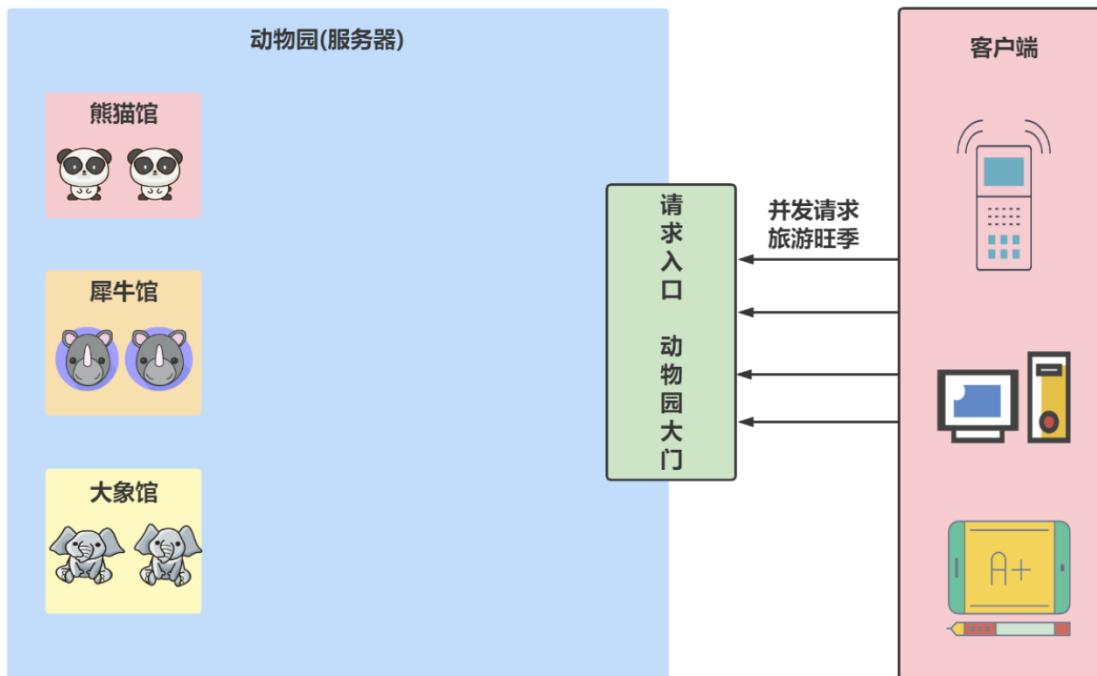
答案

1=>D

服务断路器_服务雪崩解决方案之服务限流



服务熔断和服务隔离都属于出错后的容错处理机制，而限流模式则可以称为预防模式。



限流模式主要是提前对各个类型的请求设置最高的QPS阈值，若高于设置的阈值则对该请求直接返回，不再调用后续资源。

注意：

限流的目的是通过对并发访问/请求进行限速，或者对一个时间窗口内的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务、排队或等待、降级等处理。

流量控制

- 网关限流：防止大量请求进入系统，Mq实现流量消峰
- 用户交流限流：提交按钮限制点击频率限制等

实时效果反馈

1.限流模式则可以称为_____。

- A 事前模式
- B 预防模式
- C 事后模式
- D 以上都是错误

答案

1=>B

服务断路器_Resilience4j介绍



Resilience4j是受到
Netflix Hystrix的
启发



什么是Hystrix

我们耳熟能详的就是Netflix Hystrix,这个断路器是SpringCloud中最早支持的一种容错方案，现在这个断路器已经处于维护状态，已经不再更新了，你仍然可以使用这个断路器，但是呢，我不建议你去使用，因为这个已经不再更新，所以Spring官方已经出现了Netflix Hystrix的替换方案。

CURRENT	REPLACEMENT
Hystrix	Resilience4j
Hystrix Dashboard / Turbine	Micrometer + Monitoring System
Ribbon	Spring Cloud Loadbalancer
Zuul 1	Spring Cloud Gateway
Archaius 1	Spring Boot external config + Spring Cloud Config

什么是Resilience4j

Resilience4j是一个轻量级的容错组件，其灵感来自于Hystrix，但主要为Java 8和函数式编程所设计,也就是我们的lambda表达式。轻量级体现在其只用 [Vavr](#) 库（前身是 Javaslang），没有任何外部依赖。而Hystrix依赖了Archaius，Archaius本身又依赖很多第三方包，例如 Guava、Apache Commons Configuration 等。

Resilience4j官网

<https://resilience4j.readme.io/>

Resilience4J 提供了一系列增强微服务的可用性功能：

- resilience4j-circuitbreaker: 熔断
- resilience4j-ratelimiter: 限流
- resilience4j-bulkhead: 隔离
- resilience4j-retry: 自动重试
- resilience4j-cache: 结果缓存
- resilience4j-timelimiter: 超时处理

注意：

在使用Resilience4j的过程中，不需要引入所有的依赖，只引入需要的依赖即可。

实时效果反馈

1.Resilience4j是一个轻量级的容错组件，其灵感来自于__。

- A Eureka
- B Ribbon
- C Hystrix
- D OpenFeign

答案

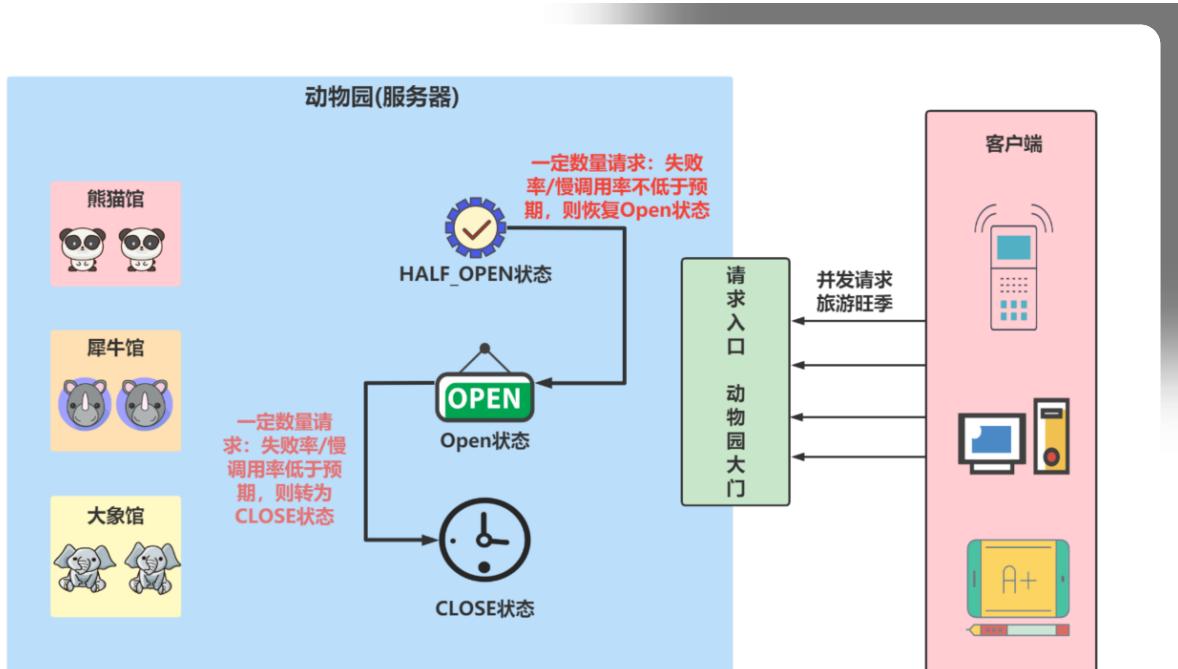
1=>C

服务断路器_Resilience4j的断路器



我只说一句话
家里保险丝

断路器 (CircuitBreaker) 相对于前面几个熔断机制更复杂, CircuitBreaker通常存在三种状态 (CLOSE、OPEN、HALF_OPEN) , 并通过一个时间或数量窗口来记录当前的请求成功率或慢速率, 从而根据这些指标来作出正确的容错响应。



6种状态:

- CLOSED: 关闭状态, 代表正常情况下的状态, 允许所有请求通过, 能通过状态转换为OPEN
- HALF_OPEN: 半开状态, 即允许一部分请求通过, 能通过状态转换为CLOSED和OPEN
- OPEN: 熔断状态, 即不允许请求通过, 能通过状态转为为HALF_OPEN
- DISABLED: 禁用状态, 即允许所有请求通过, 出现失败率达到给定的阈值也不会熔断, 不会发生状态转换。
- METRICS_ONLY: 和DISABLED状态一样, 也允许所有请求通过不会发生熔断, 但是会记录失败率等信息, 不会发生状态转换。
- FORCED_OPEN: 与DISABLED状态正好相反, 启用CircuitBreaker, 但是不允许任何请求通过, 不会发生状态转换。

主要介绍3种状态

- closed -> open : 关闭状态到熔断状态, 当失败的调用率 (比如超时、异常等) 默认50%, 达到一定的阈值服务转为open状态, 在open状态下, 所有的请求都被拦截。
- open-> half_open: 当经过一定的时间后, CircuitBreaker中默认为60s服务调用者允许一定的请求到达服务提供者。
- half_open -> open: 当half_open状态的调用失败率超过给定的阈值, 转为open状态
- half_open -> closed: 失败率低于给定的阈值则默认转换为closed状态

实时效果反馈

1. Resilience4j的断路器共有____状态。

- A 3
- B 4
- C 5
- D 6

2. 断路器最开始状态是____。

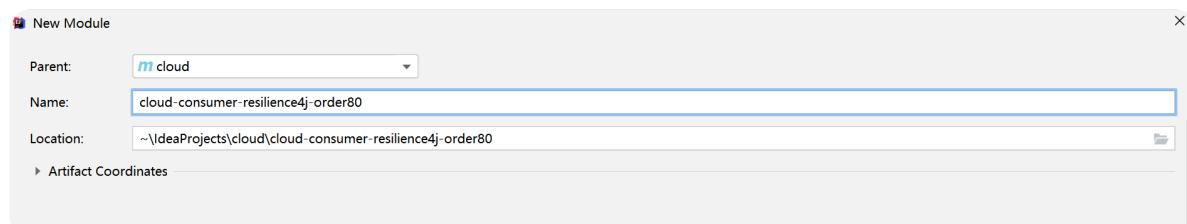
- A 关闭
- B 打开
- C 半开
- D 以上都错误

答案

1=>D 2=>A

服务断路器_Resilience4j超时降级

创建模块cloud-consumer-resilience4j-order80



POM引入依赖

```
1 <dependencies>
2     <!-- 引入Eureka 客户端依赖 -->
3     <dependency>
4
5         <groupId>org.springframework.cloud</groupId>
6     >
7         <artifactId>spring-cloud-
8             starter-netflix-eureka-client</artifactId>
9         </dependency>
10    <!-- 引入服务调用依赖 OpenFeign -->
11    <dependency>
12
13        <groupId>org.springframework.cloud</groupId>
14    >
15        <artifactId>spring-cloud-
16            starter-openfeign</artifactId>
17        </dependency>
18        <dependency>
19
20            <groupId>org.springframework.boot</groupId>
21            <artifactId>spring-boot-starter-
22                web</artifactId>
23            </dependency>
24            <dependency>
25
26                <groupId>org.projectlombok</groupId>
27                <artifactId>lombok</artifactId>
28                <version>1.18.22</version>
29                </dependency>
30            <!-- actuator监控信息完善 -->
```

```

22 <dependency>
23
24     <groupId>org.springframework.boot</groupId>
25         <artifactId>spring-boot-starter-
26             actuator</artifactId>
27             </dependency>
28     <dependency>
29
30         <groupId>io.github.resilience4j</groupId>
31             <artifactId>resilience4j-spring-
32                 cloud2</artifactId>
33             </dependency>
34     <dependency>
35
36         <groupId>org.springframework.cloud</groupId>
37             <artifactId>spring-cloud-starter-
38                 circuitbreaker-resilience4j</artifactId>
39             </dependency>
40     </dependencies>

```

修改YML文件

```

1 # 超时机制
2 resilience4j:
3     timelimiter:
4         instances:
5             delay:
6                 # 设置超时时间 5秒
7                 timeoutDuration: 2

```

编写服务提供者提供超时方法

```

1  /**
2   * 超时服务
3   * @return
4   */
5 @GetMapping("timeout")
6 public String timeout(){
7     try {
8         TimeUnit.SECONDS.sleep(5);
9     } catch (InterruptedException e) {
10        e.printStackTrace();
11    }
12    return "success";
13 }
```

编写服务提供者Controller

```

1 @GetMapping("/timeout")
2 @TimeLimiter(name = "delay", fallbackMethod =
3 "timeoutfallback")
4     public CompletableFuture<String>
5 timeout() {
6     log.info("***** 进入方法 *****");
7     //异步操作
8     CompletableFuture<String>
9     completableFuture = CompletableFuture
10
11     .supplyAsync((Supplier<String>) () ->
12     (paymentFeignService.timeout()));
13     log.info("***** 离开方法 *****");
14     return completableFuture;
15 }
```

编写服务降级方法

```

1  /**
2   * 超时服务降级方法
3   * @param e
4   * @return
5  */
6  public CompletableFuture<ResponseEntity>
7  timeoutfallback(Exception e){
8      e.printStackTrace();
9      return
CompletableFuture.completedFuture(ResponseEnt
ity.ok("超时啦"));
}

```

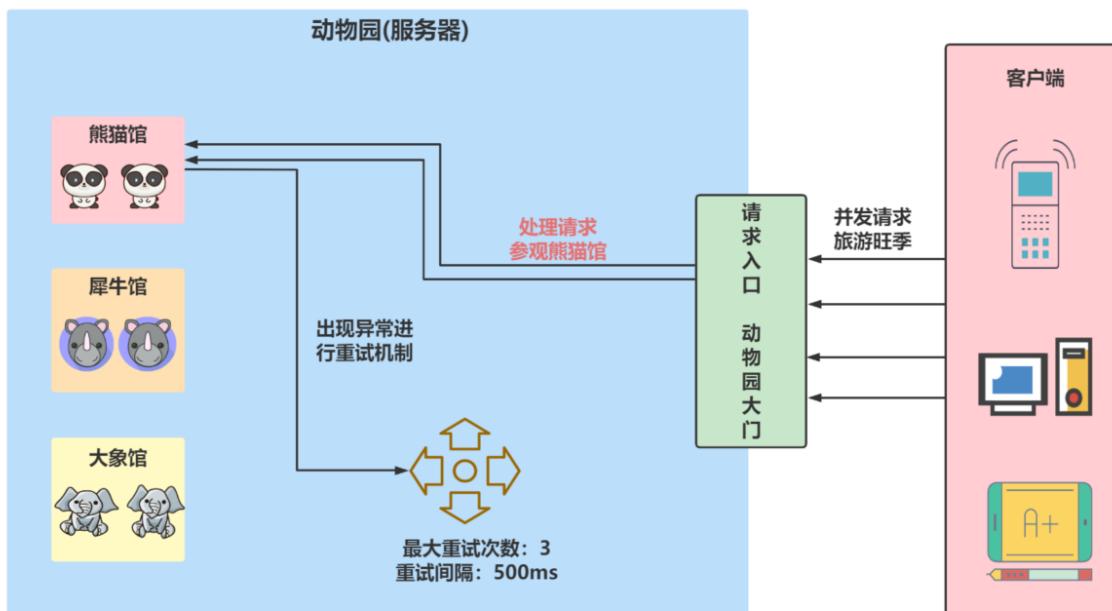
测试

```

java.util.concurrent.TimeoutException Create breakpoint : TimeLimiter 'delay' recorded a timeout exception.
at io.github.resilience4j.timelimiter.TimeLimiter.createTimeoutExceptionWithName(TimeLimiter.java:221)
at io.github.resilience4j.timelimiter.internal.TimeLimiterImpl$Timeout.lambda$of$0(TimeLimiterImpl.java
at java.lang.Thread.run(Thread.java:748)

```

服务断路器_Resilience4j重试机制



重试机制比较简单，当服务端处理客户端请求异常时，服务端将会开启重试机制，重试期间内，服务端将每隔一段时间重试业务逻辑处理。如果最大重试次数内成功处理业务，则停止重试，视为处理成功。如果在最大重试次数内处理业务逻辑依然异常，则此时系统将拒绝该请求。

修改YML文件

```

1 resilience4j:
2   retry:
3     instances:
4       backendA:
5         # 最大重试次数
6         maxRetryAttempts: 3
7         # 固定的重试间隔
8         waitDuration: 10s
9         enableExponentialBackoff: true
10        exponentialBackoffMultiplier: 2

```

服务提供者新增controller方法

```

1 /**
2 * 重试机制
3 * @return
4 */
5 @GetMapping("/retry")
6 @Retry(name = "backendA")
7 public CompletableFuture<String> retry() {
8     log.info("***** 进入方法 *****");
9     //异步操作
10    CompletableFuture<String>
completableFuture = CompletableFuture

```

```

11     .supplyAsync((Supplier<String>) () ->
12         paymentFeignService.index()));
13         log.info("***** 离开方法 *****");
14         return completableFuture;
15     }

```

服务断路器_Resilience4j异常比例熔断降级

给coud-consumer-feign-order80添加resilience4j依赖

修改yml文件

```

1 resilience4j.circuitbreaker:
2     configs:
3         default:
4             # 熔断器打开的失败阈值
5             failureRateThreshold: 30
6             # 默认滑动窗口大小, circuitbreaker使用基于
7             # 计数和时间范围滑动窗口聚合统计失败率
8             slidingWindowSize: 10
9             # 计算比率的最小值, 和滑动窗口大小去最小值, 即
10            # 当请求发生5次才会计算失败率
11            minimumNumberOfCalls: 5
12            # 滑动窗口类型, 默认为基于计数的滑动窗口
13            slidingWindowType: TIME_BASED
14            # 半开状态允许的请求数
15            permittedNumberOfCallsInHalfOpenState:
16                3
17            # 是否自动从打开到半开

```

```

15     automaticTransitionFromOpenToHalfOpenEnabled
16         : true
17             # 熔断器从打开到半开需要的时间
18             waitDurationInOpenState: 2s
19             recordExceptions:
20                 - java.lang.Exception
21             instances:
22                 backendA:
23                     baseConfig: default

```

编写OrderController

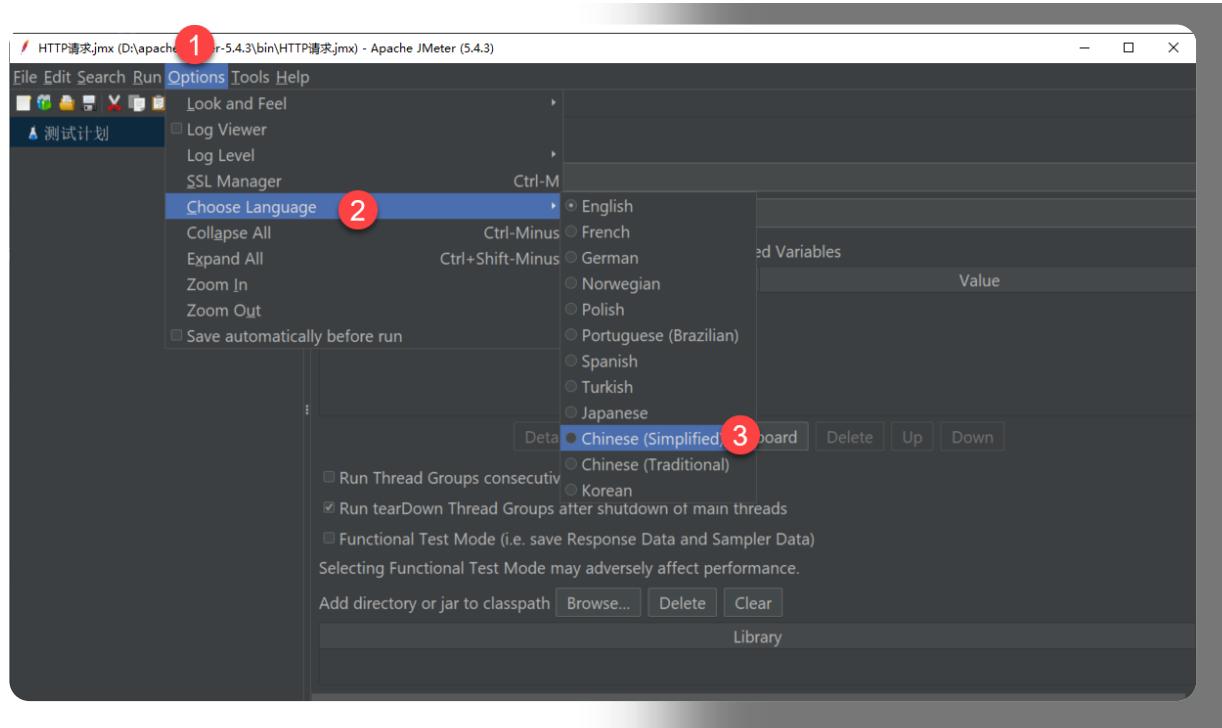
```

1 /**
2 * 异常比例熔断降级
3 * @return
4 */
5 @GetMapping("/circuitBackend")
6 @CircuitBreaker(name = "backendA")
7 public String circuitBackend(){
8
9     log.info("***** 进入方法
*****");
10    String index =
11    paymentFeignService.index();
12    log.info("***** 离开方法
*****");
13
14    return index;
}

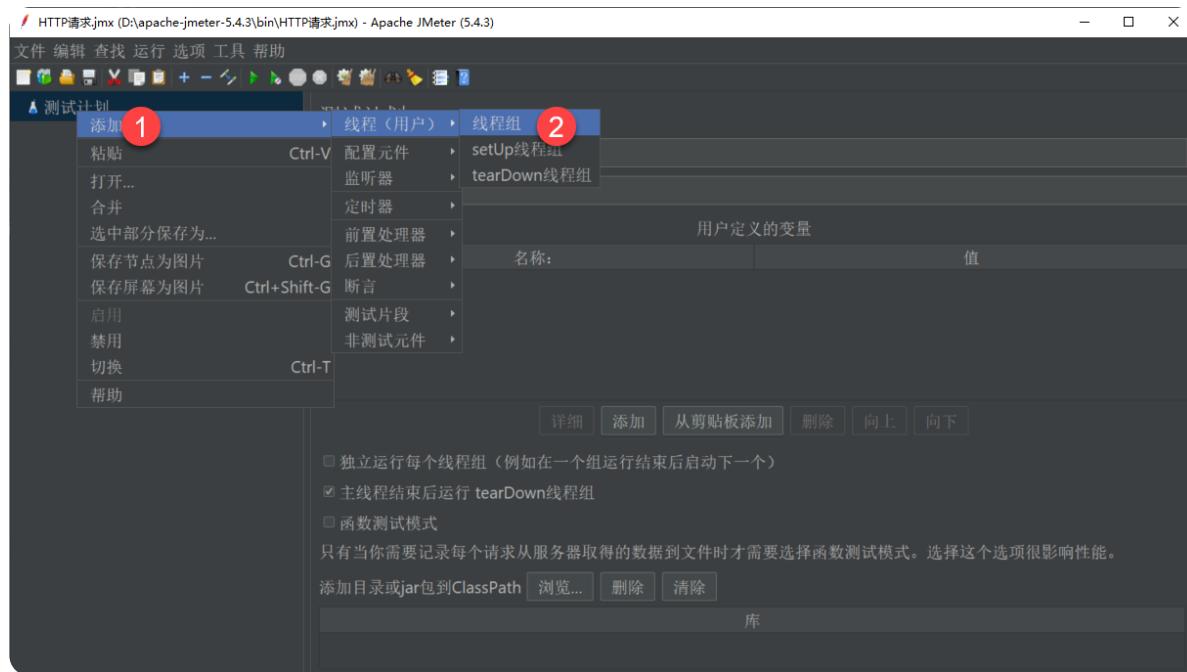
```

使用JMeter进行压力测试

修改语言



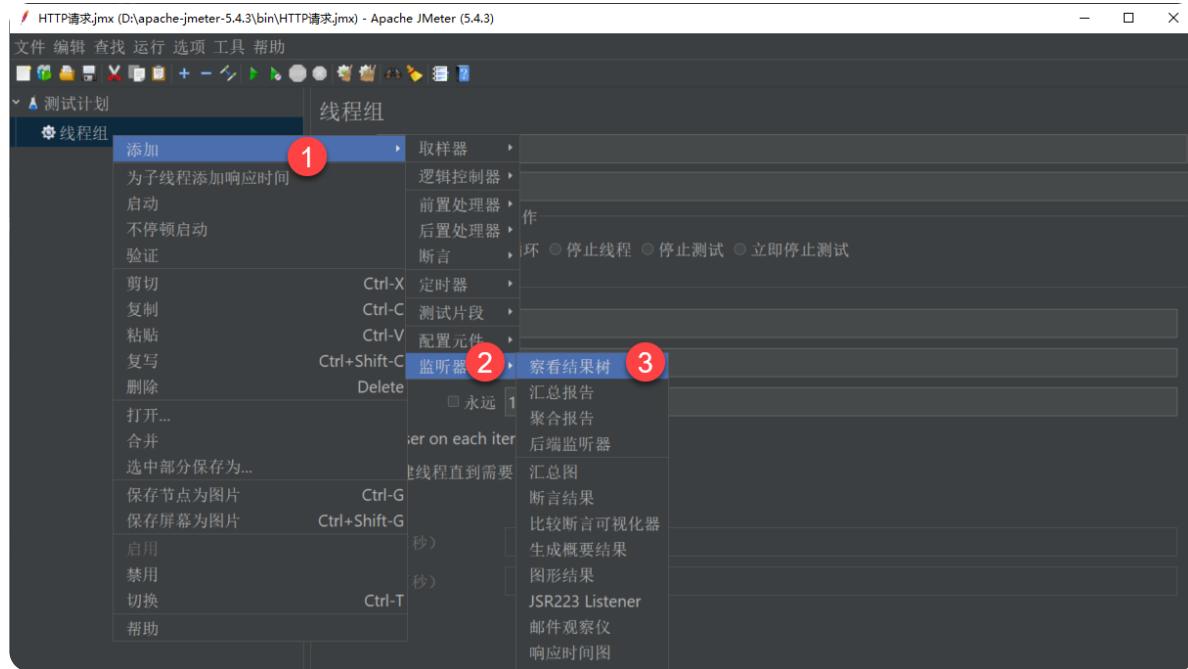
创建线程组



创建取样器HTTP请求



添加查看结果树



修改线程数量

线程组

名称: 线程组
注释:

在取样器错误后要执行的动作

继续 启动下一进程循环 停止线程 停止测试 立即停止测试

线程属性

线程数: 10 (1)

Ramp-Up时间 (秒): 1

循环次数 永远 1

Same user on each iteration
 延迟创建线程直到需要
 调度器

持续时间 (秒):

启动延迟 (秒):

修改HTTP请求参数

HTTP请求

名称: HTTP请求
注释:

基本 高级

Web服务器

协议: http 服务器名称或IP: localhost 端口号: 80

HTTP请求

GET 路径: /order/index **路由地址** 内容编码:

自动重定向 跟随重定向 使用 KeepAlive 对POST使用multipart / form-data 与浏览器兼容的头

参数 消息体数据 文件上传

同请求一起发送参数:

名称:	值	编码?	内容类型	包含等于?
-----	---	-----	------	-------

半开状态只有三次请求

```

OrderController : **** 进入方法 ****
OrderController : **** 进入方法 ****
OrderController : **** 进入方法 ****
[atcherServlet] : Servlet.service() for servlet [dispatcherServlet] in context with path |

CircuitBreaker 'backendA' is HALF_OPEN and does not permit further calls
NotPermittedException(CallNotPermittedException.java:48) ~[resilience4j-circuitbreaker-1.7.0]
HalfOpenState.acquirePermission(CircuitBreakerStateMachine.java:1031) ~[resilience4j-circuitbreaker-1.7.0]
acquirePermission(CircuitBreakerStateMachine.java:206) ~[resilience4j-circuitbreaker-1.7.0]

```

编写降级方法

```
1  /**
2  * 异常比例熔断降级
3  * @return
4  */
5  @GetMapping("/circuitBackend")
6  @CircuitBreaker(name = "backendA")
7  public String circuitTest(){
8
9      log.info("***** 进入方法
*****");
10     String index =
11         paymentFeignService.index();
12     log.info("***** 离开方法
*****");
13     return index;
14 }
15 /**
16 * 服务降级方法
17 * @param e
18 * @return
19 */
20 public String fallback(Throwable e){
21     e.printStackTrace();
22     return "客官服务繁忙，稍等一
会。。。。";
23 }
```

```

    @GetMapping("/index")
    @CircuitBreaker(name = "backendA", fallbackMethod = "fallback")
    public String get(){

        Log.info("***** 进入方法 *****")
        String index = paymentFeignService.index();
        Log.info("***** 离开方法 *****");

        return index;
    }

    /**
     * 服务降级方法
     * @param e
     * @return
     */
    public String fallback(Throwable e){
        e.printStackTrace();
        return "客官服务繁忙，稍等一会。。。。";
    }

```

方法名字得一致

测试降级方法

- ① 关闭服务提供者
- ② 服务消费者发起请求

产生服务降级

察看结果树

名称: 察看结果树

注释:

所有数据写入一个文件

文件名:

查找: 区分大小写 正则表达式

Text

取样器结果 请求 响应数据

Response Body Response headers

客官服务繁忙，稍等一会。。。。

服务断路器_Resilience4j慢调用比例熔断降级

编写OrderController

```

1  /**
2   * 慢调用比例熔断降级
3   * @return
4   */
5  @GetMapping("/slowcircuitbackend")
6  @CircuitBreaker(name =
7  "backendB", fallbackMethod = "slowfallback")
8  public String slowcircuitbackend(){
9
10    log.info("***** 进入方法
11    *****");
12    try {
13      TimeUnit.SECONDS.sleep(10);
14    } catch (InterruptedException e) {
15      e.printStackTrace();
16    }
17    String index =
18    paymentFeignService.index();
19    log.info("***** 离开方法
20    *****");
21
22    return index;
23  }

```

编写yml文件

```

1  resilience4j.circuitbreaker:
2    configs:
3      default:
4        # 熔断器打开的失败阈值
5        failureRateThreshold: 30

```

```

6      # 默认滑动窗口大小, circuitbreaker使用基于
7      # 计数和时间范围滑动窗口聚合统计失败率
8          slidingWindowSize: 10
9          # 计算比率的最小值, 和滑动窗口大小去最小值, 即
10         当请求发生5次才会计算失败率
11         minimumNumberOfCalls: 5
12         # 滑动窗口类型, 默认为基于计数的滑动窗口
13         slidingwindowType: TIME_BASED
14         # 半开状态允许的请求数
15         permittedNumberOfCallsInHalfOpenState:
16             3
17             # 是否自动从打开到半开
18
19         automaticTransitionFromOpenToHalfOpenEnabled
20         : true
21             # 熔断器从打开到半开需要的时间
22             waitDurationInOpenState: 2s
23             recordExceptions:
24                 - java.lang.Exception
25             instances:
26                 backendA:
27                     baseConfig: default
28                 backendB:
29                     # 熔断器打开的失败阈值
30                     failureRateThreshold: 50
31                     # 慢调用时间阈值 高于这个阈值的
32                     slowCallDurationThreshold: 2s
33                     # 慢调用百分比阈值, 断路器吧调用事件大于slow
34                     slowCallRateThreshold: 30
35                     slidingWindowSize: 10
36                     slidingwindowType: TIME_BASED
37                     minimumNumberOfCalls: 2

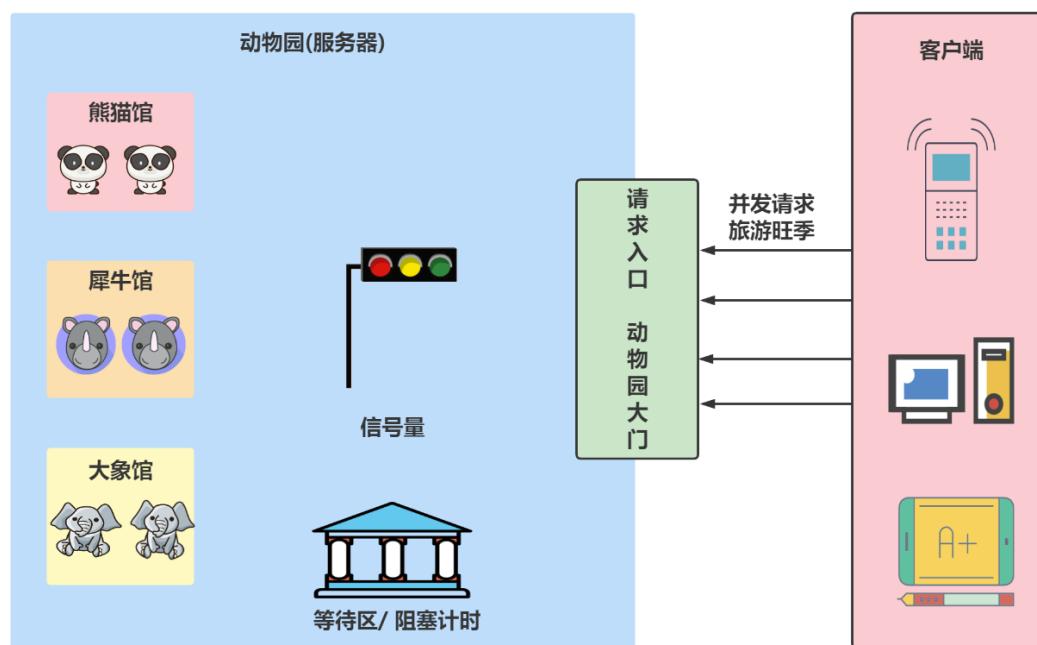
```

```

33     permittedNumberOfCallsInHalfOpenState:
34         2
35     waitDurationInOpenState: 2s
36     eventConsumerBufferSize: 10

```

服务断路器_Resilience4j信号量隔离实现



POM引入依赖

```

1 <dependency>
2
3     <groupId>io.github.resilience4j</groupId>
4         <artifactId>resilience4j-
5             bulkhead</artifactId>
6         <version>1.7.0</version>
7     </dependency>

```

信号量隔离修改YML文件

```

1 resilience4j:
2   #信号量隔离
3   bulkhead:
4     instances:
5       backendA:
6         # 隔离允许并发线程执行的最大数量
7         maxConcurrentCalls: 5
8         # 当达到并发调用数量时，新的线程的阻塞时间
9         maxwaitDuration: 20ms
10

```

编写controller

```

1 /**
2  * 测试信号量隔离
3  * @return
4  */
5 @Bulkhead(name = "backendA", type =
6 Bulkhead.Type.SEMAPHORE)
7 @GetMapping("bulkhead")
8 public String bulkhead() throws
9 InterruptedException {
10     log.info("***** 进入方法
11 *****");
12     TimeUnit.SECONDS.sleep(10);
13     String index =
14     paymentOpenFeignService.index();
15     log.info("***** 离开方法
16 *****");
17     return index;
18 }

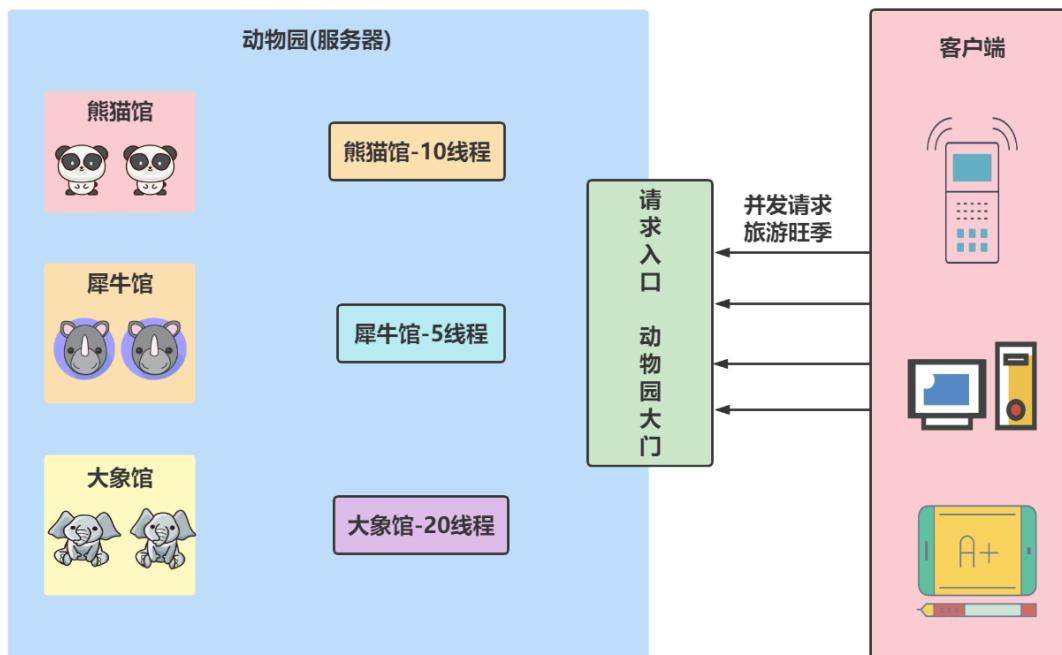
```

测试

配置隔离并发线程最大数量为5

```
[  : Initializing Spring DispatcherServlet 'dispatcherServlet'
  : Initializing Servlet 'dispatcherServlet'
  : Completed initialization in 1 ms
oller : **** 进入方法 *****
let]  : Servlet.service() for servlet [dispatcherServlet] in context with pat
```

服务断路器_Resilience4j线程池隔离实现



线程池隔离配置修改YML文件

```

1 resilience4j:
2   thread-pool-bulkhead:
3     instances:
4       backendA:
5         # 最大线程池大小
6         maxThreadPoolsSize: 4
7         # 核心线程池大小
8         coreThreadPoolsSize: 2
9         # 队列容量
10        queueCapacity: 2

```

编写controller

```

1 /**
2  * 测试线程池服务隔离
3  * @return
4 */
5 @Bulkhead(name = "backendA", type =
6 Bulkhead.Type.THREADPOOL)
7 @GetMapping("/futtrue")
8 public CompletableFuture future(){
9   log.info("***** 进入方法
10 *****");
11   try {
12     TimeUnit.SECONDS.sleep(5);
13   } catch (InterruptedException e) {
14     e.printStackTrace();
15   }
16   log.info("***** 离开方法
17 *****");
18   return
19   CompletableFuture.supplyAsync(() -> "线程池隔
20   离信息.....");

```

测试

配置文件设置核心线程2个最大4个服务会一次处理4个请求

```
: Initializing Spring DispatcherServlet 'dispatcher'
: Initializing Servlet 'dispatcherServlet'
: Completed initialization in 1 ms
:e : **** 进入方法 ****
```

服务断路器_Resilience4j限流

限流YML配置

```
1  ratelimiter:
2    instances:
3      backendA:
4        # 限流周期时长。          默认: 500纳秒
5        limitRefreshPeriod: 5s
6        # 周期内允许通过的请求数量。    默认: 50
7        limitForPeriod: 2
```

先写Controller

```
1  /**
2   * 限流
3   * @return
4   */
5  @GetMapping("/limiter")
```

```

6     @RateLimiter(name = "backendA")
7         public CompletableFuture<String>
8             RateLimiter() {
9                 log.info("***** 进入方法 *****");
10                //异步操作
11                CompletableFuture<String>
12                    CompletableFuture = CompletableFuture
13
14                .supplyAsync((Supplier<String>) () ->
15                    (paymentFeignService.index()));
16                log.info("***** 离开方法 *****");
17                return CompletableFuture;
18            }

```

JMeter压测

- 线程数 - 5



服务网关Gateway_微服务中的应用



没有服务网关

订单微服务 http://192.168.66.100:8001/order/*



支付微服务 http://192.168.66.102:8006/payment/*

用户微服务 http://192.168.66.103:8005/user/*

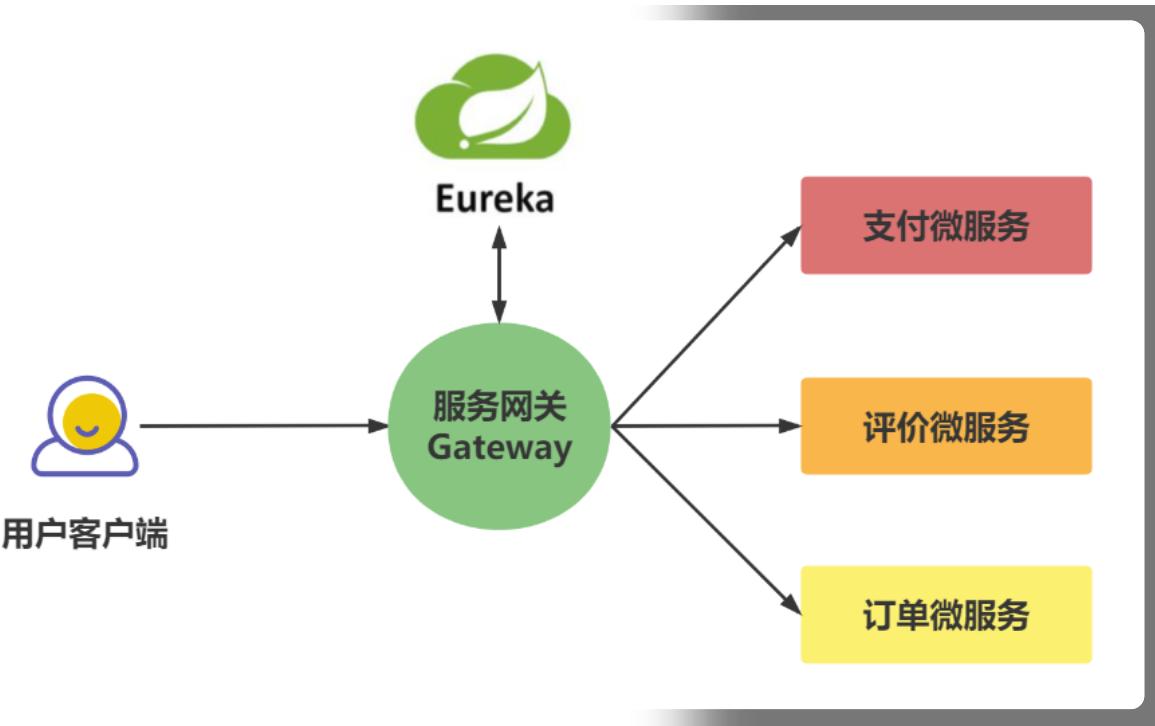
问题：

- ① 地址太多
- ② 安全性
- ③ 管理问题

为什么要使用服务网关



网关是微服务架构中不可或缺的部分。使用网关后，客户端和微服务之间的网络结构如下。

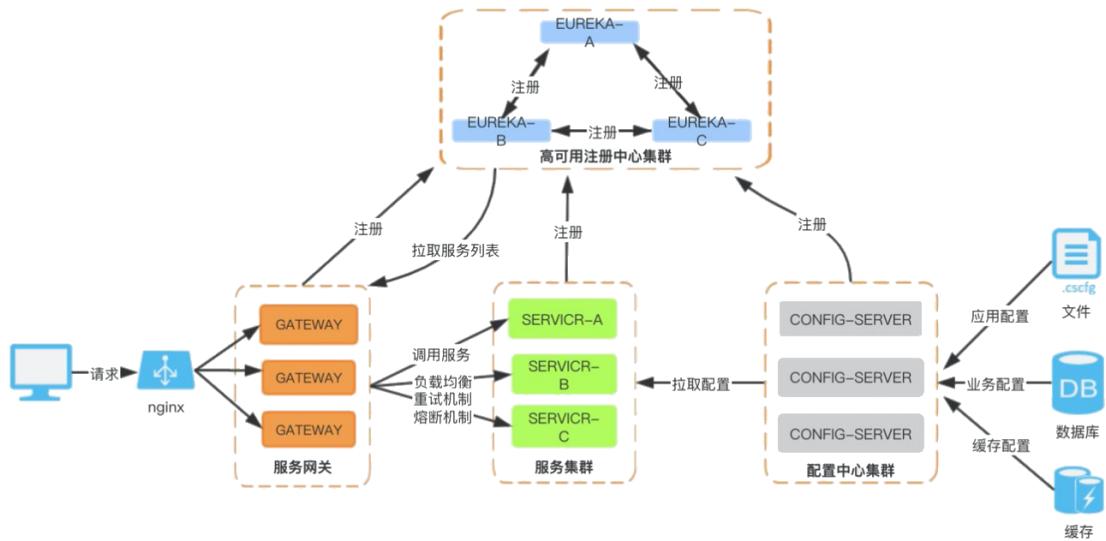


注意：

网关统一向外部系统（如访问者、服务）提供REST API。在SpringCloud中，使用Zuul、Spring Cloud Gateway等作为API Gateway来实现动态路由、监控、回退、安全等功能。

认识Spring Cloud Gateway

Spring Cloud Gateway 是 Spring Cloud生态系统中的网关，它是基于Spring 5.0、SpringBoot 2.0和Project Reactor等技术开发的，旨在为微服务架构提供一种简单有效的、统一的API路由管理方式，并为微服务架构提供安全、监控、指标和弹性等功能。其目标是替代Zuul。



注意：

Spring Cloud Gateway 用"Netty + Webflux"实现，不要加入 Web依赖，否则会报错，它需要加入Webflux依赖。

什么是WebFlux

Webflux模式替换了旧的Servlet线程模型。用少量的线程处理request和response io操作，这些线程称为Loop线程，而业务交给响应式编程框架处理，响应式编程是非常灵活的，用户可以将业务中阻塞的操作提交到响应式框架的工作线程中执行，而不阻塞的操作依然可以在Loop线程中进行处理，大大提高了Loop线程的利用率。

@Controller, @RequestMapping

Router Functions

spring-webmvc

spring-webflux

Servlet API

HTTP / Reactive Streams

Servlet Container

Tomcat, Jetty, Netty, Undertow

注意：

Webflux虽然可以兼容多个底层的通信框架，但是一般情况下，底层使用的还是Netty，毕竟，Netty是目前业界认可的最高性能的通信框架。而Webflux的Loop线程，正好就是著名的Reactor模式IO处理模型的Reactor线程，如果使用的是高性能的通信框架Netty。

温馨提示：

什么是Netty，Netty是一个基于NIO的客户、服务器端的编程框架。提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠的网络服务器和客户端程序。

Spring Cloud Gateway特点

- 易于编写谓词(Predicates)和过滤器 (Filters)。其Predicates和Filters可作用于特定路由。
- 支持路径重写。
- 支持动态路由。
- 集成了Spring Cloud DiscoveryClient。

实时效果反馈

1. Spring Cloud Gateway 是 Spring Cloud生态系统中的_____。

- A 断路器
- B 注册发现服务
- C 网关
- D 以上都不是

2. 下列不属于服务网关Gateway特点得是_____。

- A 支持路径重写
- B 支持动态路由

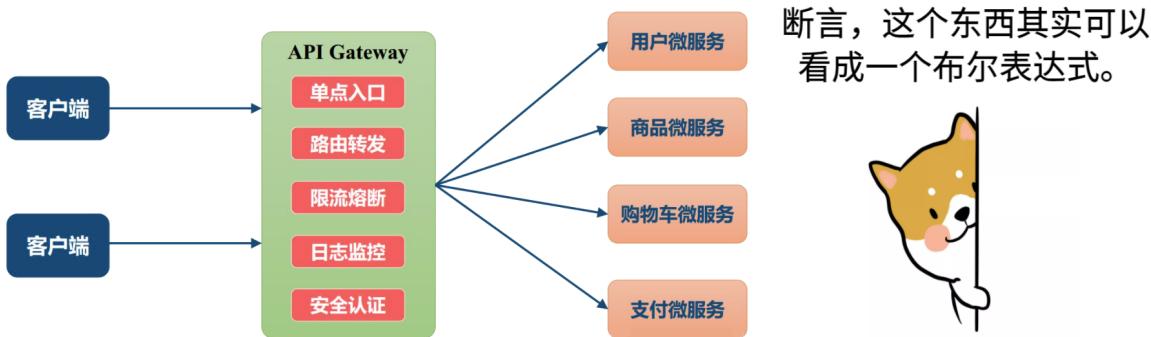
C 服务熔断

D 集成服务发现

答案

1=>C 2=>C

服务网关Gateway_三大核心概念



路由(Route)

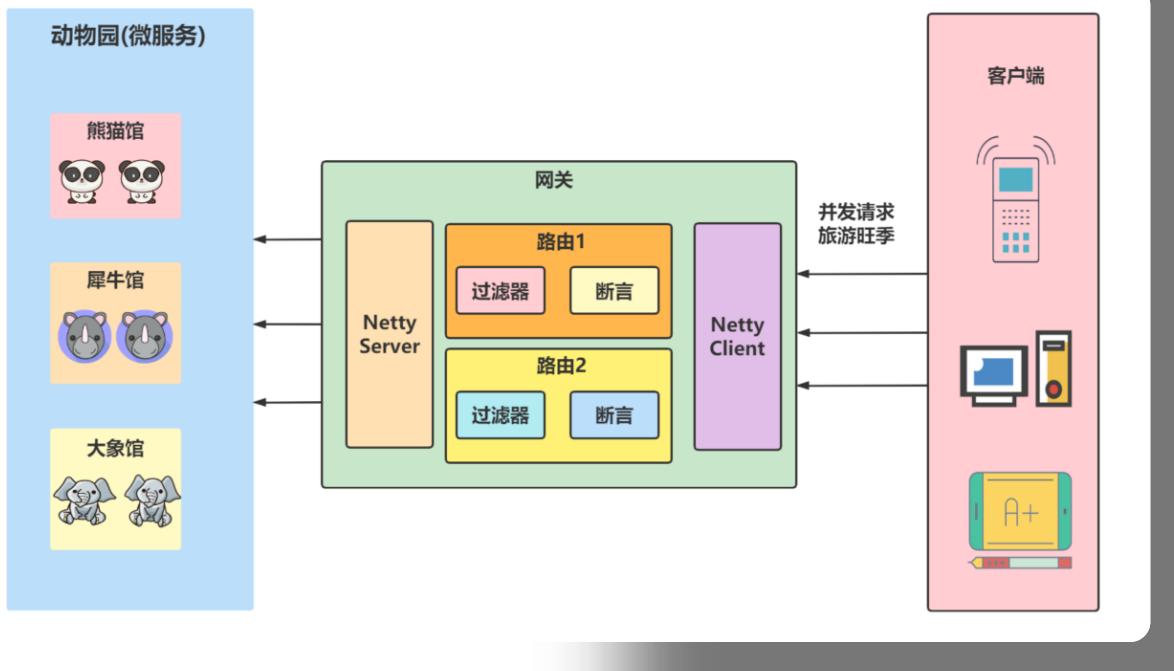
这是网关的基本构建块。它由一个ID，一个目标URI，一组断言和一组过滤器定义。如果断言为真，则路由匹配。

断言(predicate)

输入类型是一个ServerWebExchange。我们可以使用它来匹配来自HTTP请求的任何内容，例如headers或参数。

过滤(filter)

可以在请求被路由前或者之后对请求进行修改。



举个例子：

你想去动物园游玩，那么你买了一张熊猫馆的门票，只能进入熊猫馆的区域，而不能去犀牛管瞎转。因为没有犀牛馆的门票，进不去，就算走到门口，机器也不能识别。这里门票就相当于请求URL，熊猫馆相当于**路由**，而门口识别门卡的机器就是**断言**。然后我进入熊猫馆里面看到工作人员我想很有可能是**过滤器**，结果还真是，他在我进入馆之前拿手持设备对我全身扫描看看有没有危险品(请求前改代码)，并且在我出熊猫馆之后要求再次检察看看我是否携带熊猫出馆(请求后改动代码)。这个工作人员相当于**过滤器**。

总结

首先任何请求进来，网关都会把它们拦住。根据请求的URL把它们分配到不同的**路由**上，**路由**上面会有**断言**，来判断请求能不能进来。进来之后会有一系列的**过滤器**对请求被转发前或转发后进行改动。具体怎么个改动法，那就根据业务不同而自定义了。一般就是监控，限流，日志输出等等。

实时效果反馈

1.服务网关Gateway中路由作用是____。

- A 路径匹配
- B 匹配请求内容
- C 被路由前或者之后对请求进行修改
- D 以上都是错误

2.服务网关Gateway中断言作用是____。

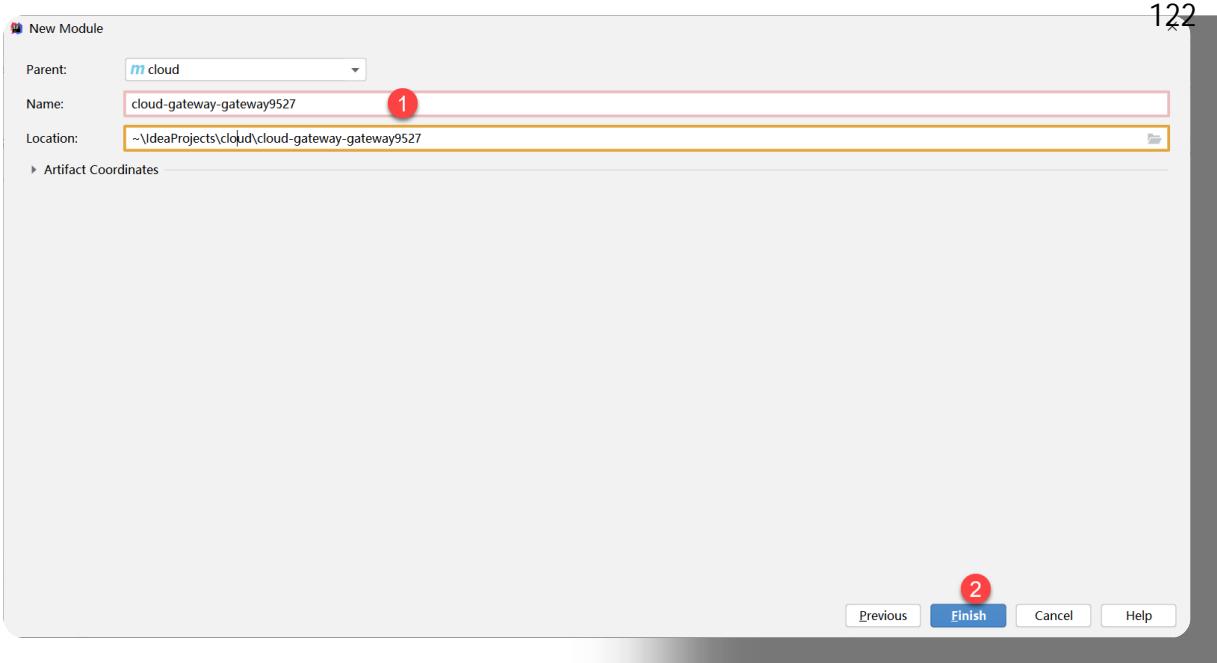
- A 路径匹配
- B 匹配请求内容
- C 被路由前或者之后对请求进行修改
- D 以上都是错误

答案

1=>D 2=>A

服务网关Gateway_入门案例

创建cloud-gateway-gateway9527工程



pom文件引入依赖

```
1 <dependencies>
2     <!-- 引入网关Gateway依赖 -->
3     <dependency>
4
5         <groupId>org.springframework.cloud</groupId>
6     >
7         <artifactId>spring-cloud-starter-gateway</artifactId>
8     </dependency>
9     <dependency>
10
11         <groupId>org.projectlombok</groupId>
12         <artifactId>lombok</artifactId>
13         <version>1.18.22</version>
14     </dependency>
15 </dependencies>
```

新增application.yml

```
1 server:
2   port: 9527
```

```

3 spring:
4   cloud:
5     gateway:
6       routes:
7         # 路由ID，没有固定规则但要求唯一，建议配合
8         服务名
9           - id: payment_provider
10          # 匹配后提供服务的路由地址
11          uri: http://localhost:8001
12          # 断言
13          predicates:
14            # 路径相匹配的进行路由
             - Path=/payment/get/**
```

主启动类

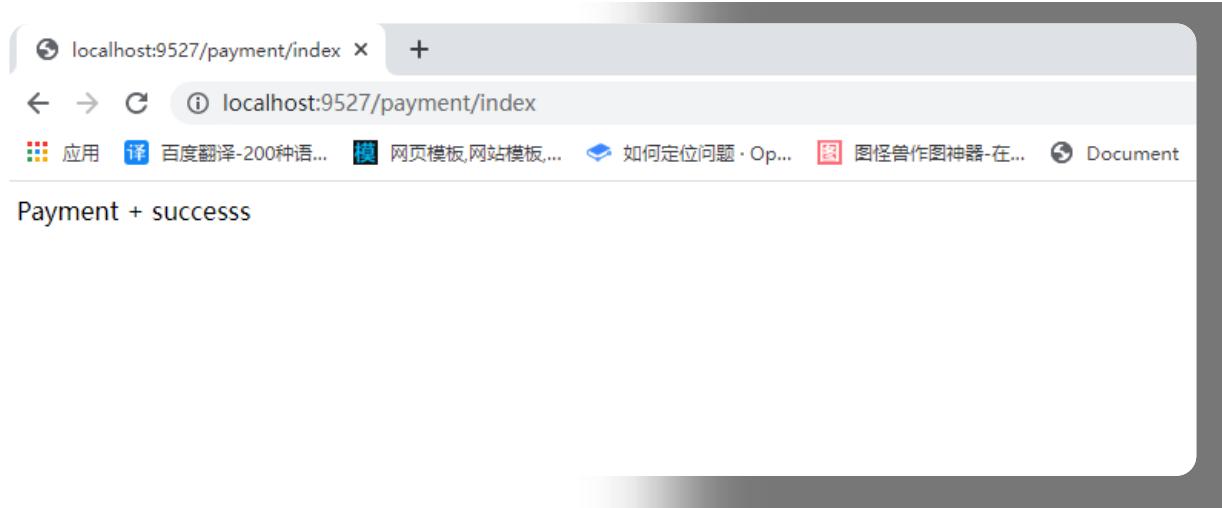
```

1 @SpringBootApplication
2 @EnableEurekaClient
3
4 public class GatewayMain {
5   public static void main(String[] args) {
6
7     SpringApplication.run(GatewayMain.class, args);
8     log.info("***** GatewayMain 服务
9     启动成功 *****");
10    }
11 }
```

测试

- 启动注册中心 7001,7002
- 启动服务提供者8001
- 启动网关服务9527

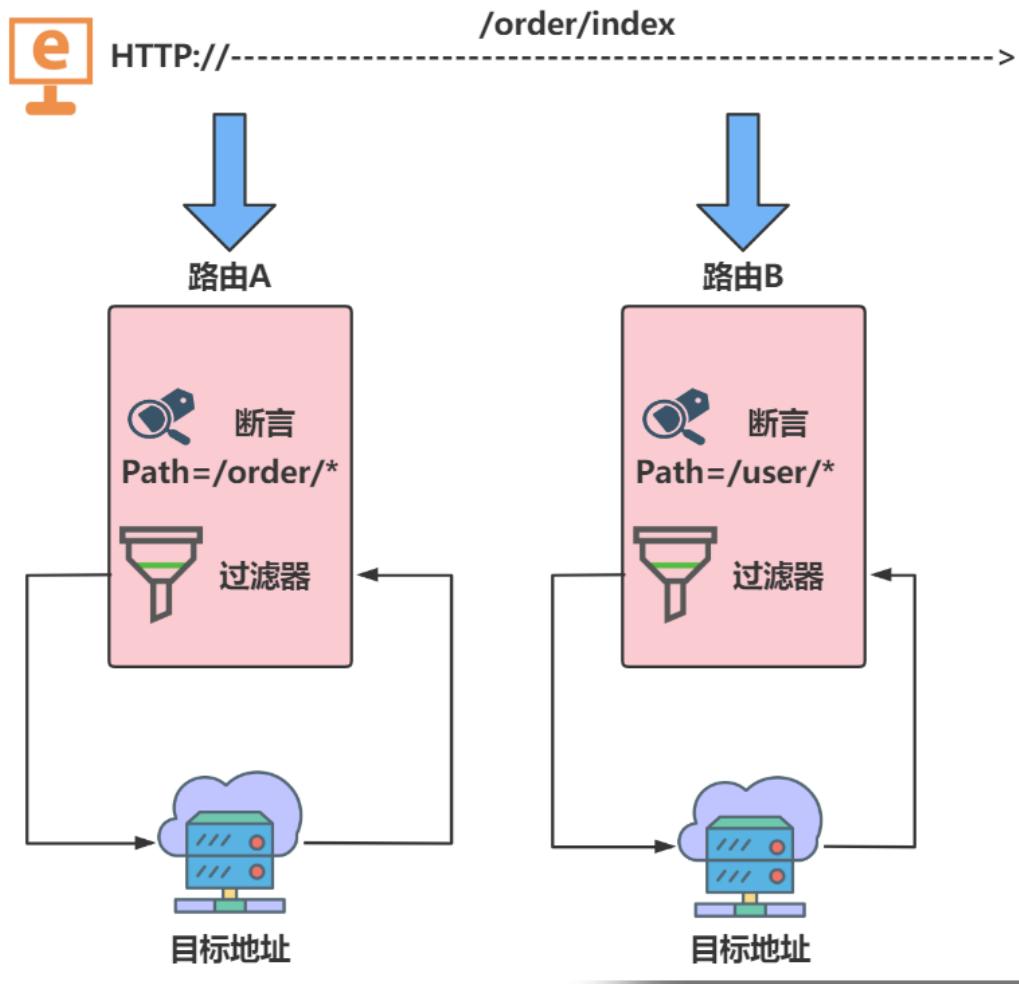
请求<http://localhost:9527/payment/index>



服务网关Gateway_路由规则



Gateway 的路由规则主要有三个部分，分别是路由、断言(谓词)和过滤器。



路由

路由是 Gateway 的一个基本单元。

断言

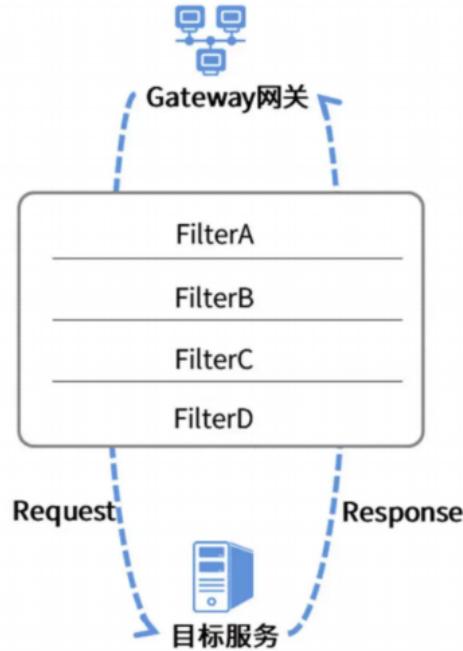
也称谓词，实际上是路由的判断规则。一个路由中可以添加多个谓词的组合。

提示：

打个比方，你可以为某个路由设置一条谓词规则，约定访问路径的匹配规则为

`Path=/bingo/*`，在这种情况下只有以 `/bingo` 打头的请求才会被当前路由选中。

过滤器



Gateway 组件使用了一种 FilterChain 的模式对请求进行处理，每一个服务请求（Request）在发送到目标服务之前都要被一串 FilterChain 处理。同理，在 Gateway 接收服务响应（Response）的过程中也会被 FilterChain 处理一把。

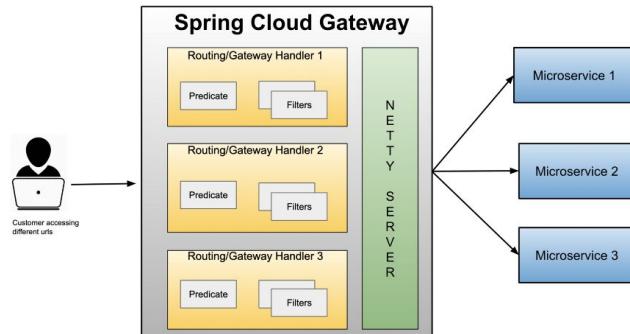
实时效果反馈

1. Spring Cloud Gateway 中断言主要作用_____。

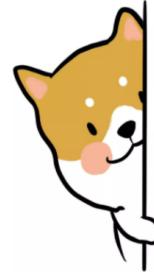
- A 判断规则
- B 注册发现服务
- C 过滤请求
- D 以上都不是

答案

1=>A



让我看看路由第二种配置方式



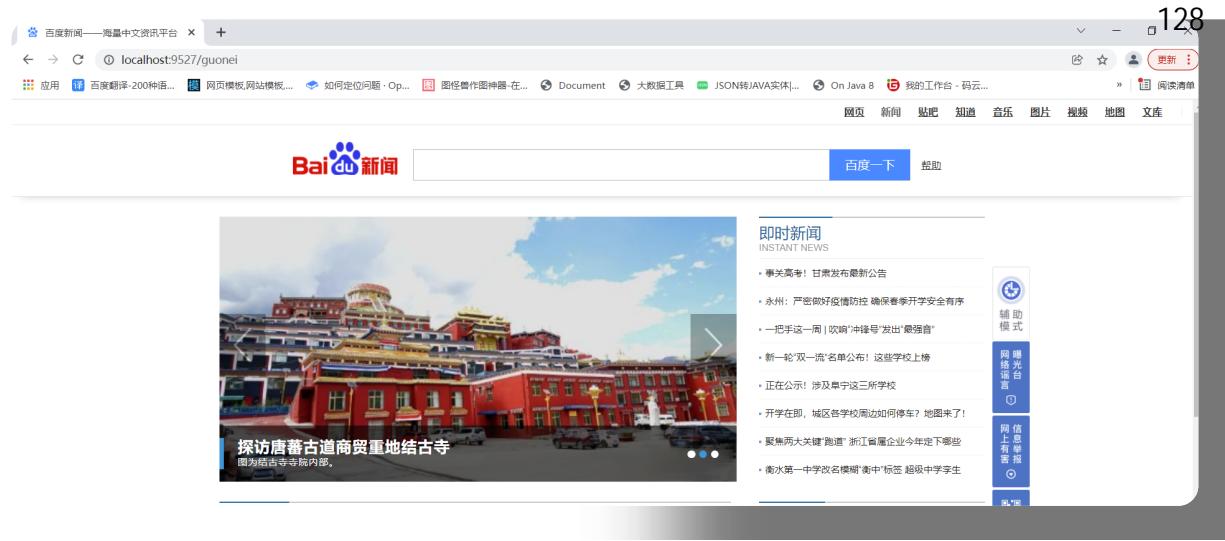
代码注入RouteLocator

```

1 @Configuration
2 public class GatewayConfig {
3
4     @Bean
5     public RouteLocator
6         customRouteLocator(RouteLocatorBuilder
7             routeLocatorBuilder){
8
9         RouteLocatorBuilder.Builder routes =
10        routeLocatorBuilder.routes();
11
12        //设置路由
13        routes.route("path_route", r ->
14            r.path("/guonei").uri("http://news.baidu.com
15            /guonei")).build();
16
17        return routes.build();
18    }
19
20 }

```

测试



实时效果反馈

1. 服务网关Gateway通过Java API构建时需要实现__接口构建路由规则。

A Route

B RouteLocator

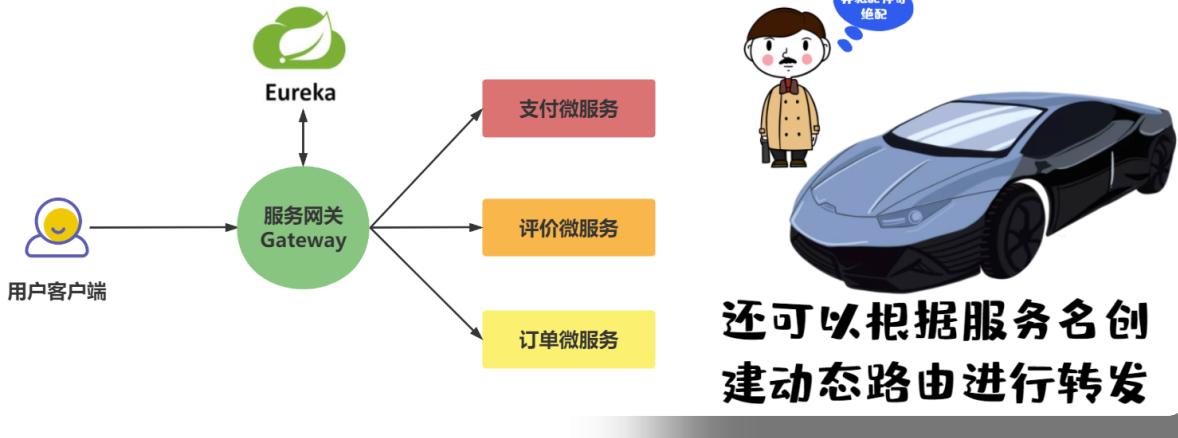
C RouteLocatorBuilder

D Builder

答案

1=>B

服务网关Gateway_动态路由



默认情况下Gateway会根据注册中心的服务列表，以注册中心上微服务名为路径创建动态路由进行转发，从而实现动态路由的功能。

添加eureka依赖

```

1 <!-- 引入Eureka client依赖 -->
2 <dependency>
3
4     <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-
6             netflix-eureka-client</artifactId>
7     </dependency>

```

添加yml配置

```

1 spring:
2   cloud:
3     gateway:
4       routes:
5         # 路由ID，没有固定规则但要求唯一，建议配合
6         # 服务名
7           - id: payment_provider
8             # 匹配后提供服务的路由地址 1b后跟提供服
9             # 务的微服务的名
10            uri: lb://CLOUD-PAYMENT-PROVIDER
11            # 断言
12            predicates:
13              # 路径相匹配的进行路由
14              - Path=/payment/**
```

注意：

需要注意的是uri的协议lb，表示启用Gateway的负载均衡的功能。

服务提供者payemt8001和payment8002工程controller添加方法

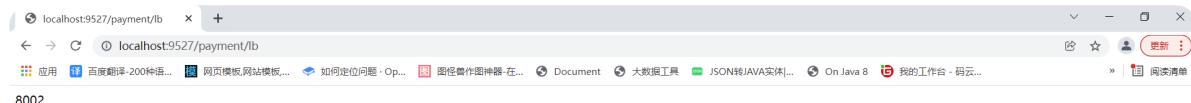
```

1
2 @value("${server.port}")
3 private String port;
4
5 /**
6 * 测试动态负载均衡
7 * @return
8 */
9 @GetMapping("lb")
10 public String lb() {
11     return port;
12 }

```

测试

- 启动eureka服务注册发现 7001,7002
- 启动服务提供者payment8001,8002
- 启动网关服务测试



实时效果反馈

1.服务网关Gateway配置中lb指__。

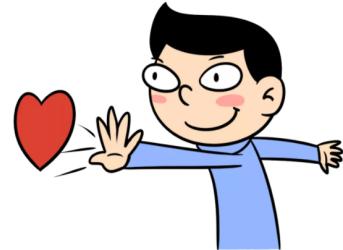
- A 过滤器
- B 路由
- C 目标地址
- D 提供服务的微服务的名

答案

1=>D

服务网关Gateway_断言功能详解

```
Loaded RoutePredicateFactory [After]
Loaded RoutePredicateFactory [Before]
Loaded RoutePredicateFactory [Between]
Loaded RoutePredicateFactory [Cookie]
Loaded RoutePredicateFactory [Header]
Loaded RoutePredicateFactory [Host]
Loaded RoutePredicateFactory [Method]
Loaded RoutePredicateFactory [Path]
Loaded RoutePredicateFactory [Query]
Loaded RoutePredicateFactory [ReadBody]
Loaded RoutePredicateFactory [RemoteAddr]
Loaded RoutePredicateFactory [Weight]
Loaded RoutePredicateFactory [CloudFoundryRouteService]
```



一个请求在抵达网关层后，首先就要进行断言匹配。

一个请求在抵达网关层后，首先就要进行断言匹配，在满足所有断言之后才会进入Filter阶段。说白了Predicate就是一种路由规则，通过Gateway中丰富的内置断言的组合，我们就能让一个请求找到对应的Route来处理。

After路由断言 Factory

After Route Predicate Factory采用一个参数——日期时间。在该日期时间之后发生的请求都将被匹配。

YML文件添加配置

```
1 predicates:
2   # 路径相匹配的进行路由
3   - Path=/payment/**
4   - After=2030-02-
5
6 15T14:54:23.317+08:00[Asia/Shanghai]
```

注意：

UTC时间格式的时间参数。

UTC时间格式的时间参数时间生成方法

```

1 public static void main(String[] args) {
2     ZonedDateTime now = ZonedDateTime.now();
3     System.out.println(now);
4 }
```

Before路由断言 Factory

Before Route Predicate Factory采用一个参数——日期时间。在该日期时间之前发生的请求都将被匹配。

YML文件添加配置

```

1 predicates:
2   - Before=2030-02-
3     15T14:54:23.317+08:00[Asia/Shanghai]
```

Between 路由断言 Factory

Between 路由断言 Factory有两个参数，datetime1和datetime2。在datetime1和datetime2之间的请求将被匹配。datetime2参数的实际时间必须在datetime1之后。

```

1 predicates:
2   - Between=2030-02-
3     15T14:54:23.317+08:00[Asia/Shanghai],2030-02-
4     15T14:54:23.317+08:00[Asia/Shanghai]
```

Cookie路由断言 Factory

顾名思义，Cookie验证的是Cookie中保存的信息，Cookie断言和上面介绍的两种断言使用方式大同小异，唯一的不同是它必须连同属性值一同验证，不能单独只验证属性是否存在。

YML文件添加配置

```

1 predicates:
2   # 路径相匹配的进行路由
3   - Cookie=username,zzyy

```

使用postman测试

The screenshot shows the Postman interface with a GET request to `http://localhost:9527/payment/lb`. The 'Headers' tab is selected, and a cookie named `username` is set to `zzyy`. The response details show a 200 OK status with 8 ms response time and 119 B size.

Header路由断言 Factory

这个断言会检查Header中是否包含了响应的属性，通常可以用来验证请求是否携带了访问令牌。

YML文件添加配置

```

1 # 请求头要有X-Request-Id属性并且值为整数的正则表达式
2 predicates:
3   - Header=X-Request-Id, \d+

```

测试

http://localhost:9527/payment/lb

Save Send

GET http://localhost:9527/payment/lb

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Headers 6 hidden

KEY	VALUE	DESCRIPTION	Bulk Edit	Presets
<input type="checkbox"/> Cookie	username=zzyy			
<input checked="" type="checkbox"/> X-Request-Id	123			
Key	Value	Description		

Host路由断言 Factory

Host 路由断言 Factory包括一个参数：host name列表。使用Ant路径匹配规则，`.`作为分隔符。访问的主机匹配http或者https, baidu.com 默认80端口, 就可以通过路由。多个`.`号隔开。

YML文件添加配置

```
1 predicates:
2 - Host=itbaizhan
```

Host文件新增配置

```
1 127.0.0.1 itbaizhan
```

Method路由断言 Factory

这个断言是专门验证HTTP Method的，在下面的例子中，我们把Method断言和Path断言通过一个and连接符合并起来，共同作用于路由判断，当我们访问“/gateway/sample”并且HTTP Method是GET的时候，将适配下面的路由。

YML文件添加配置

```

1 spring:
2   cloud:
3     gateway:
4       routes:
5         # 路由ID, 没有固定规则但要求唯一, 建议配合
6         # 服务名
7         - id: payment_provider
8           # 匹配后提供服务的路由地址
9           uri: lb://CLOUD-PAYMENT-PROVIDER
10          # 断言
11          predicates:
12            # 路径相匹配的进行路由
13            - Path=/payment/**
```

Query路由断言 Factory

请求断言也是在业务中经常使用的，它会从ServerHttpRequest中的Parameters列表中查询指定的属性，有如下两种不同的使用方式。

YML文件添加配置

```

1 spring:
2   cloud:
3     gateway:
4       routes:
5         # 路由ID, 没有固定规则但要求唯一, 建议配合
6         # 服务名
7         - id: payment_provider
8           # 匹配后提供服务的路由地址
9           uri: lb://CLOUD-PAYMENT-PROVIDER
10          # 断言
11          predicates:
```

```

11      # 路径相匹配的进行路由
12      - Path=/payment/***
13      # 要有参数名称并且是正整数才能路由
14      - Query=username,\d+

```

测试

The screenshot shows the Postman interface with a GET request to `http://localhost:9527/payment/lb?username=123`. The 'Params' tab is active, displaying a table with one row: `username` (KEY) and `123` (VALUE). Other tabs like 'Authorization', 'Headers (8)', and 'Body' are visible at the top.

KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/> username	123			
Key	Value	Description		

实时效果反馈

1.服务网关Gateway中在该日期时间之后发生的请求都将被匹配，使用路由断言。

- A After
- B Before
- C Cookie
- D Header

2.服务网关Gateway中检查Header中是否包含了响应的属性，使用路由断言。

- A After
- B Before

C

Cookie

D

Header

答案

1=>A 2=>D

服务网关Gateway_过滤器详解



微服务系统中的服务非常多。如果每个服务都自己做鉴权、限流、日志输出，则非常不科学。所以，可以通过网关的过滤器来处理这些工作。在用户访问各个服务前，应在网关层统一做好鉴权、限流等工作。

Filter的生命周期

根据生命周期可以将Spring Cloud Gateway中的Filter分为"PRE"和"POST"两种。

- **PRE**: 代表在请求被路由之前执行该过滤器，此种过滤器可用来实现参数校验、权限校验、流量监控、日志输出、协议转换等功能。
- **POST**: 代表在请求被路由到微服务之后执行该过滤器。此种过滤器可用来实现响应头的修改（如添加标准的HTTP Header）、收集统计信息和指标、将响应发送给客户端、输出日志、流量监控等功能。

Filter分类

根据作用范围， Filter可以分为以下两种。

- GatewayFilter：网关过滤器，此种过滤器只应用在单个路由或者一个分组的路由上。
- GlobalFilter：全局过滤器，此种过滤器会应用在所有的路由上。

实时效果反馈

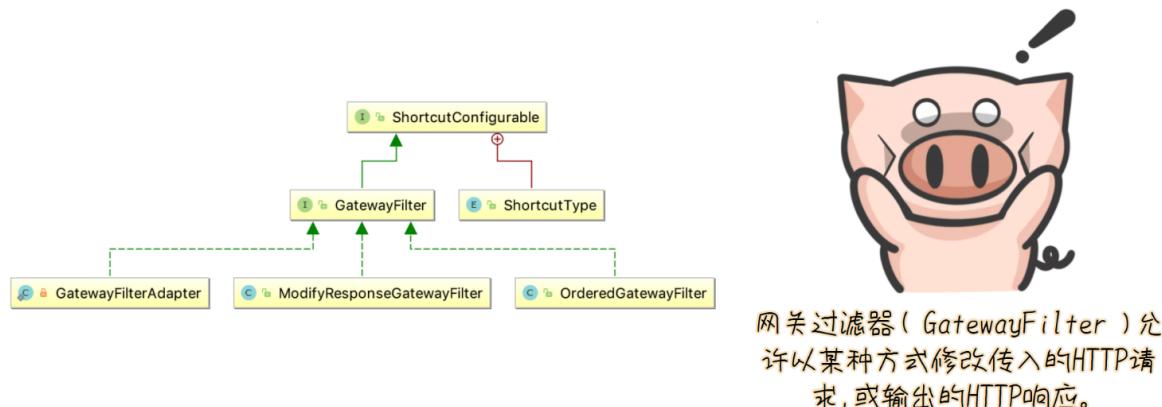
1.下列不属于服务网关Gateway过滤器应用场景。

- A 鉴权
- B 限流
- C 日志输出
- D 服务注册发现

答案

1=>D

服务网关Gateway_网关过滤器GatewayFilter



网关过滤器(GatewayFilter)允许以某种方式修改传入的HTTP请求, 或输出的HTTP响应。网关过滤器作用于特定路由。Spring Cloud Gateway内置了许多网关过滤器工厂来编写网关过滤器。

过滤器工厂	作用	参数
AddRequestHeader	为原始请求添加Header	Header的名称及值
AddRequestParameter	为原始请求添加请求参数	参数名称及值
AddResponseHeader	为原始响应添加Header	Header的名称及值
DedupeResponseHeader	剔除响应头中重复的值	需要去重的Header名称及去重策略
Hystrix	为路由引入Hystrix的断路器保护 HystrixCommand的名称	
FallbackHeaders	为fallbackUri的请求头中添加具体的异常信息	Header的名称
PrefixPath	为原始请求路径添加前缀	前缀路径
PreserveHostHeader	为请求添加一个preserveHostHeader=true的属性, 路由过滤器会检查该属性以决定是否要发送原始的Host	无
RequestRateLimiter	用于对请求限流, 限流算法为令牌桶	keyResolver、rateLimiter、statusCode、denyEmptyKey、emptyKeyStatus
RedirectTo	将原始请求重定向到指定的URL	http状态码及重定向的url
RemoveHopByHopHeadersFilter	为原始请求删除IETF组织规定的一系列Header	默认就会启用, 可以通过配置指定仅删除哪些Header
RemoveRequestHeader	为原始请求删除某个Header	Header名称
RemoveResponseHeader	为原始响应删除某个Header	Header名称
RewritePath	重写原始的请求路径	原始路径正则表达式以及重写后路径的正则表达式
RewriteResponseHeader	重写原始响应中的某个Header	Header名称, 值的正则表达式, 重写后的值
SaveSession	在转发请求之前, 强制执行WebSession::save操作	无
secureHeaders	为原始响应添加一系列起安全作用的响应头	无, 支持修改这些安全响应头的值
SetPath	修改原始的请求路径	修改后的路径
SetResponseHeader	修改原始响应中某个Header的值	Header名称, 修改后的值
Status	修改原始响应的状态码	HTTP 状态码, 可以是数字, 也可以是字符串
StripPrefix	用于截断原始请求的路径	使用数字表示要截断的路径的数量
Retry	针对不同的响应进行重试	retries、statuses、methods、series
RequestSize	设置允许接收最大请求包的大小。如果请求包大小超过设置的值, 则返回 413 Payload Too Large	请求包大小, 单位为字节, 默认值为5M
ModifyRequestBody	在转发请求之前修改原始请求体内容	修改后的请求体内容
ModifyResponseBody	修改原始响应体的内容	修改后的响应体内容
Default	为所有路由添加过滤器	过滤器工厂名称及值

常用内置过滤器的使用

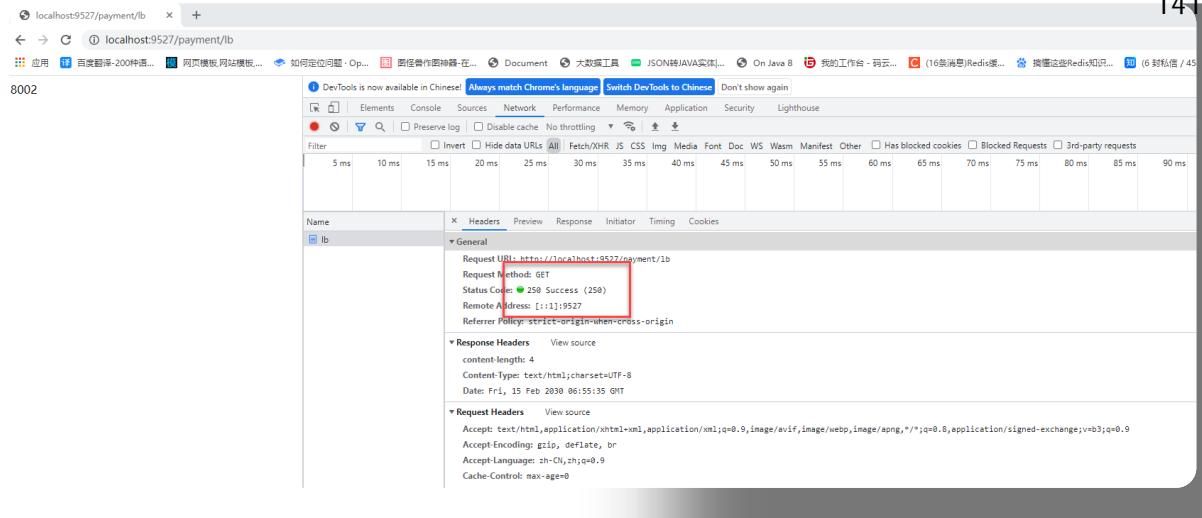
配置文件中加入setStatus过滤器

```

1 #过滤器, 请求在传递过程中可以通过过滤器对其进行一定的修改
2 filters:
3   - SetStatus=250 # 修改原始响应的状态码

```

启动测试



实时效果反馈

1.服务网关Gateway为原始请求加入Header使用____过滤器。

- A AddRequestHeader
- B AddRequestParameter
- C SetPath
- D Retry

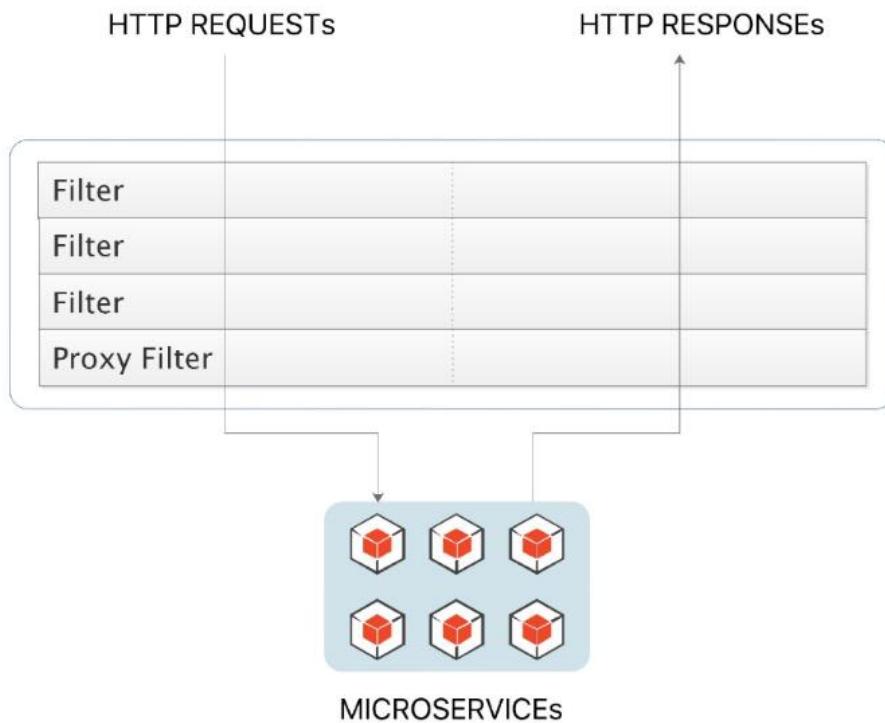
2.服务网关Gateway为原始请求添加请求参数____过滤器。

- A AddRequestHeader
- B AddRequestParameter
- C SetPath
- D Retry

答案

1=>A 2=>B

服务网关Gateway_自定义网关过滤器



需求：通过过滤器，配置是否在控制台输出日志信息，以及是否记录日志。

实现步骤：

- 1、类名必须叫做XxxGatewayFilterFactory，注入到Spring容器后使用时的名称就叫做Xxx。
- 2、创建一个静态内部类Config，里面的属性为配置文件中配置的参数， - 过滤器名称=参数1,参数2...
- 2、类必须继承 AbstractGatewayFilterFactory，让父类帮实现配置参数的处理。
- 3、重写shortcutFieldOrder()方法，返回List参数列表为Config中属性集合

```
return Arrays.asList("参数1",参数2...)
```

- 4、无参构造方法中super(Config.class)
- 5、编写过滤逻辑 public GatewayFilter apply(Config config)

在配置文件中，添加一个Log的过滤器配置

```

1 # 过滤器，请求在传递过程中可以通过过滤器对其进行一定的修改
2 filters:
3   # 控制日志是否开启
4   - Log=true

```

自定义一个过滤器工厂，实现里面的方法

```

1 /**
2  * 自定义局部过滤器
3 */
4 @Component
5 public class LogGatewayFilterFactory extends
AbstractGatewayFilterFactory<LogGatewayFilterFactory.Config> {
6
7     public LogGatewayFilterFactory() {
8         super(Config.class);
9     }
10
11    @Override
12    public List<String> shortcutFieldOrder() {
13        return Arrays.asList("consoleLog");
14    }
15
16    @Override
17    public GatewayFilter apply(Config config) {

```

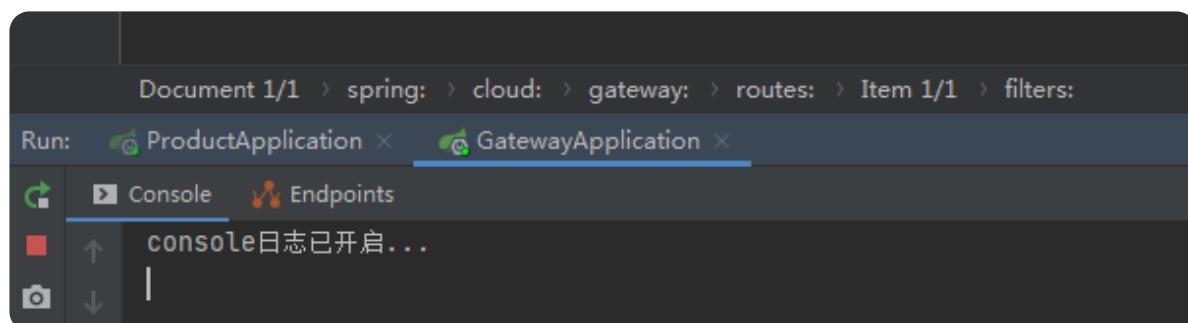
```

18     return ((exchange, chain) -> {
19         if (config.consoleLog) {
20             System.out.println("console
21                 日志已开启...");;
22         }
23         return chain.filter(exchange);
24     });
25
26     @Data
27     public static class Config{
28         private boolean consoleLog;
29     }
30
31 }

```

运行测试

请求<http://localhost:9527/payment/lb>



实时效果反馈

1. 服务网关Gateway自定义网关过滤器需要继承_____类。

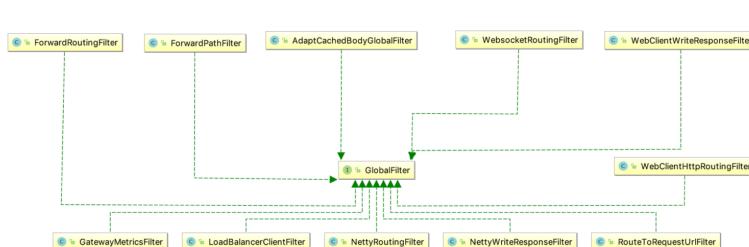
- A AbstractGatewayFilterFactory
- B GatewayFilterFactory
- C GatewayFilter

D 以上都错误

答案

1=>A

服务网关Gateway_过滤器之全局过滤器



全局过滤器由一系列特殊的过滤器组成。它会应用到所有路由中。

全局过滤器作用于所有路由，无需配置。通过全局过滤器可以实现对权限的统一校验，安全性验证等功能。

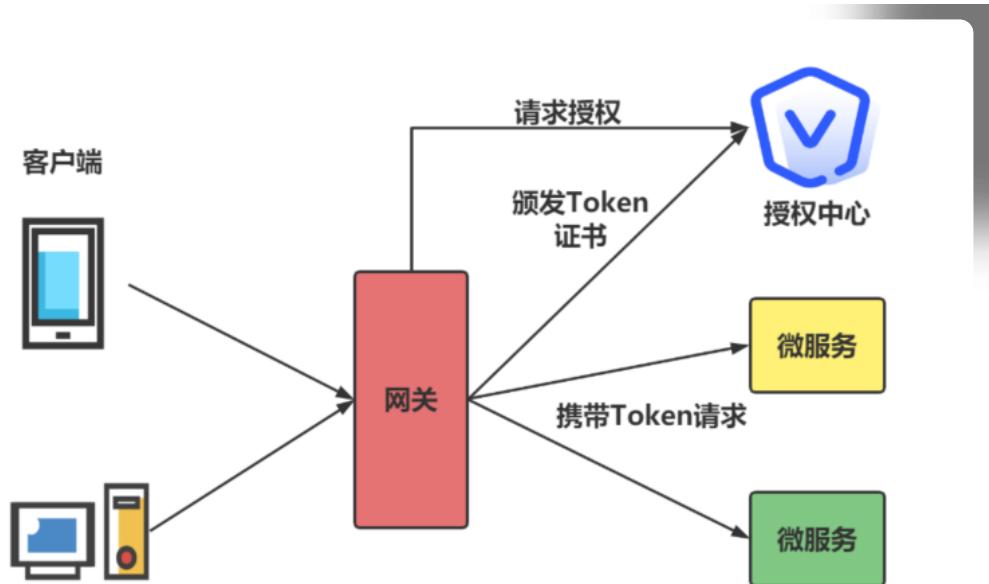
内置的全局过滤器

SpringCloud Gateway内部也是通过一系列的内置全局过滤器对整个路由转发进行处理的。

- 路由过滤器 (Forward)
- 路由过滤器 (LoadBalancerClient)
- Netty路由过滤器
- Netty写响应过滤器 (Netty Write Response F)
- RouteToRequestUrl 过滤器

- 路由过滤器 (Websocket Routing Filter)
- 网关指标过滤器 (Gateway Metrics Filter)
- 组合式全局过滤器和网关过滤器排序 (Combined Global Filter and GatewayFilter Ordering)
- 路由 (Marking An Exchange As Routed)

自定义全局过滤器



开发中的鉴权逻辑

- 当客户端第一次请求服务时，服务端对用户进行信息认证（登录）
- 认证通过，将用户信息进行加密形成token，返回给客户端，作为登录凭证
- 以后每次请求，客户端都携带认证的token
- 服务端对token进行解密，判断是否有效。

对于验证用户是否已经登录及鉴权的过程，可以在网关统一校验。

下面我们通过自定义一个GlobalFilter，去校验所有请求的请求参数中是否包含“token”，如果不包含请求参数“token”则不转发路由，否则执行正常逻辑。

```

1 package com.itbaizhan.filter;
2
3 import
org.springframework.cloud.gateway.filter.Gat
ewayFilterChain;
  
```

```
4 import  
5     org.springframework.cloud.gateway.filter.Glo  
6     balFilter;  
7 import  
8     org.springframework.core.Ordered;  
9 import  
10    org.springframework.http.HttpStatus;  
11  
12    /**  
13     * 自定义全局过滤器，需要实现GlobalFilter和  
14     * Ordered接口  
15     */  
16 @Component  
17 public class AuthGlobalFilter implements  
18     GlobalFilter, Ordered {  
19     @Override  
20     public Mono<Void>  
21     filter(ServerWebExchange exchange,  
22             GatewayFilterChain chain) {  
23         String token =  
24             exchange.getRequest().getQueryParams().getFi  
25             rst("token");  
26         if (StringUtils.isEmpty(token)) {  
27             System.out.println("鉴权失败。确少  
28             token参数。");  
29         }  
30     }  
31 }
```

```
22     exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
23         return
24     exchange.getResponse().setComplete();
25
26     if (!"jack".equals(token)) {
27         System.out.println("token无
28 效...");
29
30     exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
31         return
32     exchange.getResponse().setComplete();
33
34 }
35
36 /**
37 * 顺序，数值越小，优先级越高
38 * @return
39 */
40 @Override
41 public int getOrder() {
42     return 0;
43 }
44 }
45
```

测试

The screenshot shows the Postman application interface. At the top, the URL is set to `http://localhost:9527/payment/lb?token=jack`. Below the URL, there are tabs for `GET`, `Authorization`, `Headers (8)`, `Body`, `Pre-request Script`, `Tests`, and `Settings`. The `Params` tab is currently active, showing a table for `Query Params`. The table has columns for `KEY`, `VALUE`, `DESCRIPTION`, and `Bulk Edit`. It contains two rows: one for `username` with value `123` and another for `token` with value `jack`. Below the table, there are tabs for `Body`, `Cookies`, `Headers (3)`, and `Test Results`. The `Test Results` tab is selected, showing a summary: `250 Success (250)`, `5.21 s`, `130 B`. At the bottom, there are buttons for `Pretty`, `Raw`, `Preview`, `Visualize`, `Text`, and a search bar.

实时效果反馈

1. 服务网关Gateway自定义全局过滤器需要实现_____接口。

A `AbstractGatewayFilterFactory`

B `GatewayFilterFactory`

C `GlobalFilter`

D 以上都错误

2. 服务网关Gateway自定义全局过滤器实现`Ordered`接口含义是_____。

A 排序

B 设置比较器

C 设置全局过滤器执行顺序

D 以上都错误

答案

1=>C 2=>C

服务网关Gateway_网关的cors跨域配置



同源策略是一种约定

为什么会出现跨域问题

出于浏览器的同源策略限制。同源策略是一种约定，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，则浏览器的正常功能可能都会受到影响。可以说Web是构建在同源策略基础之上的，浏览器只是针对同源策略的一种实现。

什么是跨域

当一个请求url的**协议、域名、端口**三者之间任意一个与当前页面url不同即为跨域

当前页面url	被请求页面url	是否跨域	原因
http://www.test.com/	http://www.test.com/index.html	否	同源 (协议、域名、端口号相同)
http://www.test.com/	https://www.test.com/index.html	跨域	协议不同 (http/https)
http://www.test.com/	http://www.baidu.com/	跨域	主域名不同 (test/baidu)
http://www.test.com/	http://blog.test.com/	跨域	子域名不同 (www/blog)
http://www.test.com:8080/	http://www.test.com:7001/	跨域	端口号不同 (8080/7001)

跨域问题演示

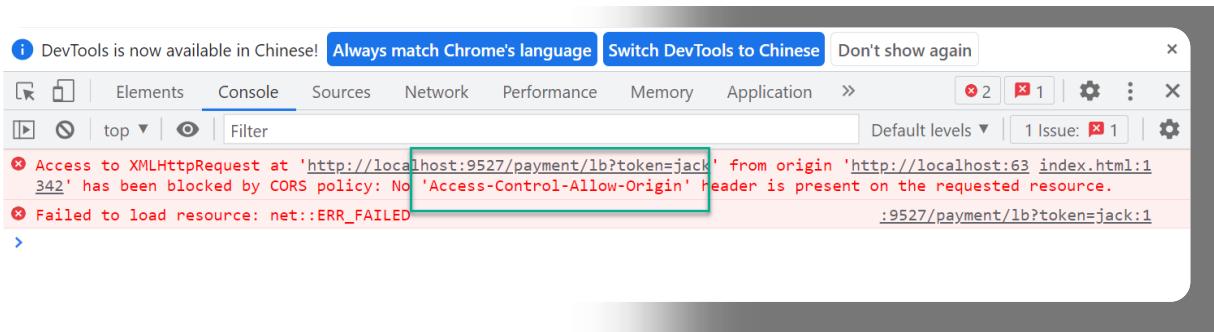
编写index页面

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Title</title>
6   </head>
7   <body>
8
9   </body>
10  <script
11    src="http://libs.baidu.com/jquery/2.0.0/jquery.min.js"></script>
12
13  <script>
14    $.get("http://localhost:9527/payment/1b?token=jack", function(data, status){
15      alert("Data: " + data + "\nStatus: "
16      + status);
17    });
18  </script>

```

问题出现



Gateway解决如何允许跨域



常见的用于解决跨域调用接口的问题就是CORS

CORS

- 如何允许跨域，一种解决方法就是**目的域告诉请求者允许什么来源域来请求**，那么浏览器就会知道B域是否允许A域发起请求。
- CORS ("跨域资源共享"(Cross-origin resource sharing)) 就是这样一种解决手段。

CORS使得浏览器在向目的域发起请求之前先发起一个OPTIONS方式的请求到目的域获取目的域的信息，比如获取目的域允许什么域来请求的信息。

```

1 spring:
2   cloud:
3     gateway:
4       globalcors:
5         cors-configurations:
6           '[/**]':
7             allowCredentials: true
8             allowedOriginPatterns: "*"
9             allowedMethods: "*"
10            allowedHeaders: "*"
11            add-to-simple-url-handler-mapping:
12              true

```

实时效果反馈

1.当一个请求url的____之间任意一个与当前页面url不同即为跨域。

- A 协议、域名
- B 协议、端口
- C 协议、域名、请求类型
- D 协议、域名、端口

2.最常用解决跨域的手段是____。

- A CORS
- B COS
- C COR
- D 以上都是错误

答案

1=>D 2=>A

服务网关Gateway实现用户鉴权_什么是JWT



{ JWT }
JSON Web Token



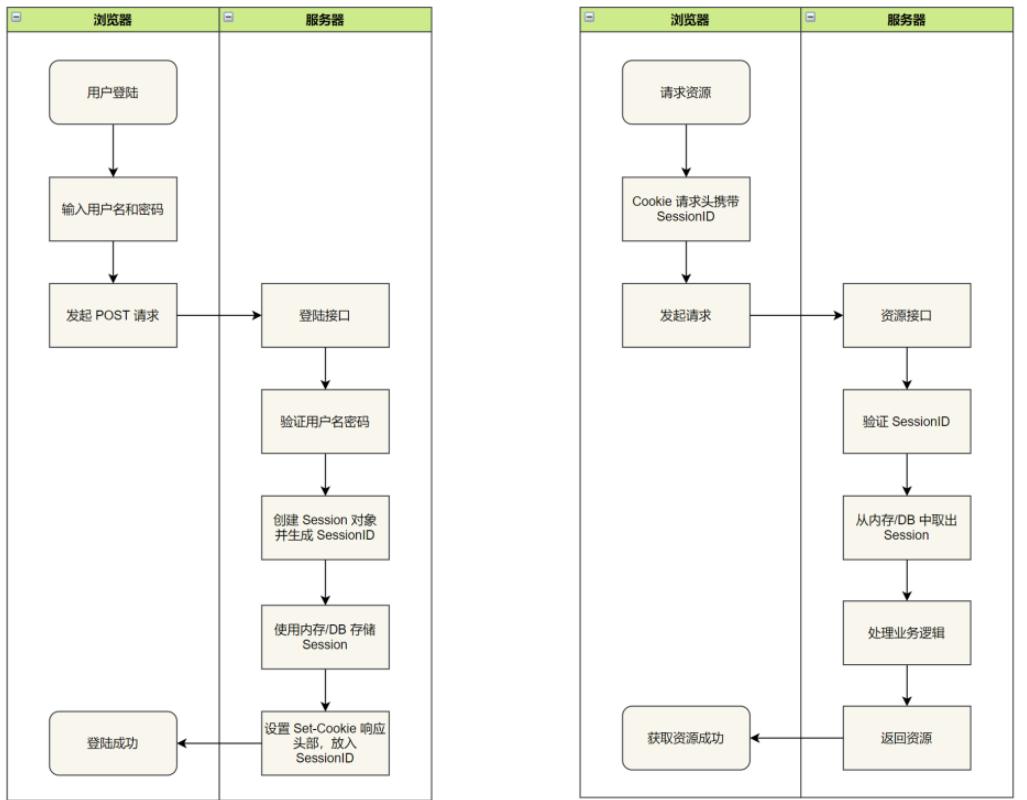
我们使用JWT在用户和
服务器之间传递安全可
靠的信息。

什么是JWT

JWT是一种用于双方之间传递安全信息的简洁的、[URL](#)安全的声明规范。定义了一种简洁的，自包含的方法用于通信双方之间以Json对象的形式安全的传递信息。特别适用于分布式站点的单点登录（SSO）场景。

传统的session认证

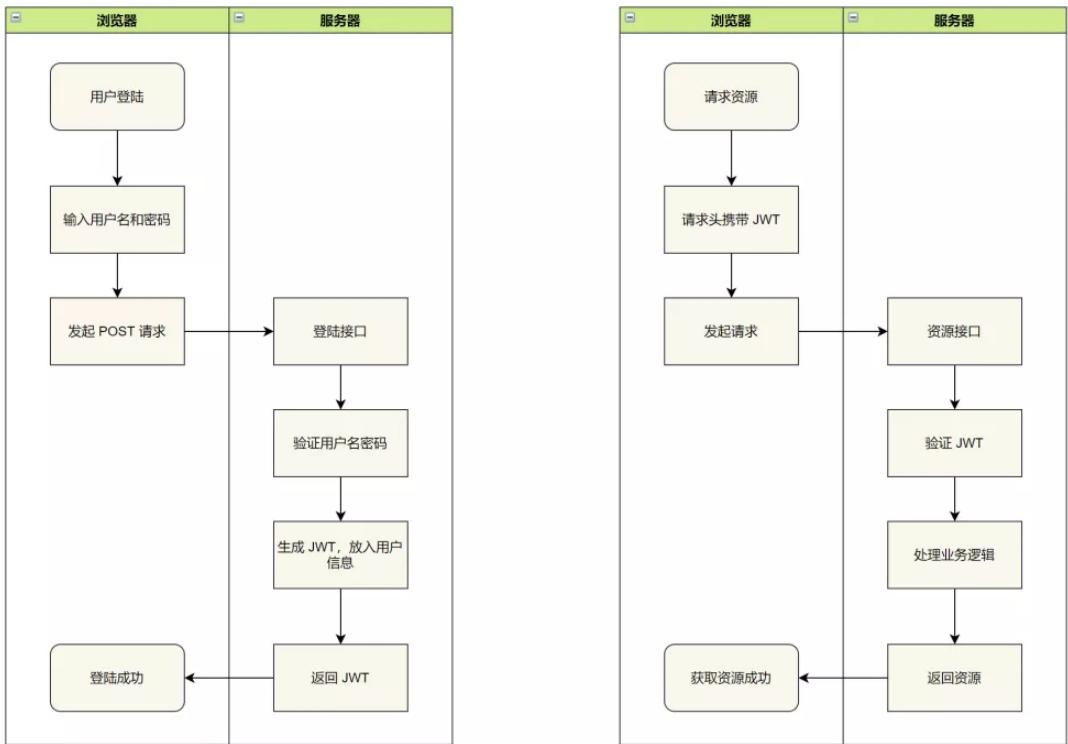
每次提到无状态的JWT时相信都会看到另一种基于Session的用户认证方案介绍，这里也不例外，Session的认证流程通常会像这样：



缺点：

- 安全性**: CSRF攻击因为基于cookie来进行用户识别, cookie如果被截获, 用户就会很容易受到跨站请求伪造的攻击。
- 扩展性**: 对于分布式应用, 需要实现 session 数据共享
- 性能**: 每一个用户经过后端应用认证之后, 后端应用都要在服务端做一次记录, 以方便用户下次请求的鉴别, 通常而言session都是保存在内存中, 而随着认证用户的增多, 服务端的开销会明显增大, 与REST风格不匹配。因为它在一个无状态协议里注入了状态。

JWT方式



优点：

- 无状态
- 适合移动端应用
- 单点登录友好

实时效果反馈

1. 传统的session认证缺点的是__。

- A 内存容量
- B 网络安全
- C 分布式应用session共享
- D 以上都正确

2. 下列属于JWT优点的是__。

- A 无状态
- B 适合移动端应用

C 单点登录友好

D 以上都正确

答案

1=>D 2=>D

服务网关Gateway实现用户鉴权 JWT原理



JWT 的原理是，服务器认证以后，生成一个 JSON 对象



JWT 的原理是，服务器认证以后，生成一个 JSON 对象，发回给用户，就像下面这样。

```

1  {
2    "姓名": "张三",
3    "角色": "管理员",
4    "到期时间": "2030年7月1日0点0分"
5 }
```

注意：

用户与服务端通信的时候，都要发回这个 JSON 对象。服务器完全只靠这个对象认定用户身份。为了防止用户篡改数据，服务器在生成这个对象的时候会加上签名，服务器就不保存任何 session 数据了，也就是说，服务器变成无状态了，从而比较容易实现扩展。

JWT的结构

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

注意：

它是一个很长的字符串，中间用点（.）分隔成三个部分。注意，JWT 内部是没有换行的，这里只是为了便于展示，将它写成了几行。

JWT 的三个部分依次如下：

- 头部 (header)
- 载荷 (payload)
- 签证 (signature)

Header

JSON对象，描述 JWT 的元数据。其中 alg 属性表示签名的算法 (algorithm)，默认是 HMAC SHA256 (写成 HS256)；typ 属性表示这个令牌 (token) 的类型 (type)，统一写为 JWT。

```

1
2 {
3   "alg": "HS256",
4   "typ": "JWT"
5 }
```

注意：

上面代码中，`alg` 属性表示签名的算法（algorithm），默认是 HMAC SHA256（写成 HS256）；`typ` 属性表示这个令牌（token）的类型（type），JWT 令牌统一写为`JWT` 然后将头部进行**Base64编码**构成了第一部分，Base64是一种用64个字符来表示任意二进制数据的方法，Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量二进制数据。

Payload



载荷载荷，啥意思？

内容又可以分为3中标准

- 标准中注册的声明
- 公共的声明
- 私有的声明

payload-标准中注册的声明 (建议但不强制使用) :

- `iss`: jwt签发者
- `sub`: jwt所面向的用户

- **aud**: 接收jwt的一方
- **exp**: jwt的过期时间，这个过期时间必须要大于签发时间
- **nbf**: 定义在什么时间之前，该jwt都是不可用的.
- **iat**: jwt的签发时间
- **jti**: jwt的唯一身份标识，主要用来作为一次性token,从而回避重放攻击。

payload-公共的声明 :

公共的声明可以添加任何的信息。一般这里我们会存放一下用户的基本信息（非敏感信息）。

payload-私有的声明 :

私有声明是提供者和消费者所共同定义的声明。需要注意的是，**不要存放敏感信息，不要存放敏感信息，不要存放敏感信息！！！**

我是好人，你们一定要信我！



因为：这里也是base64编码，任何人获取到jwt之后都可以解码！！

```

1  {
2    "sub": "1234567890",
3    "name": "John Doe",
4    "iat": 1516239022
5 }
```

注意：

sub和iat是标准声明，分别代表所面向的用户和jwt签发时间。

- **sub**: 这个是发给一个账号是1234567890的用户（也许是ID）
- **name**: 名字叫John Doe
- **iat**: 签发时间是1516239022 (2030/1/18 9:30:22)

Signature

这部分就是 JWT 防篡改的精髓，其值是对前两部分 base64UrlEncode 后使用指定算法签名生成，以默认 HS256 为例，指定一个密钥 (secret)，就会按照如下公式生成：

```

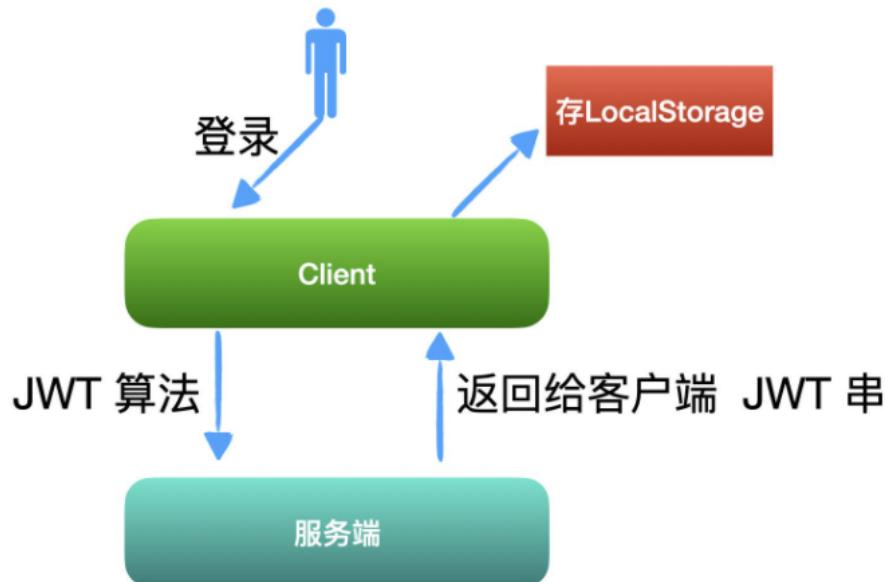
1 HMACSHA256(
2   base64UrlEncode(header) + "." +
3   base64UrlEncode(payload),
4   secret,
5 )

```

注意：

算出签名以后，把 Header、Payload、Signature 三个部分拼成一个字符串，每个部分之间用"点" (.) 分隔，就可以返回给用户。

JWT 的使用方式



流程：

客户端收到服务器返回的 JWT，可以储存在 Cookie 里面，也可以储存在 localStorage。此后，客户端每次与服务器通信，都要带上这个 JWT。你可以把它放在 Cookie 里面自动发送，但是这样不能跨域，所以更好的做法是放在 HTTP 请求的头信息 Authorization 字段里面。

实时效果反馈

1. 客户端收到服务器返回的 JWT 把数据保存到_____。

- A load
- B session
- C localStorage
- D 以上都是错误

2. JWT 签证默认算法_____。

- A HS256

B Base64

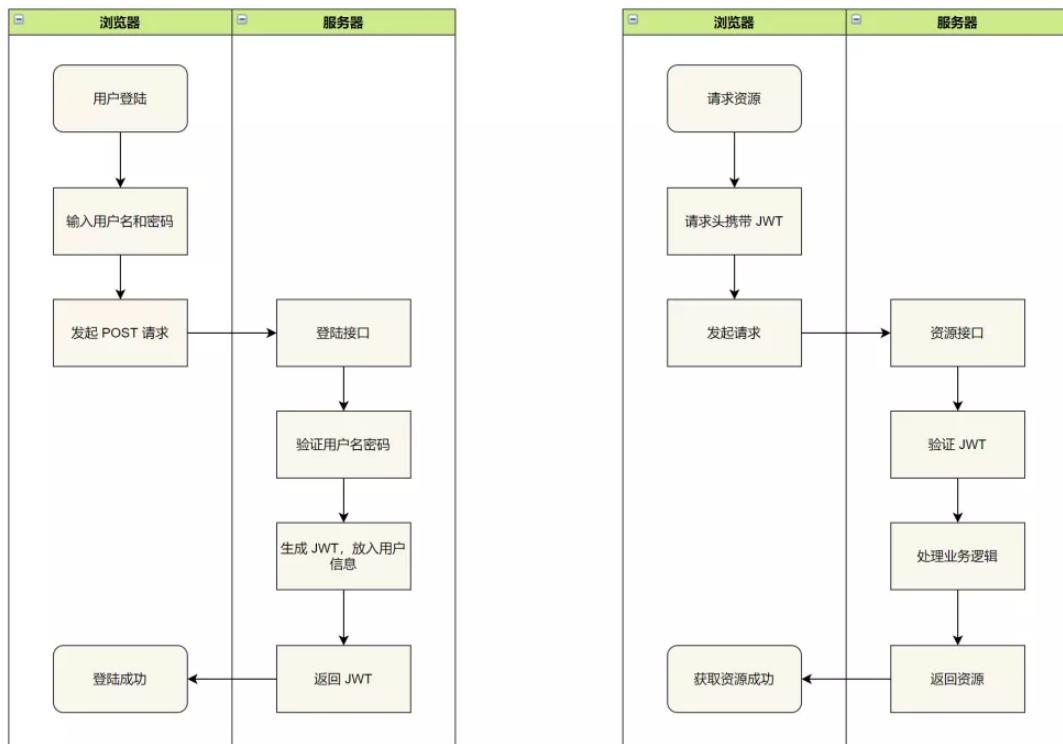
C HS384

D SR384

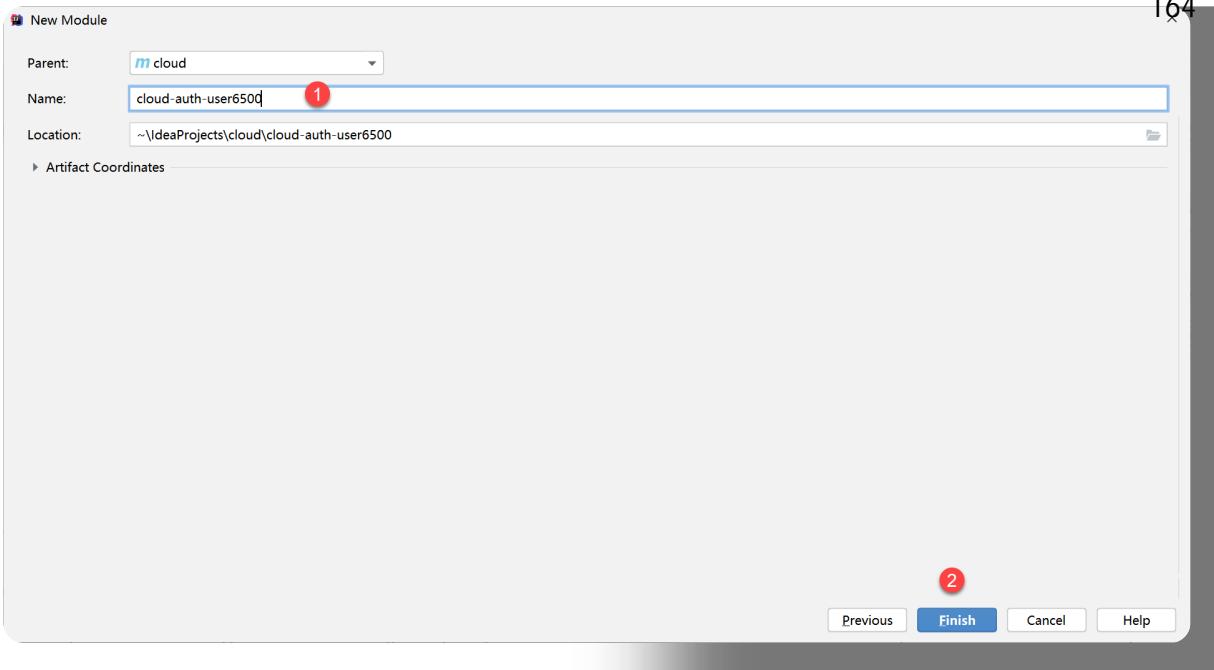
答案

1=>C 2=>A

服务网关Gateway实现用户鉴权_用户微服务



创建cloud-auth-user6500工程



引入POM依赖

```

1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4       <artifactId>spring-boot-starter-
5         web</artifactId>
6       </dependency>
7       <!-- redis -->
8       <dependency>
9         <groupId>org.springframework.boot</groupId>
10        <artifactId>spring-boot-starter-
11          data-redis</artifactId>
12        </dependency>
13        <!-- eureka client 依赖 -->
14        <dependency>
15          <groupId>org.springframework.cloud</groupId>
16        >
17          <artifactId>spring-cloud-
18            starter-netflix-eureka-client</artifactId>

```

```
15 </dependency>
16 <dependency>
17
18 <groupId>org.projectlombok</groupId>
19 <artifactId>lombok</artifactId>
20 <version>1.18.22</version>
21 </dependency>
22 <dependency>
23 <groupId>org.springframework.boot</groupId>
24 <artifactId>spring-boot-starter-
actuator</artifactId>
25 </dependency>
</dependencies>
```

服务网关Gateway实现用户鉴权 JWT工具类

引入JWT依赖

```

1 <dependency>
2   <groupId>com.alibaba</groupId>
3   <artifactId>fastjson</artifactId>
4   <version>1.2.79</version>
5 </dependency>
6 <dependency>
7   <groupId>com.auth0</groupId>
8   <artifactId>java-jwt</artifactId>
9   <version>3.7.0</version>
10 </dependency>

```

创建JWT工具类JWTUtils

```

1 public class JWTUtil {
2     // 秘钥
3     public static final String SECRET_KEY =
4         "erbadagang-123456";
5     // token过期时间
6     public static final long
7     TOKEN_EXPIRE_TIME = 5 * 60 * 1000;
8     // 签发人
9     private static final String ISSUER =
10        "issuer";
11    // 用户名
12    private static final String USER_NAME =
13        "username";
14 }

```

生成签名方法

```

1     public static String token(String
2       username) {

```

```

2     Date now = new Date();
3         //SECRET_KEY是用来加密数据签名秘钥
4         Algorithm algorithm =
5             Algorithm.HMAC256(SECRET_KEY);
6         String token = JWT.create()
7             // 签发人
8             .withIssuer(ISSUER)
9             // 签发时间
10            .withIssuedAt(now)
11            // 过期时间
12            .withExpiresAt(new
13                Date(now.getTime() + TOKEN_EXPIRE_TIME))
14                // 保存权限标记
15                .withClaim(USER_NAME,
16                    username)
17                .sign(algorithm);
18
19
20         log.info("jwt generated user={}", 
21             username);
22         return token;
23     }

```

验证签名

```

1     public static boolean verify(String token)
2     {
3         try {
4             //SECRET_KEY是用来加密数据签名秘钥
5             Algorithm algorithm =
6                 Algorithm.HMAC256(SECRET_KEY);
7                 JWTVerifier verifier =
8                     JWT.require(algorithm)
9                         .withIssuer(ISSUER)

```

```

7         .build();
8             //如果校验有问题会抛出异常。
9             verifier.verify(token);
10            return true;
11        } catch (Exception ex) {
12            ex.printStackTrace();
13        }
14        return false;
15    }

```

测试JWT

```

1 public static void main(String[] args) {
2
3     // 生成Token
4     String token =
5         JWTUtil.token("itbaizhan");
6         System.out.println(token);
7
8     //验证Token
9     boolean verify =
10        JWTUtil.verify("eyJ0eXAiOiJKV1QiLCJhbGciOiJI
11        UzI1NiJ9.eyJpc3MiUiJpc3N1ZXIiLCJleHAiOjE2NDU
12        wNjMyNzYsImhlhdCI6MTY0NTA2Mjk3NiwidXNlcm5hbWU
13        iOiJpdGJhaXpoYW4ifQ.uKKVEMCTw0-e_bPHLyb-
14        JY8CwRU7ciU8vP5B781DY3s");
15        System.out.println(verify);
16    }

```

服务网关Gateway实现用户鉴权_用户服务实现JWT鉴权

编写LoginController类

```

1  @RequestMapping("user")
2  @RestController
3  @Slf4j
4  public class LoginController {
5
6 }
```

编写统一返回实体类

```

1 /**
2  * AuthResult 作用 : 统一返回实体类
3 */
4 @Data
5 @NoArgsConstructor
6 @Builder
7 public class AuthResult {
8
9     // 返回状态码
10    private int code;
11    // 返回描述信息
12    private String msg;
13    // 返回Token签名
14    private String token;
15
16    public AuthResult(int i, String s) {
17        this.code = i;
18        this.msg = s;
19    }
20
21    public AuthResult(int i, String success,
String token) {
```

```

22     this.code = i;
23     this.msg = success;
24     this.token = token;
25 }
26 }
```

编写用户登录接口

```

1 /**
2  * 用户登录
3  * @param username 用户名
4  * @param password 密码
5  * @return
6 */
7 @PostMapping("/login")
8 public AuthResult login(String
username, String password) {
9
10    // TODO 模拟数据库验证用户名密码
11    if ("admin".equals(username) &&
12        "admin".equals(password)) {
13        //生成token
14        String token = JWTUtil.token();
15        return new AuthResult(0,
16            "success", token, refreshToken);
17    } else {
18        return new AuthResult(1001,
19            "username or password error");
20    }
21 }
```

编写验证令牌接口

```

1  /**
2   * 验证令牌
3   * @param token 令牌
4   * @return
5   */
6  @GetMapping("/verify")
7  public AuthResult verify(String token) {
8      // 验证令牌
9      boolean success =
10     JWTUtil.verify(token);
11     if (success){
12         return AuthResult.builder()
13             .code(200).msg("Token未失
14            效").build();
15     }else {
16         return AuthResult.builder()
17             .code(500).msg("Token失
18            效").build();
19     }
20 }
```

YML文件编写

```

1 eureka:
2   client:
3     # 表示是否将自己注册到Eureka Server
4     register-with-eureka: true
5     # 表示是否从Eureka Server获取注册的服务信息
6     fetch-registry: true
7     # Eureka Server地址
8     service-url:
```

```

9     defaultZone:
10    http://eureka7001.com:7001/eureka,http://eur
11    eka7002.com:7002/eureka
12
13    instance:
14      instance-id: user-service
15      prefer-ip-address: true
16
17
18    spring:
19      application:
20        # 设置应用名词
21        name: user-service
22
23
24    server:
25      port: 6510

```

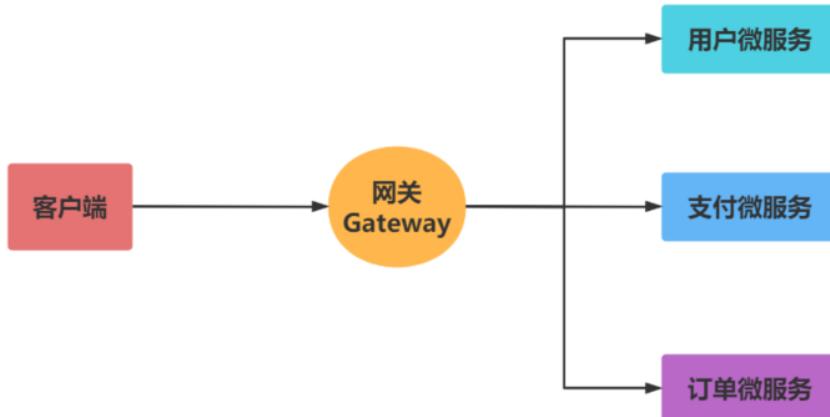
Postman测试

测试登录

The screenshot shows the Postman interface with a POST request to `http://localhost:6510/user/login`. The request body is set to `form-data` (indicated by circle 2). There are two form fields: `username` (containing 'admin') and `password` (containing 'admin') (indicated by circle 3). The 'Send' button is highlighted with circle 1.

KEY	VALUE	DESCRIPTION	Bulk Edit
<input checked="" type="checkbox"/> username	admin		
<input checked="" type="checkbox"/> password	admin		
Key	Value	Description	

服务网关Gateway实现用户鉴权_网关全局过滤器加入JWT 鉴权



配置跳过验证路由

```

1 org:
2   my:
3     jwt:
4       #跳过认证的路由
5       skipAuthUrls:
6         - /user/login
  
```

创建LoginGlobalFilter全局过滤器

```

1 @Data
2 @Slf4j
3 @Component
4 @ConfigurationProperties("org.my.jwt")
5 public class LoginGlobalFilter implements
GlobalFilter, Ordered {
6
7   // 跳过路由数组
8   private String[] skipAuthUrls;
9
10  @Override
  
```

```
11     public Mono<Void>
12         filter(ServerWebExchange exchange,
13             GatewayFilterChain chain) {
14             //获取请求url地址
15             String url =
16                 exchange.getRequest().getURI().getPath();
17
18             //跳过不需要验证的路径
19             if (null != skipAuthUrls &&
20                 isSkipUrl(url)) {
21                 return chain.filter(exchange);
22             }
23
24             //从请求头中取得token
25             String token =
26                 exchange.getRequest().getHeaders().getFirst(
27                     "Authorization");
28             if (StringUtils.isEmpty(token)) {
29                 return
30                     createResponseObj(exchange, 500, "token参数缺
31                     失");
32             }
33
34             //请求中的token是否有效
35             boolean verifyResult =
36                 JWTUtil.verify(token);
37             if (!verifyResult) {
38                 return
39                     createResponseObj(exchange, 500, "token 失效");
40             }
41
42             //如果通过了验证，那么就将token放入到header中
43             Map<String, String> map = new HashMap<String, String>();
44             map.put("token", token);
45             exchange.getHeaders().setAll(map);
46             return Mono.empty();
47         }
48     }
```

```

32         //如果各种判断都通过，执行chain上的其他业
33         //务逻辑
34
35     }
36
37     @Override
38     public int getOrder() {
39         return 0;
40     }
41
42
43     /**
44      * 判断当前访问的url是否开头URI是在配置的忽略
45      * url列表中
46      *
47      * @param url
48      * @return
49      */
50     public boolean isSkipUrl(String url) {
51         for (String skipAuthUrl :
52             skipAuthUrls) {
53             if (url.startsWith(skipAuthUrl))
54             {
55                 return true;
56             }
57         }
58         return false;
59     }
60
61     // 组装返回数据

```

```
59     private Mono<Void>
60         createResponseObj(ServerWebExchange
61             exchange, Integer code, String message){
62             ServerHttpResponse response =
63                 exchange.getResponse();
64                 // 设置响应状态码200
65
66             response.setStatus(HttpStatus.OK);
67                 // 设置响应头
68                 response.getHeaders().add("Content-
69                     Type", "application/json;charset=UTF-8");
70                     // 创建响应对象
71                     Response res = new Response(code,
72                         message);
73                     // 把对象转成字符串
74                     byte[] responseByte =
75                         JSONObject.toJSONString(res).toString().getBytes(StandardCharsets.UTF_8);
76                     DataBuffer buffer =
77                         response.bufferFactory().wrap(responseByte);
78                     return
79                         response.writeWith(Flux.just(buffer));
80                 }
81             }
82 }
```

测试

POST http://localhost:... ● | POST http://localhost:... ● | GET http://localhost:... ● | GET http://localhost:... ● | GET http://localhost:... ● | + | ... | No Environment

http://localhost:9527/payment/lb

GET http://localhost:9527/payment/lb

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Headers 6 hidden

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Authorization	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJpc...				
<input type="checkbox"/> user-name	wcc				
Key	Value	Description			

Body Cookies Headers (2) Test Results

Status: 200 OK Time: 1m 0.18 s Size: 130 B Save Response

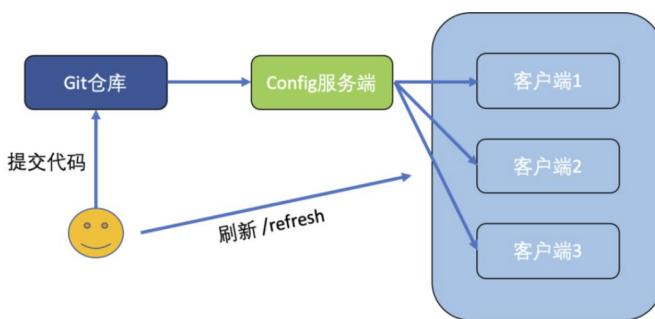
Pretty Raw Preview Visualize JSON

```

1
2 "code": 500,
3 "message": "token 失效"
4

```

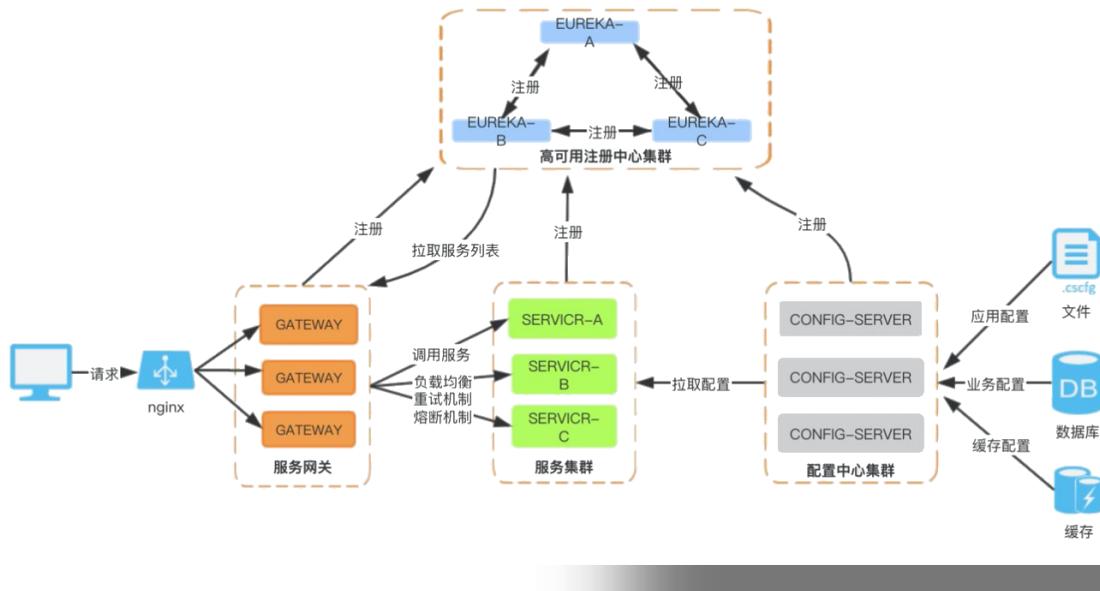
分布式配置中心_Spring Cloud Config



Config项目是一个解决分布
式系统的配置管理方案

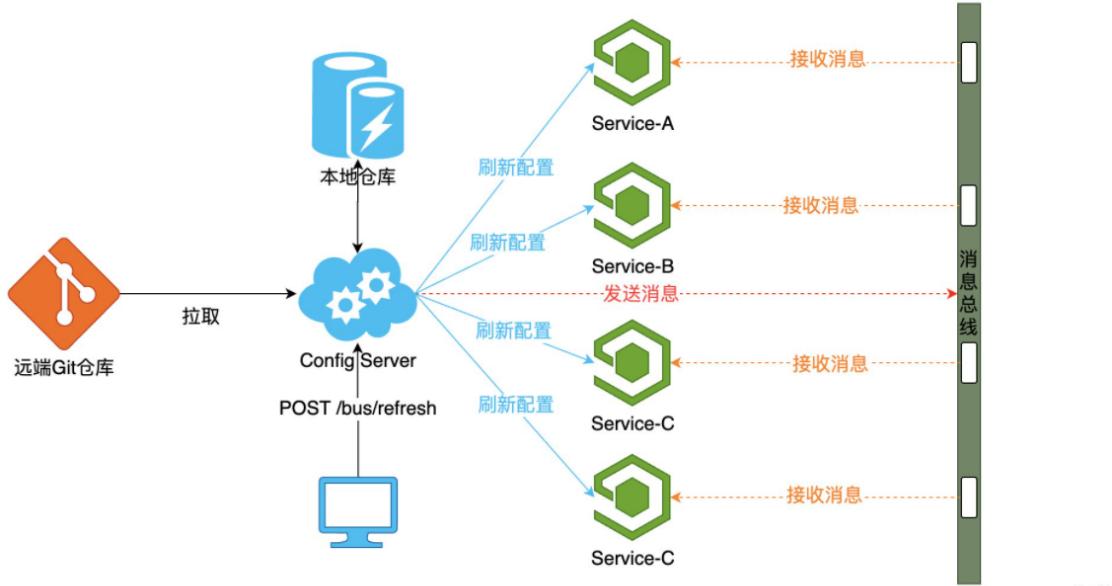
分布式系统面临问题

在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。



什么是Spring Cloud Config

Spring Cloud Config项目是一个解决分布式系统的配置管理方案。



整个结构包括三个部分，客户端（各个微服务应用），服务端（中介者），配置仓库（可以是本地文件系统或者远端仓库，包括git,svn等）。

- 配置仓库中放置各个配置文件 (.yml 或者.properties)
- 服务端指定配置文件存放的位置
- 客户端指定配置文件的名称

Config能干什么

- 提供服务端和客户端支持
- 集中管理各环境的配置文件
- 配置文件修改之后，可以快速的生效
- 可以进行版本管理
- 支持大的并发查询
- 支持各种语言

对比主流配置中心

开源的配置中心有很多，比如，360的QConf、淘宝的nacos、携程的Apollo等。在Spring Cloud中，有分布式配置中心组件spring cloud config，它功能全面、强大，可以无缝地和Spring体系相结合，使用方便简单。

实时效果反馈

1. Spring Cloud Config项目是一个解决分布式系统的__问题。

- A 服务注册发现
- B 负载均衡
- C 服务熔断
- D 配置管理

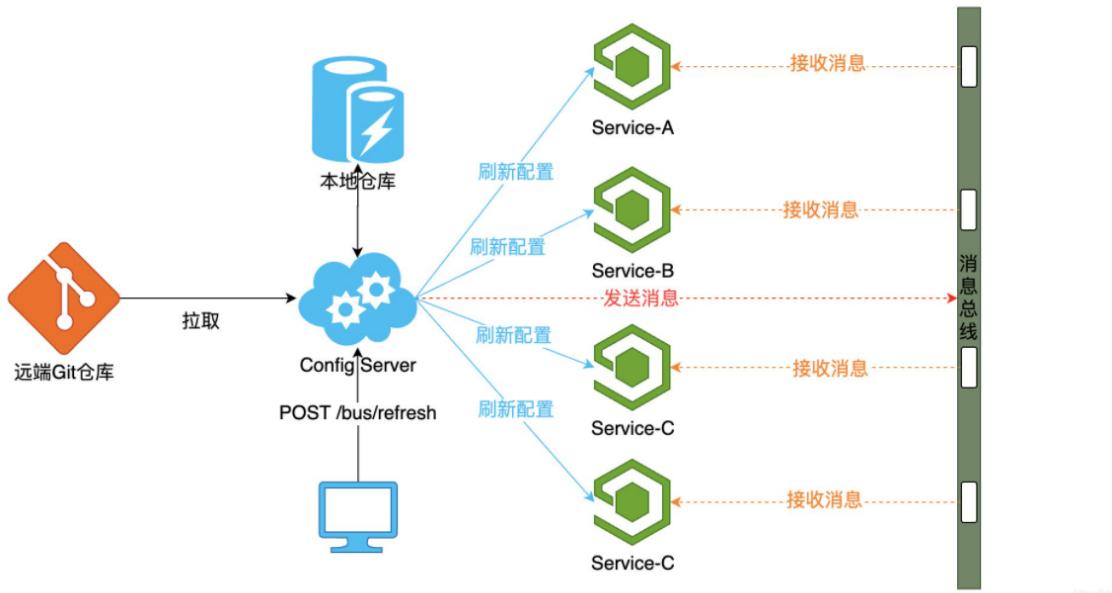
2. Spring Cloud Config项目包含了__和__两个部分。

- A client,service
- B client,server
- C master,slave
- D 以上都错误

答案

1=>D 2=>B

分布式配置中心_Config配置总控中心搭建



服务端开发

服务端开发最主要的任务是配置从哪里读取对应的配置文件，我们将配置从Git仓库读取配置文件。

在码云新建一个名为springcloud-config的新的仓库

新建仓库

在其他网站已经有仓库了吗？[点击导入](#)

仓库名称 * ✓

cloud-config 1

归属 路径 * ✓

 kalista / cloud-config

仓库地址: <https://gitee.com/WCCRegistered/cloud-config>

仓库介绍 0/200

用简短的语言来描述一下吧

开源 (所有人可见) ②

私有 (仅仓库成员可见)

企业内部开源 (仅企业成员可见) ②

初始化仓库 (设置语言、.gitignore、开源许可证)

设置模板 (添加 Readme、Issue、Pull Request 模板文件)

选择分支模型 (仓库创建后将根据所选模型创建分支)

创建 2

项目开源

The screenshot shows the GitHub repository interface for 'kalista / cloud-config'. The top navigation bar includes links for '代码', 'Issues' (highlighted with a red circle), 'Pull Requests', 'Wiki', '统计', 'DevOps', '服务', '设置', and '贡献者' (with a red circle). The repository name 'kalista / cloud-config' is at the top left. The top right corner shows status icons for 'Watching' (1), 'Star' (0), 'Fork' (0). A red circle highlights the 'Issues' tab. The main content area shows a message: '你当前开源项目尚未选择许可证 (LICENSE)，点此选择并创建开源许可证'. Below this are sections for '分支' (branches), '提交' (commits), and '文件' (files). The 'Issues' tab is selected, showing a list of issues with columns for '状态' (Status), '标题' (Title), '最后更新' (Last updated), and '操作' (Actions). One issue is highlighted with a red circle. To the right, there are sections for '简介' (Introduction), '仓库公开须知' (Repository public notice), '发行版' (Release), and '贡献者' (Contributors). The '贡献者' section has a red circle highlighting the '贡献者 (1)' link.

新建模块cloud-config-server3344

New Module

Parent:

Name:

Location:

▶ Artifact Coordinates

仓库中新建3个文件

config-dev.yml

```

1 config:
2   info: "master branch,config-dev.yml
  version=1"

```

config-test.yml

```

1 config:
2   info: "master branch,config-test.yml
  version=1"

```

config-prod.yml

```

1 config:
2   info: "master branch,config-prod.yml
  version=1"

```

POM文件引入依赖

```

1 <dependencies>
2   
3   <dependency>
4     <groupId>org.springframework.cloud</groupId>
5     <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
6   </dependency>
7   <!-- 引入 config 依赖--&gt;
8   &lt;dependency&gt;
9     &lt;groupId&gt;org.springframework.cloud&lt;/groupId&gt;
10    &lt;artifactId&gt;spring-cloud-config-server&lt;/artifactId&gt;
11    &lt;/dependency&gt;
12    <!-- 引入 web 依赖--&gt;
</pre>

```

```

13 <dependency>
14
15   <groupId>org.springframework.boot</groupId>
16     <artifactId>spring-boot-starter-
17       web</artifactId>
18     </dependency>
19   <dependency>
20
21     <groupId>org.projectlombok</groupId>
22       <artifactId>lombok</artifactId>
23         <version>1.18.22</version>
24       </dependency>
25   </dependencies>

```

编写YML文件

新增application.yml

```

1 server:
2   port: 3344
3 spring:
4   application:
5     name: cloud-config-center
6   cloud:
7     config:
8       server:
9         git:
10          uri:
11            https://gitee.com/wCCRegistered/cloud-
12              config.git
13          search-paths:
14            - cloud-config
15        label: master

```

```

14 eureka:
15   client:
16     # 表示是否将自己注册到Eureka Server
17     register-with-eureka: true
18     # 表示是否从Eureka Server获取注册的服务信息
19     fetch-registry: true
20     # Eureka Server地址
21     service-url:
22       defaultZone:
23       http://eureka7001.com:7001/eureka,http://eur
24       eka7002.com:7002/eureka
25     instance:
26       instance-id: cloud-config-center
27       prefer-ip-address: true

```

编写主启动类

```

1 package com.itbaizhan;
2
3 import lombok.extern.slf4j.Slf4j;
4 import
5 org.springframework.boot.SpringApplication;
6 import
7 org.springframework.boot.autoconfigure.Sprin
8 gBootApplication;
9 import
10 org.springframework.cloud.config.server.Enab
11 leConfigServer;
12
13 /**
14  * 主启动类
15  */
16 @Slf4j

```

```

12 @SpringBootApplication
13 @EnableConfigServer
14 public class ConfigCenterMain3344 {
15     public static void main(String[] args) {
16
17
18         SpringApplication.run(ConfigCenterMain3344.
19             class, args);
20         log.info("***** 配置中心服务启动
21 成功 *****");
22     }
23 }
```

测试通过config微服务是否可以从码云上获取配置

<http://localhost:3344/master/config-dev.yml>

实时效果反馈

1. Spring Cloud Config项目配置具体存放在_____。

- A redis
- B mysql
- C git
- D svn

答案

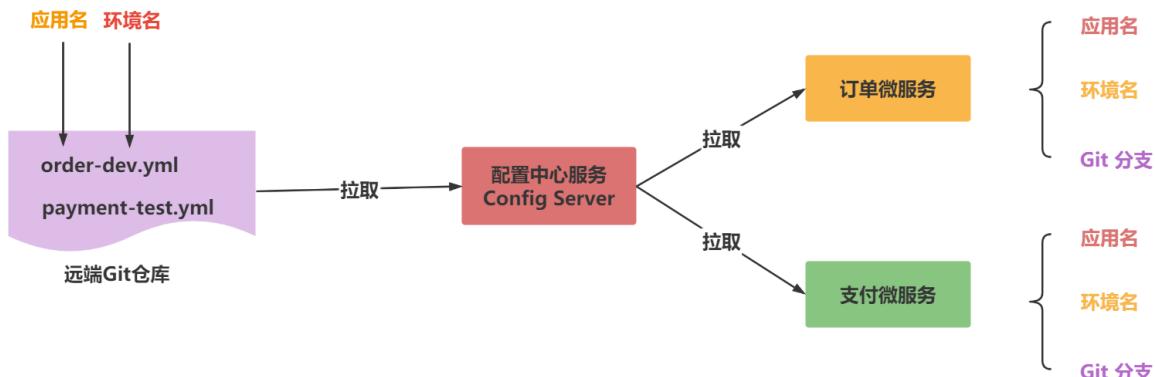
1=>C

Config配置 读取规则



Config支持的请求的参数规则

- /{application}/{profile}[/{label}]
- /{application}-{profile}.yml
- /{label}/{application}-{profile}.yml
- /{application}-{profile}.properties
- /{label}/{application}-{profile}.properties



注意：

- {application} 就是应用名称，对应到配置文件上来，就是配置文件的名称部分，例如我上面创建的配置文件。
- {profile} 就是配置文件的版本，我们的项目有开发版本、测试环境版本、生产环境版本，对应到配置文件上来就是以 application-{profile}.yml 加以区分，例如 application-dev.yml、application-test.yml、application-prod.yml。
- {label} 表示 git 分支，默认是 master 分支，如果项目是以分支做区分也是可以的，那就可以通过不同的 label 来控制访问不同的配置文件了。

最推荐使用方式

/ { 分支名 } / { 应用名 } - { 环境名 }.yml

实时效果反馈

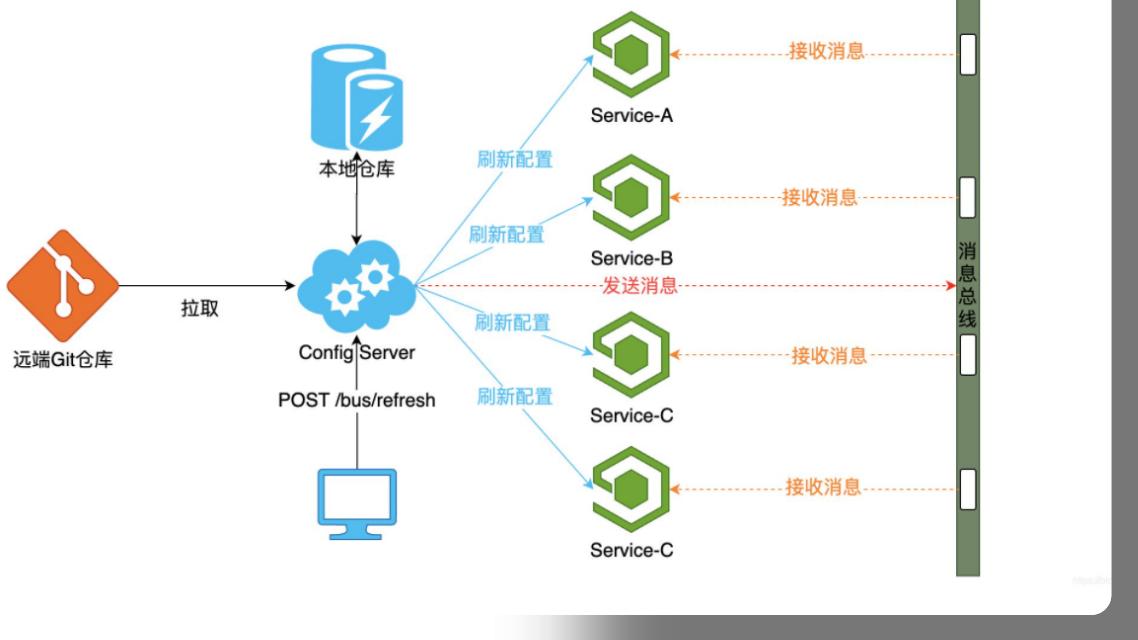
1. Spring Cloud Config客户端在指定配置文件时profile表示_含义。

- A 应用名称
- B 配置文件的版本
- C git 分支
- D 以上都是错误

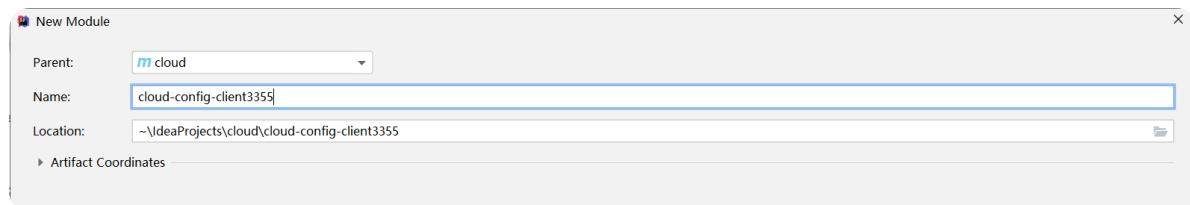
答案

1=>D

分布式配置中心_Config客户端配置与测试



新建cloud-config-client3355



POM文件添加依赖

```

1 <dependencies>
2     
3     <dependency>
4
5         <groupId>org.springframework.cloud</groupId>
6
7             <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
8
9             </dependency>
10            <!-- 引入 config 依赖--&gt;
11            &lt;dependency&gt;
12
13                &lt;groupId&gt;org.springframework.cloud&lt;/groupId&gt;
14
15            &lt;/dependency&gt;
</pre>

```

```

10          <artifactId>spring-cloud-
11          starter-config</artifactId>
12          </dependency>
13          <!-- 引入 web 依赖-->
14          <dependency>
15
16              <groupId>org.springframework.boot</groupId>
17                  <artifactId>spring-boot-starter-
18                  web</artifactId>
19                  </dependency>
20                  <dependency>
21
22                      <groupId>org.projectlombok</groupId>
23                          <artifactId>lombok</artifactId>
24                          <version>1.18.22</version>
25                          </dependency>
26                  </dependencies>

```

新增bootstrap.yml配置

```

1 spring:
2     application:
3         name: config-client
4     cloud:
5         config:
6             label: master
7             name: config
8             profile: dev
9             # 上述综合 master分支上config-dev.yml的
10            # 配置文件
11            # http://localhost:3344/master/config-
12            # dev.yml
13            uri: http://localhost:3344

```

```

12 server:
13   port: 3355
14 eureka:
15   client:
16     # Eureka Server地址
17     service-url:
18       defaultZone:
19         http://eureka7001.com:7001/eureka,http://eure
20         ka7002.com:7002/eureka
21       instance:
22         instance-id: cloud-config-center
23         prefer-ip-address: true

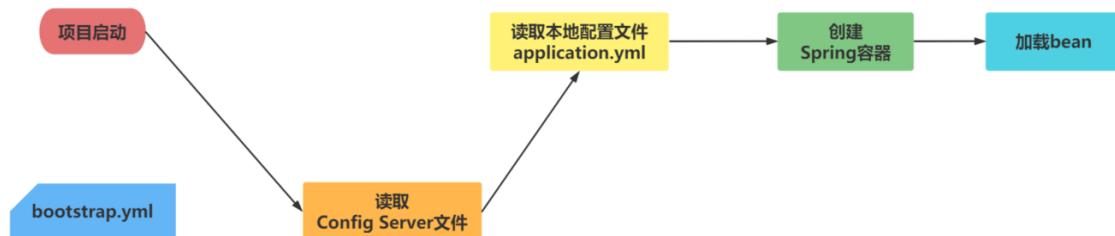
```

注意:

- `application.yml`: 是用户级的资源配置项
- `bootstrap.yml`: 是系统级的, 优先级更加高

`bootstrap.yml`优先级高于`application.yml`

为什么要引入bootstrap



注意:

要将Client模块下的`application.yml`文件改为`bootstrap.yml`, 这是很关键的,

因为`bootstrap.yml`是比`application.yml`先加载的。

`bootstrap.yml`优先级高于`application.yml`

必抗指南

错误提示：

```

1 Application failed to start due to an
exception
2 org.springframework.cloud.config.client.Confi
gServerConfigDataMissingEnvironmentPostProces
sor$ImportException: No spring.config.import
set

```

诞生原因：

springcloud2020 版本把Bootstrap被默认禁用，同时
spring.config.import加入了对解密的支持。

解决办法

```

1 <dependency>
2
3     <groupId>org.springframework.cloud</groupId>
4         <artifactId>spring-cloud-starter-
bootstrap</artifactId>
5     </dependency>

```

业务类controller编写

```

1 @RestController
2 public class ConfigClientController {
3
4     @value("${config.info}")
5     private String configInfo;
6
7     /**
8      * 获取配置

```

```
9     * @return
10    */
11    @GetMapping("/configinfo")
12    public String getConfigInfo(){
13        return configInfo;
14    }
15
16 }
```

启动config配置中心3355测试

<http://localhost:3355/configinfo>



难道每次运维修改配置文件，客户端都需要重启？



分布式配置的动态刷新问题

修改码云上的配置文件内容做调整。

问题：

- 刷新3344，发现ConfigServer配置中心立刻响应
- 刷新3355，发现ConfigServer客户端没有任何响应
- 3355没有变化除非自己重启或者重新加载
- 难道每次运维修改配置文件，客户端都需要重启

cloud-config-client3355工程引入actuator监控

```
1 <dependency>
2
3     <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-
5             actuator</artifactId>
6     </dependency>
```

修改bootstrap.yml暴露监控端口

```

1 management:
2   endpoints:
3     web:
4       exposure:
5         include: "*"

```

业务类Controller修改 加入注解@RefreshScope

```

1 @RefreshScope
2 @RestController
3 public class ConfigClientController {
4
5   @Value("${config.info}")
6   private String configInfo;
7
8   /**
9    * 获取配置
10   * @return
11   */
12   @GetMapping("/configinfo")
13   public String getConfigInfo(){
14     return configInfo;
15   }
16
17 }

```

手动刷新配置

<http://localhost:3355/actuator/refresh>

The screenshot shows the Postman interface. At the top, it says "localhost:3355/actuator/refresh". Below that, there's a dropdown menu set to "POST" and a red box highlights the URL field. The "Body" tab is selected. In the "Query Params" section, there is one entry: "Key" is "Key" and "Value" is "Value". The "Description" column has "Description". On the right, there are "Save" and "Send" buttons. Below the main interface, a status bar shows "Status: 200 OK Time: 4.29 s Size: 231 B Save Response". The response body is displayed in JSON format:

```

1
2   "config.client.version",
3   "config.info"
4

```

注意：

必修是post请求。

实时效果反馈

1. Spring Cloud Config实现动态刷新功能需要引入____依赖。

- A actuator
- B eureka
- C gateway
- D config

2. Spring Cloud Config实现动态刷新功能时进行手动刷新配置请求地址必须____请求。

- A get
- B post
- C put

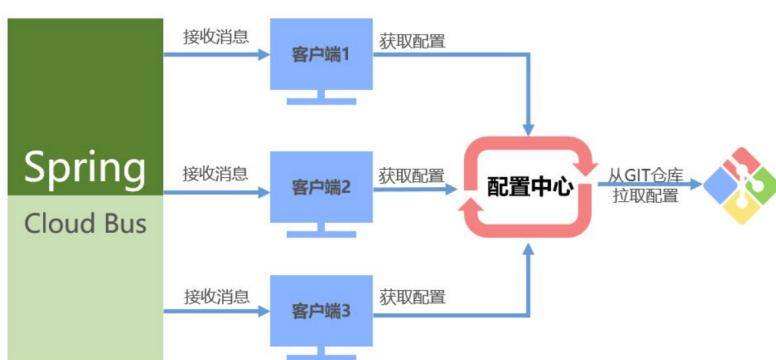
D

delete

答案

1=>A 2=>B

消息总线_什么是Spring Cloud Bus

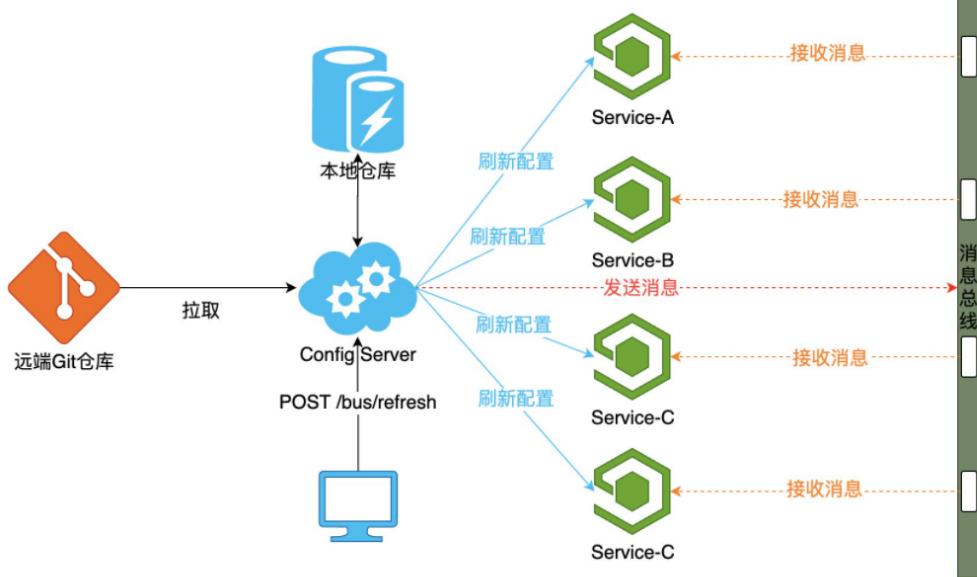


Config配置中心遇到的问题

当我们在更新码云上面的配置以后，如果想要获取到最新的配置，需要手动刷新机制每次提交代码发送请求来刷新客户端，客户端越来越多的时候，需要每个客户端都执行一遍，这种方案就不太适合了。Spring Cloud作为微服务架构的一个综合解决方案，也提供了对应的解决方案Spring Cloud Bus，即消息总线。

什么是Spring Cloud Bus

Spring Cloud Bus通过建立多个应用之间的通信频道，管理和传播应用间的消息，从技术角度来说，应用了AMQP消息代理作为通道，通过MQ的广播机制实现消息的发送和接收。Bus支持两种消息代理：RabbitMQ和Kafka。



Spring Cloud Bus做配置更新的步骤:

- ① 修改配置文件，提交代码触发post给客户端A发送bus/refresh
- ② 客户端A接收到请求从Server端更新配置并且发送给Spring Cloud Bus
- ③ Spring Cloud bus接到消息并通知给其它客户端
- ④ 其它客户端接收到通知，请求Server端获取最新配置
- ⑤ 全部客户端均获取到最新的配置

实时效果反馈

1. Spring Cloud Bus组件主要解决____问题。

- A 注册中心
- B 负载均衡
- C 手动刷新
- D 服务熔断

2. Spring Cloud Bus组件支持____消息代理。

- A RabbitMQ
- B ActiveMQ



- D 以上都是错误

答案

1=>C 2=>A

消息总线_Docker安装RabbitMQ



下载镜像

```
1 docker pull docker.io/macintoshplus/rabbitmq-management
```

启动容器

```
1 docker run -d --name rabbitmq -e
RABBITMQ_DEFAULT_USER=guest -e
RABBITMQ_DEFAULT_PASS=guest -p 15672:15672 -p
5672:5672 docker.io/macintoshplus/rabbitmq-
management
```

参数:

- -d: 守护进行运行
- --name: 容器名字
- -e: 配置用户名和密码
- -p: 设置端口号

测试

<http://192.168.66.100:15672>

The screenshot shows the RabbitMQ Management Console's Overview page. At the top right, there is a red box around the user information 'User jia' and 'Log out'. Below that, another red box highlights the 'Message rates' section. A large red arrow points from the question text to this section. The 'Message rates' section displays metrics for the last minute, including 'Currently idle' and 'Global counts'.

Connections	Channels	Exchanges	Queues	Consumers
0	0	7	0	0

实时效果反馈

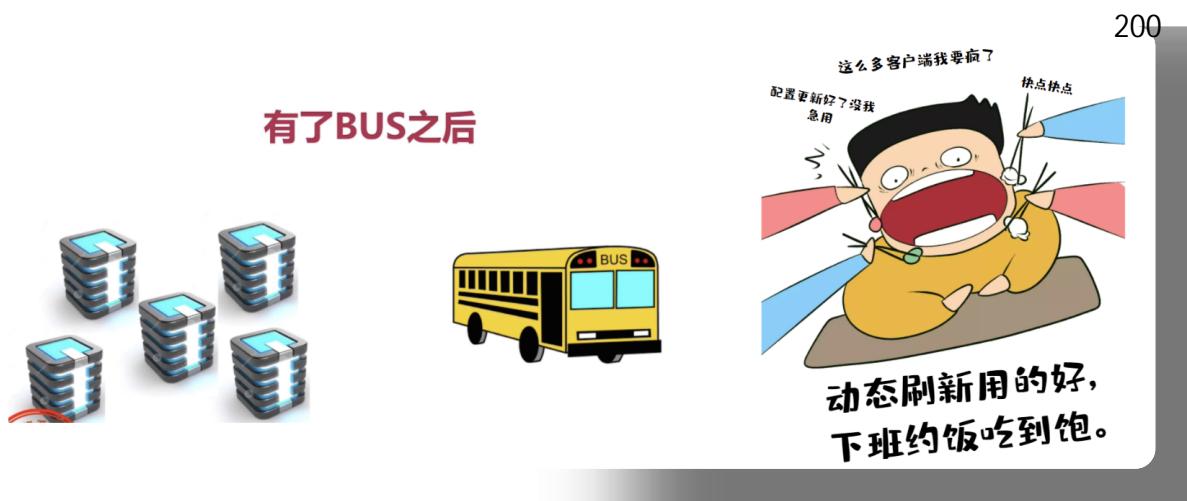
1.RabbitMQ消息中间件可视化服务端口号是_____。

- A 5672
- B 15672
- C 3306
- D 8080

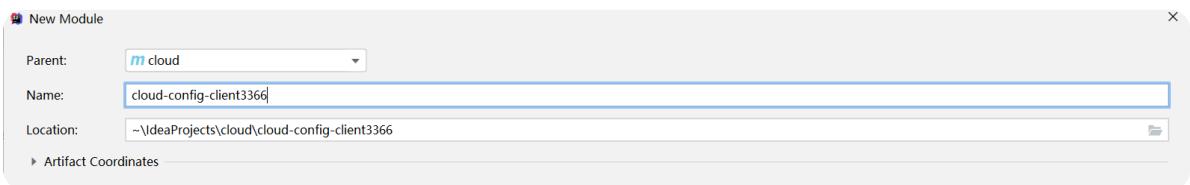
答案

1=>B

消息总线_Bus动态刷新全局广播



新增cloud-config-client3366工程



POM文件新增依赖

```
1 <dependencies>
2     <!-- 引入Eureka client依赖 -->
3     <dependency>
4
5         <groupId>org.springframework.cloud</groupId>
6     >
7         <artifactId>spring-cloud-
8             starter-netflix-eureka-client</artifactId>
9
10        </dependency>
11        <!-- 引入 config 依赖-->
12        <dependency>
13
14            <groupId>org.springframework.cloud</groupId>
15        >
16            <artifactId>spring-cloud-
17                starter-config</artifactId>
18
19            </dependency>
20            <!-- 引入 web 依赖-->
21            <dependency>
```

```

14 <groupId>org.springframework.boot</groupId>
15     <artifactId>spring-boot-starter-
16     web</artifactId>
17     </dependency>
18     <dependency>
19
20         <groupId>org.springframework.boot</groupId>
21             <artifactId>spring-boot-starter-
22             actuator</artifactId>
23             </dependency>
24             <dependency>
25
26             <groupId>org.springframework.cloud</groupId>
27         >
28             <artifactId>spring-cloud-
29             starter-bootstrap</artifactId>
30             </dependency>
31             <dependency>
32
33             <groupId>org.projectlombok</groupId>
34                 <artifactId>lombok</artifactId>
35                 <version>1.18.22</version>
36                 </dependency>
37             </dependencies>

```

新增yml配置文件bootstrap.yml

```

1 spring:
2     application:
3         name: config-client
4     cloud:
5         config:

```

```

6      label: master
7      name: config
8      profile: dev
9          # 上述综合 master分支上config-dev.yml的
配置文件
10         # http://localhost:3344/master/config-
dve.yml
11         uri: http://localhost:3344
12 server:
13     port: 3366
14 eureka:
15     client:
16         # 表示是否将自己注册到Eureka Server
17         register-with-eureka: true
18         # 表示是否从Eureka Server获取注册的服务信息
19         fetch-registry: true
20         # Eureka Server地址
21         service-url:
22             defaultZone:
23                 http://eureka7001.com:7001/eureka,http://eur
eka7002.com:7002/eureka
24             instance:
25                 instance-id: cloud-config-center
26                 prefer-ip-address: true
27 management:
28     endpoints:
29         web:
30             exposure:
31                 include: "*"

```

主启动类

```

1  /**
2  * 主启动类
3  */
4 @Slf4j
5 @EnableEurekaClient
6 @SpringBootApplication
7 public class ConfigClient3366 {
8     public static void main(String[] args) {
9
10        SpringApplication.run(ConfigClient3366.class,
11        args);
12        log.info("*****"
13        "ConfigClient3366 启动成功 *****");
14    }
15 }

```

业务controller

```

1 @RefreshScope
2 @RestController
3 public class ConfigClientController {
4
5     @Value("${config.info}")
6     private String configInfo;
7
8     /**
9      * 获取配置
10     * @return
11     */
12     @GetMapping("/configinfo")

```

```

13     public String getConfigInfo(){
14         return configInfo;
15     }
16
17 }
```

注意：

利用消息总线触发一个服务端ConfigServer的/bus/refresh端点,而刷新所有客户端的配置。

给3344,3355,3366工程添加消息总线支持

POM文件添加依赖

```

1 <dependency>
2
3     <groupId>org.springframework.cloud</groupId>
4         <artifactId>spring-cloud-starter-bus-
5             amqp</artifactId>
6     </dependency>
```

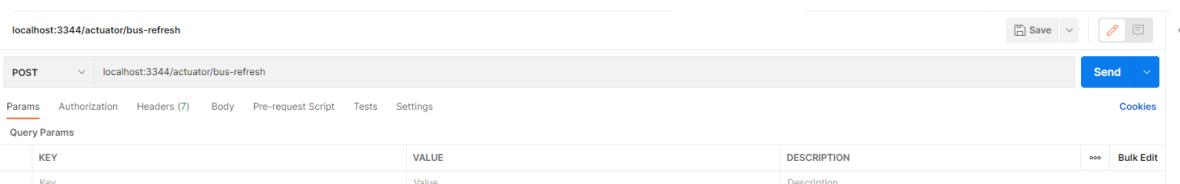
YML文件添加配置

```

1 rabbitmq:
2     host: 192.168.66.101
3     port: 5672
4     username: guest
5     password: guest
```

一次发送处生效

<http://localhost:3355/actuator/busrefresh>



Bus动态刷新定点通知

指具体某个实例生效而不是全部

示例

```
1 | http://localhost:3355/actuator/busrefresh/config-client:3355
```

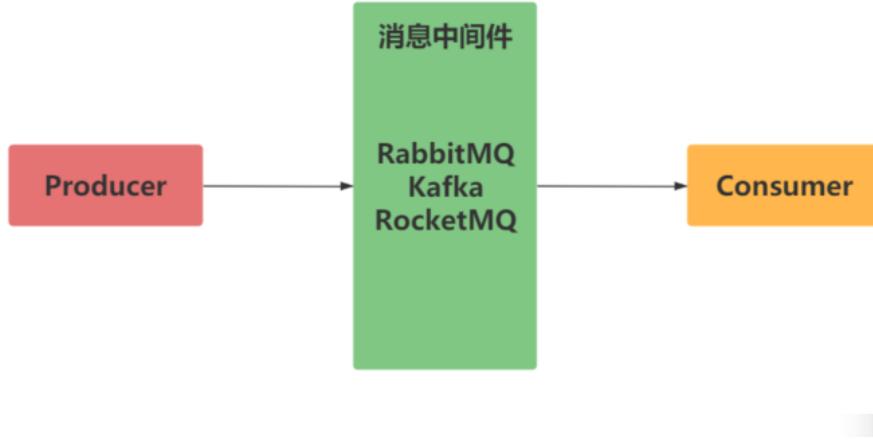
消息驱动_什么是Spring Cloud Stream



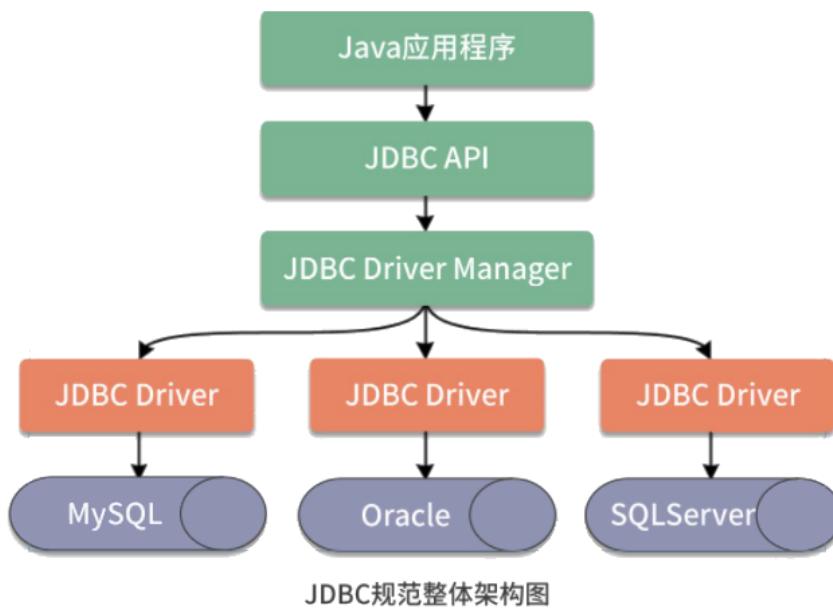
屏蔽底层消息中间件的
差异
嘿嘿！暗爽中

为什么使用Spring Cloud Stream

流行的消息中间件过多，有可能一个工程中使用MQ，比方说我们用到了RabbitMQ和Kafka，由于这两个消息中间件的架构上的不同，像RabbitMQ有exchange，kafka有Topic，partitions分区，这些中间件的差异性导致我们实际项目开发给我们造成了一定的困扰，我们如果用了两个消息队列的其中一种，后面的业务需求，我想往另外一种消息队列进行迁移，这时候无疑就是一个灾难性的，一大堆东西都要重新推倒重新做，因为它跟我们的系统耦合了，这时候springcloud Stream给我们提供了一种解耦合的方式。



我们之前使用的数据库链接工具

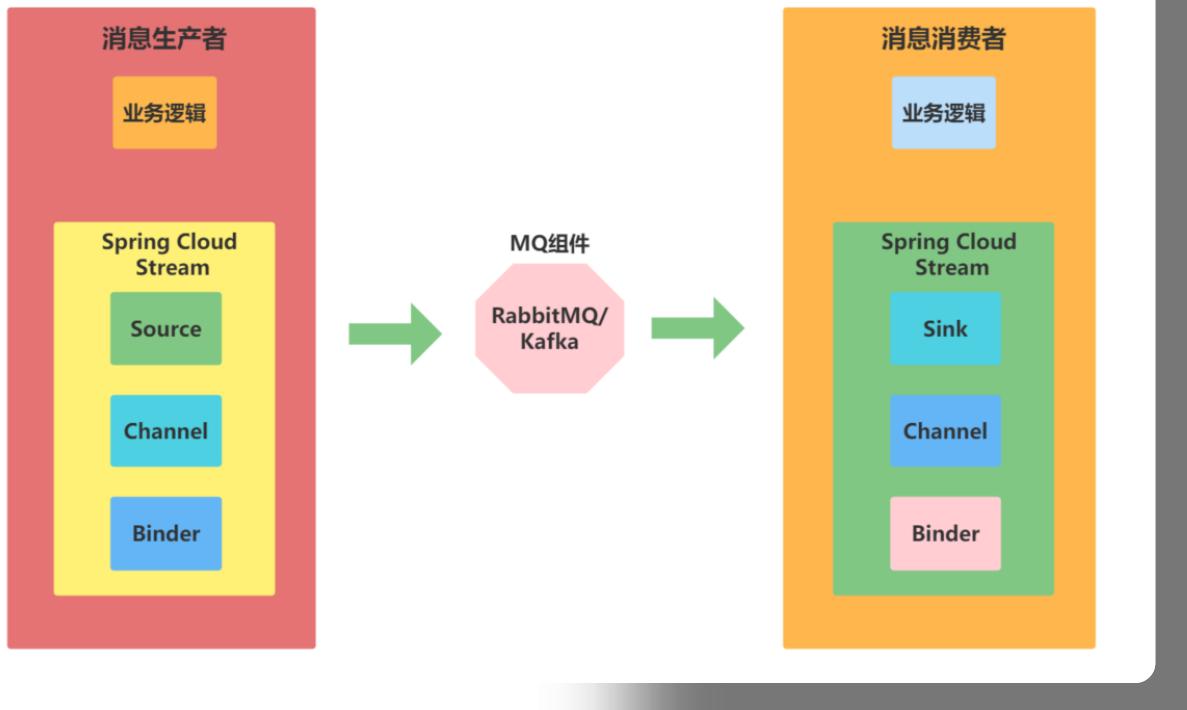


注意：

Stream解决了开发人员无感知的使用消息中间件的问题，因为Stream对消息中间件的进一步封装，可以做到代码层面对中间件的无感知，甚至于动态的切换中间件，使得微服务开发的高度解耦，服务可以关注更多自己的业务流程。

什么是Spring Cloud Stream

官方定义Spring Cloud Stream 是一个构建消息驱动微服务的框架。实现了一套轻量级的消息驱动的微服务框架。通过使用Spring Cloud Stream,可以有效简化开发人员对消息中间件的使用复杂度，让系统开发人员可以有更多的精力关注于核心业务逻辑的处理。



实时效果反馈

1. Spring Cloud Stream 组件主要解决____问题。

- A 注册中心
- B 负载均衡
- C 服务熔断
- D 屏蔽底层消息中间件的差异

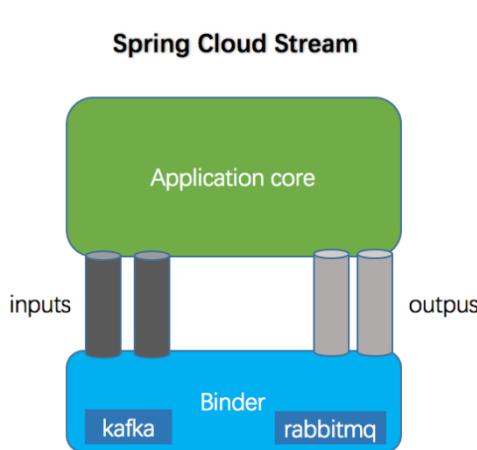
2. Spring Cloud Stream 是一个构建____微服务的框架。

- A 服务网关
- B 消息代理
- C 消息驱动
- D 以上都是错误

答案

1=>D 2=>C

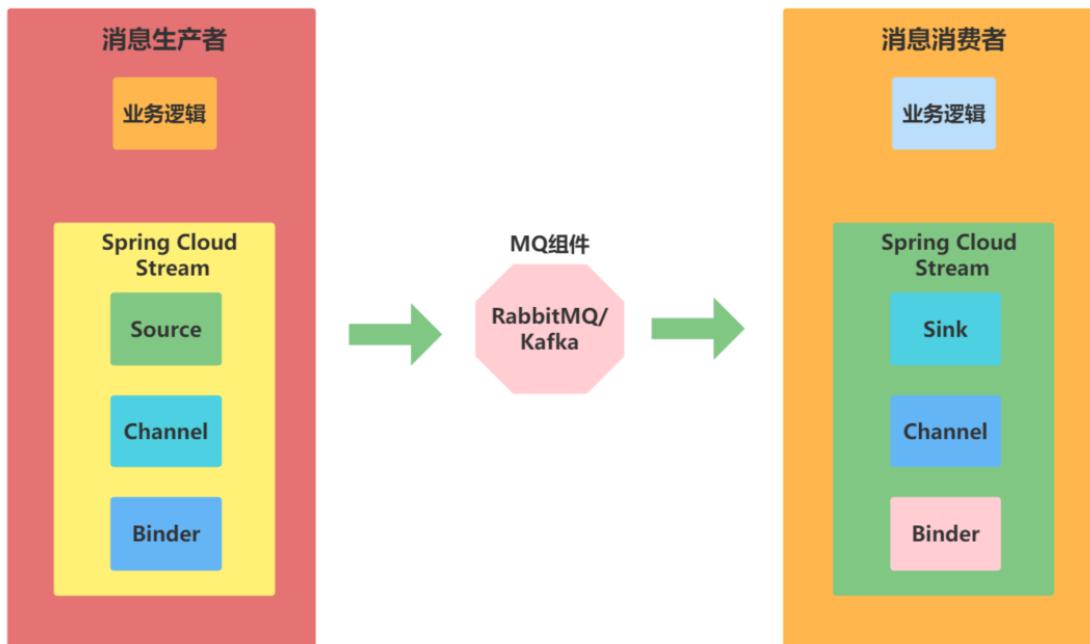
消息驱动_SpringCloud Stream核心概念



Binder 方便连接中间件，屏蔽差异

绑定器

Binder 绑定器是Spring Cloud Stream中一个非常重要的概念。



注意：

- Source: 当需要发送消息时, 我们就需要通过Source.java, 它会把我们所要发送的消息进行序列化 (默认转换成JSON格式字符串), 然后将这些数据发送到channel 中;
- Sink: 当我们需要监听消息时就需要通过Sink.java, 它负责从消息通道中获取消息, 并将消息反序列化成消息对象, 然后交给具体的消息监听处理;
- Channel: 通常我们向消息中间件发送消息或者监听消息时需要指定主题 (Topic) 和消息队列名称, 一旦我们需要变更主题的时候就需要修改消息发送或消息监听的代码。通过Channel对象, 我们的业务代码只需要对应Channel就可以了, 具体这个Channel对应的是哪个主题, 可以在配置文件中来指定, 这样当主题变更的时候我们就不用对代码做任何修改, 从而实现了与具体消息中间件的解耦;
- Binder: 通过不同的 Binder可以实现与不同的消息中间件整合,Binder提供统一的消息收发接口, 从而使得我们可以根据实际需要部署不同的消息中间件, 或者根据实际生产中所部署的消息中间件来调整我们的配置。

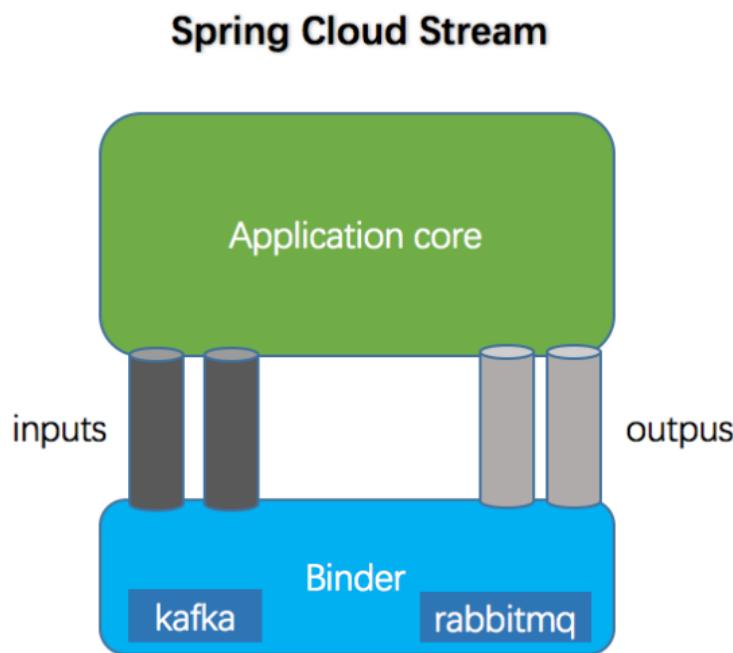
Message

生产者/消费者之间靠消息媒介传递消息内容

消息通道MessageChannel

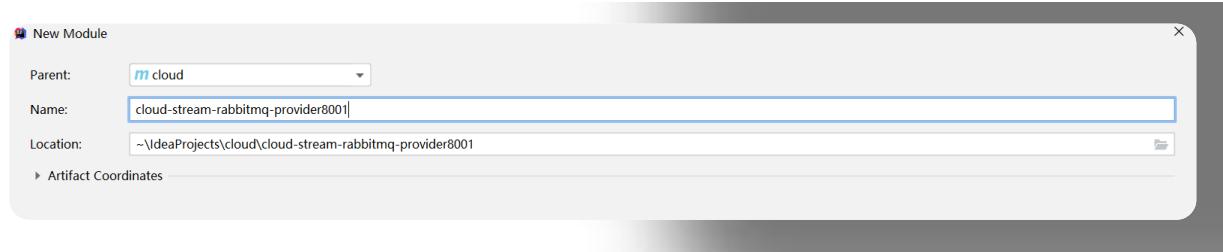
消息必须走特定的通道。

常用注解



消息驱动_入门案例之消息生产者

新建Module工程cloud-stream-rabbitmq-provider8001



POM添加依赖

要使用RabbitMQ绑定器，可以通过使用以下Maven坐标将其添加到 Spring Cloud Stream应用程序中

```

1 <dependency>
2
3   <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-stream-binder-
5       rabbit</artifactId>
6   </dependency>

```

或者

```

1 <dependency>
2
3   <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-stream-
5       rabbit</artifactId>
6   </dependency>

```

完整依赖如下：

```

1 <dependencies>
2   <dependency>

```

```

3   <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-
5       starter-stream-rabbit</artifactId>
6       </dependency>
7       <!-- 引入Eureka client依赖 -->
8       <dependency>
9
10      <groupId>org.springframework.cloud</groupId>
11    <artifactId>spring-cloud-
12      starter-netflix-eureka-client</artifactId>
13      </dependency>
14      <dependency>
15
16      <groupId>org.springframework.boot</groupId>
17        <artifactId>spring-boot-starter-
18          web</artifactId>
19        </dependency>
20        <dependency>
21
22          <groupId>org.projectlombok</groupId>
23            <artifactId>lombok</artifactId>
24            <version>1.18.22</version>
25          </dependency>
26        </dependencies>

```

编写YML文件

```

1 server:
2   port: 800
3 eureka:

```

```

4   instance:
5     # 实例名字
6     instance-id: stream-producer8001
7   client:
8     service-url:
9       # 指定单机eureka server地址
10      #defaultZone:
11        http://localhost:7001/eureka/
12        defaultZone:
13          http://localhost:7001/eureka/,http://localhost:7002/eureka/
14
15   spring:
16     application:
17       name: stream-producer
18
19   rabbitmq:
20     port: 5672
21     host: 192.168.66.101
22     username: guest
23     password: guest
24
25   cloud:
26     stream:
27       bindings:
28         # 广播消息
29         myBroadcast-out-0: # 生产者绑定名称,
30           myBroadcast是自定义的绑定名称, out代表生产者, 0是
31           固定写法
32           destination: my-broadcast-topic
33           # 对应的真正的 RabbitMQ Exchange
34
35
36

```

注意:

spring.cloud.function.definition 属性非常重要，务必配置。

消息实体类

```

1  @Setter
2  @Getter
3  public class MyMessage implements
4      java.io.Serializable {
5
6      // 消息体
7      private String payload;
8
9  }

```

消息生产类实现

```

1  package com.itbaizhan.service.impl;
2
3  import
4      com.itbaizhan.service.IMessageProvider;
5  import
6      org.springframework.beans.factory.annotation
7          .Autowired;
8  import
9      org.springframework.cloud.stream.function.St
10     reamBridge;
11  import
12      org.springframework.stereotype.Service;
13
14 /**
15 * 定义消息的推送管道
16 */
17 @Service
18 public class MessageProviderImpl implements
19     IMessageProvider {
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1
```

```
14     @Autowired
15     private StreamBridge streamBridge;
16
17     @Override
18     public String send(String message) {
19
20         MyMessage myMessage = new
21         MyMessage();
22
23         // 生产消息
24         // 第一个参数是绑定名称，格式为：自定义的绑
25         // 定名称-out-0，myBroadcast是自定义的绑定名称，out
26         // 代表生产者，0是固定写法
27         // 自定义的绑定名称必须与消费方法的方法名保
28         // 持一致
29         // 第二个参数是发送的消息实体
30         streamBridge.send("myBroadcast-out-
31         0", myMessage);
32         return "SUCCESS";
33     }
34 }
```

编写业务类

```

1  @RestController
2  public class SendMessageController {
3
4      @Resource
5      private IMessageProvider
6      messageProvider;
7
8      @GetMapping("/send")
9      public String sendMessage(String
10     message){
11         return
12     messageProvider.send(message);
13 }
14 }
```

测试

Overview	Connections	Channels	Exchanges	Queues	Admin
exchange-testdlx	topic		D ha-all		
fallback-topic	topic		D ha-all		
input	topic		D ha-all		
my-broadcast-topic	topic		D ha-all	0.00/s	0.00/s
my-group-topic	topic		D ha-all	0.00/s	0.00/s
my-send-topic	topic		D ha-all	0.00/s	0.00/s
my-send2-topic	topic		D ha-all	0.00/s	0.00/s
my-topic-new	topic		D ha-all	0.00/s	0.00/s
myGroup	topic		D ha-all		
myTopic	topic		D ha-all		
requeue-topic	topic		D ha-all		
springCloudBus	topic		D ha-all		

自动生成队列

User: guest Cluster: rabbit@b054d02930d7 (change) Log out
RabbitMQ 3.6.2, Erlang_18,3

Overview Connections Channels Exchanges Queues Admin

Queues

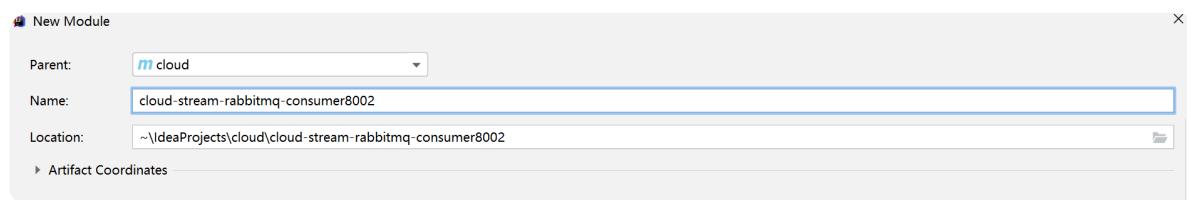
All queues (1)

Pagination Page 1 of 1 Filter: Regex (?)(?) Displaying 1 item , page size up to: 100

Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
my-broadcast-topic.anonymous.yDcwtYU4R5iQHq73BVcWiA	Excl AD Args idle		0	0	0			

消息驱动_入门案例之消息消费者

新建Module工程cloud-stream-rabbitmq-consumer8002,8003



POM添加依赖

```

1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-stream-starter-rabbit</artifactId>
5   </dependency>
6   <!-- 引入Eureka client依赖 -->
7   <dependency>
8     <groupId>org.springframework.cloud</groupId>
9   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
10  </dependency>
11  <dependency>
```

```

12 <groupId>org.springframework.boot</groupId>
13     <artifactId>spring-boot-starter-
14     web</artifactId>
15     </dependency>
16     <dependency>
17
18 <groupId>org.projectlombok</groupId>
19     <artifactId>lombok</artifactId>
20     <version>1.18.22</version>
21     </dependency>
22 </dependencies>

```

编写YML文件

```

1 server:
2     port: 8002
3 eureka:
4     instance:
5         # 实例名字
6         instance-id: stream-consumer8002
7     client:
8         service-url:
9             # 指定单机eureka server地址
10            #defaultZone:
11                http://localhost:7001/eureka/
12                defaultZone:
13                    http://localhost:7001/eureka/,http://localhost:7002/eureka/
14        spring:
15            application:
16                name: stream-consumer

```

```

15 rabbitmq:
16     port: 5672
17     host: 192.168.66.101
18     username: guest
19     password: guest
20 cloud:
21     stream:
22         bindings:
23             # 广播消息
24             myBroadcast-in-0: # 消费者绑定名称,
myBroadcast是自定义的绑定名称, in代表消费者, 0是固定写法
25                 destination: my-broadcast-topic
# 对应的真正的 RabbitMQ Exchange

```

注意:

spring.cloud.function.definition 属性非常重要, 务必配置。

消息实体类

```

1 @Setter
2 @Getter
3 public class MyMessage implements
java.io.Serializable {
4
5     // 消息体
6     private String payload;
7 }

```

消费者

```

1 @Slf4j
2 @Component
3 // 消费者
4 public class Concumer {
5
6     // 消费广播消息
7     @Bean
8     public Consumer<MyMessage> myBroadcast()
9     { // 方法名必须与生产消息时自定义的绑定名称一致
10
11         return message -> {
12             log.info("接收广播消息: {}", message.getPayload());
13         };
14     }

```

测试

将 Stream 工程启动两个实例，调用发送广播消息接口，查看消息消费情况。

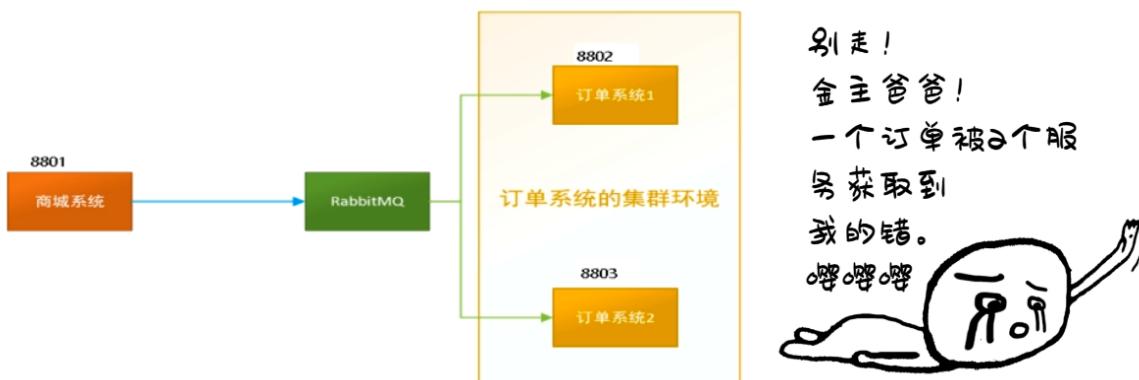
发送广播消息接口：GET <http://localhost:8001/send?message=hello> 这是一条广播消息！

KEY	VALUE	DESCRIPTION	Bulk Edit
<input checked="" type="checkbox"/> message	hello		
Key	Value	Description	

实例的消费情况

```
:接收广播消息: hello  
:接收广播消息: hello  
:接收广播消息: hello  
:接收广播消息: hello  
:接收广播消息: hello  
:接收广播消息: hello
```

消息驱动_分组消费



什么是消息分组

比如在电商场景中，订单系统我们做集群部署，都会从RabbitMQ中获取订单信息，那如果一个订单同时被两个服务获取到，那么就会造成数据错误，我们得避免这种情况。

注意：

我们可以使用Stream中的消息分组来解决。在Stream中处于同一个group中的多个消费者是竞争关系，就能够保证消息只会被其中一个应用消费一次。

服务生产者工程8001

编写controller新增接口

```

1 // 发送分组消息
2     @PostMapping("/group")
3         public String
4             sendGroupMessage(@RequestParam("body")
5                 String body) {
6
7
8             MyMessage myMessage = new
9                 MyMessage();
10            myMessage.setPayload(body);
11
12            // 生产消息
13            streamBridge.send("myGroup-out-0",
14                myMessage);
15            return "SUCCESS";
16        }

```

application.yml 配置

```

1 server:
2     port: 800
3 eureka:
4     instance:
5         # 实例名字
6         instance-id: stream-producer8001
7     client:
8         service-url:
9             # 指定单机eureka server地址
10            #defaultZone:
11                http://localhost:7001/eureka/
12                defaultZone:
13                    http://localhost:7001/eureka/,http://localhost:7002/eureka/
14
15 spring:

```

```

13 application:
14     name: stream-producer
15 rabbitmq:
16     port: 5672
17     host: 192.168.66.101
18     username: guest
19     password: guest
20 cloud:
21     stream:
22         bindings:
23             # 广播消息
24             myBroadcast-out-0: # 生产者绑定名称,
25                 myBroadcast是自定义的绑定名称, out代表生产者, 0是
26                 固定写法
27                 destination: my-broadcast-topic
28             # 对应的真实的 RabbitMQ Exchange
29             # 分组消息
30             myGroup-out-0: # 生产者绑定名称
31                 myGroup是自定义的绑定名称, out代表生产者, 0是
32                 固定写法
33                 destination: my-group-topic    # 对
34                 应的真实的 RabbitMQ Exchange

```

消费者工程8002,8003

新增接收信息方法

```

1 // 消费分组消息
2     @Bean
3     public Consumer<MyMessage> myGroup() {
4         // 方法名必须与生产消息时自定义的绑定名称一致
5         return message -> {
6             log.info("接收分组消息: {}", message.getPayload());
7         };
8     }

```

application.yml 配置

```

1 server:
2     port: 8003
3 eureka:
4     instance:
5         # 实例名字
6         instance-id: stream-consumer8003
7     client:
8         service-url:
9             # 指定单机eureka server地址
10            #defaultZone:
11                http://localhost:7001/eureka/
12                defaultZone:
13                    http://localhost:7001/eureka/,http://localhost:7002/eureka/
14 spring:
15     application:
16         name: stream-consumer
17     rabbitmq:
18         port: 5672
19         host: 192.168.66.101

```

```

18     username: guest
19     password: guest
20     cloud:
21     function:
22         # 定义消费者，多个用分号分隔，当存在大于1个的
23         # 消费者时，不定义不会生效
24         definition: myBroadcast;myGroup
25     stream:
26         bindings:
27             # 广播消息
28             myBroadcast-in-0: # 消费者绑定名称,
29             # myBroadcast是自定义的绑定名称，in代表消费者，0是固
30             # 定写法
31             destination: my-broadcast-topic
32             # 对应的真实的 RabbitMQ Exchange
33
34         # 分组消息
35         myGroup-in-0: # 消费者绑定名称
36             destination: my-group-topic    # 对
37             # 应的真实的 RabbitMQ Exchange
38             group: my-group-1    # 同一分组的消费
39             # 服务，只能有一个消费者消费到消息

```

验证分组消息

将 Stream 工程启动两个实例，调用发送分组消息接口，查看消息消费情况。

发送分组消息接口：GET<http://localhost:8001/group?message=hello> 这是一条分组消息！

http://localhost:8001/group?message=hello

GET http://localhost:8001/group?message=hello

Params

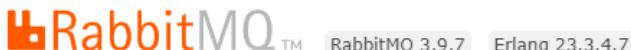
KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/> message	hello			
Key	Value	Description		

自动生成的 Exchange



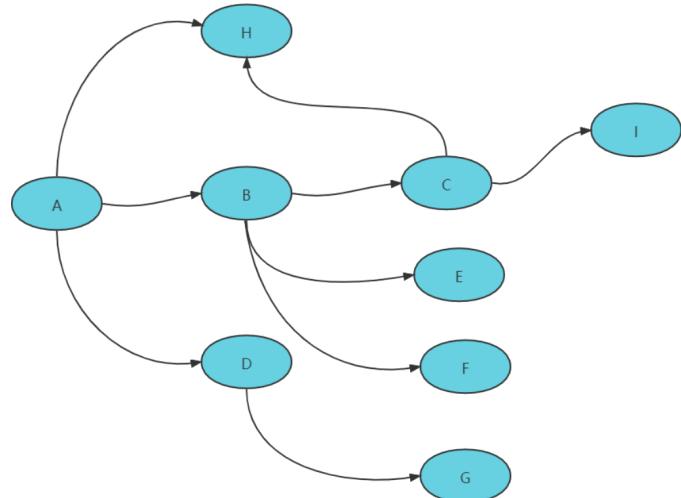
Overview	Connections	Channels	Exchanges	Queues	Admin
exchange-testdlx	topic		D ha-all		
fallback-topic	topic		D ha-all		
input	topic		D ha-all		
my-broadcast-topic	topic		D ha-all	0.00/s	0.00/s
my-group-topic	topic		D ha-all	0.00/s	0.00/s
my-send-topic	topic		D ha-all	0.00/s	0.00/s
my-send2-topic	topic		D ha-all	0.00/s	0.00/s
my-topic-new	topic		D ha-all	0.00/s	0.00/s
myGroup	topic		D ha-all		

自动生成的 Queue

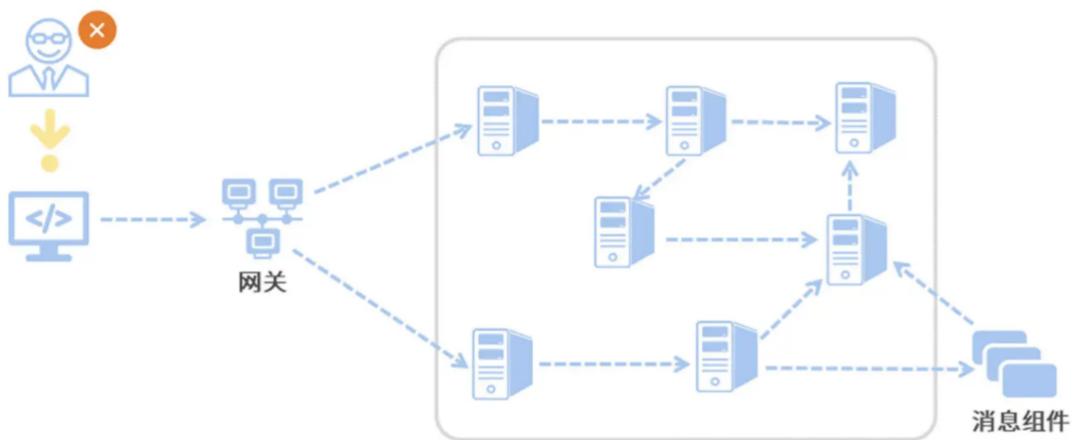


Overview	Connections	Channels	Exchanges	Queues	Admin
			+2		
fallback-topic.fallback-group	rabbit@zhufengren3	classic	D ha-all	idle	
my-broadcast-topic.anonymous.C3PWGee4TNiOVTGohu5gaQ	rabbit@zhufengren1	classic	AD Excl ML ha-all	idle	
my-broadcast-topic.anonymous.SOUV5bxMRpGtcCGP41hrMg	rabbit@zhufengren1	classic	AD Excl ML ha-all	idle	
my-group-topic.my-group-1	rabbit@zhufengren1	classic	D ha-all	idle	

分布式请求链路追踪_为什么需要链路追踪

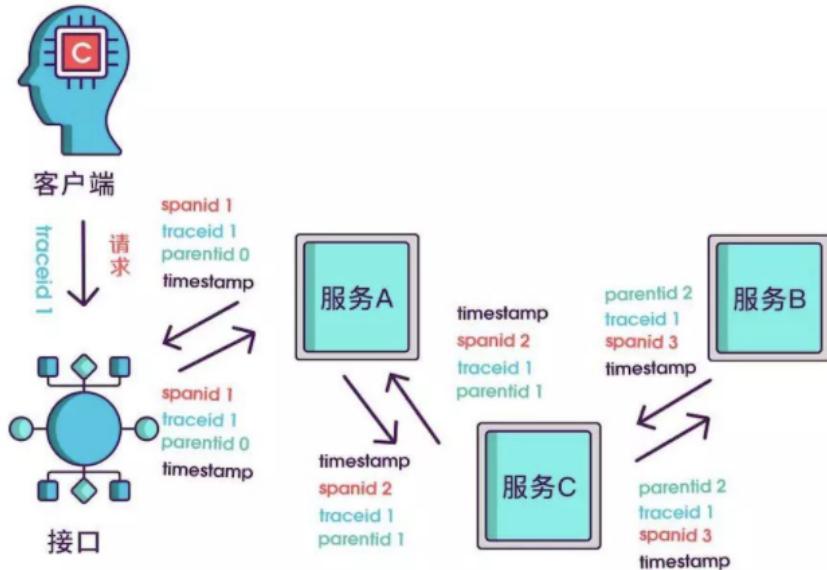


在这个微服务系统中，用户通过浏览器的 H5 页面访问系统，这个用户请求会先抵达微服务网关组件，然后网关再把请求分发给各个微服务。所以你会发现，用户请求从发起到结束要经历很多个微服务的处理，这里面还涉及到消息组件的集成。



存在的问题：

- 服务之间的依赖与被依赖的关系如何能够清晰的看到？
- 出现异常时如何能够快速定位到异常服务？
- 出现性能瓶颈时如何能够迅速定位哪个服务影响的？



解决：

为了能够在分布式架构中快速定位问题，分布式链路追踪应运而生。将一次分布式请求还原成调用链路，进行日志记录，性能监控并将一次分布式请求的调用情况集中展示。比如各个服务节点上的耗时、请求具体到达哪台机器上、每个服务节点的请求状态等等。

常见链路追踪技术有那些

市面上有很多链路追踪的项目，其中也不乏一些优秀的，如下：

- **Sleuth**: SpringCloud 提供的分布式系统中链路追踪解决方案。很可惜的是阿里系并没有链路追踪相关的开源项目，我们可以采用Spring Cloud Sleuth+Zipkin来做链路追踪的解决方案。
- **zipkin**: 由Twitter公司开源，开放源代码分布式的跟踪系统，用于收集服务的定时数据，以解决微服务架构中的延迟问题，包括：数据的收集、存储、查找和展现。该产品结合 `spring-cloud-sleuth` 使用较为简单，集成很方便，但是功能较简单。
- **pinpoint**: 韩国人开源的基于字节码注入的调用链分析，以及应用监控分析工具。特点是支持多种插件，UI功能强大，接入端无代码侵入

- **skywalking**: SkyWalking是本土开源的基于字节码注入的调用链分析，以及应用监控分析工具。特点是支持多种插件，UI功能较强，接入端无代码侵入。目前已加入Apache孵化器。

实时效果反馈

1. Spring Cloud Sleuth组件主要解决____问题。

A 注册中心

B 负载均衡

C 服务熔断

D 链路追踪

2. 下列不是分布式链路追踪技术的是____。

A Sleuth

B SkyWalking

C Eureka

D Zipkin

答案

1=>D 2=>C

分布式请求链路追踪_Spring Cloud Sleuth是什么

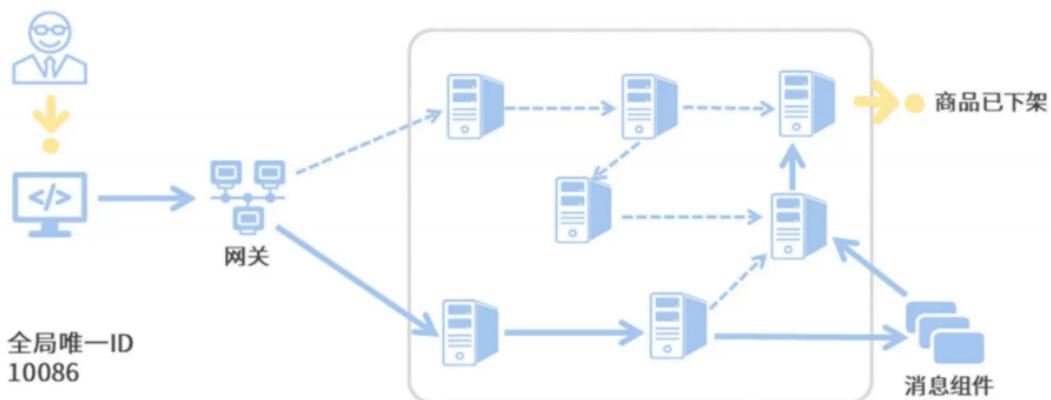


Spring Cloud Sleuth实现了一种分布式的服务链路跟踪解决方案，通过使用Sleuth可以让我们快速定位某个服务的问题。简单来说，Sleuth相当于调用链监控工具的客户端，集成在各个微服务上，负责产生调用链监控数据。

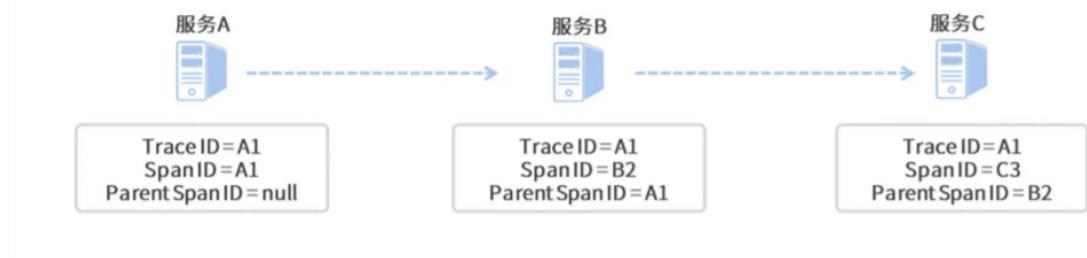
注意：

Spring Cloud Sleuth只负责产生监控数据，通过日志的方式展示出来，并没有提供可视化的UI界面。

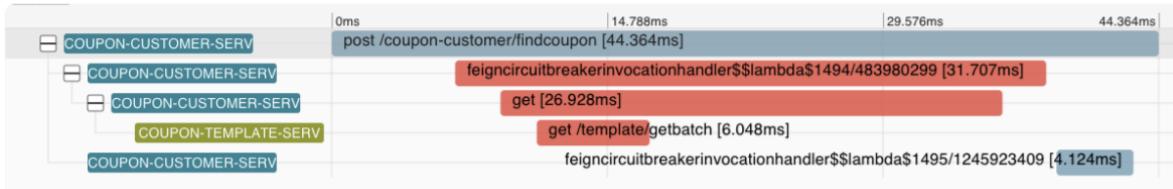
Sleuth核心概念



- **Span**: 基本的工作单元，相当于链表中的一个节点，通过一个唯一ID标记它的开始、具体过程和结束。我们可以通过其中存储的开始和结束的时间戳来统计服务调用的耗时。除此之外还可以获取事件的名称、请求信息等。
- **Trace**: 一系列的Span串联形成的一个树状结构，当请求到达系统的入口时就会创建一个唯一ID (traceId)，唯一标识一条链路。这个traceId始终在服务之间传递，直到请求的返回，那么就可以使用这个traceId将整个请求串联起来，形成一条完整的链路。



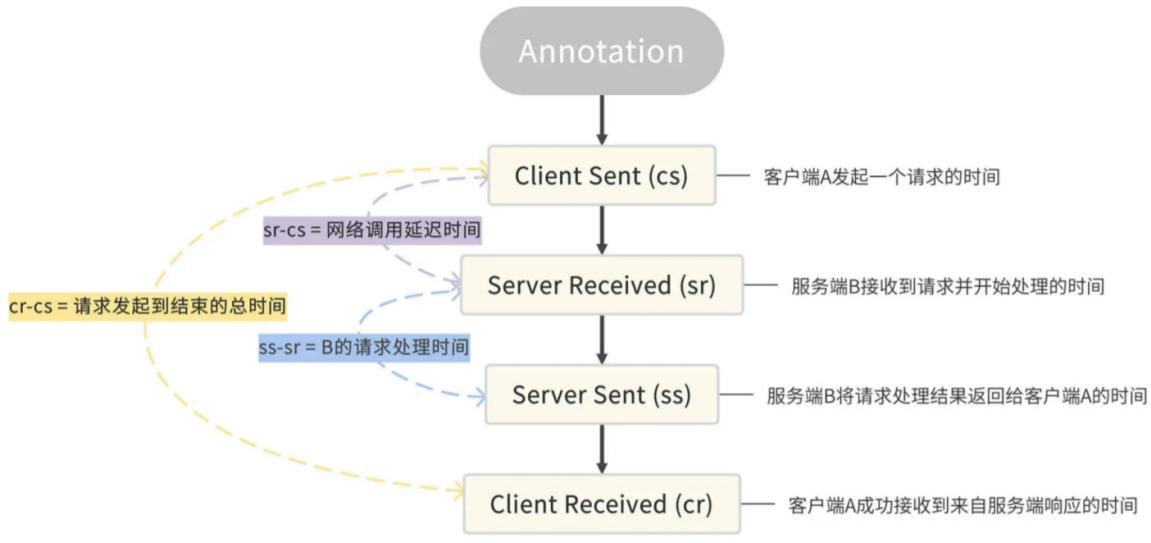
通过这些Sleuth 的特殊标记，我们就可以根据时间顺序，将一次服务请求经过的调用节点都梳理出来，这样你就能迅速发现报错信息发生在哪个阶段。这是使用Zipkin 生成的链路追踪的可视化信息。你可以看出，每个服务调用都以时间先后顺序规整好了，红色的部分就是发生线上Exception的服务。



除了Trace和Span之外，Sleuth还有一个特殊的数据结构，叫做Annotation，被用来记录一个具体的“事件”。我把Sleuth所支持的四种事件做成了一个表格，你可以参考一下。

Annotation名称	作用
Client Sent (cs)	客户端发起一个请求的时间。
Server Received (sr)	服务端接收到用户请求并开始处理的时间。
Server Sent (ss)	服务端将请求处理结果返回给客户端的时间。
Client Received (cr)	客户端成功接收到来自服务端响应的时间戳。

在这里我举个例子，来帮你理解怎么使用这四种事件。



流程：

如果你用服务B的ss减去 sr，你就可以得到请求在服务B阶段的处理时间。如果用服务B的sr减去服务A的cs，就可以得到服务A到服务B之间的网络调用延迟时间。如果用服务A的 cr减去 cs，就可以得到当次请求从发起到结束所花费的总时间。

实时效果反馈

1. Spring Cloud Sleuth只负责_____数据。

- A 服务熔断
- B 负载均衡
- A 注册中心
- B 负载均衡
- C 产生监控数据
- D 链路追踪

2. Spring Cloud Sleuth是通过_____来实现链路追踪。

- A 打标记
- B 打日志
- C 保存数据库
- D 以上都是错误

答案

1=>C 2=>A

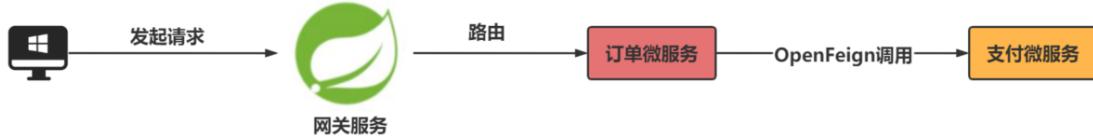
分布式请求链路追踪_微服务集成Sleuth实现链路打标



整合Spring Cloud Sleuth其实没什么的难的，在这之前需要准备以下三个服务：

- `cloud-gateway-gateway9527`: 作为网关服务
- `cloud-consumer-feign-order80`: 订单微服务
- `cloud-provider-payment8001`: 支付微服务

三个服务的调用关系如下图：



流程：

客户端请求网关发起订单的请求，网关路由给订单服务，订单服务调用支付服务进行支付。

实现步骤

第一步，我们需要将 Sleuth 的依赖项添加到三个微服务的 pom.xml 文件

```

1 <!-- sleuth依赖项 -->
2 <dependency>
3
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-
sleuth</artifactId> </dependency>
  
```

第二步，我们打开微服务模块的 application.yml 配置文件，在配置文件中添加采样率和每秒采样记录条数。

```

1 spring:
2   sleuth:
3     sampler:
4       # 采样率的概率，100%采样
5       probability: 1.0
6       # 每秒采样数字最高为100
7       rate: 1000
  
```

注意：

在配置文件里设置了一个 probability，它应该是一个 0 到 1 的浮点数，用来表示采样率。我这里设置的 probability 是 1，就表示对请求进行 100% 采样。如果我们把 probability 设置成小于 1 的数，就说明有的请求不会被采样。如果一个请求未被采样，那么它将不会被调用链追踪系统 Track 起来。

三个服务中调整日志级别

由于sleuth并没有UI界面，因此需要调整一下日志级别才能在控制台看到更加详细的链路信息。在三个服务的配置文件中添加以下配置：

```

1 ## 设置openFeign和sleuth的日志级别为debug，方便查看日志信息
2 logging:
3   level:
4     org.springframework.cloud.openfeign:
5       debug
6     org.springframework.cloud.sleuth: debug

```

演示接口完善

<http://localhost:9527/order/index>

```

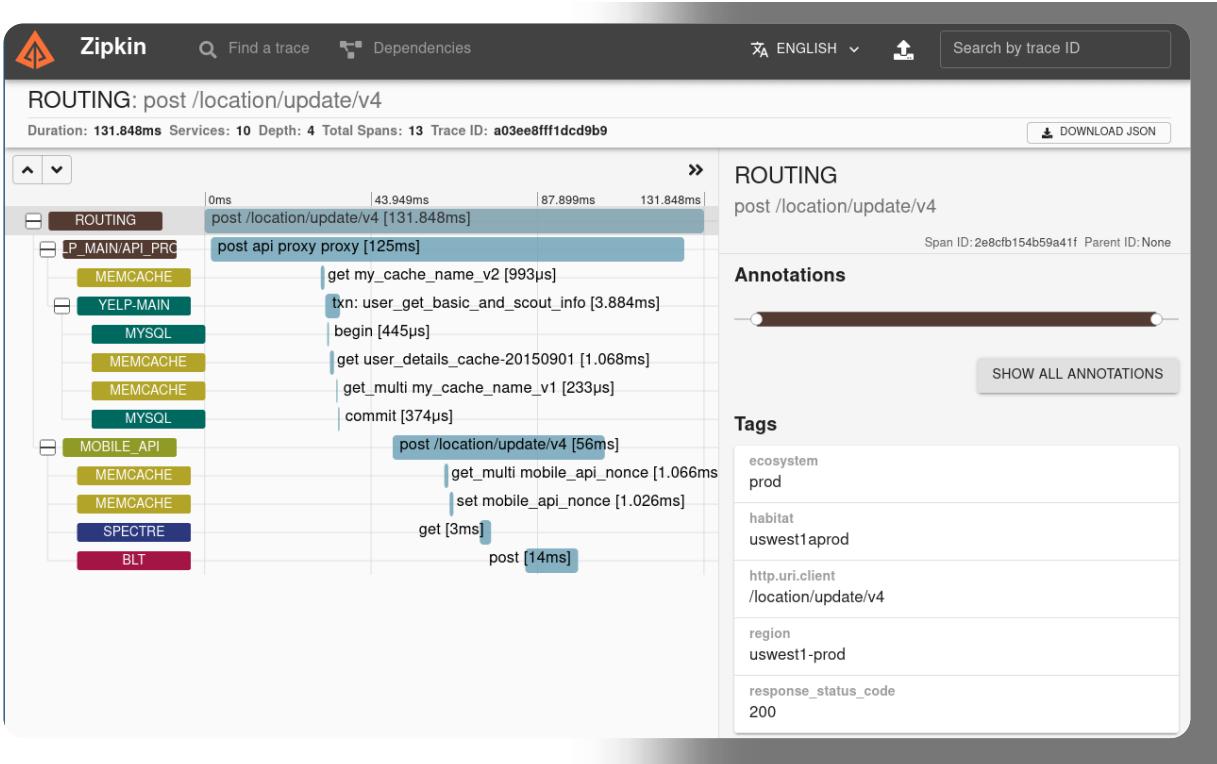
I Console Endpoints
DEBUG [gateway-service,,] 7432 --- [ctor-http-nio-2] o.s.c.s.instrument.web.TraceWebFilter : Received a requ
DEBUG [gateway-service,,] 7432 --- [ctor-http-nio-2] o.s.c.s.instrument.web.TraceWebFilter : Handled receive
DEBUG [gateway-service,2755137b85ec746a,2755137b85ec746a] 7432 --- [ctor-http-nio-2] ientBeanPostProcessor$Trac
DEBUG [gateway-service,,] 7432 --- [ctor-http-nio-2] PostProcessor$AbstractTracingDoOnHandler : Handle receive
DEBUG [gateway-service,,] 7432 --- [ctor-http-nio-2] o.s.c.s.instrument.web.TraceWebFilter : Handled send of

```

日志格式中总共有四个参数，含义分别如下：

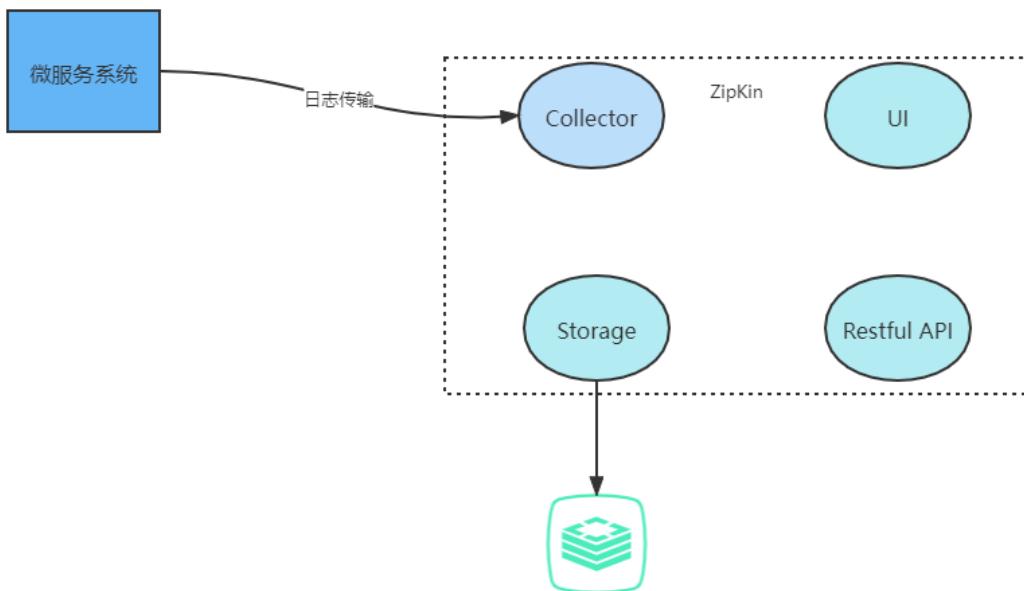
- 第一个：服务名称
- 第二个：traceId，唯一标识一条链路
- 第三个：spanId，链路中的基本工作单元id

分布式请求链路追踪_什么是ZipKin



什么是Zipkin

Zipkin是Twitter 的一个开源项目，它基于Google Dapper实现，它致力于收集服务的定时数据，以解决微服务架构中的延迟问题，包括数据的收集、存储、查找和展现。



Zipkin4个核心的组件

- **Collector**: 收集器组件，它主要用于处理从外部系统发送过来的跟踪信息，将这些信息转换为Zipkin内部处理的Span格式，以支持后续的存储、分析、展示等功能。
- **Storage**: 存储组件，它主要对处理收集器接收到的跟踪信息，默认会将这些信息存储在内存中，我们也可以修改此存储策略，通过使用其他存储组件将跟踪信息存储到数据库中
- **RESTful API**: API组件，它主要用来提供外部访问接口。比如给客户端展示跟踪信息，或是外接系统访问以实现监控等。
- **UI**: 基于API组件实现的上层应用。通过UI组件用户可以方便而直观地查询和分析跟踪信息



注意：

zipkin分为服务端和客户端，服务端主要用来收集跟踪数据并且展示，客户端主要功能是发送给服务端，微服务的应用也就是客户端，这样一旦发生调用，就会触发监听器将sleuth日志数据传输给服务端。

实时效果反馈

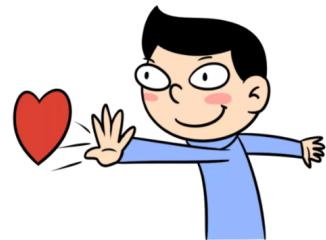
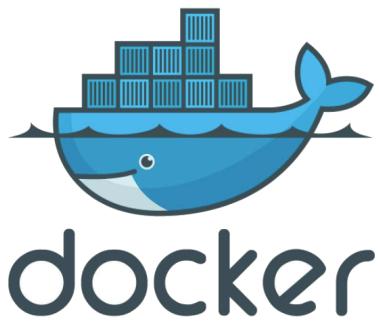
1.zipkin主要作用_____数据。

- A 服务熔断
- B 负载均衡
- C 收集并查看链路
- D 链路追踪

答案

1=>C

分布式请求链路追踪_Docker搭建Zipkin服务



Docker技术用的好，
下班约会回家早。

为了搭建一条高可用的链路信息传递通道，我将使用RabbitMQ作为中转站，让各个应用服务器将服务调用链信息传递给 RabbitMQ，而Zipkin 服务器则通过监听 RabbitMQ 的队列来获取调用链数据。相比于让微服务通过 Web 接口直连 Zipkin，使用消息队列可以大幅提高信息的送达率和传递效率。



安装RabbitMQ服务

```
1 docker run -d --name rabbitmq -e
RABBITMQ_DEFAULT_USER=guest -e
RABBITMQ_DEFAULT_PASS=guest -p 15672:15672 -p
5672:5672 docker.io/macintoshplus/rabbitmq-
management
```

下载Zipkin镜像

```
1 docker pull openzipkin/zipkin
```

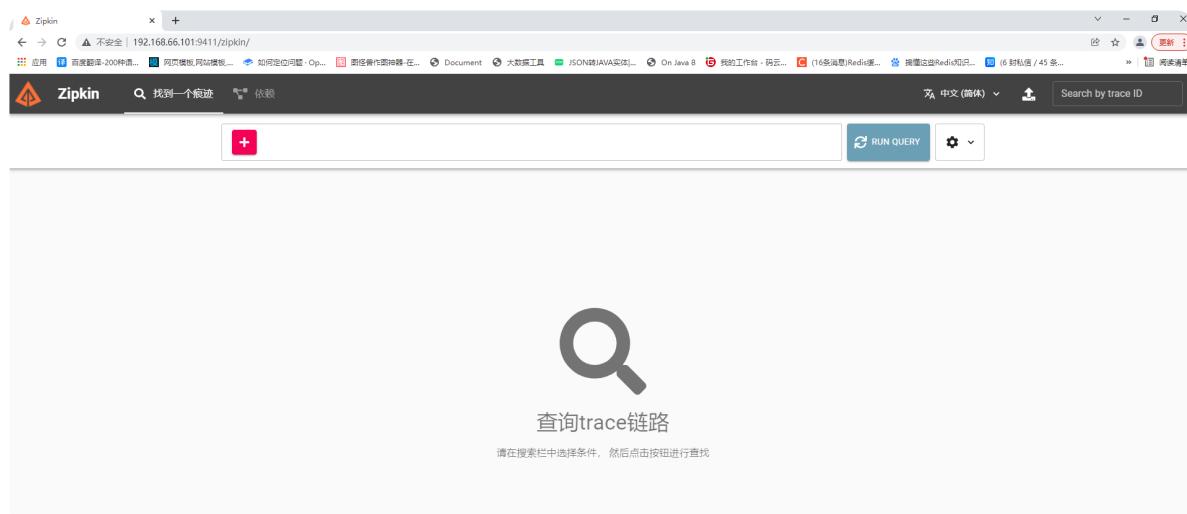
启动容器

```
1 docker run --name rabbit-zipkin -d -p
9411:9411 --link rabbitmq -e
RABBIT_ADDRESSES=rabbitmq:5672 -e
RABBIT_USER=guest -e RABBIT_PASSWORD=guest
openzipkin/zipkin
```

测试

此时可以访问zipkin的UI界面，地址：

<http://192.168.66.101:9411>，界面如下：



最后，我们只需要验证消息监听队列是否已就位就可以了。我们使用 guest 账号登录 RabbitMQ，并切换到“Queues”面板，如果 Zipkin 和 RabbitMQ 的对接一切正常，那么你会在 Queues 面板下看到一个名为 zipkin 的队列。

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
mv-group-topic.mv-group-1	D	idle	0	0	0			
zipkin	D	idle	0	0	0	0.00/s	0.00/s	0.00/s

Add a new queue

Name: *

Durability: Durable

Auto delete: No

Arguments: = String

Add Message TTL (?) | Auto expire (?) | Max length (?) | Max length bytes (?)
Dead letter exchange (?) | Dead letter routing key (?) | Maximum priority (?)

Add queue

[HTTP API](#) | [Command Line](#)

分布式请求链路追踪_Zipkin客户端搭建



三个服务添加依赖

首先，我们需要在每个微服务模块的 pom.xml 中添加 Zipkin 适配插件和 Stream 的依赖。其中，Stream 是 Spring Cloud 中专门对接消息中间件的组件。

```
1 <!--链路追踪 zipkin依赖，其中包含sleuth的依赖-->
2 <dependency>
3
4     <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-sleuth-
6             zipkin</artifactId>
7     </dependency>
8     <dependency>
9         <groupId>org.springframework.cloud</groupId>
10    </dependency>
11    <dependency>
12        <artifactId>spring-cloud-stream-binder-
13            rabbit</artifactId>
14    </dependency>
```

配置文件需要配置一下zipkin服务端的地址，配置如下：

```
1 spring:
2   sleuth:
3     sampler:
4       # 日志数据采样百分比, 默认0.1(10%), 这里为了
5       # 测试设置成了100%, 生产环境只需要0.1即可
6         probability: 1.0
7
8 zipkin:
9   sender:
10     type: rabbit
11
12 rabbitmq:
13   addresses: 192.168.66.101:5672
14   queue: zipkin
```

测试

请求<http://localhost:9527/order/index>调用接口之后，再次访问zipkin的UI界面

服务	开始时间	Spans	持续时间
Root			
cloud-payment-provider: get /payment/lb	a minute ago (02/18 16:05:48:441)	1	50.408ms
cloud-payment-provider: get /payment/lb	a few seconds ago (02/18 16:06:01:893)	1	2.822ms
cloud-payment-provider: get /payment/lb	a few seconds ago (02/18 16:06:02:028)	1	2.166ms

可以看到刚才调用的接口已经被监控到了，点击 **SHOW** 进入详情查看，如下图：

分布式请求链路追踪_什么是SkyWalking



什么是SkyWalking

SkyWalking 是中国人吴晟（华为）开源的一款 APM 工具是一个开源的可观测平台，主要用于收集，分析，聚合和可视化服务和云原生基础设施的数据。SkyWalking 提供了一种简单的方式来维护分布式系统的视图关系，它甚至能够支持跨云的服务。特别为微服务、云原生和基于容器（Docker, Kubernetes, Mesos）体系结构而设计。

The screenshot displays the SkyWalking interface. At the top, there's a navigation bar with tabs for Dashboard, Topology, Trace (selected), Profile, Log, Alarm, Event, and Debug. Below the navigation is a search bar with fields for Trace ID, Duration, and Time Range, and a note to input tags. The main area shows a list of traces and a detailed trace view. The detailed view for trace ID 3f172d4f-10fd-4fe1-ad4a-add780a061fb shows a timeline from 11:07 11:52:38 to 11:57:06. It lists several spans, each with a duration (e.g., 62028 ms, 62024 ms, 62023 ms, 62020 ms, 62019 ms) and a timestamp. The visualization part shows a call graph with nodes for /projectA/test, /projectB/test, and /projectC/value, connected by lines representing the flow of data between them. Labels indicate the protocol (Http - Nginx, SpringMVC, SpringRestTemplate) and the specific method being called.

SkyWalking与ZipKin对比

Zipkin

Twitter公司开源的一个分布式追踪工具，被Spring Cloud Sleuth集成，使用广泛而稳定。

优点:

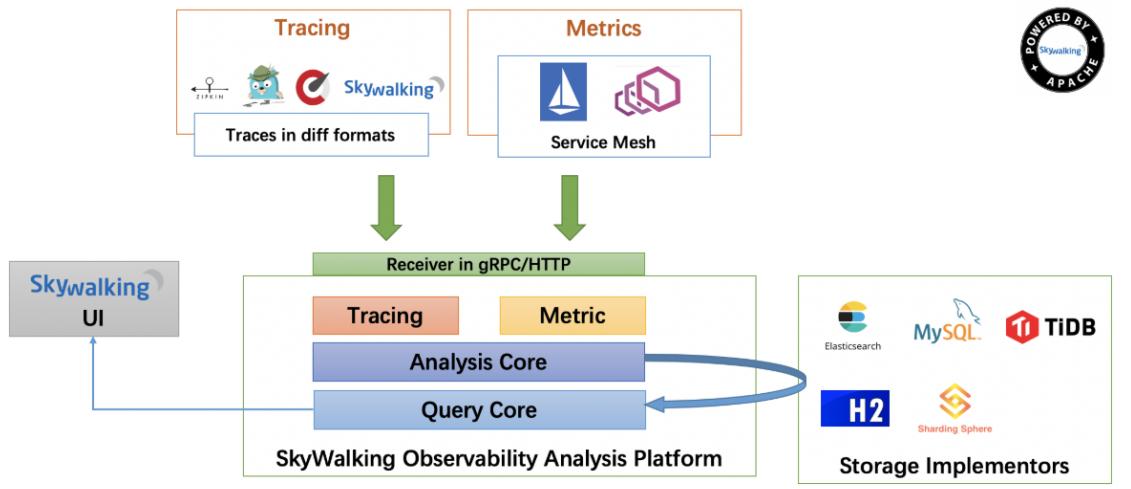
- 轻量级，SpringCloud集成，使用人数多，成熟

缺点:

- 侵入性
- 功能简单
- 欠缺APM报表能力（能力弱）

SkyWalking

SkyWalking是开源的一款分布式追踪，分析，告警的工具，现已属于Apache旗下开源项目，SkyWalking为服务提供了自动探针代理，将数据通过gRPC或者HTTP传输给后端平台，后端平台将数据存储在Storage中，并且分析数据将结果展示在UI中。



优点:

- 多种监控手段多语言自动探针，Java，.NET Core 和 Node.JS
- 轻量高效，不需要大数据
- 模块化，UI、存储、集群管理多种机制可选，
- 支持告警
- 社区活跃

缺点:

- 较为新兴，成熟度不够高

SkyWalking能够解决什么问题

- SkyWalking提供了非常完善的分布式链路追踪功能
- SkyWalking提供了非常完善的错误诊断功能
- SkyWalking提供了非常完善的实时链路大屏功能
- SkyWalking提供了非常完善的服务拓扑关系分析功能
- SkyWalking提供了非常完善的链路指标及错误告警功能
- SkyWalking提供了在线性能诊断功能

实时效果反馈

1. 分布式链路追踪SkyWalking主要作用_____。

- A 服务熔断
- B 负载均衡
- C 收集并查看链路
- D 链路追踪

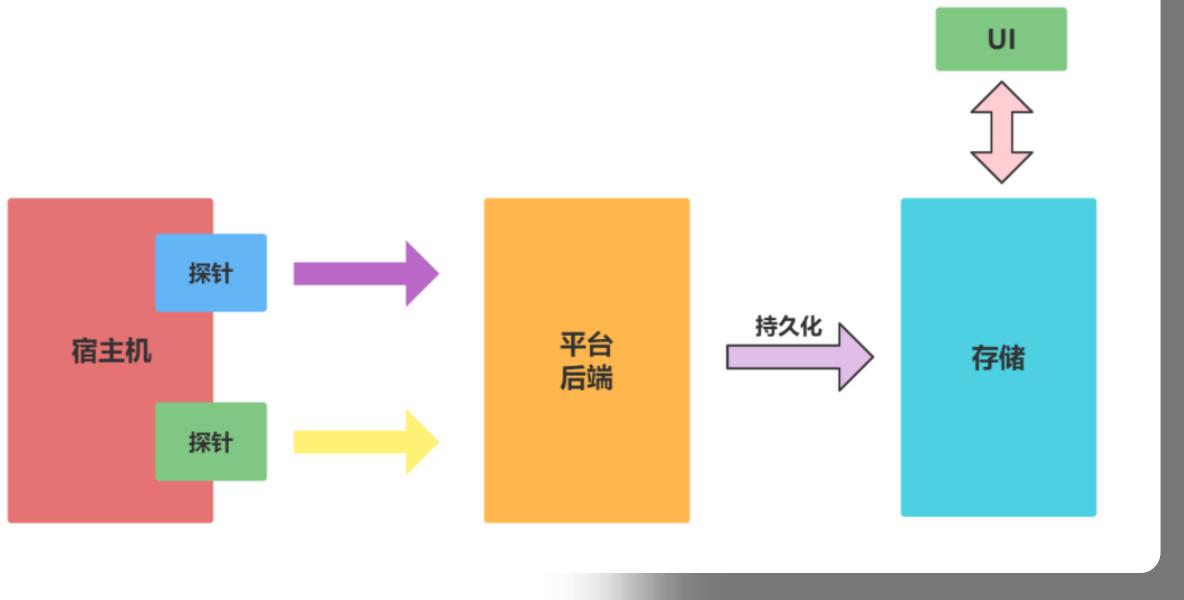
2. 下列属于分布式链路追踪SkyWalking优点的是_____。

- A 无代码侵入
- B 多种监控手段多语言自动探针
- C 支持告警
- D 以上都是正确

答案

1=>C 2=>D

分布式请求链路追踪_SkyWalking核心概念



注意：

- 宿主应用：被探针通过字节码技术引入应用。从探针的视角来看，应用是探针寄生的宿主。
- 探针：收集从应用采集到的链路数据，并将其转换为SkyWalking能够识别的数据格式。
- RPC：宿主应用和平台后端之间的通信渠道。
- 平台后端：支持数据聚合、分析和流处理，包括跟踪，指标和日志等。
- 存储：通过开放的，可插拔的接口来存储SkyWalking的链路数据。SkyWalking目前支持 ElasticSearch、H2、MySQL、TiDB、InfluxDB。
- UI：一个可定制的基于WEB的界面，允许SkyWalking终端用户管理和可视化SkyWalking的链路数据。

实时效果反馈

1. 下列不是SkyWalking支持的存储方式_____。

- A ES
- B MySQL
- C H2
- D Redis

答案

1=>C

分布式请求链路追踪_什么是探针Java Agent



探针产生的背景

在开发过程中，开发人员经常会使用IDEA的Debug功能（包含本地和远程）调试应用，在JVM进程期间获取应用运行的JVM信息，变量信息等。这些个技术通过Java Agent来实现的，那么Java Agent到底是啥，为啥这么吊？

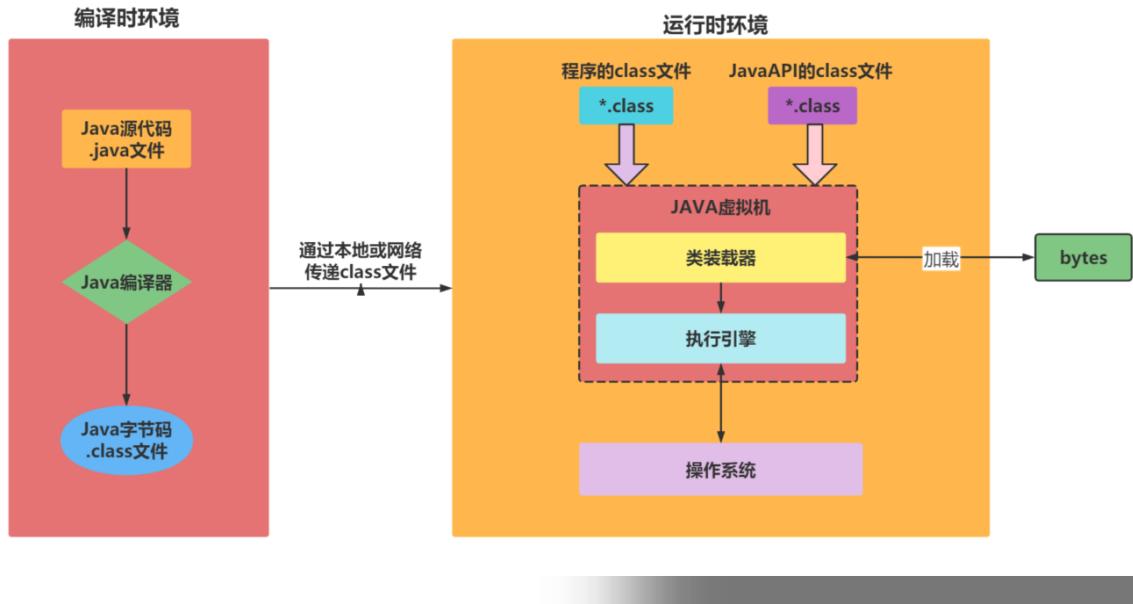
什么是探针

JavaAgent提供了一种在加载字节码时对字节码进行修改的方式。通常使用ASM Javassist字节码工具修改class文件。

注意：

javassist是一个库，实现ClassFileTransformer接口中的transform()方法。ClassFileTransformer这个接口的目的就是在class被装载到JVM之前将class字节码转换掉，从而达到动态注入代码的目的。

Java探针工具技术原理



流程：

- ① 在 JVM 加载 class 二进制文件的时候，利用 ASM 动态的修改加载的 class 文件，在监控的方法前后添加计时器功能，用于计算监控方法耗时；
- ② 将监控的相关方法和耗时及内部调用情况，按照顺序放入处理器；
- ③ 处理器利用栈先进后出的特点对方法调用先后顺序做处理，当一个请求处理结束后，将耗时方法轨迹和入参 map 输出到文件中；
- ④ 然后区分出耗时的业务，转化为 XML 格式进行解析和分析。

Java 探针工具功能

- 1、支持方法执行耗时范围抓取设置，根据耗时范围抓取系统运行时出现在设置耗时范围的代码运行轨迹。
- 2、支持抓取特定的代码配置，方便对配置的特定方法进行抓取，过滤出关系的代码执行耗时情况。
- 3、支持 APP 层入口方法过滤，配置入口运行前的方法进行监控，相当于监控特有的方法耗时，进行方法专题分析。
- 4、支持入口方法参数输出功能，方便跟踪耗时高的时候对应的入参数。
- 5、提供 WEB 页面展示接口耗时展示、代码调用关系图展示、方法耗时百分比展示、可疑方法凸显功能。

实时效果反馈

1. Java Agent Java 代理，又叫_____。

- A Java 探针
- B 虚拟机
- C 栈

D 以上都是错误

答案

1=>A

分布式请求链路追踪 Java探针日志监控实现之环境搭建



引入依赖

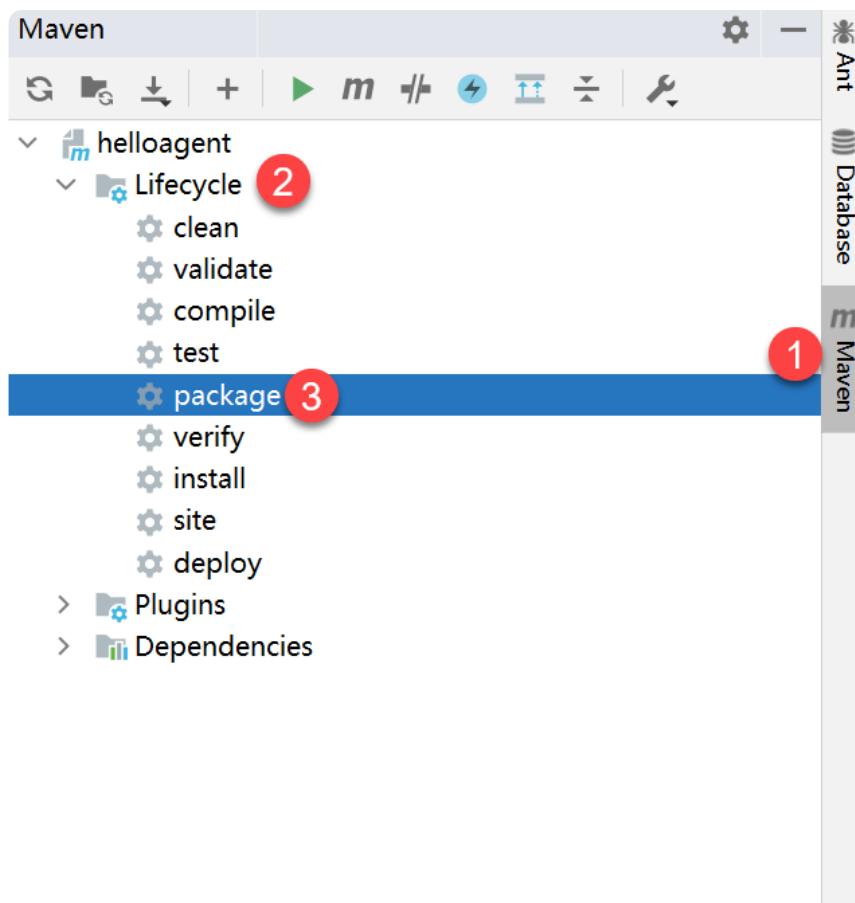
```
1 <dependency>
2
3   <groupId>org.projectlombok</groupId>
4     <artifactId>lombok</artifactId>
5   </dependency>
```

编写HelloController

```

1 @RestController
2 @Slf4j
3 public class HelloController {
4
5     @GetMapping("/hello")
6     public String hello() {
7         return "hello agent";
8     }
9 }
```

打包SpringBoot项目



分布式请求链路追踪 Java探针日志监控实现之探针实现



请尽快落实
“探针实现”

如何使用Java Agent

JDK1.5引入了java.lang.instrument包，开发者可以很方便的实现字节码增强。其核心功能由java.lang.instrument.Instrumentation接口提供。

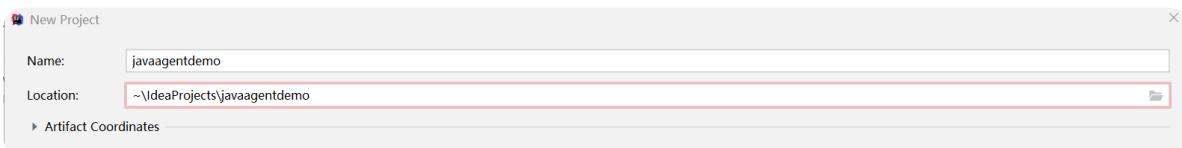
Instrumentation 有两种使用方式：

- 在应用运行之前，通过premain()方法来实现，在应用启动时侵入并代理应用。
- 在应用运行之后，通过调用Attach AP和agentmain()方法来实现

注意：

javassist是一个库，实现ClassFileTransformer接口中的transform()方法。ClassFileTransformer 这个接口的目的就是在class被装载到JVM之前将class字节码转换掉，从而达到动态注入代码的目的。

新建Maven项目javaagentdemo



Pom引入依赖

```

1   <dependencies>
2     <dependency>
3       <groupId>org.javassist</groupId>

```

```

4   <artifactId>javassist</artifactId>
5       <version>3.22.0-GA</version>
6   </dependency>
7 </dependencies>
8
9 <build>
10    <finalName>javaagent</finalName>
11    <plugins>
12        <!--
13            META-INF 下 MANIFEST.MF 文件 内容
14            Manifest-Version: 1.0
15            Premain-Class:
16            com.jenson.TestAgent
17            下面Maven插件可以自动实现
18            -->
19            <plugin>
20                <groupId>org.apache.maven.plugins</groupId>
21                    <artifactId>maven-shade-
22 plugin</artifactId>
23                    <version>3.0.0</version>
24                    <executions>
25                        <execution>
26                            <phase>package</phase>
27                                <goals>
28                                    <goal>shade</goal>
29                                        </goals>
<configuration>
    <transformers>

```

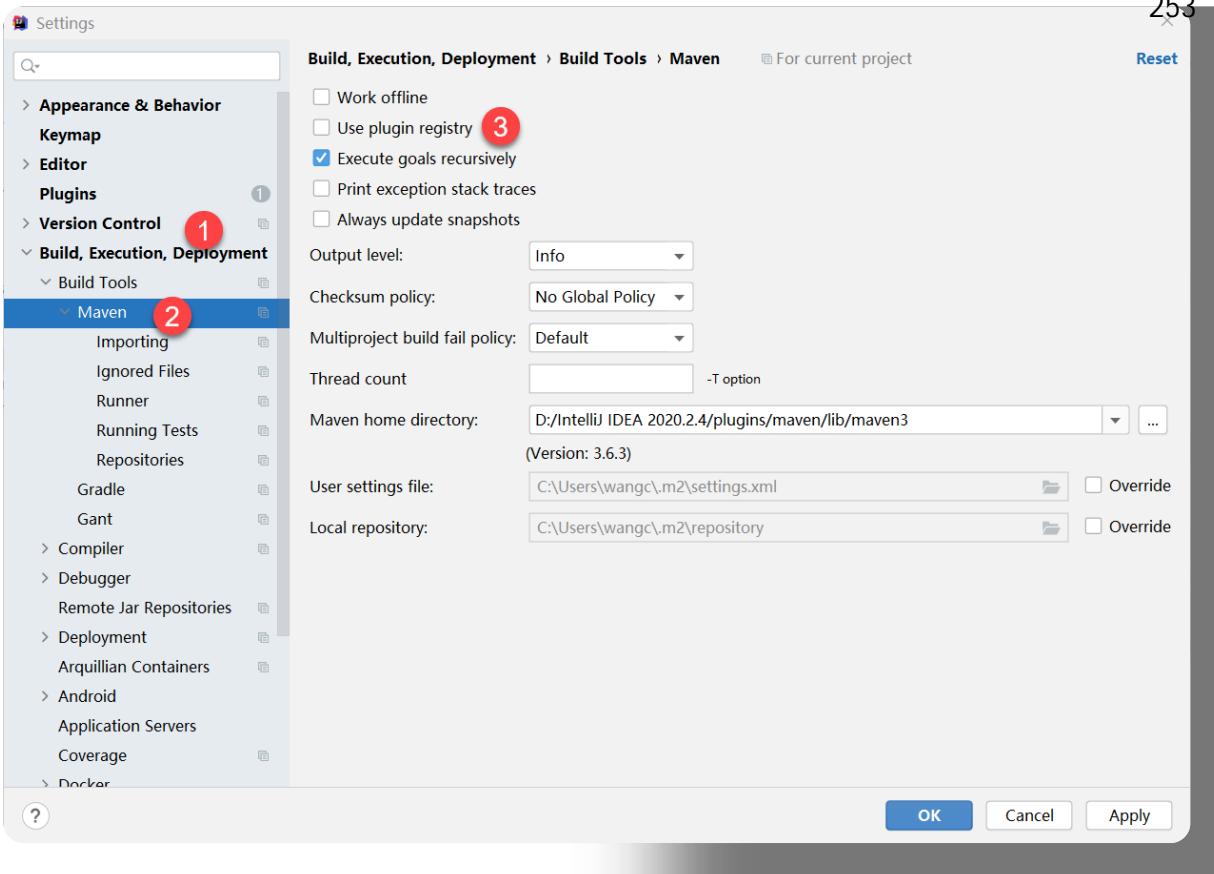
```
30
31     implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
32
33     <manifestEntries>
34         <Premain-
35             class>com.itbaizhan.PreMainTraceAgent</Premain-
36         </manifestEntries>
37     </transformer>
38
39         </transformers>
40
41         </configuration>
42
43         </execution>
44
45         </executions>
46
47         </plugin>
48
49         </plugins>
50
51     </build>
```

Maven插件引入失败

在IDEA中使用maven-shade-plugin插件时会提示"Plugin 'maven-shade-plugin': not found"

解决方法:

- File -> Setting -> Build, Execution, Deployment -> Build Tools -> Maven -> 勾选Use plugin registry选项
- File -> Invalidate Caches -> Invalidate and Restart



编写探针类TestAgent

```

1 package com.jenson;
2
3 import java.lang.instrument.Instrumentation;
4
5 /**
6 * @author Jenson
7 */
8 public class TestAgent {
9
10    /**
11     * 在main方法运行前，与main方法运行于同一JVM中
12     *
13     * @param agentArgs agentArgs是premain函数得到的程序参数，随同"-javaagent"一同传入，
14     *                   与main函数不同的是，该参数是一个字符串而不是一个字符串数组，

```

如果程序参数有多个，程序

```

15     *
16     * @param inst      一个
17     * java.lang.instrument.Instrumentation实例，由
18     * JVM自动传入，
19     *
20     *          也是核心部分，集中了其中
21     *几乎所有的功能方法，例如类定义的转换和操作
22     */
23     public static void premain(String
24         agentArgs, Instrumentation inst) {
25
26         System.out.println("=====premain1
执行=====");
27         System.out.println("agentArgs : " +
agentArgs);
28         // 添加Transformer
29         inst.addTransformer(new
30             TestTransformer());
31     }
32 }
```

编写TestTransformer

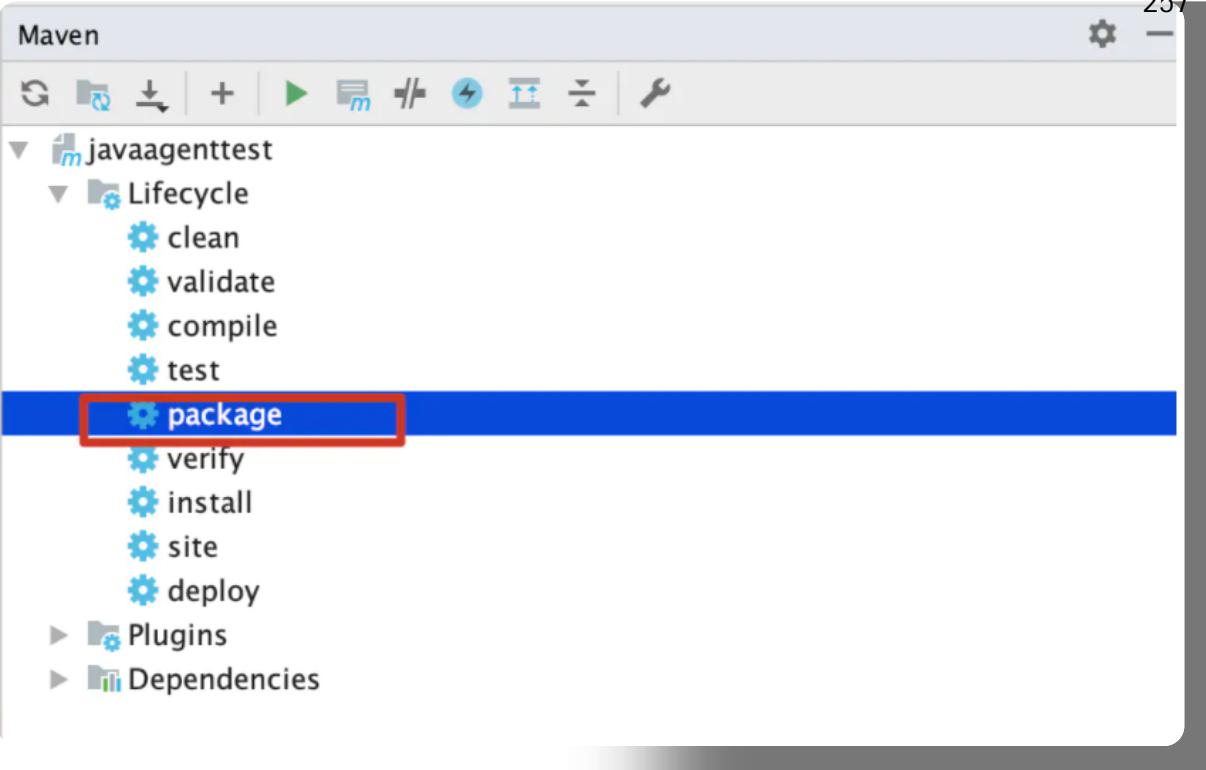
```

1 /**
2  * @author Jenson
3 */
4 public class TestTransformer implements
5     ClassFileTransformer {
```

```
6     public final String TEST_CLASS_NAME =
7         "com.itbaizhan.controller.HelloController";
8
9
10    public byte[] transform(ClassLoader
11        loader,
12        String
13        className,
14        Class<?>
15        classBeingRedefined,
16        ProtectionDomain
17        protectionDomain,
18        byte[]
19        classfileBuffer) throws
20        IllegalClassFormatException {
21
22        //      System.out.println("className : " +
23        //      className);
24
25        String finalClassName =
26        className.replace("/", ".");
27
28        if
29        (TEST_CLASS_NAME.equals(finalClassName)) {
30            System.out.println("class name
31            匹配上了 !");
32
33        CtClass ctClass;
```

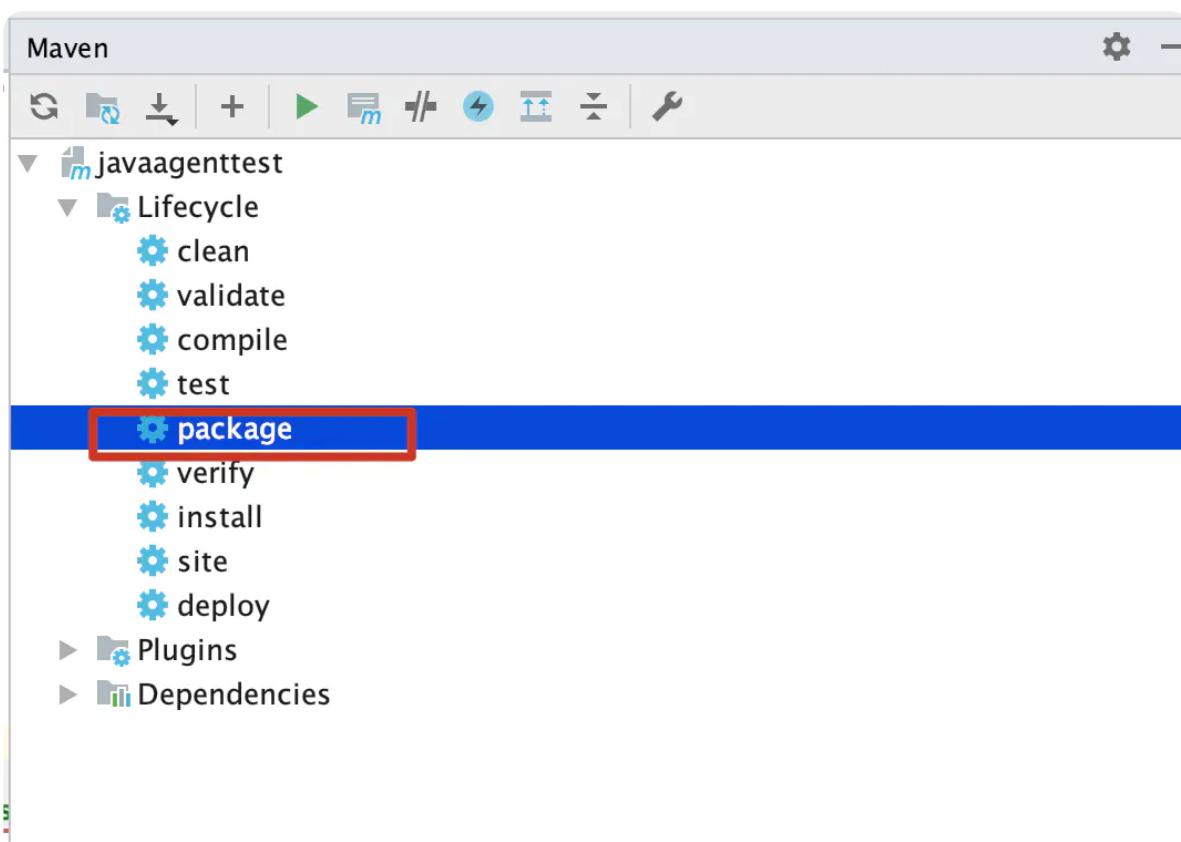
```
26     try {
27         ctclass =
28             classPool.getDefault().get(finalClassName);
29         System.out.println("ctclass
30             is OK !");
31         CtMethod ctMethod =
32             ctClass.getDeclaredMethod(METHOD_NAME);
33         System.out.println("ctMethod
34             is OK !");
35
36         ctMethod.insertBefore("System.out.println(
37             "字节码添加成功，打印日志 !\\");");
38
39         return ctclass.toBytecode();
40     } catch (Exception e) {
41         e.printStackTrace();
42
43         System.out.println(e.getMessage());
44     }
45
46     return null;
47 }
```

项目测试打包



添加javaagent启动参数

```
1 | java -javaagent:agent路径 -jar 项目名.jar
```



启动，打印出日志

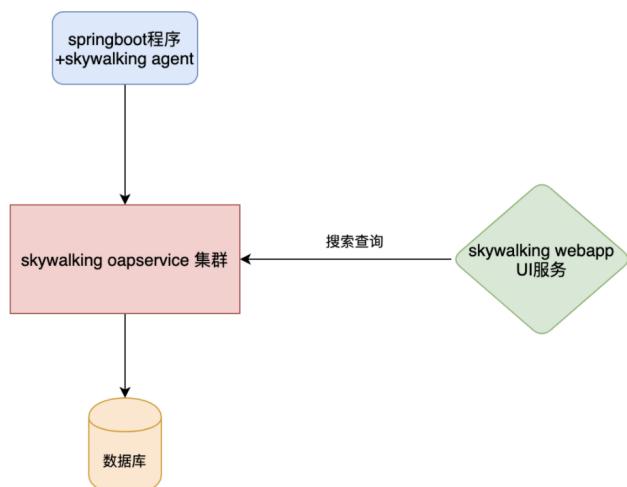
```
C:\Windows\System32\cmd.exe - java -javaagent:test-agent.jar -jar javaagentdemo2-0.0.1-SNAPSHOT.jar
V [ ] [ ] [ ] [ ( ) / ]
: Spring Boot : (v2.6.4)

18:51:55.508 INFO 2624 --- [           main] com.itbaizhan.Javaagentdemo2Application : Starting Javaagentdemo2Application using Java 1.8.0_181 on DESKTOP-637RQ4D with PID 2624 (C:\Users\wangc\Desktop\新建文件夹\javaagentdemo2-0.0.1-SNAPSHOT.jar started by wangc in C:\Users\wangc\Desktop\新建文件夹)
18:51:55.510 INFO 2624 --- [           main] com.itbaizhan.Javaagentdemo2Application : No active profile set, falling back to 1 default profile: "default"
class name 匹配上了！
ctClass is OK !
ctMethod is OK !
```

发送请求<http://localhost:8080/hello>

```
application in 0 ms
字节码添加成功，打印日志！
```

分布式请求链路追踪_SkyWalking服务环境搭建



**SkyWalking需要
安装两个服务**

下载SkyWalking包

Download

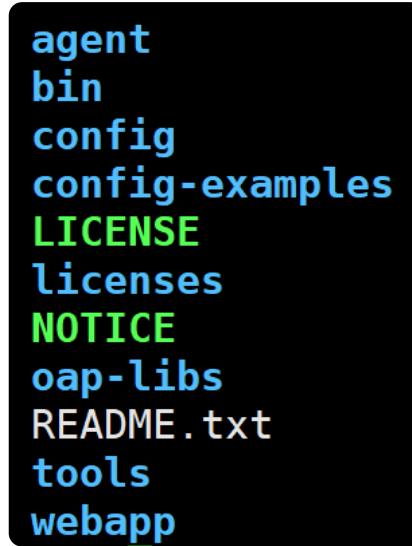
Use the links below to download Apache SkyWalking releases from one of our mirrors. Don't forget to verify the files downloaded. Please note that only source code releases are official Apache releases, binary distributions are just for end user convenience.

	Foundations	Agents	Operating	tools	Docker images
	v8.6.0 for H2/MySQL/TiDB/InfluxDB/ElasticSearch 7 [June, 10th, 2021] [tar] [asc] [sha512]				
	v8.5.0 for ElasticSearch 6 [Apr, 12th, 2021] [tar] [asc] [sha512]				
	v8.5.0 for H2/MySQL/TiDB/InfluxDB/ElasticSearch 7 [Apr, 12th, 2021] [tar] [asc] [sha512]	 2.0 for ElasticSearch 6 [Feb, 4th] [tar] [asc] [sha512]	Included in the main repo release		Deployed
					
 1					
	Agents				
	Java Agent		Nginx LUA Agent		Kong Agent
The Java Agent for Apache SkyWalking, which provides the native tracing/metrics/logging abilities for Java		SkyWalking Nginx Agent provides the native tracing capability for Nginx powered by Nginx LUA module.		SkyWalking Kong Agent provides the native tracing capability.	

解压SkyWalking包

```
1 tar -zxvf apache-skywalking-apm-es7-  
8.5.0.tar.gz -C /usr/local/
```

SkyWalking包目录介绍



介绍：

- webapp: UI前端(web 监控页面)的jar包和配置文件
 - oap-libs: 后台应用的jar包, 以及它的依赖jar包
 - config: 启动后台应用程序的配置文件, 是使用的各种配置
 - bin: 各种启动脚本, 一般使用脚本startup.*来启动web页面和对应的后台应用
 - agent: 代理服务jar包

修改端口号

vim /usr/local/apache-skywalking-apm-bin-es7/webapp/webapp.yml

```
# "License"); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

server:
  port: 8068

collector:
  path: /graphql
  ribbon:
    ReadTimeout: 10000
    # Point to all backend's restHost:restPort, split by ,
    listOfServers: 127.0.0.1:12800
```

26,0-1

Bot

启动服务

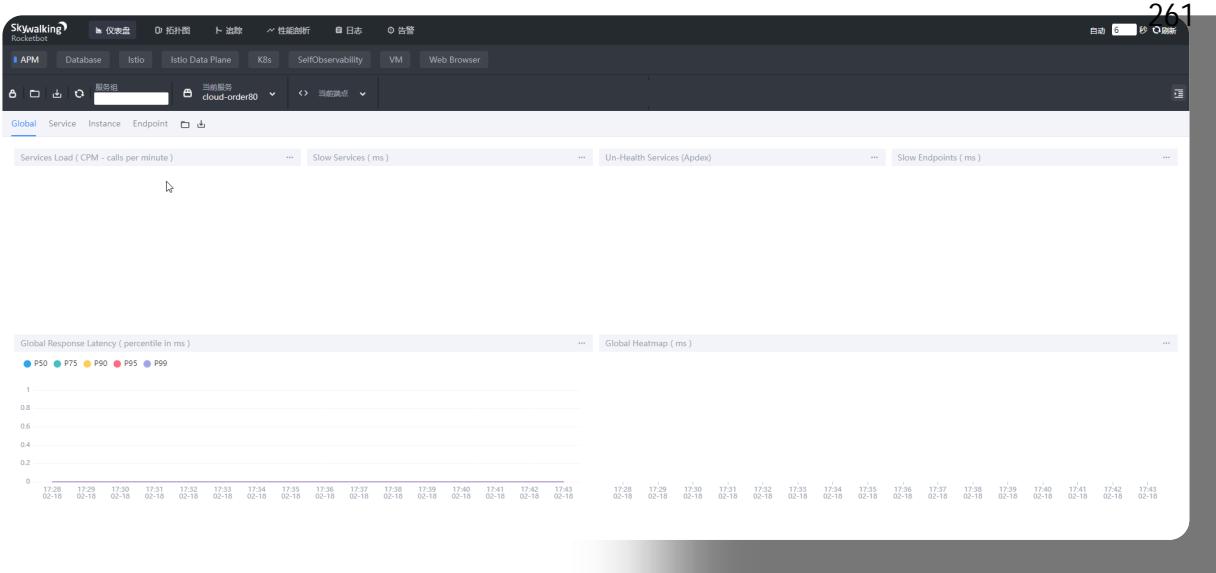
```
[root@localhost bin]# ./startup.sh
SkyWalking OAP started successfully!
SkyWalking Web Application started successfully!
[root@localhost bin]# jps
6945 OAPServerStartUp
6972 Jps
6959 skywalking-webapp.jar
```

注意：

启动成功后会启动两个服务，一个是skywalking-oap-server，一个skywalking-web-ui : 8868，skywalking-oap-server服务启动后会暴露11800和12800两个端口，分别为收集监控数据的端口11800和接受前端请求的端口12800，修改端口可以修改 config/application.yml。

测试服务

请求<http://192.168.66.100:8068>



实时效果反馈

1. 分布式链路追踪SkyWalking如何修改UI服务端口号____。

- A appcation.yml
- B webapp.yml
- C app.yml
- D 以上都是错误

2. 分布式链路追踪SkyWalking在__文件配置持久化方式。

- A appcation.yml
- B webapp.yml
- C app.yml
- D 以上都是错误

答案

1=>C 2=>A

分布式请求链路追踪_微服务接入SkyWalking探针



探针，用来收集和发送数据到归集器。

下载官方提供探针

网址<https://skywalking.apache.org/downloads/>

Skywalking

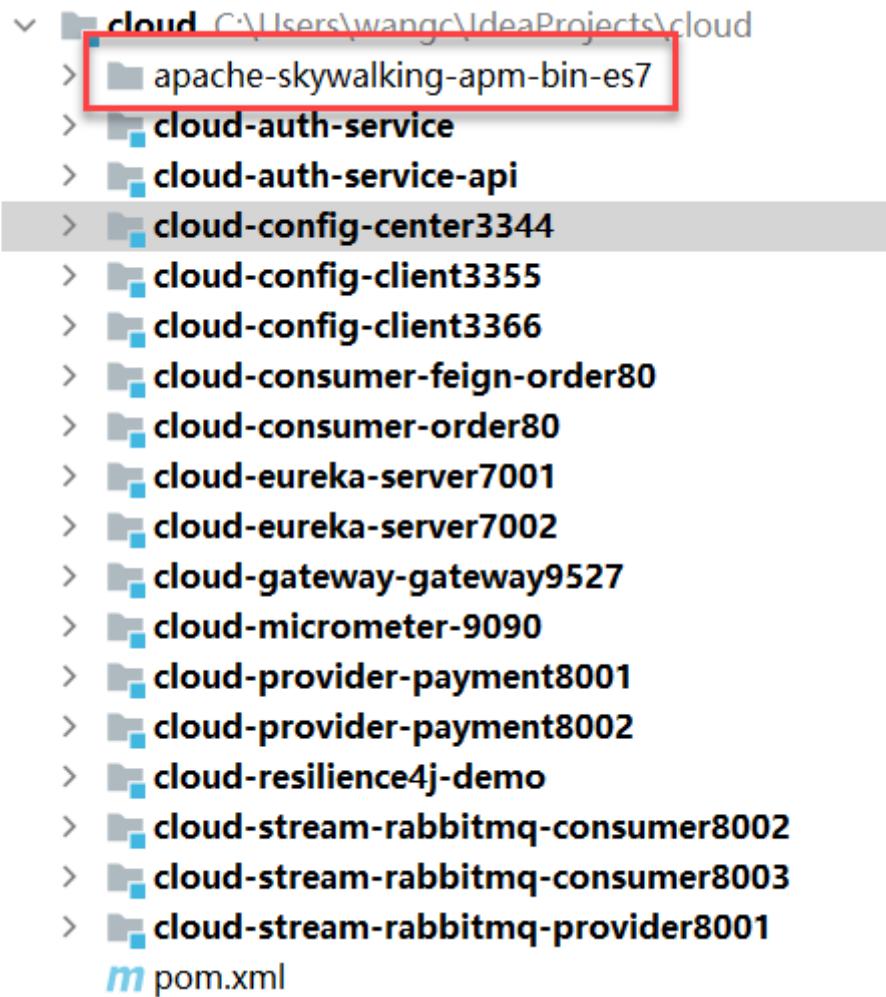
Projects and Docs Events Blogs Downloads Team Users CN 中文博客

Foundations Agents Operation tools Docker images

Agents

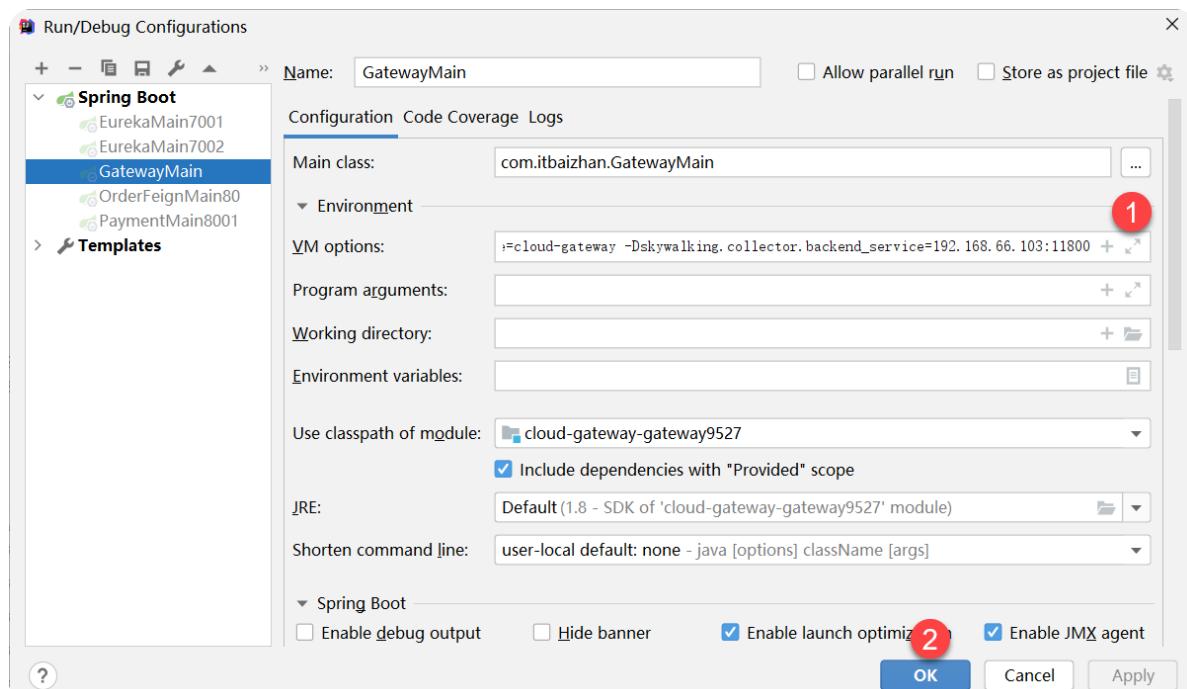
Java Agent The Java Agent for Apache SkyWalking, which provides the native tracing/metrics/logging abilities for Java projects. Source ▾ Distribution ▾ v8.9.0 [Jan. 30th, 2022] [tar] [asc] [sha512] v8.8.0 [Oct. 31th, 2021] [tar] [asc] [sha512]	Nginx LUA Agent SkyWalking Nginx Agent provides the native tracing capability for Nginx powered by Nginx LUA module. Source ▾ Distribution ▾	Kong Agent SkyWalking Kong Agent provides the native tracing capability. Source ▾ Distribution ▾	Python Agent The Python Agent for Apache SkyWalking, which provides the native tracing abilities for Python. Source ▾ Distribution ▾
NodeJS Agent The NodeJS Agent for Apache SkyWalking, which provides the native tracing abilities for NodeJS backend. Source ▾ Distribution ▾	JavaScript Agent Apache SkyWalking Client-side JavaScript exception and tracing library. Source ▾ Distribution ▾	SkyWalking Rust The Rust Agent for Apache SkyWalking, which provides the native tracing abilities for Rust project. Source ▾ Distribution ▾	SkyWalking Satellite A lightweight collector/sidebar could be deployed closing to the target monitored system, to collect metrics, traces, and logs. Source ▾ Distribution ▾

拷贝探针文件到项目中

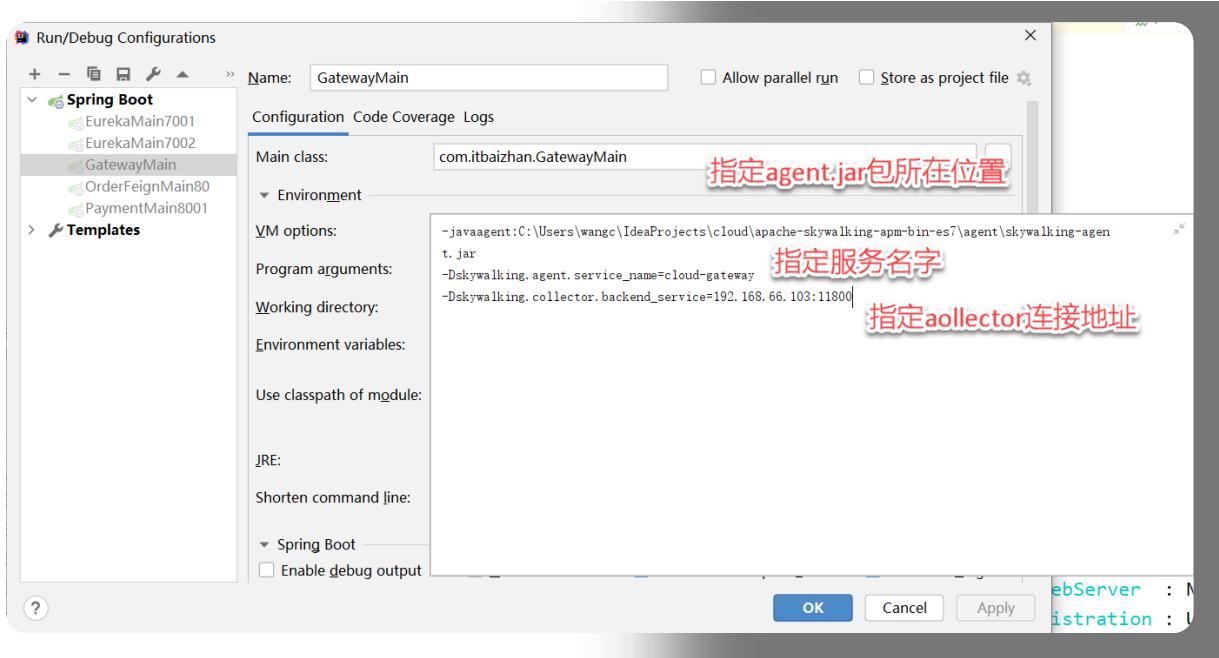


修改项目的VM运行参数

点击菜单栏中的 **Run** -> **EditConfigurations...**，此处我们以 **cloud-gateway-gateway9527** 项目为例，修改参数如下：



添加参数



```

1 java -
  javaagent:C:\Users\wangc\IdeaProjects\cloud\agent\skywalking-agent.jar
2 -DSW_AGENT_NAME=consumer-order
3 -
  DSW_AGENT_COLLECTOR_BACKEND_SERVICES=192.168.
  66.101:11800

```

参数:

- **-javaagent**: 用于指定探针路径。
- **-DSW_AGENT_NAME**: 服务名字
- **-DSW_AGENT_COLLECTOR_BACKEND_SERVICES**: 连接地址

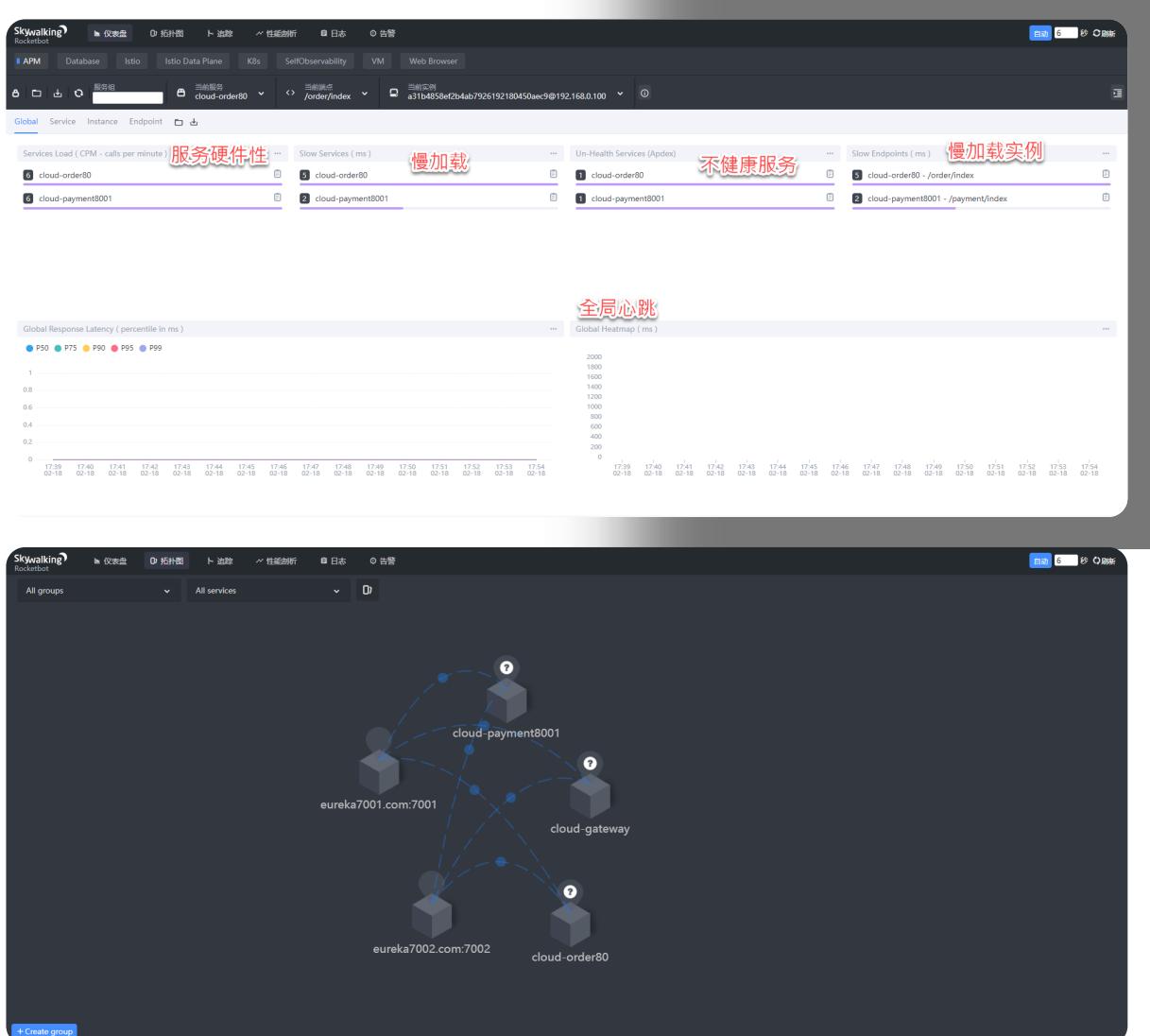
Java 命令行启动方式

```

1 java -javaagent:/path/to/skywalking-
  agent/skywalking-agent.jar -
  DSW_AGENT_NAME=nacos-provider -
  DSW_AGENT_COLLECTOR_BACKEND_SERVICES=localhost
  :11800 -jar yourApp.jar

```

测试监控



实时效果反馈

1. 微服务通过____参数接入Skywalking探针。

- A -javapath
- B -agent
- C -javaagent
- D -jar

2. 微服务通过____参数指定Skywalking服务端地址。

- A -COLLECTOR_BACKEND_SERVICES

B -DSW_AGENT_SERVICES

C -DSW_COLLECTOR_BACKEND_SERVICES

D -DSW_AGENT_COLLECTOR_BACKEND_SERVICES

答案

1=>C 2=>D

分布式请求链路追踪_Docker搭建Elasticsearch环境



拉取镜像

```
1 docker pull
docker.elastic.co/elasticsearch/elasticsearch
:7.1.1
```

启动容器

```
1 docker run -d --name es -p 9200:9200 -p
9300:9300 -e ES_JAVA_OPTS="-Xms512m -Xmx512m"
-e "discovery.type=single-node"
docker.elastic.co/elasticsearch/elasticsearch
:7.1.1
```

参数:

- ① -d: 守护进程运行

- ② ES_JAVA_OPTS: 设置堆内存
- ③ discovery.type: 设置单节点启动

测试

访问地址: <http://192.168.66.101:9200>

分布式请求链路追踪_SkyWalking使用Elasticsearch持久化



elasticsearch



Skywalking 官方推荐
使用ES

修改config目录下application.yml

/usr/local/apache-skywalking-apm-bin-es7/config

```
or_execution:
  maxSyncOperationNum: ${SW_CORE_MAX_SYNC_OPERATION_NUM:50000}
storage:
  selector: ${SW_STORAGE:mysql}
  elasticsearch:
    nameSpace: ${SW_NAMESPACE:""}
    clusterNodes: ${SW_STORAGE_ES_CLUSTER_NODES:localhost:9200}
    protocol: ${SW_STORAGE_ES_HTTP_PROTOCOL:"http"}
    user: ${SW_ES_USER:""}
    password: ${SW_ES_PASSWORD:""}
    trustStorePath: ${SW_STORAGE_ES_SSL_JKS_PATH:""}
```

修改命名空间

```
elasticsearch7:
  nameSpace: ${SW_NAMESPACE:"skywalking"}
  clusterNodes: ${SW_STORAGE_ES_CLUSTER_NODES:192.168.66.101:9200}
  protocol: ${SW_STORAGE_ES_HTTP_PROTOCOL:"http"}
  trustStorePath: ${SW_STORAGE_ES_SSL_JKS_PATH:""}
  trustStorePass: ${SW_STORAGE_ES_SSL_JKS_PASS:""}
  dayStep: ${SW_STORAGE_DAY_STEP:1} # Represent the number of days in the
```

修改ES数据连接地址

```

elasticsearch7:
  nameSpace: ${SW_NAMESPACE:""}
  clusterNodes: ${SW_STORAGE_ES_CLUSTER_NODES:192.168.66.101:9200}
  protocol: ${SW_STORAGE_ES_HTTP_PROTOCOL:"http"}
  trustStorePath: ${SW_STORAGE_ES_SSL_JKS_PATH:""}
  trustStorePass: ${SW_STORAGE_ES_SSL_JKS_PASS:""}
  dayStep: ${SW_STORAGE_DAY_STEP:1} # Represent the number of days in the o
y index.I
  indexShardsNumber: ${SW_STORAGE_ES_INDEX_SHARDS_NUMBER:1} # Shard number
  indexReplicasNumber: ${SW_STORAGE_ES_INDEX_REPLICAS_NUMBER:1} # Replicas
  ...

```

重启服务

```

1 [root@localhost bin]# ./startup.sh
2 SkyWalking OAP started successfully!
3 Skywalking Web Application started
  successfully!

```

```

[root@localhost bin]# ./startup.sh
SkyWalking OAP started successfully!
SkyWalking Web Application started successfully!

```

实时效果反馈

1. 分布式链路追踪Skywalking官方推荐_____数据库。

- A Elasticsearch
- B MySQL
- C H2
- D TiDB

答案

1=>A



当我们工程中，有些重要的方法，没有添加在链路中，而我们又需要时，就可以添加自定义链路追踪的Span。

工程cloud-provider-skywalking-payment8001引入依赖

```
1 <dependency>
2     <groupId>mysql</groupId>
3         <artifactId>mysql-connector-
4             java</artifactId>
5             <version>5.1.49</version>
6         </dependency>
7     <dependency>
8         <groupId>com.baomidou</groupId>
9             <artifactId>mybatis-plus-boot-
10            starter</artifactId>
11            <version>3.5.1</version>
12        </dependency>
13    <dependency>
14        <groupId>org.apache.skywalking</groupId>
15            <artifactId>apm-toolkit-
16            trace</artifactId>
17            <version>8.5.0</version>
```

```

15   </dependency>
16   <!--数据库连接池 HikariCP-->
17   <dependency>
18     <groupId>com.zaxxer</groupId>
19
20     <artifactId>HikariCP</artifactId>
21     <version>2.7.8</version>
22   </dependency>

```

YML文件添加配置

```

1 eureka:
2   client:
3     # 表示是否将自己注册到Eureka Server
4     register-with-eureka: true
5     # 表示是否从Eureka Server获取注册的服务信息
6     fetch-registry: true
7     # Eureka Server地址
8     service-url:
9       defaultZone:
10      http://eureka7001.com:7001/eureka,http://eure
11      ka7002.com:7002/eureka
12
13   instance:
14     instance-id: payment8003
15     prefer-ip-address: true
16
17 spring:
18   application:
19     # 设置应用名词
20     name: cloud-payment-provider
21
22     # 数据库配置
23     datasource:
24       name: sonice1024

```

```
20     driver-class-name: com.mysql.jdbc.Driver
21     url:
22       jdbc:mysql://192.168.66.101:3306/test?
23       useSSL=false&useUnicode=true&characterEncoding=utf8&useSSL=false&useTimezone=true&serverTimezone=GMT%2B8
24     username: root
25     password: 123456
26     type: com.zaxxer.hikari.HikariDataSource
27     hikari:
28       minimum-idle: 3
29       auto-commit: true
30       idle-timeout: 10000
31       max-lifetime: 1800000
32       connection-timeout: 30000
33       connection-test-query: SELECT 1
34
35   server:
36     port: 8003
```

编写主启动类

```

1 /**
2  * 主启动类
3 */
4 @EnableEurekaClient
5 @MapperScan("com.itbaizhan.mapper")
6 @SpringBootApplication
7 @Slf4j
8 public class PaymentMain8003 {
9     public static void main(String[] args) {
10
11         SpringApplication.run(PaymentMain8003.class
12         ,args);
13         log.info("***** 服务提供者启动成功
14         *****");
15     }
16 }

```

编写实体类

```

1 @Data
2 public class User {
3
4     private Long id;
5     private String name;
6     private Integer age;
7     private String email;
8
9 }

```

编写UserMapper

```

1 public interface UserMapper extends
2 BaseMapper<User> {
3 }
```

编写UserService接口

```

1 public interface UserService {
2
3     List<User> findByAllUser();
4
5 }
```

编写接口实现类

```

1 @Service
2 public class UserServiceImpl implements
3 UserService {
4
5     @Autowired
6     private UserMapper userMapper;
7
8     @Trace
9     @Override
10    public List<User> findByAllUser() {
11        return
12            userMapper.selectList(null);
13    }
14 }
```

编写用户控制层

```

1  @RequestMapping("user")
2  @RestController
3  public class UserController {
4
5      @Autowired
6      private UserService userService;
7
8      @GetMapping("findByAll")
9      public List<User> findByAll(){
10         return userService.findByAllUser();
11     }
12
13 }
```

测试

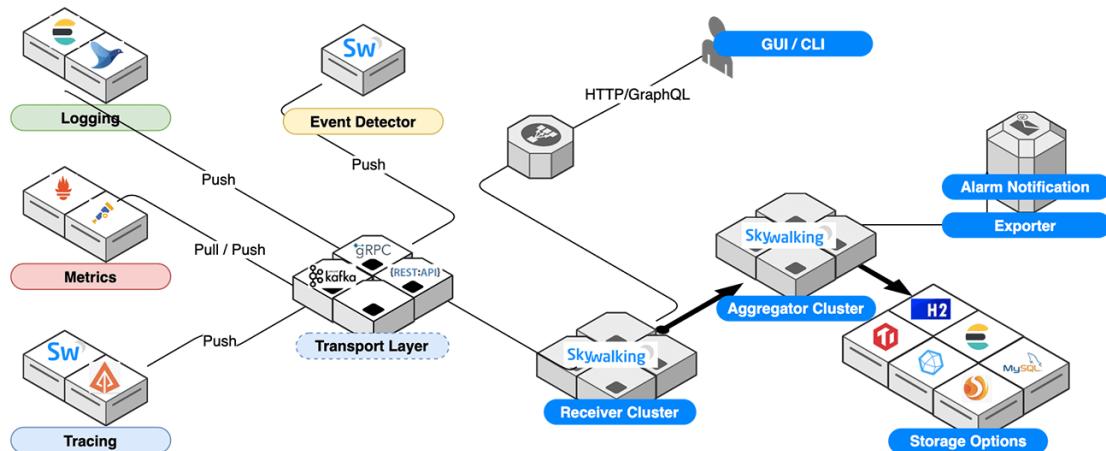
<http://localhost:8003/user/findByAll>

The screenshot shows the SkyWalking UI with the following details:

- Trace ID:** aec3a075ffbd40e78673048fa3e4a1cd@192.168.0.100
- Service:** cloud-payment8003
- Endpoint:** Mysql/JDBI/PreparedStatement/execute
- Peer:** 192.168.66.103:3306
- Failure:** false
- db.type:** sql
- db.instance:** test
- db.statement:** SELECT id,name,age,email FROM user

A blue button at the bottom says "相关的日志" (Related Log).

分布式请求链路追踪_SkyWalking日志



POM中引入相关依赖

Skywalking8.4.0版本开始才支持收集日志功能，同时pom需引用以下依赖。

```

1 <dependency>
2   <groupId>org.apache.skywalking</groupId>
3   <artifactId>apm-toolkit-logback-
4     1.x</artifactId>
5   <version>8.5.0</version>
6 </dependency>
```

Logback配置

在logback.xml中加入配置

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <configuration>
3   <appender name="console"
4     class="ch.qos.logback.core.ConsoleAppender">
5     <!-- 日志输出编码 -->
6     <encoder
7       class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
```

```

6      <layout
7          class="org.apache.skywalking.apm.toolkit.log
8              .logback.v1.x.TraceIdPatternLogbackLayout">
9                  <pattern>%d{HH:mm:ss.SSS}
10                 [%thread] %-5level logger_name:%logger{36} -
11                 [%tid] - message:%msg%n</pattern>
12             </layout>
13         </encoder>
14     </appender>
15     <root level="info">
16         <appender-ref ref="console"/>
17     </root>
18
19 </configuration>

```

Skywalking通过gRPC上报日志

gRPC报告程序可以将收集到的日志发送给Skywalking OAP服务器上。

创建logback.xml文件中添加

```

1 <appender name="log"
2     class="org.apache.skywalking.apm.toolkit.log
3         .logback.v1.x.log.GRPCLogClientAppender">
4         <!-- 日志输出编码 -->
5         <encoder
6             class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
7             <layout
8                 class="org.apache.skywalking.apm.toolkit.log
9                     .logback.v1.x.TraceIdPatternLogbackLayout">

```

```
5           <pattern>%d{HH:mm:ss.SSS}
[%thread] %-5level logger_name:%logger{36} -
[%tid] - message:%msg%n</pattern>
6           </layout>
7           </encoder>
8       </appender>
9
10
11      <root level="info">
12          <appender-ref ref="console"/>
13          <appender-ref ref="log"/>
14      </root>
```

打开你的agent/config/agent.config配置文件，添加如下配置信息，注意skywalking的log通信用的grpc：

```

1 # 指定要向其报告日志数据的grpc服务器的主机
2 plugin.toolkit.log.grpc.reporter.server_host
= ${SW_GRPC_LOG_SERVER_HOST:192.168.66.101}

3
4 # 指定要向其报告日志数据的grpc服务器的端口
5 plugin.toolkit.log.grpc.reporter.server_port
= ${SW_GRPC_LOG_SERVER_PORT:11800}

6
7 # 指定grpc客户端要报告的日志数据的最大大小
8 plugin.toolkit.log.grpc.reporter.max_message
_size=${SW_GRPC_LOG_MAX_MESSAGE_SIZE:1048576
0}

9
10 # 客户端向上游发送数据时将超时多长时间。单位是秒
11 plugin.toolkit.log.grpc.reporter.upstream_ti
meout=${SW_GRPC_LOG_GRPC_UPSTREAM_TIMEOUT:30
}

```

注意：

注：gRPC报告程序可以将收集到的日志转发到SkyWalking OAP服务器或SkyWalking Satellite卫星。

测试

The screenshot shows the SkyWalking OAP interface with the following details:

- Toolbar:** Includes tabs for Dashboard, Topology, Trace, Performance Analysis, Log, and Alert.
- Log Search Bar:** Fields for Trace ID, Time Range (03-08 18:24:49 - 03-08 18:39:49), Content Keyword, and Advanced Conditions.
- Log Table:** Displays log entries for 'provider-payment' with the following columns: Provider, Log Level, Time, Type, Content, and Level.
- Log Content Examples:**
 - level=INFO, logger=com.netflix.discovery.Disco...
 - level=INFO, logger=com.netflix.discovery.Disco...
 - level=INFO, logger=com.netflix.discovery.Disco...
 - level=INFO, logger=org.springframework.web.se...
 - level=INFO, logger=org.springframework.web.se...
 - level=INFO, logger=org.apache.catalina.core.Con...
 - level=INFO, logger=com.itbaizhan.SkywalkingPa...
 - level=INFO, logger=com.itbaizhan.SkywalkingPa...
 - level=INFO, logger=com.netflix.discovery.Disco...
 - level=INFO, logger=org.springframework.cloud...
 - level=INFO, logger=org.springframework.boot.w...
 - level=INFO, logger=org.apache.coyote.http11.H...
 - level=INFO, logger=com.netflix.discovery.Disco...

分布式请求链路追踪_SkyWalking告警



告警基本流程

每隔一段时间轮询Skywalking-collector收集到的链路追踪的数据，再根据所配置的告警规则（如服务响应时间、服务响应时间百分比）等，如果达到阈值则发送响应的告警信息。发送告警信息是以线程池异步的方式调用webhook接口完成，从而开发者可以在指定的 webhook 接口中自行编写各种告警方式，钉钉告警、邮件告警等等。

Skywalking默认支持7中通知：

web、grpc、微信、钉钉、飞书、华为weLink、slack

默认规则

Skywalking默认提供的 alarm-settings.yml，定义的告警规则如下：

- 过去3分钟内服务平均响应时间超过1秒
- 服务成功率在过去2分钟内低于80%
- 服务90%响应时间在过去3分钟内高于1000毫秒
- 服务实例在过去2分钟内的平均响应时间超过1秒
- 端点平均响应时间过去2分钟超过1秒

告警规则

- endpoint_percent_rule: 规则名称, 将会在告警消息体中展示, 必须唯一, 且以 _rule 结尾
- metrics-name: 度量名称
- include-names: 将此规则作用于匹配的实体名称上, 实体名称可以是服务名称或端点名称等
- exclude-names: 将此规则作用于不匹配的实体名称上, 实体名称可以是服务名称或端点名称等
- threshold: 阈值
- op: 操作符, 目前支持 >、<、=
- period: 多久检测一次告警规则, 即检测规则是否满足的时间窗口, 与后端开发环境匹配
- count: 在一个period窗口中, 如果实际值超过该数值将触发告警
- silence-period: 触发告警后, 在silence-period这个时间窗口中不告警, 该值默认和period相同。例如, 在时间T这个瞬间触发了某告警, 那么在(T+10)这个时间段, 不会再次触发相同告警。
- message: 告警消息体, {name} 会解析成规则名称

注意:

这些预定义的告警规则, 打开config/alarm-settings.yml文件即可看到。

Webhook

Webhook表达的意思是, 当告警发生时, 将会请求的地址URL (用POST方法)。警报消息将会以 `application/json` 格式发送出去。

举个栗子:

比如你的好友发了一条朋友圈, 后端将这条消息推送给所有其他好友的客户端, 就是 Webhook 的典型场景。

```

1  [
2    {
3      "scopeId": 1,
4      "scope": "SERVICE",
5      "name": "serviceA",
6      "id0": 12,
7      "id1": 0,
8      "ruleName": "service_resp_time_rule",
9      "alarmMessage": "alarmMessage xxxx",
10     "startTime": 1560524171000
11   ]

```

参数:

- scopeld、scope: 作用域
- name: 目标作用域下的实体名称;
- id0: 作用域下实体的ID, 与名称匹配;
- id1: 暂不使用;
- ruleName: alarm-settings.yml 中配置的规则名称;
- alarmMessage: 告警消息体;
- startTime: 告警时间 (毫秒), 时间戳形式。

实时效果反馈

1.下列不属于Skywalking告警通知的是_____。

- A 钉钉
- B 邮件
- C 微信
- D 抖音

2.Skywalking发送告警的基本原理是每隔一段时间
_____Skywalking-collector收集到的链路追踪的数据。

- A 轮询
- B 定时
- C 记录
- D 以上都是错误

答案

1=>D 2=>A

分布式请求链路追踪_Skywalking自定义告警规则



默认规则

- 过去3分钟内服务平均响应时间超过1秒
- 服务成功率在过去2分钟内低于80%
- 服务90%响应时间在过去3分钟内高于1000毫秒
- 服务实例在过去2分钟内的平均响应时间超过1秒
- 端点平均响应时间过去2分钟超过1秒

自定义告警规则

```

1 service_response_time_rule:
2     #指定的规则
3     metrics-name: service_resp_time
4     op: ">"
5     # 阈值
6     threshold: 1 # 单位毫秒
7     # 多久检查一次当前的指标数据是否符合告警规则
8     period: 5
9     # 达到多少次告警后，发送告警消息
10    count: 1
11    # 告警消息内容
12    message: 服务{name}最近5分钟以内响应时间超过了
13    1ms

```

测试

The screenshot shows the Skywalking interface with the '报警' (Alert) tab selected. The alert details are as follows:

- Date: 2022-03-09
- Time: 16:05:22
- Message: 服务provider-payment最近2分钟以内, 最近一分钟的响应时间超过了1ms
- Service: 服务 (highlighted in blue)

实时效果反馈

1. 自定义Skywalking告警规则该count参数含义是_____。

- A 告警阈值
- B 多久检查一次当前的指标数据是否符合告警规则
- C 告警消息内容
- D 达到多少次告警后，发送告警消息

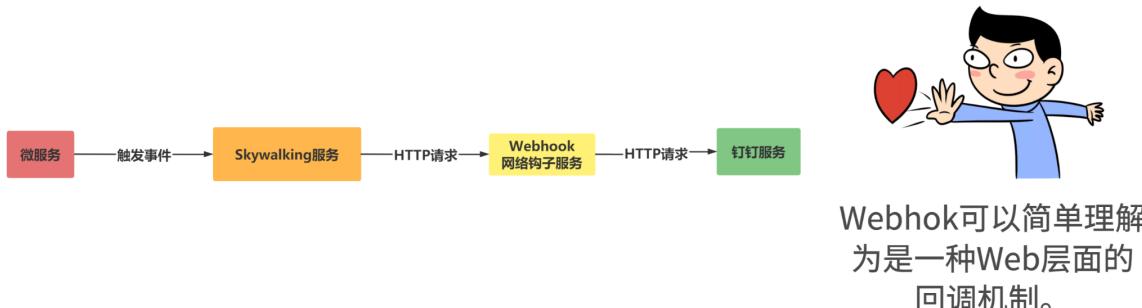
2.自定义Skywalking告警规则该threshold参数含义是____。

- A 告警阈值
- B 多久检查一次当前的指标数据是否符合告警规则
- C 告警消息内容
- D 达到多少次告警后，发送告警消息

答案

1=>B 2=>A

分布式请求链路追踪_SkyWalking网络钩子Webhooks



Webhooks网络钩子

Webhook可以简单理解为是一种Web层面的回调机制。告警就是一个事件，当事件发生时Skywalking会主动调用一个配置好的接口，这个接口就是所谓的Webhook；

注意：

Skywalking的告警消息会通过借HTTP请求进行发送,请求方法为POST (Content-Type 为application/json。其JSON数据实基于List<org.apache.skywalking.oap.server.core.alarm.AlarmMessage>进行序列化的。

JSON数据示例

```

1
2 [{ 
3     "scopeId": 1,
4     "scope": "SERVICE",
5     "name": "serviceA",
6     "id0": "12",
7     "id1": "",
8     "ruleName": "service_resp_time_rule",
9     "alarmMessage": "alarmMessage xxxx",
10    "startTime": 1560524171000
11 }]

```

创建项目cloud-alarm9090



引入依赖

```

1 <dependency>
2
3     <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-
5             web</artifactId>
6     </dependency>
7     <dependency>
8
9         <groupId>org.projectlombok</groupId>
10            <artifactId>lombok</artifactId>
11            <version>1.18.22</version>
12        </dependency>

```

创建接收实体类AlarmMessageDto

<https://github.com/apache/skywalking/blob/v8.5.0/docs/en/setup/backend/backend-alarm.md>

```

1 import lombok.Data;
2
3 @Data
4 public class AlarmMessageDto {
5
6     private int scopeId;
7     private String scope;
8     private String name;
9     private String id0;
10    private String id1;
11    private String ruleName;
12    private String alarmMessage;
13    private List<Tag> tags;
14    private long startTime;
15    private transient int period;

```

```

16     private transient boolean
onlyAsCondition;

17
18     @Data
19     public static class Tag{
20         private String key;
21         private String value;
22     }
23
24 }
25

```

编写钩子接口

```

1 /**
2 * 订单机器人通知的
3 */
4 @PostMapping("dingding")
5 public void sendDingding(@RequestBody
List<AlarmMessageDto> alarmMessageDtoList) {
6     StringBuilder builder = new
StringBuilder();
7     alarmMessageDtoList.forEach(info ->
8     {
9         builder.append("\nscopeId:")
.append(info.getScopeId())
10            .append("\nScope实
体:")
.append(info.getScope())
11            .append("\n告警消
息:")
.append(info.getAlarmMessage())
12            .append("\n告警规
则:")
.append(info.getRuleName())

```

```

12         .append("\n\n-----\n-----\n");
13     });
14 }

```

配置网络钩子

alarm-settings.yml 增加alarm接口

```

webhooks:
  - http://192.168.66.10:9090/alarm/dingding
#  - http://127.0.0.1/go-wechat/

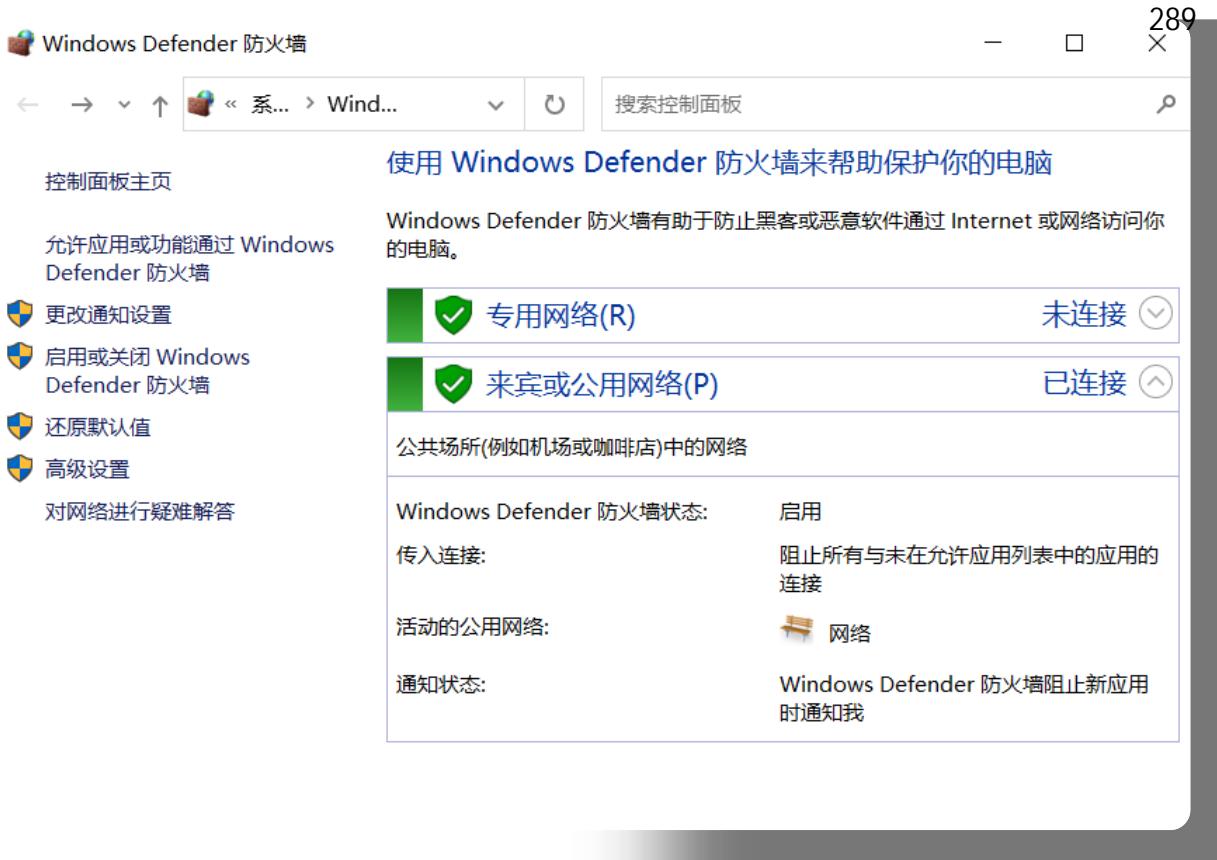
```

回调失败

关闭windows防火墙



搜索防火墙



关闭防火墙

自定义各类网络的设置

你可以修改使用的每种类型的网络的防火墙设置。

专用网络设置

- 启用 Windows Defender 防火墙
 阻止所有传入连接，包括位于允许应用列表中的应用
 Windows Defender 防火墙阻止新应用时通知我
 关闭 Windows Defender 防火墙(不推荐)

公用网络设置

- 启用 Windows Defender 防火墙
 阻止所有传入连接，包括位于允许应用列表中的应用
 Windows Defender 防火墙阻止新应用时通知我
 关闭 Windows Defender 防火墙(不推荐)

实时效果反馈

1. Webhook可以简单理解为是一种Web层面的_____机制。

A 回调

- B 请求
- C 告警
- D 以上都是错误

答案

1=>A

分布式请求链路追踪_SkyWalking钉钉告警



前言

The screenshot shows the SkyWalking interface with a dark header bar. The header includes the SkyWalking logo, navigation links for 仪表盘 (Dashboard), 拓扑图 (Topology), 追踪 (Tracing), 性能剖析 (Performance Analysis), 日志 (Logs), and 告警 (Alerts). Below the header is a search bar labeled '关键字搜索' (Key Word Search) and a pagination area showing page 1 of 1. The main content area displays a single alert entry:

时间	告警信息
03-09 16:23:31	服务 provider-payment 最近5分钟以内响应时间超过了1ms [服务]

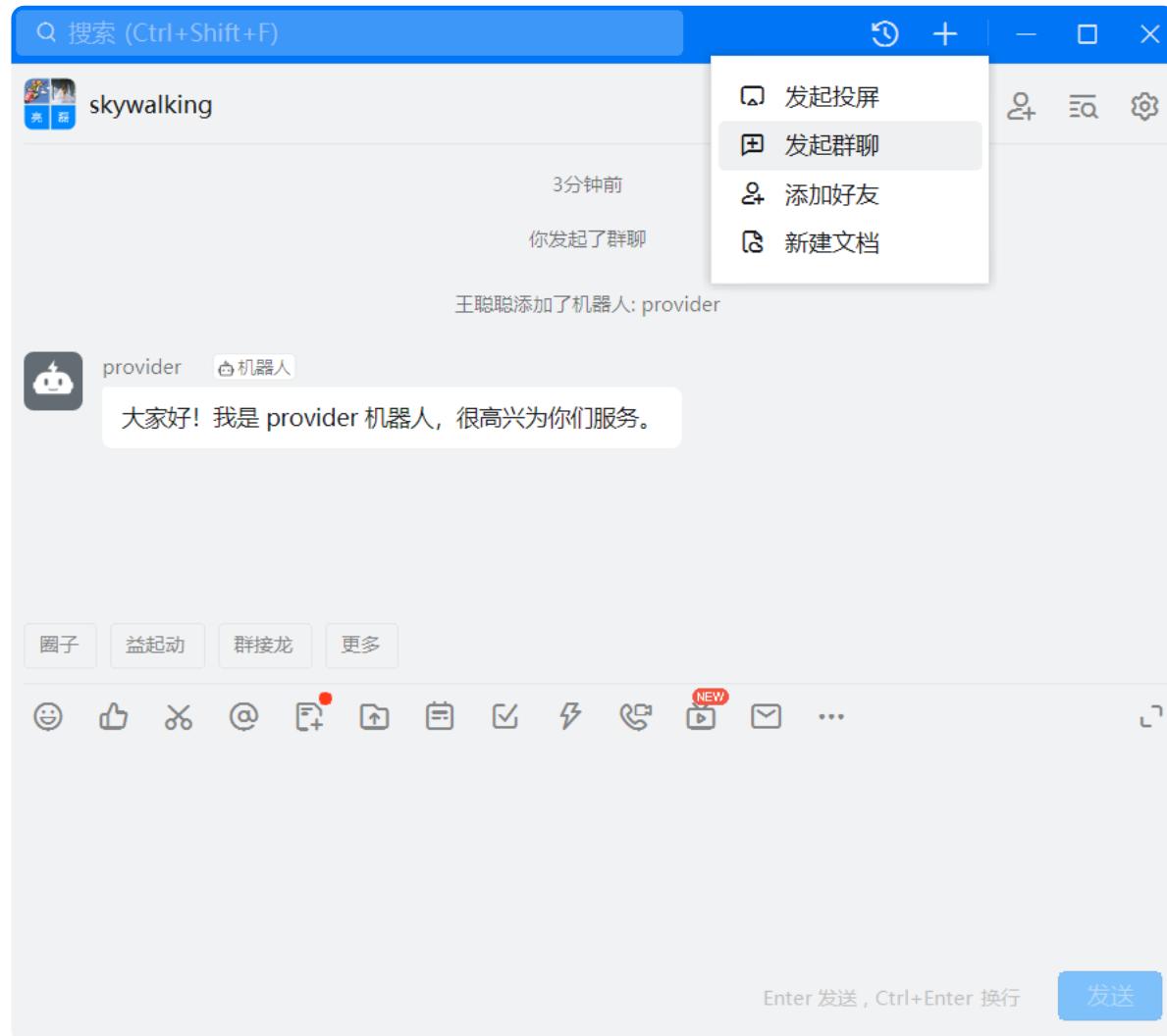
缺点：

实际项目中，我们不会一直看着告警菜单。希望有告警信息产生时，将告警信息通过邮件或者短信发送给相关负责人。



钉钉告警

创建群聊



添加智能助手

群设置

X



skywalking



普通群

群成员 4人



设置群身份



王聪聪



梅景瑶



郭心亮



雷磊

群管理



升级群



智能群助手



我在本群的昵称

未设置



置顶聊天

添加机器人



群机器人

钉钉机器人可以把你需要的消息及通知，自动推送到钉钉群 [了解更多](#)

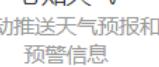
 添加机器人
目前群里最多添加 10 个机器人 +

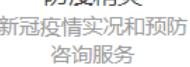
本群的机器人

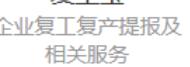
 小钉
由 王聪聪 添加，接受群组：skywalking

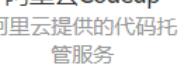
选择机器人

群机器人

 自动推送天气预报和预警信息

 新冠疫情实况和预防咨询服务

 企业复工复产提报及相关服务

 阿里云提供的代码托管服务

 GitHub
基于Git的代码托管服务

 极狐GitLab
基于ROR的开源代码托管软件

 JIRA
出色的项目与事务跟踪工具

 Travis
出色的项目与事务跟踪工具

 Trello
实时的卡片墙，管理任何事情

 自定义
通过Webhook接入自定义服务

企业机器人

配置加签



添加机器人

机器人名字: Skywalking监控报警

* 添加到群组: skywalking

* 安全设置 ? 自定义关键词

[说明文档](#) 加签

SECc5914063558dfffab1eb74107dcf6b1

密钥如上，签名方法请参考 [说明文档](#)

IP地址 (段)

我已阅读并同意 [《自定义机器人服务及免责条款》](#)

POM引入钉钉工具包依赖

```

1 <!--钉钉工具包-->
2 <dependency>
3   <groupId>com.aliyun</groupId>
4     <artifactId>alibaba-dingtalk-service-
5       <version>2.0.0</version>
6 </dependency>

```

创建application.yml

```

1 server:
2   port: 9090
3
4
5 dingding:
6   #地址
7   webhook:
8     https://oapi.dingtalk.com/robot/send?
9       access_token=7915a428336dd933247d019420032bb7
10      2e920f459920cc581c42c61d46da7e46
11      #密钥
12      secret:
13      SECa38500986415fc1404ad36415d8846f432db49936f
14      9fb7f4d0ab5260e69ca82e

```

编写发送接口

```

1 @Slf4j
2 @RestController
3 @RequestMapping("alarm")
4 public class AlarmController {
5
6   @Value("${dingding.webhook}")
7   private String webhook;
8
9   @Value("${dingding.secret}")
10  private String secret;
11
12  /**
13   * 钉钉机器人通知
14   * @param alarmMessageList
15   */
16  @PostMapping("pushData")

```

```

17     public void alarm(@RequestBody
18         List<AlarmMessageDto> alarmMessageList) {
19             log.info("alarmMessage:{}",
20                 alarmMessageList.toString());
21             alarmMessageList.forEach(info -> {
22                 try {
23                     // 当前时间戳
24                     Long timestamp =
25                         System.currentTimeMillis();
26                     String stringToSign =
27                         timestamp + "\n" + secret;
28                     /**
29                      * Mac算法是带有密钥的消息摘要算
30                      * 法
31                      * 初始化HmacMD5摘要算法的密钥产
32                      * 生器
33                     */
34                     Mac mac =
35                         Mac.getInstance("HmacSHA256");
36                     // 初始化mac
37                     mac.init(new
38                         SecretKeySpec(secret.getBytes("UTF-8"),
39                             "HmacSHA256"));
40                     // 执行消息摘要
41                     byte[] signData =
42                         mac.doFinal(stringToSign.getBytes("UTF-8"));
43                     // 拼接签名
44                     String sign = "&timestamp="
45                         + timestamp + "&sign=" +
46                         URLEncoder.encode(new
47                             String(Base64.encodeBase64(signData)), "UTF-
48                             8");

```

```

35 // 构建钉钉发送客户端工具
36 DingTalkClient client = new
37 DefaultDingTalkClient(webhook + sign);
38 // 设置消息类型
39 OapiRobotSendRequest request
40 = new OapiRobotSendRequest();
41 request.setMsgtype("text");
42 // 设置告警信息
43 OapiRobotSendRequest.Text
44 text = new OapiRobotSendRequest.Text();
45 text.setContent("业务告警:\n"
46 + info.getAlarmMessage());
47 request.setText(text);
48 // 接受人
49 OapiRobotSendRequest.At at =
50 new OapiRobotSendRequest.At();
51 at.setAtMobiles(Arrays.asList("所有人"));
52 request.setAt(at);
53 OapiRobotSendResponse
54 response = client.execute(request);
55 } catch (Exception e) {
56     e.printStackTrace();
57 }
58 });
59 }
60 }
61 }
62 }
63 }
64 }
65 }

```

测试



百战 百机器人

业务告警:
服务provider-payment最近5分钟以内响应时间超过了1ms



百战 百机器人

业务告警:
Response time of service instance d28735d62d3b4980ab2345311ecbbe71@192.168.0.100
of provider-payment is more than 1000ms in 2 minutes of last 10 minutes



百战 百机器人 18:21

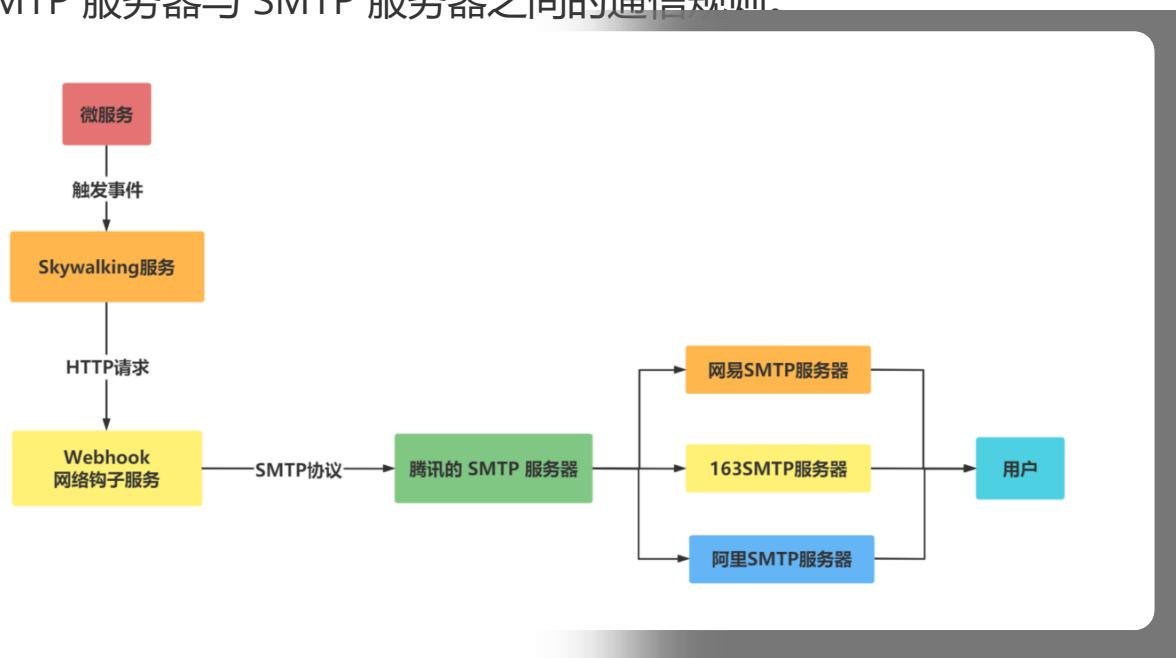
业务告警:
Response time of endpoint relation User in User to {GET}/user/findById in provider-
payment is more than 1000ms in 2 minutes of last 10 minutes

分布式请求链路追踪_SkyWalking邮件告警



邮件发送原理

SMTP 协议全称为 Simple Mail Transfer Protocol，译作简单邮件传输协议，它定义了邮件客户端软件与 SMTP 服务器之间，以及 SMTP 服务器与 SMTP 服务器之间的通信规则。



授权过程

所以在使用springboot发送邮件之前，要开启POP3和SMTP协议，需要获得邮件服务器的授权码，这里以qq邮箱为例，展示获取授权码的过程：



在账户的下面有一个开启SMTP协议的开关并进行密码验证：



成功后会出现

开启POP3/SMTP

成功开启POP3/SMTP服务,在第三方客户端
登录时,密码框请输入以下授权码:

xcxu snzd msmo bcef

取消	QQ	下一步
名称	Mail team	
电子邮件	mailteam@qq.com	
密码	*****	
描述	Mail Team	

提示: 你可拥有多个授权码, 所以无需记住该授权码, 也不要告诉其他人。[了解更多](#)

确定

POM引入依赖

```

1 <dependency>
2
3   <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-
5       mail</artifactId>
6   </dependency>

```

配置邮箱基本信息

```

1 spring:
2   mail:
3     # 配置 SMTP 服务器地址
4     host: smtp.qq.com
5     # 发送者邮箱
6     username: 877910962@qq.com
7     # 配置密码, 注意不是真正的密码, 而是刚刚申请到的
8     # 授权码
9     password: izkmheghgpvmbfeg
10    # 默认的邮件编码为UTF-8
11    default-encoding: UTF-8
12    properties:

```

```

12     mail:
13         smtp:
14             #需要验证用户名密码
15             auth: true
16             starttls:
17                 # 设置为配置SMTP连接的属性。要使用
18                 STARTTLS, 必须设置以下属性
19                     enable: true
20                     required: true

```

注意:

- 126邮箱SMTP服务器地址:smtp.126.com,端口号:465或者994
- 163邮箱SMTP服务器地址:smtp.163.com,端口号:465或者994
- yeah邮箱SMTP服务器地址:smtp.yeah.net,端口号:465或者994
- qq邮箱SMTP服务器地址: smtp.qq.com,端口号465或587

编写接口

```

1 @GetMapping("sendMail")
2     public void sendEmail(@RequestBody
3         List<AlarmMessage> alarmMessages) {
4         alarmMessages.forEach(info->{
5             SimpleMailMessage
6             simpleMailMessage = new SimpleMailMessage();
7                 // 发件人
8
8             simpleMailMessage.setFrom("877910962@qq.com
9             ");
10                // 收件人
11
12             simpleMailMessage.setTo("877910962@qq.com")
13             ;
14                 // 邮件主题

```

```
10     simpleMailMessage.setSubject(info.getScope());
11         // 邮件内容
12
13     simpleMailMessage.setText(info.getAlarmMessage());
14 }
15 }
```

实时效果反馈

1.简单的邮件协议_____。

- A RPC
- B HTTP
- C TCP
- D SMTP

答案

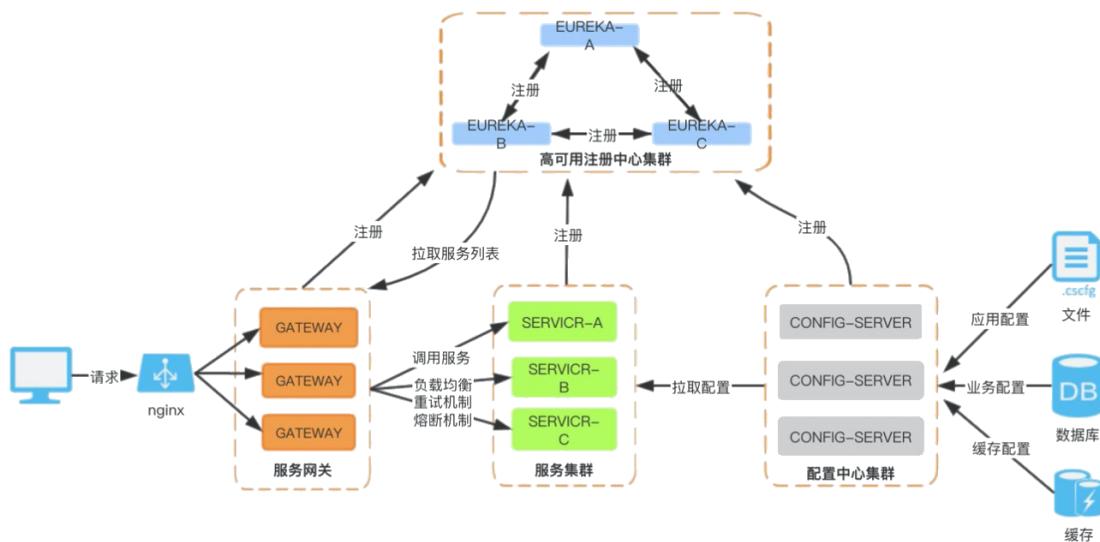
1=>D

全方位的监控告警系统_为什么需要监控系统



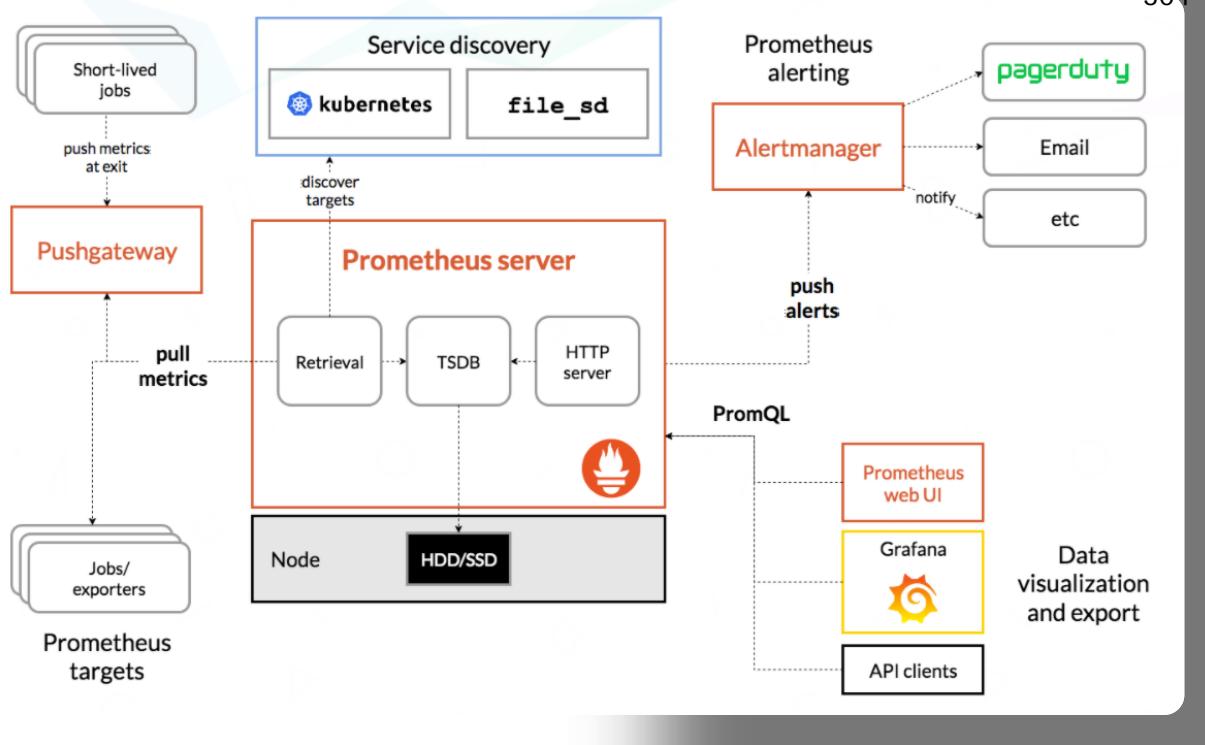
前言

一个服务上线了后，你想知道这个服务是否可用，需要监控。假如线上出故障了，你要先于顾客感知错误，你需要监控。等等各层面的监控。



什么是Prometheus

Prometheus 是一套开源的系统监控报警框架。



具体流程：

- Prometheus Server 用于定时抓取数据指标(metrics)、存储时间序列数据(TSDB)
- Jobs/exporter 收集被监控端数据并暴露指标给Prometheus
- Pushgateway 监控端的数据会用push的方式主动传给此组件，随后被Prometheus 服务定时pull此组件数据即可
- Alertmanager 报警组件，可以通过邮箱、微信等方式
- Web UI 用于多样的UI展示，一般为Grafana
- 还有一些例如配置自动发现目标的小组件和后端存储组件

Prometheus的特点

- 多维度数据模型。
- 灵活的查询语言。
- 不依赖分布式存储，单个服务器节点是自主的。
- 通过基于HTTP的pull方式采集时序数据。
- 可以通过中间网关进行时序数据推送。
- 通过服务发现或者静态配置来发现目标服务对象。
- 支持多种多样的图表和界面展示，比如Grafana等。



Grafana介绍

Grafana是一个跨平台的开源的度量分析和可视化工具，可以通过将采集的数据查询然后可视化的展示。Grafana提供了对prometheus的友好支持，各种工具帮助你构建更加炫酷的数据可视化。



Grafana特点

- 可视化：快速和灵活的客户端图形具有多种选项。面板插件为许多不同的方式可视化指标和日志。
- 报警：可视化地为最重要的指标定义警报规则。Grafana将持续评估它们，并发送通知。
- 通知：警报更改状态时，它会发出通知。接收电子邮件通知。
- 动态仪表盘：使用模板变量创建动态和可重用的仪表板，这些模板变量作为下拉菜单出现在仪表板顶部。
- 混合数据源：在同一个图中混合不同的数据源！可以根据每个查询指定数据源。这甚至适用于自定义数据源。
- 注释：注释来自不同数据源图表。将鼠标悬停在事件上可以显示完整的事件元数据和标记。
- 过滤器：过滤器允许您动态创建新的键/值过滤器，这些过滤器将自动应用于使用该数据源的所有查询。

实时效果反馈

1.全方位的监控告警系统主要作用____。

- A 监控链路
- B 监控微服务运行状况
- C 监控服务调用情况
- D 以上都是错误

2.Prometheus 是一套开源的系统____框架。

- A 服务网关
- B 服务注册
- C 监控报警
- D 服务熔断

答案

1=>B 2=>C

全方位的监控告警系统_Prometheus环境搭建



Prometheus下载

Prometheus下载地址: <https://prometheus.io/download>

File name	OS	Arch	Size	SHA256 Checksum
prometheus-2.34.0-rc.1.darwin-amd64.tar.gz	darwin	amd64	72.79 MiB	6f5112b9cd90421431ab5150a3bbc7c6a0e20d1902af2f9f5440eebdf7bc3cec
prometheus-2.34.0-rc.1.linux-amd64.tar.gz	linux	amd64	72.75 MiB	c4e60a680ddef438fe2419704a44ecd88093b190490e1f62a99616a3c38fad49
prometheus-2.34.0-rc.1.windows-amd64.zip	windows	amd64	74.10 MiB	2c62765c6363392ad611a10396d08a3251a89c1ce3ea7aa3e9bc864db2719173

解压Prometheus

```
1 | tar -zxvf prometheus-2.34.0-rc.1.linux-
   | amd64.tar.gz -C /usr/local
```

Prometheus启动服务

运行，指定prometheus.yml配置文件

```
1 | ./prometheus --config.file=prometheus.yml
```

Prometheus关闭服务

```
1 | pgrep -f prometheus
```

Prometheus验证

启动后在浏览器中访问: <http://192.168.66.101:3000/>，用户名密码默认都是admin

A screenshot of the Prometheus web interface. At the top, there's a navigation bar with links for Prometheus, Alerts, Graph, Status, and Help. On the far right, it shows the number '308' and icons for refresh, search, and help. Below the navigation is a toolbar with checkboxes for 'Use local time', 'Enable query history', 'Enable autocomplete' (which is checked), 'Enable highlighting' (which is checked), and 'Enable linter'. A search bar labeled 'Expression (press Shift+Enter for newlines)' is followed by a blue 'Execute' button. Under the search bar, there are tabs for 'Table' and 'Graph', with 'Graph' being the active tab. A date range selector shows 'Evaluation time' with arrows for navigating between dates. Below this, a message says 'No data queried yet'. In the bottom right corner of the main area, there's a 'Remove Panel' link. At the very bottom left, there's a blue 'Add Panel' button.

全方位的监控告警系统_Grafana环境搭建



下载Grafana镜像

```
1 docker pull grafana/grafana-enterprise
```

运行Grafana镜像

```
1 docker run -d --name=grafana -p 3000:3000  
grafana/grafana-enterprise
```

Grafana验证

启动后在浏览器中访问：<http://192.168.66.101:3000/>，用户名密码默认都是admin

The screenshot shows the Grafana homepage. On the left, there's a sidebar with a "Basic" section containing a "TUTORIAL" card about "DATA SOURCE AND DASHBOARDS" and "Grafana fundamentals". To the right of the sidebar are three cards: "COMPLETE" for "Add your first data source" and "Create your first dashboard", both with "Learn how in the docs" links. Below the sidebar, there are sections for "Starred dashboards" and "Recently viewed dashboards", with one entry: "1 SLS JVM监控大盘". On the right side, there's a "Latest from the blog" section with a post by "3月 10" titled "Best practices for alerting on Synthetic Monitoring metrics in Grafana Cloud".

Grafana datasource配置

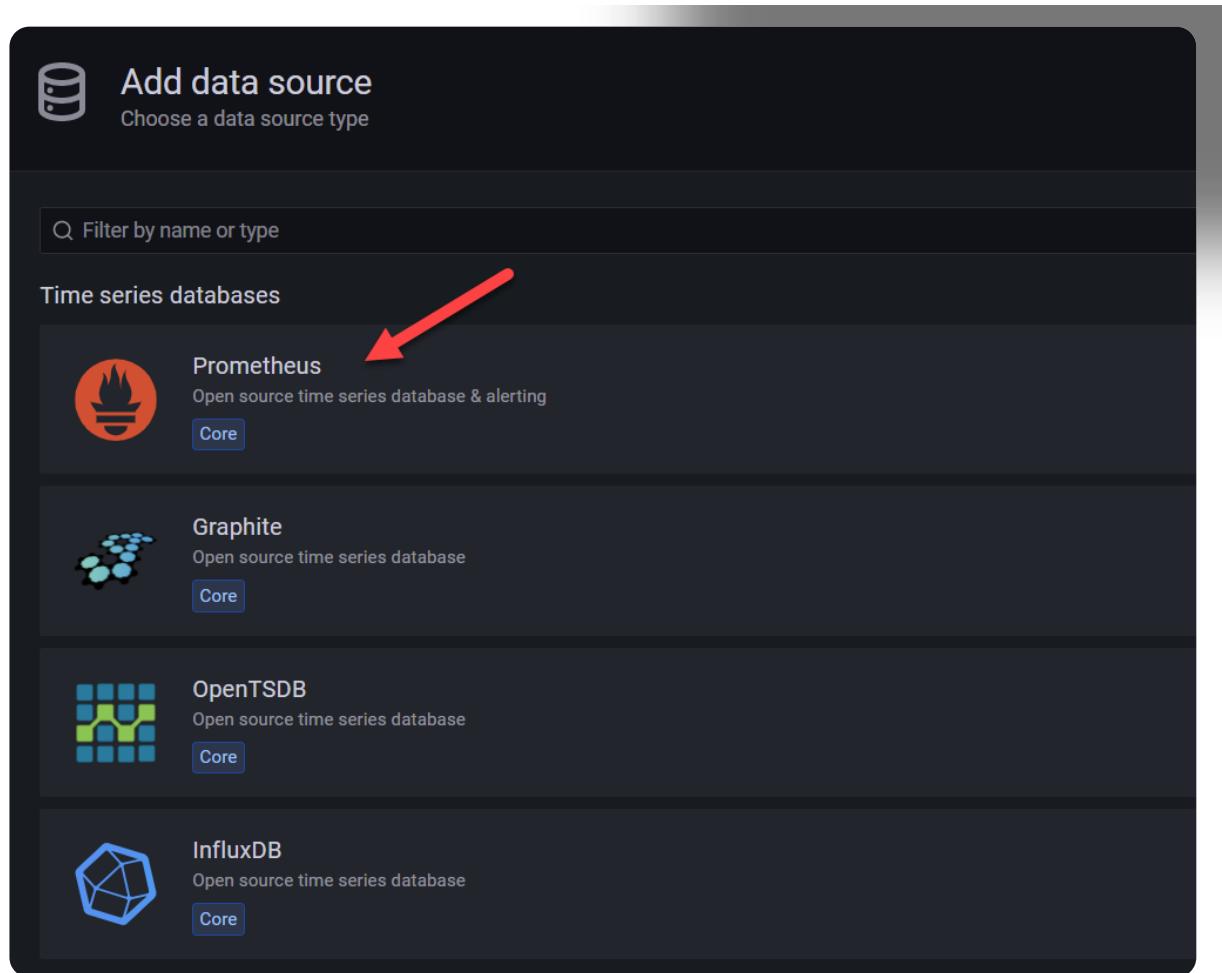
The screenshot shows the Grafana Configuration page. On the left, a sidebar menu is open, showing options like "General / Home", "Dashboards", and "Configuration". Under "Configuration", there are links for "Data sources" (marked with a red circle and number 2), "Users", "Teams", "Plugins", "Preferences", "API keys", and "Recorded queries". A red circle with the number 1 is placed over the "Configuration" link itself. The main content area displays the "Welcome to Grafana" screen with the "Basic" tutorial card visible.

进入到Data Sources配置页面

The screenshot shows the "Data sources" configuration page. At the top, there's a header with "Configuration" and "Organization: Main Org.". Below the header, a navigation bar includes "Data sources" (which is highlighted with an orange underline), "Users", "Teams", "Plugins", "Preferences", "API keys", and "Recorded queries". A green success message box says "Data source deleted" with a checkmark icon. In the main content area, it says "No data sources defined" and has a blue "Add data source" button. At the bottom, there's a note: "ProTip: You can also define data sources through configuration files. Learn more".

添加监控数据源配置

点击add data source按钮，进入添加监控数据源配置页面，数据源类型选择Prometheus（因为Grafana UI的数据来源于Prometheus）



配置Prometheus地址

Data Sources / Prometheus

Type: Prometheus

Settings Dashboards

Name: Prometheus Default

HTTP 配置普罗米修斯地址

URL: http://192.168.66.101:9090

Access: Server (default) Help >

Allowed cookies: New tag (enter key to add)

Timeout: Timeout in seconds

Auth

Basic auth: Off With Credentials: Off

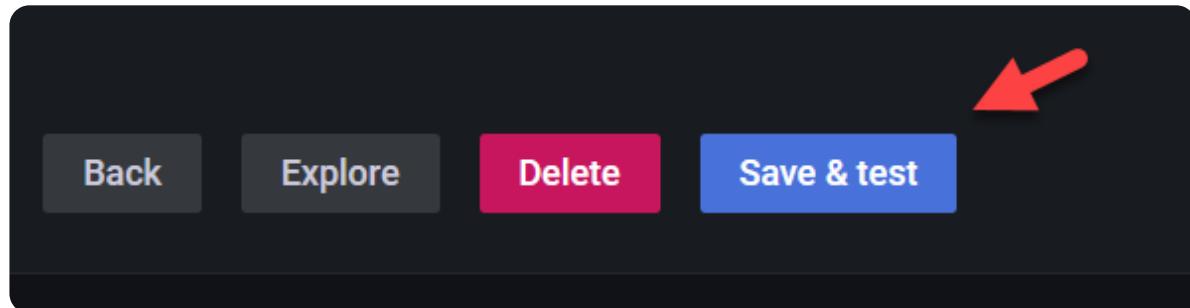
TLS Client Auth: Off With CA Cert: Off

Skip TLS Verify: On

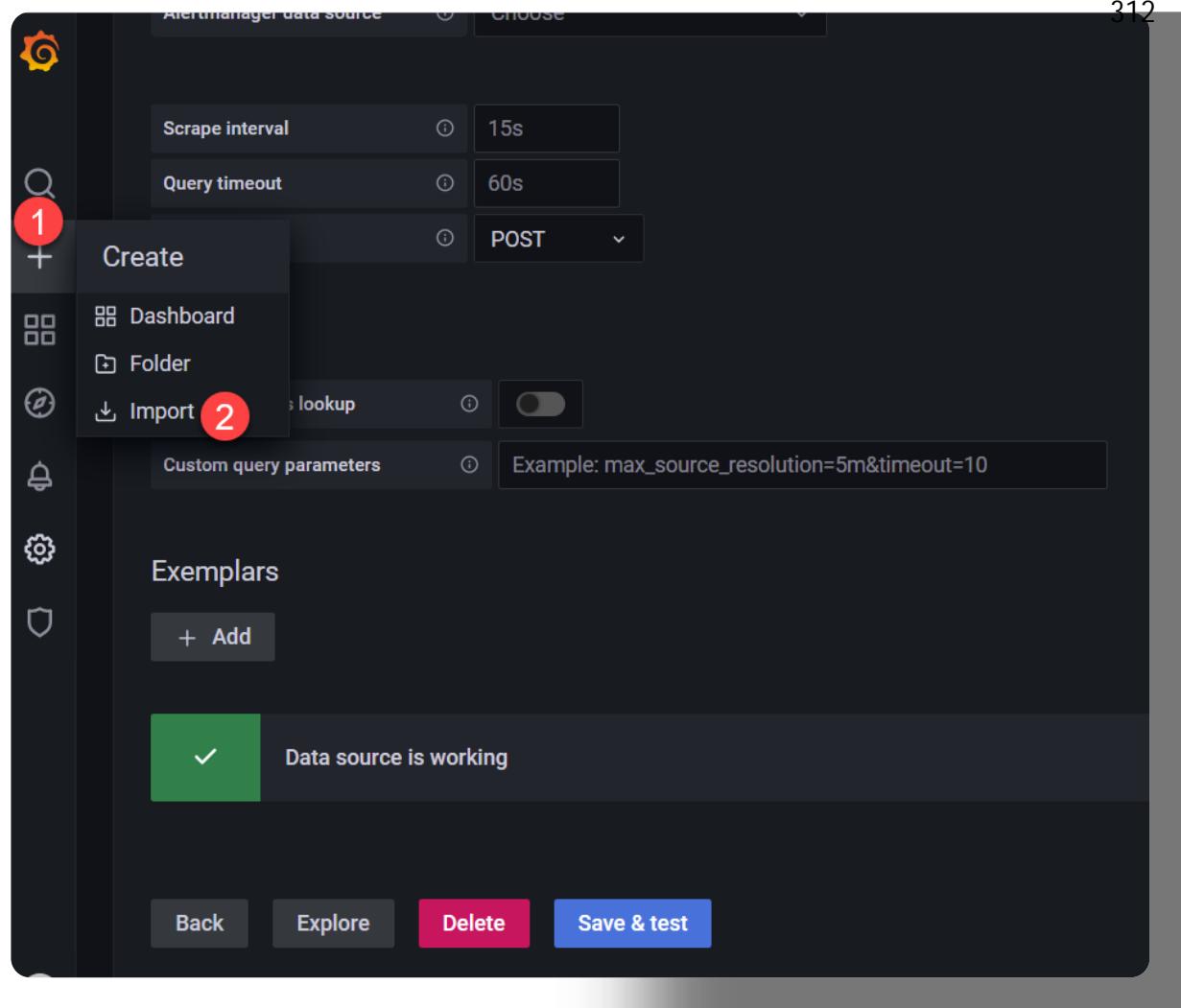
Forward OAuth Identity: Off

保存配置

填写URL后，点击页面最下方的save & Test按钮



点击左侧import按钮



配置Grafana模板

在输入框中填写12856，然后点击load。12856是Grafana模板ID，
更多模板请参考：<https://grafana.com/grafana/dashboards>

 Import
Import dashboard from file or Grafana.com

 Upload JSON file

12856是Grafana模板ID

Import via grafana.com
12856 

Import via panel json



选择数据源对象

 Import
Import dashboard from file or Grafana.com

Importing dashboard from [Grafana.com](#)

Published by

Updated on

Options

Name

Folder

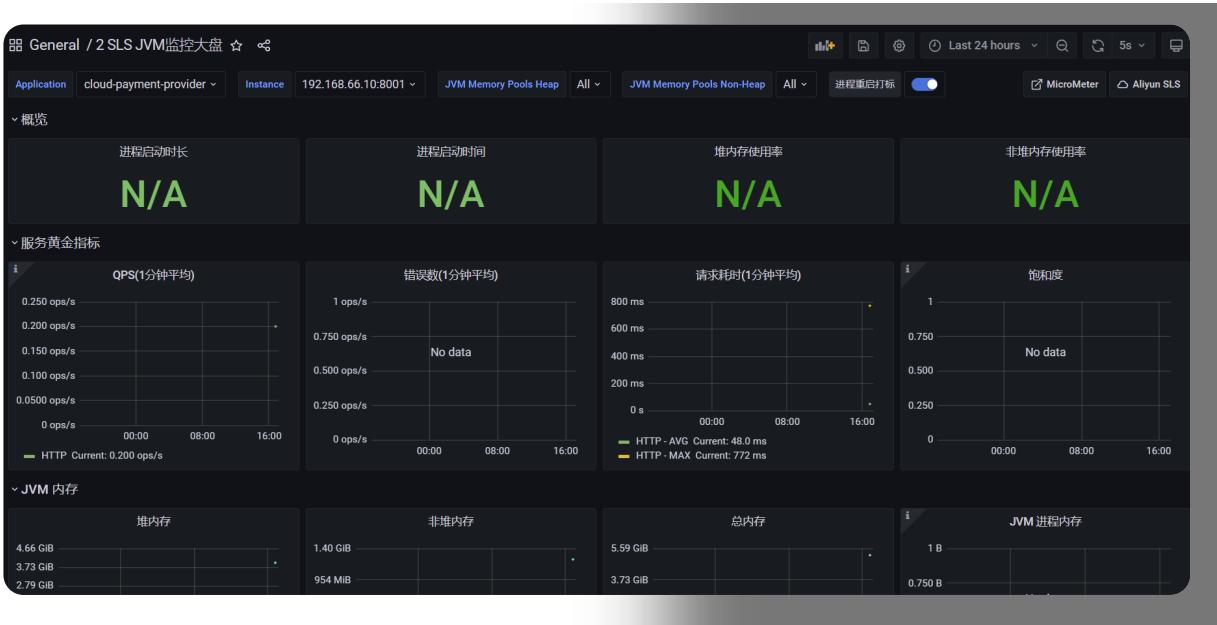
Unique identifier (UID)
The unique identifier (UID) of a dashboard can be used for uniquely identify a dashboard between multiple Grafana installs. The UID allows having consistent URLs for accessing dashboards so changing the title of a dashboard will not break any bookmarked links to that dashboard.


 Prometheus

 Import  Cancel

进入DashBoard

就可以查看JVM监控大盘了



全方位的监控告警系统_微服务应用接入监控



创建工程cloud-provider-prometheus-payment8001

The screenshot shows the IntelliJ IDEA 'New Project' dialog. The 'Parent' dropdown is set to 'cloud'. The 'Name' field contains 'cloud-provider-prometheus-payment8001'. The 'Location' field shows the path 'C:\Users\user\IdeaProjects\cloud\cloud-provider-prometheus-payment8001'. Below the location field is a collapsed section titled 'Artifact Coordinates'.

引入以来

```

1 <dependency>
2   <groupId>io.micrometer</groupId>
3     <artifactId>micrometer-registry-
4       prometheus</artifactId>
5   </dependency>

```

修改application.yml配置文件

```

1 management:
2   endpoints:
3     web:
4       exposure:
5         include: '*'
6     endpoint:
7       health:
8         show-details: ALWAYS
9   metrics:
10    tags:
11      application:
12        ${spring.application.name}

```

注意：

management.endpoints.web.exposure.include=* 配置为开启 Actuator 服务，因为 Spring Boot Actuator 会自动配置一个 URL 为 /actuator/Prometheus 的 HTTP 服务来供 Prometheus 抓取数据，不过默认该服务是关闭的，该配置将打开所有的 Actuator 服务。

management.metrics.tags.application 配置会将该工程应用名称添加到 Grafana UI，方便后边根据应用名称来区分不同的服务。

修改主启动类

```

1  @EnableEurekaClient
2  @SpringBootApplication
3  @Slf4j
4  public class PaymentMain8001 {
5      public static void main(String[] args) {
6
7          SpringApplication.run(PaymentMain8001.class
8          ,args);
9          log.info("*****PaymentMain8001
10         服务启动成功 *****");
11     }
12
13     @Bean
14     MeterRegistryCustomizer<MeterRegistry>
15     configurer(@value("${spring.application.name
16     }") String applicationName) {
17         return registry ->
18             registry.config().commonTags("application",
19             applicationName);
20     }
21
22 }

```

关闭Prometheus服务

因为 Prometheus 是一个 Unix [二进制](#)程序，我们可以向 Prometheus 进程发送 [SIGTERM](#) 关闭信号。

- 1 使用 `pgrep -f prometheus` 找到运行的 Prometheus 进程号
- 2 使用 `kill -9 1234` 来关闭

修改prometheus.yml配置文件

```

1   - job_name: "payment-provider"
2     scrape_interval: 5s
3     metrics_path: "/actuator/prometheus/"
4     static_configs:
5       - targets: ["192.168.66.10:8001"]

```

启动Prometheus服务

```
1 ./prometheus --config.file=prometheus.yml
```

查看微服务大盘

