

# 目录

微服务.....	11
谈谈你对微服务的理解.....	11
Spring Cloud.....	12
Spring Cloud 技术栈.....	14
Spring Cloud 版本选型.....	15
SpringBoot2.X 版 和 SpringCloud H 版 .....	15
SpringBoot 和 SpringCloud 版本约束 .....	16
关于 Cloud 各种组件的停更/升级/替换.....	16
停更引发的升级惨案.....	16
明细条目 .....	17
Eureka 集群 .....	18
Eureka 工作原理.....	18
Eureka 集群原理.....	19
搭建集群.....	19
服务注册 Eureka 集群.....	21
服务提供者集群 .....	21
actuator 微服务信息完善.....	24
服务名称修改 .....	24
设置服务的 IP 显示.....	24
服务发现 Discovery .....	25
Eureka 自我保护机制 .....	25
概念 .....	25
导致原因 .....	26
禁止自我保护 .....	27
关于 Eureka 停更.....	27
Eureka 停更后的替换.....	28
Zookeeper 替换 Eureka.....	28
Zookeeper 是什么 .....	28
搭建 Zookeeper 注册中心.....	28
Consul 替换 Eureka .....	30

简介	30
功能	31
安装	31
服务提供者注册 Consul	31
总结	32
CAP 理论	32
AP 架构	33
CP 架构	34
Ribbon 实现负载均衡	35
概念	35
LB 负载均衡是什么	36
Ribbon 本地负载均衡客户端 VS Nginx 服务端负载均衡	36
Ribbon 工作原理	36
引入 Ribbon	37
RestTemplate	38
Ribbon 核心组件 IRule	39
负载均衡算法	39
默认负载均衡算法替换	40
手写 Ribbon 负载均衡算法	41
原理	41
源码	41
手写负载均衡算法	42
OpenFeign 实现服务调用	44
概述	45
Feign 的作用	45
Feign 集成 Ribbon	45
Feign 和 OpenFeign 的区别	45
OpenFeign 使用步骤	46
引入依赖	46
修改启动类	46
添加业务逻辑接口	46

具体使用 .....	46
OpenFeign 的超时控制 .....	47
OpenFeign 日志打印功能.....	47
概念.....	47
日志级别 .....	47
修改 YML 文件 .....	48
Hystrix 断路器 .....	48
概述.....	48
分布式面临的问题.....	48
服务雪崩 .....	49
HyStrix 的诞生 .....	49
Hystrix 作用 .....	50
Hystrix 重要概念 .....	50
服务降级 .....	50
服务熔断.....	50
服务限流 .....	50
Hystrix 案例 .....	51
构建.....	51
高并发测试 .....	53
服务消费者加入.....	54
解决方案.....	56
原因.....	56
服务降级 .....	56
服务熔断 .....	60
总结 .....	62
服务限流 .....	64
Hystrix 工作流程 .....	64
服务监控 HystrixDashboard.....	65
概述.....	65
搭建.....	65
使用监控 .....	67

服务网关.....	70
前言.....	70
概念.....	70
Zuul .....	70
Gateway.....	70
能做啥.....	72
使用场景.....	73
为什么选用 Gateway.....	73
Gateway 特性 .....	73
Spring Cloud Gateway 和 Zuul 的区别.....	74
WebFlux 框架.....	75
三大核心概念 .....	75
Route 路由.....	75
Predicate 断言 .....	76
Filter 过滤 .....	76
Gateway 工作流程.....	76
入门配置.....	77
引入依赖.....	77
修改 YML .....	78
访问.....	78
路由匹配.....	79
路由配置的两种方式.....	79
通过微服务名实现动态路由 .....	79
Predicate 的使用.....	80
概念.....	80
常用的 Predicate .....	81
Filter 的使用 .....	82
概念.....	82
Spring Cloud Gateway Filter.....	82
自定义过滤器 .....	82
分布式配置中心 SpringCloudConfig.....	83
概述.....	83

面临的问题 .....	83
是什么 .....	83
怎么玩 .....	84
能做什么 .....	84
与 Github 整合部署 .....	84
Config 服务端配置与测试 .....	85
引入依赖 .....	85
修改 YML .....	85
配置读取规则 .....	86
参数总结 .....	86
Config 客户端配置与测试 .....	86
引入依赖 .....	86
bootstrap.yml .....	86
存在的问题 .....	87
Config 客户端之动态刷新 .....	88
引入了动态刷新 .....	88
消息总线 SpringCloudBUS .....	89
概述 .....	89
什么是总线 .....	90
基本原理 .....	90
RabbitMQ 环境配置 .....	91
SpringCloudBus 动态刷新全局广播 .....	91
配置 .....	91
设计思想 .....	92
服务端添加消息总线 .....	93
引入依赖 .....	93
增加配置 .....	93
客户端引入消息总线支持 .....	94
引入依赖 .....	94
修改 yml .....	94
测试 .....	95

SpringCloudBus 动态刷新定点通知 .....	95
案例.....	95
SpringCloud Stream 消息驱动 .....	96
为什么引入消息驱动? .....	96
消息驱动概述 .....	97
是什么 .....	97
什么是 SpringCloudStream.....	97
SpringCloudStrem 设计思想.....	98
案例说明 .....	102
消息驱动之生产者.....	102
引入依赖.....	102
修改 yml.....	102
业务类.....	103
消息驱动之消费者 .....	105
引入依赖.....	105
修改 yml.....	105
业务类.....	106
分组消费 .....	106
运行后有两个问题.....	106
消费 .....	106
分组.....	108
消息持久化 .....	110
案例.....	110
SpringCloudSleuth 分布式请求链路跟踪.....	111
概述.....	111
搭建.....	112
zipkin.....	112
名词解释 .....	113
引入依赖 .....	114
修改 yml.....	114
Nacos.....	114

SpringCloud Alibaba 简介 .....	114
维护模式 .....	114
意味着 .....	114
诞生 .....	115
能做啥 .....	116
引入依赖版本控制 .....	116
怎么玩 .....	117
Nacos 简介 .....	117
为什么叫 Nacos .....	117
是什么 .....	117
能干嘛 .....	117
下载 .....	118
比较 .....	118
安装并运行 .....	118
Nacos 作为服务注册中心 .....	119
服务提供者注册 Nacos .....	119
服务消费者注册到 Nacos .....	123
服务中心对比 .....	124
Nacos 作为服务配置中心演示 .....	127
Nacos 作为配置中心 - 基础配置 .....	127
Nacos 作为配置中心 - 分类配置 .....	131
Nacos 集群和持久化配置 .....	136
官网说明 .....	136
单机模式支持 mysql .....	137
Nacos 持久化配置解释 .....	138
Linux 版 Nacos + Mysql 生产环境配置 .....	139
SpringCloudAlibabaSentinel 实现熔断和限流 .....	143
Sentinel .....	143
官网 .....	143
是什么 .....	144
主要特征 .....	145

生态圈.....	145
下载.....	145
安装 Sentinel 控制台.....	146
初始化演示工程.....	147
引入依赖.....	147
修改 YML.....	148
增加业务类.....	148
流控规则.....	149
基本介绍.....	149
流控模式.....	150
流控效果.....	156
降级规则.....	160
名词介绍.....	160
概念.....	161
降级策略实战.....	162
Sentinel 热点规则.....	168
什么是热点数据.....	168
兜底的方法.....	168
配置.....	168
参数例外项.....	170
结语.....	171
Sentinel 系统配置.....	172
@SentinelResource 注解.....	173
问题.....	173
客户自定义限流处理逻辑.....	173
更多注解属性说明.....	174
服务熔断.....	175
不设置任何参数.....	175
设置 fallback .....	175
设置 blockHandler.....	176
blockHandler 和 fallback 一起配置.....	176

异常忽略 .....	177
Feign 系列 .....	177
熔断框架对比 .....	179
Sentinel 规则持久化 .....	180
是什么 .....	180
怎么玩 .....	180
解决方法 .....	180
SpringCloudAlibabaSeata 处理分布式事务 .....	183
分布式事务 .....	183
Seata 简介 .....	184
处理过程 .....	185
下载 .....	185
怎么玩 .....	188
订单/库存/账户业务微服务准备 .....	189
分布式事务的业务说明 .....	189
创建数据库 .....	189
建立业务表 .....	190
创建回滚日志表 .....	191
订单/库存/账户业务微服务准备 .....	191
业务需求 .....	191
新建 Order-Module 表 .....	191
新建 Storage-Module .....	202
新建账户 Account-Module .....	202
测试 .....	202
一部分补充 .....	204
Seata .....	204
再看 TC/TM/RM 三大组件 .....	205
分布式事务的执行流程 .....	206
AT 模式如何做到对业务的无侵入 .....	206
AT 模式 .....	206



## 微服务

### 谈谈你对微服务的理解

微服务架构下的一整套解决方案

- 服务注册与发现
- 服务调用
- 服务熔断
- 负载均衡
- 服务降级
- 服务消息队列
- 配置中心
- 服务网关
- 服务监控
- 全链路追踪
- 自动化构建部署
- 服务定时任务调度操作



## Spring Cloud

分布式微服务架构的一站式解决方案，是多种微服务架构落地技术的集合体，俗称微服务全家桶

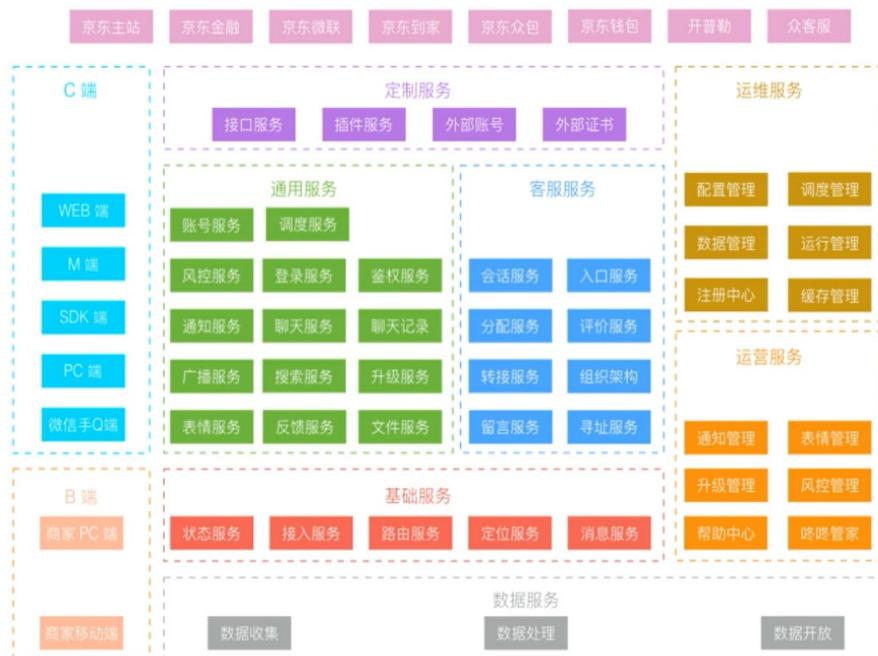
下面一张图是京东的促销架构



阿里的架构图：



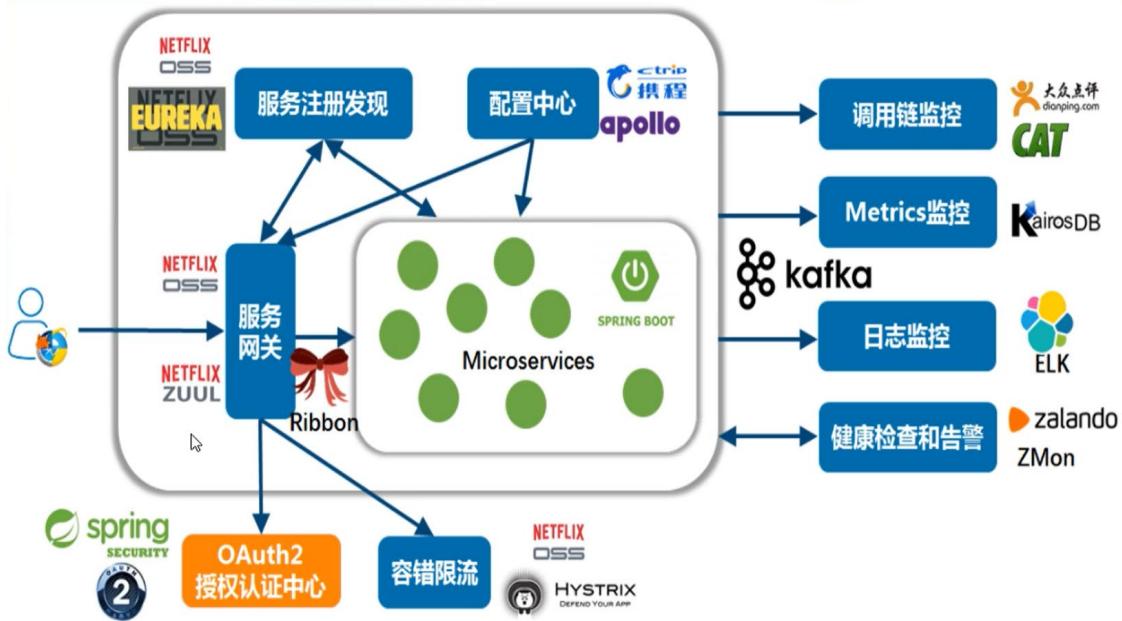
京东物流的架构图：



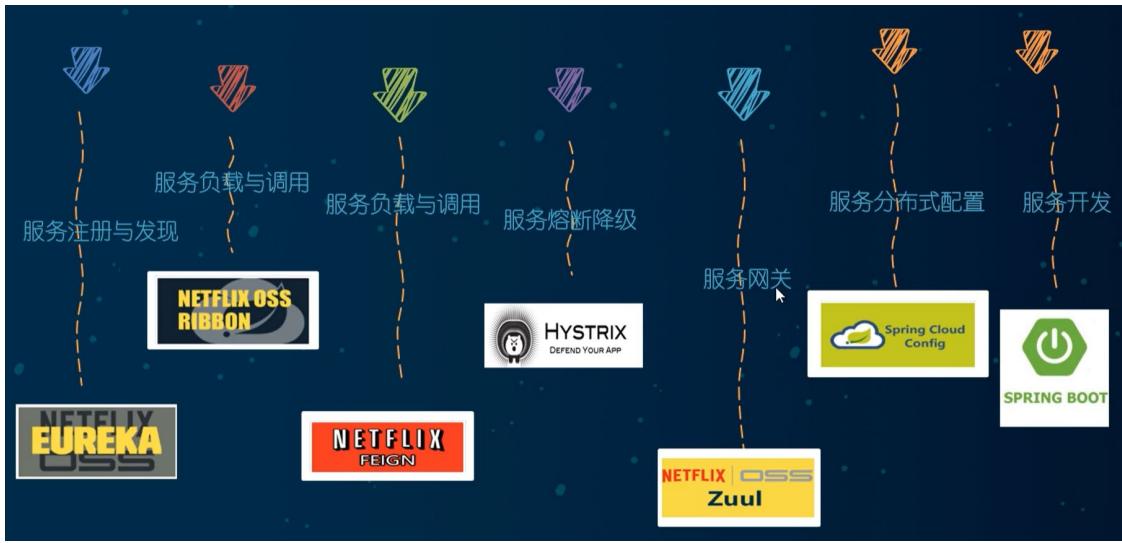
基础服务：



## Spring Cloud 技术栈



这是原来 2020 年以前的微服务方案



但是随着 Eureka 等组件的闭源，后续的一些解决方案也有了新的替换产品

## Spring Cloud 版本选型

### SpringBoot2.X 版 和 SpringCloud H 版

SpringBoot 官方已经强烈推荐 2.X 版

SpringCloud 采用英国伦敦地铁站的名称来命名，并由地铁站名称字母 A-Z 一次类推的形式发布迭代版本

SpringCloud 是由许多子项目组成的综合项目，各子项目有不同的发布节奏，为了管理 SpringCloud 与各子项目的版本依赖关系，发布了一个清单，其中包括了某个 SpringCloud 版对应的子项目版本，为了避免 SpringCloud 版本号与子项目版本号混淆，SpringCloud 版采用了名称而非版本号命名。例如 Angel, Brixton。当 SpringCloud 的发布内容积累到临界点或者一个重大 BUG 被解决后，会发布一个 Service releases 版本，俗称 SRX 版本，比如 Greenwich.SR2 就是 SpringCloud 发布的 Greenwich 版本的第二个 SRX 版本

The screenshot shows the official Spring website at [spring.io/projects/spring-cloud](https://spring.io/projects/spring-cloud). The main navigation bar includes links for Why Spring, Learn, Projects, Training, Support, and Community. On the left, a sidebar lists projects such as Spring Boot, Spring Framework, Spring Data, and Spring Cloud. The Spring Cloud section is currently selected. The main content area is titled "Spring Cloud Hoxton SR1". It features three tabs: "OVERVIEW", "LEARN" (which is highlighted in green), and "SAMPLES". Below the tabs, there's a section titled "Documentation" with a brief description. At the bottom, there are links for "Hoxton SR1 CURRENT GA", "Reference Doc.", and "API Doc.". A red box highlights the "Spring Cloud Hoxton SR1" title.

## SpringBoot 和 SpringCloud 版本约束

SpringBoot 和 SpringCloud 的版本选择也不是任意的，而是应该参考官网的约束配置

Release Train	Boot Version
Hoxton	2.2.x
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

地址: <https://spring.io/projects/spring-cloud>

版本对应: <https://start.spring.io/actuator/info>

## 关于 Cloud 各种组件的停更/升级/替换

### 停更引发的升级惨案

- 被动修复 Bugs

- 不再接受合并请求
- 不再发布新版本

## 明细条目

- 服务调用
  - Eureka
  - Zookeeper
  - Consul
  - Nacos (推荐)
- 服务调用
  - Feign
  - OpenFeign (推荐)
  - Ribbon
  - LoadBalancer
- 服务降级
  - Hystrix
  - resilience4j
  - sentinel (推荐)
- 服务网关
  - Zuul
  - Zuul2
  - Gateway (推荐)
- 服务配置
  - Config
  - Nacos (推荐)
- 服务总线
  - Bus

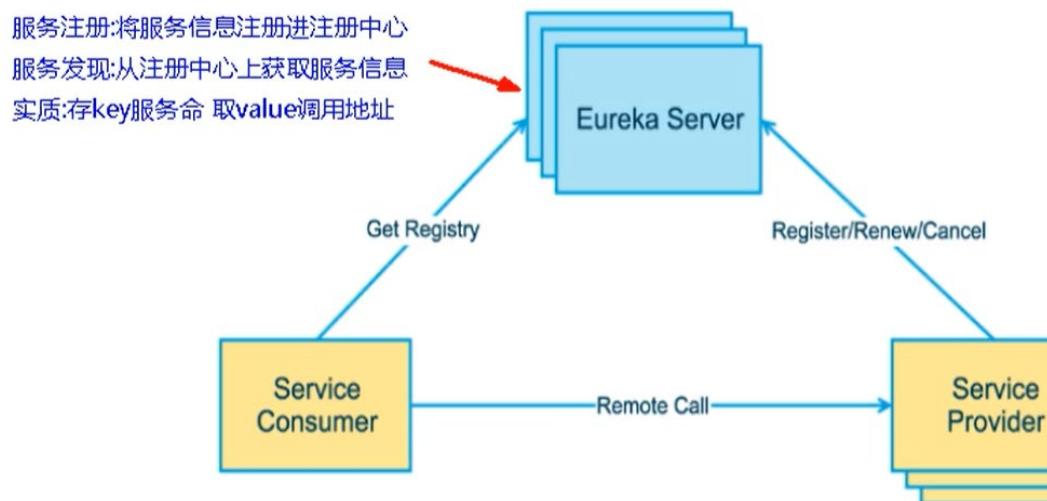
- Nacos (推荐)

## Eureka 集群

没有集群带来的高可用，会带来单点故障

### Eureka 工作原理

- 服务注册：将服务信息注册进注册中心
  - 服务发现：从注册中心上获取服务信息
  - 实质：存 key 服务命名，取 value 调用地址
1. 先启动 eureka 注册中心
  2. 启动服务提供者 payment 支付服务
  3. 支付服务启动后，会把自身信息（比如服务地址以别名方式注册进 eureka）
  4. 消费者 order 服务在调用接口时候使用服务别名去注册中心获取实际的 RPC 远程调用地址
  5. 消费者获得调用地址后，底层实际是利用 HttpClient 技术实现远程调用
  6. 消费者获取服务地址后会缓存在本地 JVM 内存中，默认每隔 30 秒更新一次服务调用地址

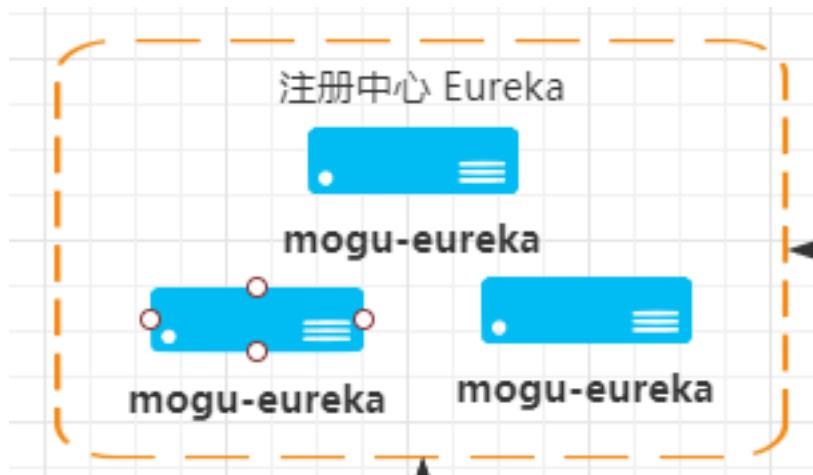


微服务 RPC 远程调用最核心的就是：高可用

因为假设注册中心只有一个，如果出现了故障，那么将会导致整个微服务不可用，所以需要搭建 Eureka 注册中心集群，实现负载均衡 + 故障容错

## Eureka 集群原理

互相注册，相互守望



## 搭建集群

原来单机版本时，我们的注册中心的配置文件为

```
server:  
  port: 7001  
eureka:  
  instance:  
    hostname: localhost #eureka 服务端实例名称  
  client:  
    register-with-eureka: false #表示不向注册中心注册自己  
    fetch-registry: false #false 表示自己就是注册中心，我的职责就是维护服务  
实例，并不检索服务  
    service-url:  
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eu  
reka/
```

但是如果使用了集群后，我们的 eureka 就需要相互注册了，也就是 7001 的需要注册到 7002，而 7002 注册 7001

同时 hostname 也不能重复，需要有两个主机的 ip

eureka 7001

```

server:
  port: 7001
eureka:
  instance:
    hostname: eureka7001.com #eureka 服务端实例名称
  client:
    register-with-eureka: false #表示不向注册中心注册自己
    fetch-registry: false #false 表示自己就是注册中心，我的职责就是维护服务
实例，并不区检索服务
  service-url:
    # 向另外一个 eureka 服务注册
    defaultZone: http://eureka7002.com:7002/eureka/

```

eureka 7002:

```

server:
  port: 7002
eureka:
  instance:
    hostname: eureka7002.com #eureka 服务端实例名称
  client:
    register-with-eureka: false #表示不向注册中心注册自己
    fetch-registry: false #false 表示自己就是注册中心，我的职责就是维护服务
实例，并不区检索服务
  service-url:
    # 向另外一个 eureka 服务注册
    defaultZone: http://eureka7001.com:7001/eureka/

```

启动后，我们能发现，在 eureka7001 上，能看到 7002 注册上去了

The screenshot shows the Spring Eureka dashboard. At the top, it displays 'spring Eureka' and 'HOME LAST 1000 SINCE STARTUP'. Below this is the 'System Status' section, which includes environment details like 'Environment: test', 'Data center: default', and various uptime metrics. The 'DS Replicas' section shows a single entry: 'eureka7002.com'. In the 'Instances currently registered with Eureka' section, there are two entries: 'CLOUD-ORDER-SERVICE' and 'CLOUD-PAYMENT-SERVICE', both marked as 'UP (1)'. At the bottom, there's a 'General Info' section.

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - order80
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - payment8001

同时在 eureka7002 上，能看到 7001，这个时候说明我们的 eureka 集群已经搭建完毕

The screenshot shows the Spring Eureka dashboard. At the top, it displays 'spring Eureka' and 'HOME LAST 1000 SINCE STARTUP'. Below this, the 'System Status' section shows environment and data center details. The 'DS Replicas' section lists a single instance: 'eureka7001.com'. The 'Instances currently registered with Eureka' section shows two services: 'CLOUD-ORDER-SERVICE' and 'CLOUD-PAYMENT-SERVICE', both marked as 'UP'.

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - order80
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - payment8001

## 服务注册 Eureka 集群

我们修改服务提供者 payment 的 yml 配置，同时将两个 eureka 地址配置在 defaultZone 中

```

eureka:
  client:
    #表示向注册中心注册自己 默认为 true
    register-with-eureka: true
    #是否从 EurekaServer 抓取已有的注册信息，默认为 true,单节点无所谓，集群必须设置为 true 才能配合 ribbon 使用负载均衡
    fetch-registry: true
    service-url:
      # 入驻地址
      # defaultZone: http://localhost:7001/eureka/
      #集群版
      defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/
      #服务名称
    instance:
      instance-id: payment8001
      #访问路径显示 IP 地址
      prefer-ip-address: true
  
```

在上面的图中，我们能发现，payment 服务已经成功注册到两台 eureka 集群中了

## 服务提供者集群

我们需要搭建多个服务提供者

例如：payment8001：

```

server:
  port: 8001

spring:
  application:
    name: cloud-payment-service #服务名称
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource #当前数据源操作类型
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/cloud2020?characterEncoding=utf8&useSSL=false&useUnicode=true
    username: root
    password: root
  eureka:
    client:
      #表示向注册中心注册自己 默认为 true
      register-with-eureka: true
      #是否从 EurekaServer 抓取已有的注册信息， 默认为 true，单节点无所谓，集群必须设置为 true 才能配合 ribbon 使用负载均衡
      fetch-registry: true
    service-url:
      # 入驻地址
      # defaultZone: http://localhost:7001/eureka/
      #集群版
      defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/

```

和 payment8002:

```

server:
  port: 8002

spring:
  application:
    name: cloud-payment-service #服务名称
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      # 采集率介于 0 到 1 之间， 1 表示全部采集
      probability: 1
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource #当前数据源操作类型
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/cloud2020?characterEncoding=utf8&useSSL=false&useUnicode=true
    username: root
    password: root

```

```

eureka:
  client:
    #表示向注册中心注册自己 默认为 true
    register-with-eureka: true
    #是否从 EurekaServer 抓取已有的注册信息， 默认为 true,单节点无所谓,集群必须设置为 true 才能配合 ribbon 使用负载均衡
    fetch-registry: true
    service-url:
      # 入驻地址
    #      defaultZone: http://localhost:7001/eureka/
      #集群版
      defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/
    #服务名称
    instance:
      instance-id: payment8001
      #访问路径显示 IP 地址
      prefer-ip-address: true
mybatis:
  mapper-locations: classpath:mapper/*.xml
  type-aliases-package: com.atguigu.springcloud.entity #所有 entity 别名所在包

```

这里需要注意的就是，为了保证这两个服务，对外暴露的都是同一个服务提供者，我们的服务名需要保持一致

```

spring:
  application:
    name: cloud-payment-service #服务名称

```

启动后，我们发现 CLOUD-PAYMENT\_SERVICE 上有两个服务提供者了，分别为：8001 和 8002

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - order80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - payment8002,payment8001
General Info			

同时我们需要服务名进行调用

[http://CLOUD-PAYMENT\\_SERVICE](http://CLOUD-PAYMENT_SERVICE)

通知在 RestTemplate 需要设置负载均衡策略，即 @LoadBalanced 注解，不然它不知道调用哪个微服务地址

```

@Configuration
public class ApplicationContextConfig {

```

```

@Bean
@LoadBalanced //赋予 RestTemplate 负载均衡的能力
public RestTemplate getRestTemplate() {
    return new RestTemplate();
}
}

```

这个就是 Ribbon 的负载均衡功能， 默认是轮询

Ribbon 和 Eureka 整合后， Consumer 可以直接调用服务而不再关心地址和端口号，且该服务还有负载均衡的功能

## actuator 微服务信息完善

要做图形化的展示这块，这两个依赖都需要导入

```

<!--web 启动器-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!--监控-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

```

actuator：主要用于 IP 信息完善

actuator 查看健康状态

<http://192.168.80.1:8002/actuator/health>

## 服务名称修改

修改后，对外暴露的就是服务名称

```

Eureka
#服务名称
instance:
    instance-id: payment8001

```

## 设置服务的 IP 显示

```

Eureka
#服务名称
instance:
    #访问路径显示 IP 地址
    prefer-ip-address: true

```

## 服务发现 Discovery

Eureka 的新的注解标签 @EurekaDiscovery

对于注册进 Eureka 里面的微服务，可以通过服务发现来获得该服务的信息

```
@Resource  
private DiscoveryClient discoveryClient;
```

获得服务列表

```
# 获取列表  
List<String> services = discoveryClient.getServices();  
  
# 获取实例  
List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-P  
AYMENT-SERVICE");  
  
# 获取 ServiceId  
instances.get(0).getServiceId();  
  
# 获取主机名  
instances.get(0).getHost();  
  
# 获取端口号  
instances.get(0).getPort();  
  
# 获取 URL  
instances.get(0).getUrl();
```

## Eureka 自我保护机制

### 概念

保护模式主要用于一组客户端和 Eureka Server 之间存在网络分区场景下的保护，一旦进入保护模式，Eureka Server 将会尝试保护其服务注册表的信息，不再删除服务注册表中的数据，也就是不会注销任何微服务。

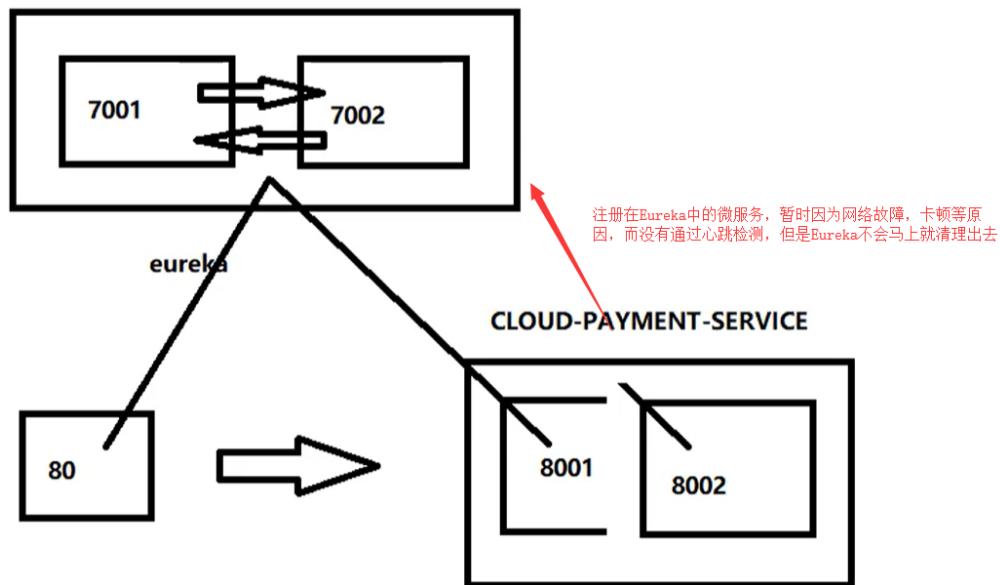
如果在 Eureka Server 的首页看到以下这段提示，说明 Eureka 进入了保护模式



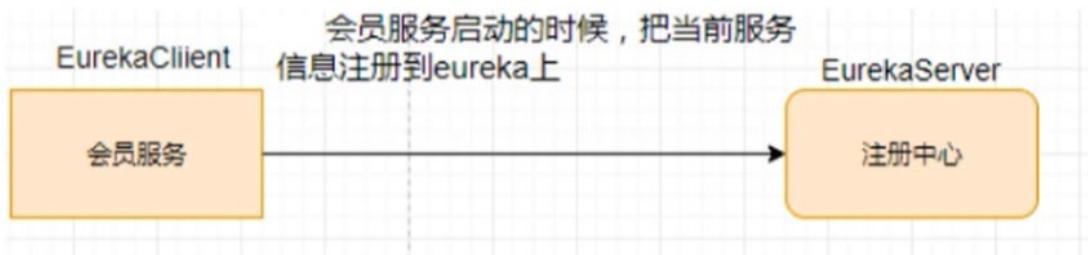
通俗的话来说：某时刻某一个微服务不可用了，Eureka 不会立刻清理，依旧会对该微服务的信息进行保存，属于 CAP 里面的 AP 分支。

## 导致原因

默认情况下，如果 EurekaServer 在一定时间内没有接收到某个微服务实例的心跳，EurekaServer 将会注销该实例， 默认 90 秒。但是当网络分区故障发生（延时，卡顿，拥挤）时，微服务与 EurekaServer 之间无法正常通信，以上行为可能变得非常危险了- 因为微服务本身其实是健康的，此时不应该注销这个微服务，Eureka 通过自我保护模式来解决这个问题，当 EurekaServer 节点在短时间丢失过多客户端，那么这个节点就会进入自我保护模式



这是一种高可用的机制



在自我保护模式下，Eureka Server 会保护服务注册表中的信息，不在注销任何服务实例

它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例

综上，自我保护模式是一种应对网络异常的安全保护措施，它的架构哲学是宁可保留所有微服务（健康的微服务和不健康的微服务都会保留）也不盲目注销任何健康的微服务。使用自我保护模式，可以让 Eureka 集群更加健壮，稳定。

## 禁止自我保护

Eureka 默认开启自我保护

```
eureka:
  server:
    enable-self-preservation: true
    peer-node-read-timeout-ms: 3000
    peer-node-connect-timeout-ms: 3000
```

同时在客户端进行设置

```
eureka:
  instance:
    # Eureka 客户端向服务端发送心跳的时间间隔，单位为秒，默认 30
    lease-renewal-interval-in-seconds: 1
    # Eureka 服务端在收到最后一次心跳后等待时间上限，单位为秒，默认为 90 秒，超时将剔除服务
    lease-expiration-duration-in-seconds: 2
```

设置完成后，只要服务宕机，会马上从服务注册列表中清除

## 关于 Eureka 停更

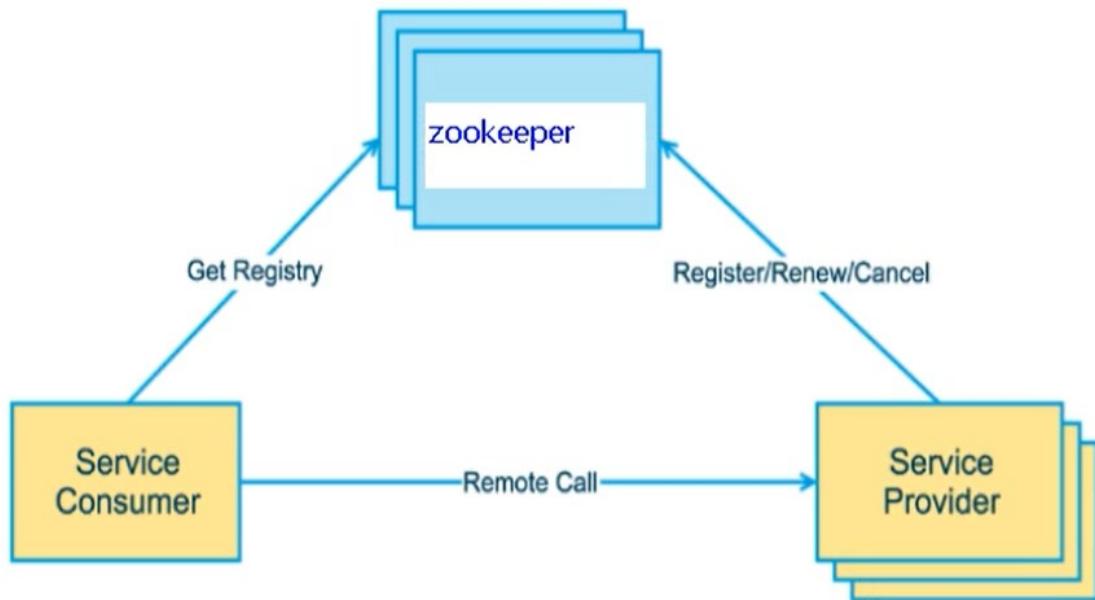
Eureka 停更后，出现了其它的替代者

- Zookeeper
- Consul

- Nacos

## Eureka 停更后的替换

### Zookeeper 替换 Eureka



### Zookeeper 是什么

Zookeeper 是一个分布式协调工具，可以实现注册中心功能

关闭 Linux 服务器防火墙后，启动 Zookeeper 服务器，Zookeeper 服务器取代 Eureka 服务器，zk 作为服务注册中心。

### 搭建 Zookeeper 注册中心

#### 引入依赖

```

<!--zookeeper 客户端-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
    <!--先排除自带的 zookeeper3.5.3-->
    <exclusions>
        <exclusion>
  
```

```
<groupId>org.apache.zookeeper</groupId>
<artifactId>zookeeper</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.9</version>
</dependency>
```

## 修改配置文件

连接上 Zookeeper 客户端

```
spring:
  application:
    name: cloud-provider-payment
  cloud:
    zookeeper:
      connect-string: 180.76.99.142:2181
```

## 修改主启动类

使用@EnableDiscoveryClient 注解

```
@SpringBootApplication
@EnableDiscoveryClient
public class PaymentMain8004 {
    public static void main(String[] args) {
        SpringApplication.run(PaymentMain8004.class);
    }
}
```

## 启动

启动成功后，把服务注册进 Zookeeper 客户端

The screenshot shows two windows side-by-side. On the left is an IDE window for IntelliJ IDEA, displaying the file `application.yml`. A red box highlights the section where the service is registered to Zookeeper:

```

server:
  port: 8004
#服务别名---注册zookeeper到注册中心名称
spring:
  application:
    name: cloud-provider-payment
  cloud:
    zookeeper:
      connect-string: 192.168.111.144:2181

```

On the right is a terminal window titled "CentOS7-01 - VMware Workstation". It shows the command `ls /services` being run, which lists a node named `cloud-provider-payment`. Another red box highlights this node.

## 思考

服务已经成功注册到 Zookeeper 客户端，那么注册上去的节点被称为临时节点，还是持久节点？

首先 Eureka 有自我保护机制，也就是某个服务下线后，不会立刻清除该服务，而是将服务保留一段时间

Zookeeper 也一样在服务下线后，会等待一段时间后，也会把该节点删除，这就说明 Zookeeper 上的节点是临时节点。

## Consul 替换 Eureka

### 简介

官网: <https://www.consul.io/>

Consul 是一套开源的分布式服务发现和配置管理系统，由 HashiCorp 公司用 Go 语言开发

提供了微服务系统中的服务治理、配置中心、控制总线等功能，这些功能中的每一个都可以根据需要单独使用，也可以一起使用构建全方位的服务网路，总之 Consul 提供了一种完整的服务网络解决方案。

它具有很多优点，包括：基于 raft 协议，比较简洁；支持健康检查，同时支持 HTTP 和 DNS 协议，支持跨数据中心的 WAN 集群，提供图形化界面，跨平台，支持 Linux, MAC, Windows

## 功能

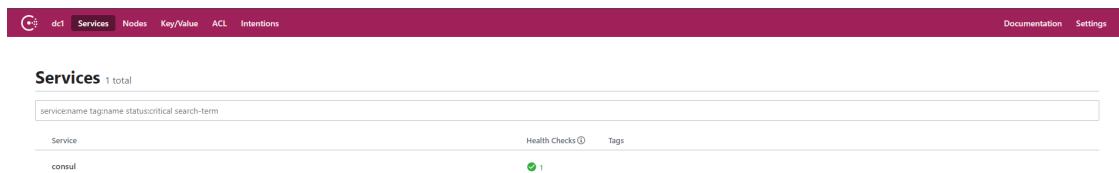
- 服务发现：提供 HTTP 和 DNS 两种发现方式
- 健康监测：支持多种方法，HTTP, TCP, Docker, Shell 脚本定制化
- KV 存储：Key, Value 的存储方式
- 多数据中心：Consul 支持多数据中心
- 可视化 Web 界面

## 安装

官网：<https://www.consul.io/downloads.html>

- 查看版本：`consul --version`
- 运行：`consul agent -dev`

运行成功后，然后访问 `http://localhost:8500`，进入 consul 的可视化界面



The screenshot shows the Consul UI interface. At the top, there is a navigation bar with tabs: Services (which is selected), dcl, Nodes, Key/Value, ACL, Intentions, Documentation, and Settings. Below the navigation bar, the main area is titled "Services" with "1 total". There is a search bar with placeholder text "servicename tagname status critical search-term". A table lists one service: "consul". The table has columns for "Service", "Health Checks (0)", and "Tags". The "consul" entry has a green circle icon with a checkmark and the number "1" next to it.

## 服务提供者注册 Consul

引入依赖

```
<!--consul-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

修改 yml

```
#consul 服务端口号
server:
  port: 8006
```

```

spring:
  application:
    name: consul-provider-payment
#consul 注册中心地址
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        service-name: ${spring.application.name}

```

然后启动项目，即可发现服务提供者已经注册到 Consul 中了

## 总结

组件名	语言	健康检查	对外暴露接口	CAP	Spring Cloud 集成
Eureka	Java	可配支持	HTTP	AP	已集成
Consul	Go	支持	HTTP/DNS	CP	已集成
Zookeeper	Java	支持	客户端	CP	已集成

## CAP 理论

Availability: 高可用

Consistency: 强一致性

Partition Tolerance: 分区容错性

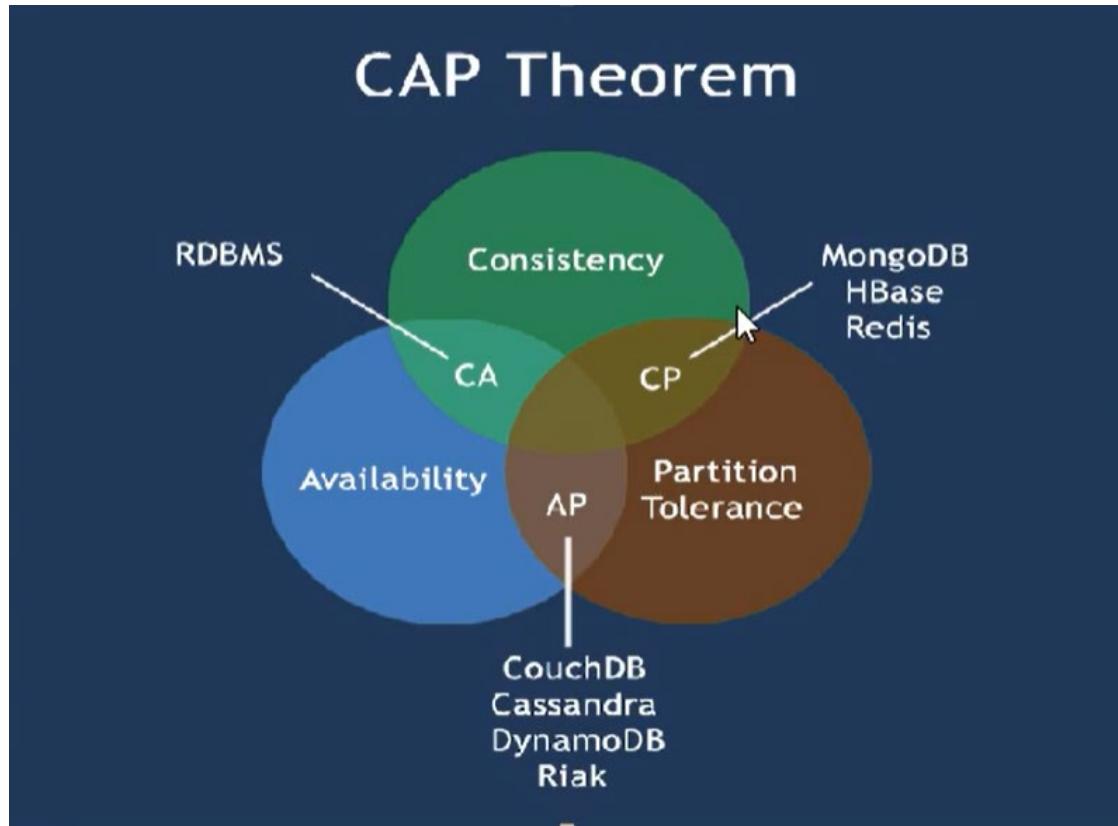
CAP 理论关注粒度是数据，而不是整体系统设计的策略

因此现在的微服务架构要么是 CP 要么是 AP，也就是 P 一定需要保证，最多只能较好的同时满足两个

CAP 理论的核心：一个分布式系统不可能同时很好的满足：一致性，可用性和分区容错性这个三个需求

因此，根据 CAP 原理将 NoSQL 数据库分成了满足 CA 原则，满足 CP 原则，满足 AP 的三大类

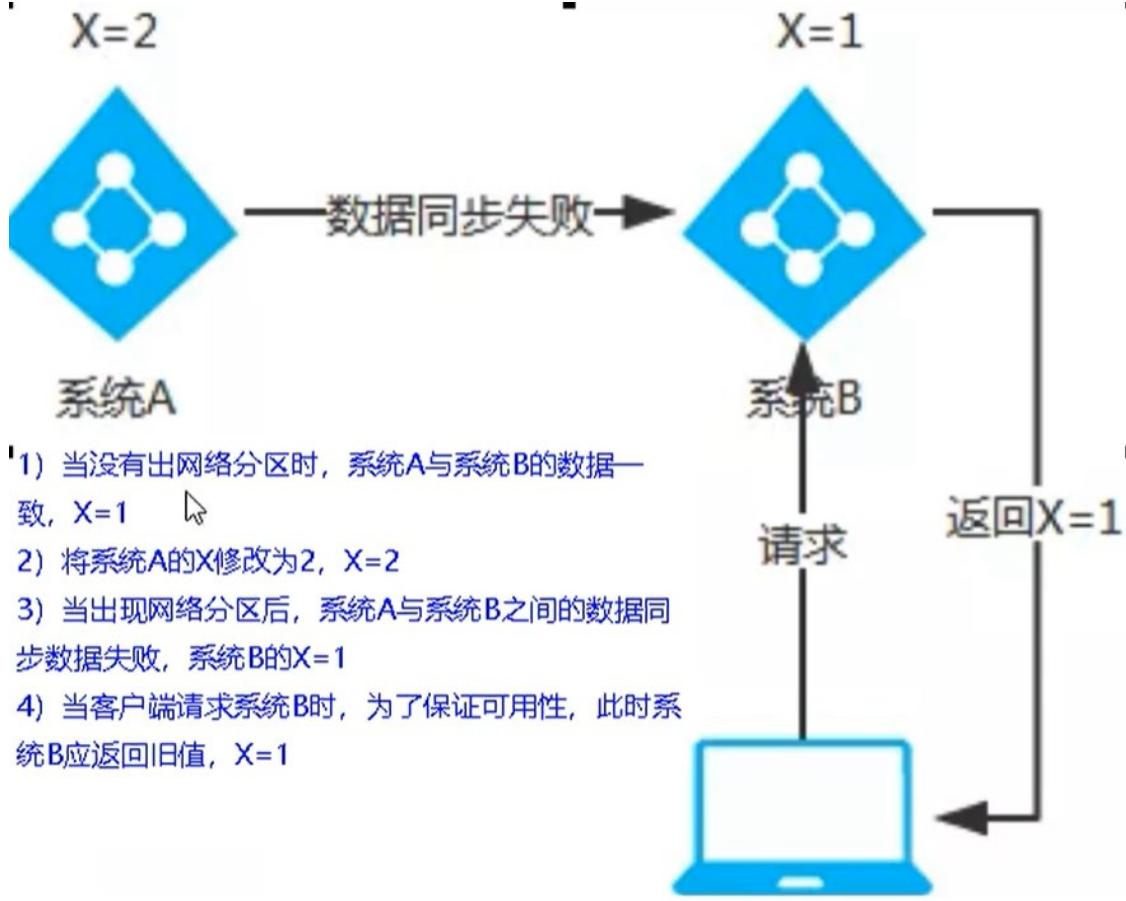
- CA：单点集群，满足一致性，可用性的系统，通常在可扩展性上不太满足
- CP：满足一致性，分区容忍性，通常性能不是特别高
- AP：满足可用性，分区容忍性，通常对一致性要求低一些



部分情况下，我们对数据一致性的要求没有这么高，比如蘑菇博客的点赞和浏览记录，都是每隔一段时间才写入数据库的。

## AP 架构

Eureka 是 AP 架构



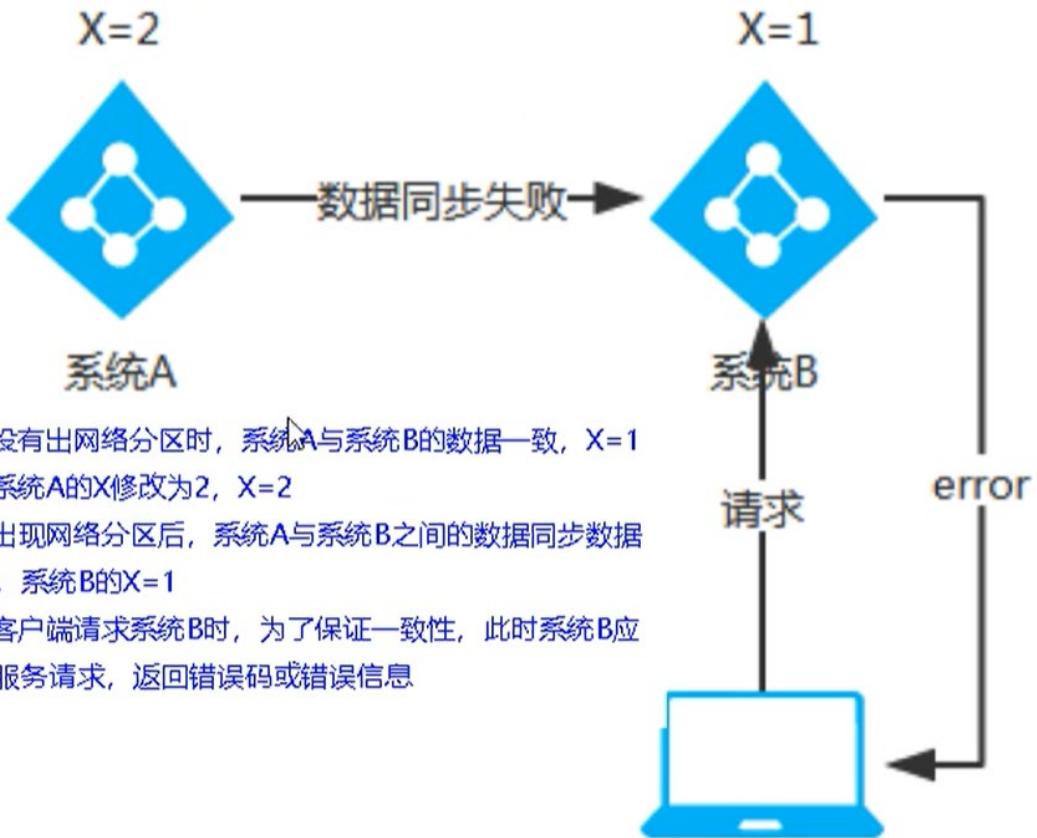
因为同步原因出现问题，而造成数据没有一致性

当出现网络分区后，为了保证高可用，系统 B 可以返回旧值，保证系统的可用性

结论：违背了一致性 C 的要求，只满足可用性和分区容错性，即 AP

## CP 架构

Zookeeper 和 Consul 是 CP 架构



当出现网络分区后，为了保证一致性，就必须拒绝请求，否者无法保证一致性

结论：违背了可用性 A 的要求，只满足一致性和分区容错性，即 CP

## Ribbon 实现负载均衡

Ribbon 目前已经进入了维护模式，但是目前主流还是使用 Ribbon

Spring Cloud 想通过 LoadBalancer 用于替换 Ribbon

### 概念

Spring Cloud Ribbon 是基于 Netflix Ribbon 实现的一套客户端，负载均衡的工具

简单的说，Ribbon 是 NetFlix 发布的开源项目，主要功能是提供客户端的软件负载均衡算法和服务调用。Ribbon 客户端组件提供了一系列完善的配置项如连接超时，重试等。简单的说，就是在配置文件中列出 Load Balancer（简称 LB）后面所有的机器，Ribbon 会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。我们很容易使用 Ribbon 实现自定义的负载均衡算法。

## LB 负载均衡是什么

Load Balance，简单来说就是将用户的请求平摊的分配到多个服务上，从而达到系统的 HA（高可用）。常见的负载均衡有软件 Nginx，LVS，硬件 F5 等。

- 集中式 LB：即在服务的消费方和提供方之间使用独立的 LB 设施（可以是硬件，如 F5，也可以是软件，如 Nginx），由该设施负责把访问请求通过某种策略转发至服务的提供方
- 进程内 LB：将 LB 逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。Ribbon 就属于进程内 LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址。

## Ribbon 本地负载均衡客户端 VS Nginx 服务端负载均衡

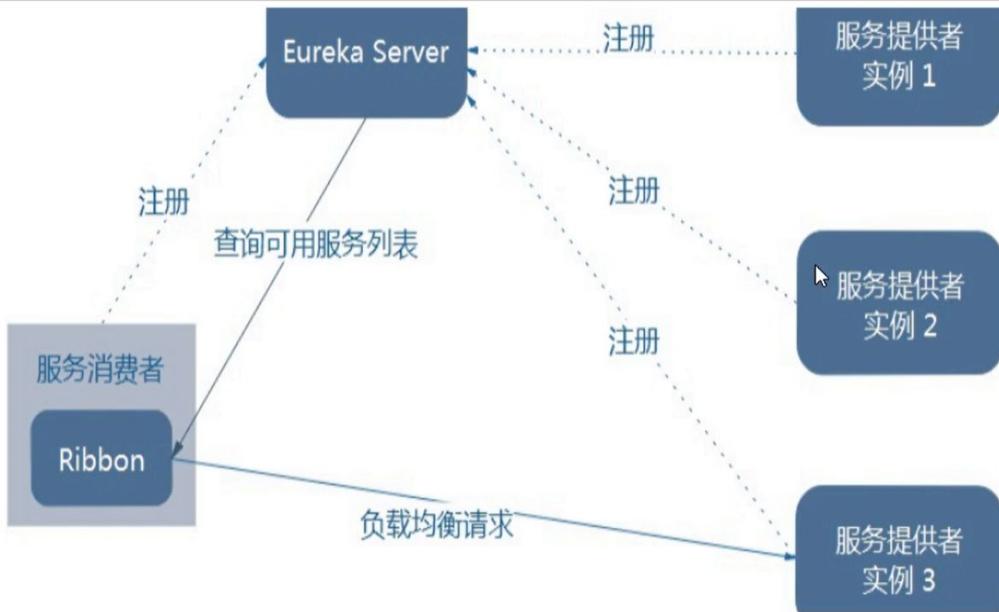
Nginx 是服务器负载均衡，客户端所有的请求都会交给 nginx，然后由 nginx 实现转发请求，即负载均衡是由服务端实现的。

Ribbon 本地负载均衡，在调用微服务接口的时候，会在注册中心上获取注册信息服务列表之后，缓存到 JVM 本地，从而在本地实现 RPC 远程调用的技术。

一句话就是：Ribbon = 负载均衡 + RestTemplate 调用

## Ribbon 工作原理

Ribbon 其实就是一个软负载均衡的客户端组件，它可以和其它所需请求的客户端结合使用，和 Eureka 结合只是其中的一个实例。



Ribbon 在工作时分成两步

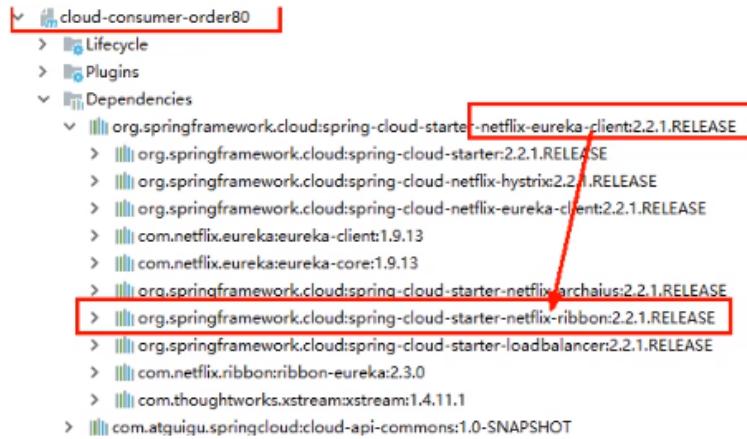
- 首先先选择 EurekaServer，它优先选择在同一个区域内负载较少的 Server
- 再根据用户的指定的策略，从 Server 取到服务注册列表中选择一个地址
- 其中 Ribbon 提供了多种策略：比如轮询，随机和根据响应时间加权

## 引入 Ribbon

新版的 Eureka 已经默认引入 Ribbon 了，不需要额外引入

```
<!--Eureka 客户端-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

可以看到spring-cloud-starter-netflix-eureka-client 确实引入了Ribbon



## RestTemplate

主要方法为：

- restTemplate.getForObject
- restTemplate.postForObject

```
@GetMapping("/consumer/payment/create")
public CommonResult<Payment> create(Payment payment) {
    return restTemplate.postForObject(PAYMENT_URL + "/payment/create", payment, CommonResult.class);
}

@GetMapping("/consumer/payment/get/{id}")
public CommonResult<Payment> getPayment(@PathVariable("id") Long id) {
    return restTemplate.getForObject(PAYMENT_URL + "/payment/get/" + id, CommonResult.class);
}
@GetMapping("/consumer/payment/getForEntity/{id}")
public CommonResult<Payment> getForEntity(@PathVariable("id") Long id) {
    ResponseEntity<CommonResult> entity = restTemplate.getForEntity(PAYMENT_URL + "/payment/get/" + id, CommonResult.class);
    if (entity.getStatusCode().is2xxSuccessful()){
        return entity.getBody();
    }else {
        return new CommonResult<>(444, "操作失败");
    }
}
```

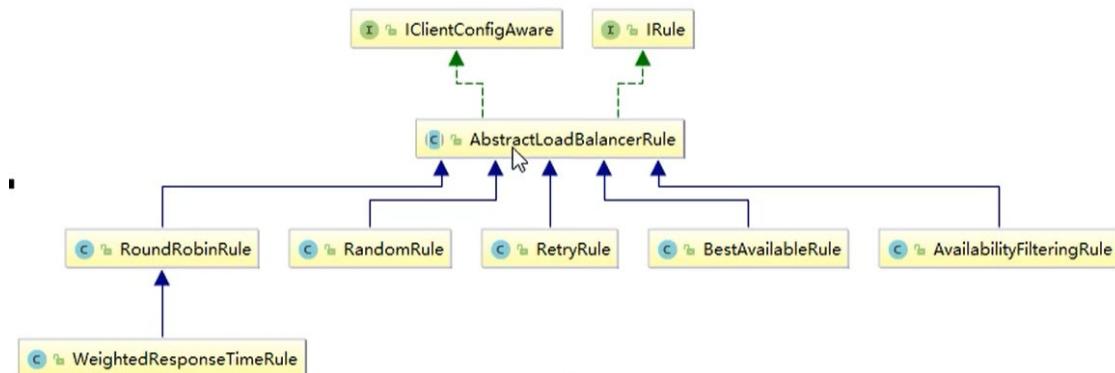
## Ribbon 核心组件 IRule

Ribbon 默认是使用轮询作为负载均衡算法

IRule 根据特定算法从服务列表中选取一个要访问的服务，IRule 是一个接口

```
public interface IRule {  
    Server choose(Object var1);  
  
    void setLoadBalancer(ILoadBalancer var1);  
  
    ILoadBalancer getLoadBalancer();  
}
```

然后对该接口，进行特定的实现



## 负载均衡算法

IRule 的实现主要有以下七种

- RoundRobinRule: 轮询
- RandomRule: 随机
- RetryRule: 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，获取可用服务
- WeightedResponseTimeRule: 对 RoundRobinRule 的扩展，响应速度越快的实例选择的权重越大，越容易被选择

- BestAvailableRule: 会先过滤掉由于多次访问故障而处于短路跳闸状态的服务，然后选择一个并发量最小的服务
- AvailabilityFilteringRule: 先过滤掉故障实例，在选择并发较小的实例
- ZoneAvoidanceRule: 默认规则，符合判断 server 所在区域的性能和 server 的可用性选择服务器

## 默认负载均衡算法替换

官网警告：自定义的配置类不能放在@ComponentScanner 所扫描的当前包下以及子包下，否者我们自定义的这个配置类就会被所有的 Ribbon 客户端所共享，达不到特殊化定制的目的了



然后我们创建自定义 Rule 接口

```
@Configuration
public class MySelfRule {
    @Bean
    public IRule myRule(){
        return new RandomRule(); //自定义为随机
    }
}
```

在主启动类中，添加@RibbonClient

```
@SpringBootApplication
@EnableDiscoveryClient
@RibbonClient(name="CLOUD-PAYMENT-SERVICE", configuration = MySelfRule.class)
public class OrderMain80 {
    public static void main(String[] args) {
        SpringApplication.run(OrderMain80.class, args);
    }
}
```

# 手写 Ribbon 负载均衡算法

## 原理

负载均衡算法：rest 接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标，每次服务重启后 rest 接口计数从 1 开始。

假设现在有 2 台机器，同时 `List = 2 instance` (也就是服务注册列表中，有两台)

`1 % 2 = 1 -> index = list.get(1)`

`2 % 2 = 0 -> index = list.get(0)`

`3 % 2 = 1 -> index = list.get(1)`

....

这就是轮询的原理，即

```
List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");
```

如：  
`List [0] instances = 127.0.0.1:8002`

`List [1] instances = 127.0.0.1:8001`

8001 + 8002 组合成为集群，它们共计2台机器，集群总数为2， 按照轮询算法原理：

当总请求数为1时：  
`1 % 2 = 1` 对应下标位置为1，则获得服务地址为`127.0.0.1:8001`

当总请求数位2时：  
`2 % 2 = 0` 对应下标位置为0，则获得服务地址为`127.0.0.1:8002`

当总请求数位3时：  
`3 % 2 = 1` 对应下标位置为1，则获得服务地址为`127.0.0.1:8001`

当总请求数位4时：  
`4 % 2 = 0` 对应下标位置为0，则获得服务地址为`127.0.0.1:8002`

如此类推.....

## 源码

我们查看 `RandomRule` 的源码发现，其实内部就是利用的取余的技术，同时为了保证同步机制，还是使用了 `AtomicInteger` 原子整型类

```
public class RandomRule extends AbstractLoadBalancerRule {  
    public RandomRule() {  
    }  
  
    @SuppressWarnings({"RCN_REDUNDANT_NONNULLCHECK_OF_NULL_VALUE"})  
    public Server choose(ILoadBalancer lb, Object key) {  
        if (lb == null) {  
            return null;  
        } else {  
            int size = lb.getNumberOfAvailable();  
            if (size <= 0) {  
                return null;  
            }  
            int index = key.hashCode() % size;  
            return lb.getAvailableServerAtIndex(index);  
        }  
    }  
}
```

```

        Server server = null;

        while(server == null) {
            if (Thread.interrupted()) {
                return null;
            }

            List<Server> upList = lb.getReachableServers();
            List<Server> allList = lb.getAllServers();
            int serverCount = allList.size();
            if (serverCount == 0) {
                return null;
            }

            int index = this.chooseRandomInt(serverCount);
            server = (Server)upList.get(index);
            if (server == null) {
                Thread.yield();
            } else {
                if (server.isAlive()) {
                    return server;
                }
            }

            server = null;
            Thread.yield();
        }

        return server;
    }
}

protected int chooseRandomInt(int serverCount) {
    return ThreadLocalRandom.current().nextInt(serverCount);
}

public Server choose(Object key) {
    return this.choose(this.getLoadBalancer(), key);
}

public void initWithNiwsConfig(IClientConfig clientConfig) {
}
}

```

## 手写负载均衡算法

原理 + JUC (CAS+自旋锁)

首先需要在 RestTemplate 的配置上将 @LoadBalanced 注解删除

```

@Bean
//@LoadBalanced 赋予 RestTemplate 负载均衡的能力
public RestTemplate getRestTemplate() {
    return new RestTemplate();
}

```

然后创建一个 LoadBalanced 接口

```

/**
 * 自定义负载均衡算法
 * @Author: TianTian
 * @Date: 2020/3/7 19:53
 */
public interface LoadBalancer {
    // 获取注册的一个实例
    ServiceInstance instances(List<ServiceInstance> serviceInstances);
}

```

创建一个实现类，首先 LoadBalanced 接口

```

@Component
public class MyLB implements LoadBalancer {

    // 创建原子整型类
    private AtomicInteger atomicInteger = new AtomicInteger(0);

    /**
     * 获取 Rest 调用的次数
     * @return
     */
    public final int getAndIncrement(){
        int current;
        int next;
        // 自旋锁
        do{
            // 获取当前值
            current=this.atomicInteger.get();

            /*2147483647:整型最大值*/
            // 发生越界，从 0 开始计数
            next= current >=2147483647 ? 0:current+1;

            // 比较并交换
        }while (!this.atomicInteger.compareAndSet(current,next));

        System.out.println("*****第几次访问 next"+next);
        return next;
    }
}

```

```

//负载均衡算法：第几次请求%服务器总数量=实际访问。服务每次启动从 1 开始
@Override
public ServiceInstance instances(List<ServiceInstance> serviceInstances) {

    // 获取当前计数 模 实例总数
    int index= getAndIncrement() % serviceInstances.size();

    // 返回选择的实例
    return serviceInstances.get(index);
}
}

```

具体使用

步骤就是，首先我们通过 `discoveryClient` 获取所有的注册实例，然后调用该实现类，获取到调用的地址

```

/**
 * 在这边我为了以上程序的正常执行：把自定义接口注释掉，不用自定义负载均衡
算法，若想再次启动
 * 请操作一下步骤：
 *          1.注释掉@LoadBalanced（在 config 下面），放开下方注释，同时
会导致上方不可用，因为找不到具体服务
 */

@GetMapping(value = "/consumer/payment/lb")
public String getPaymentLB(){
    List<ServiceInstance> instances = discoveryClient.getInstances
("CLOUD-PAYMENT-SERVICE");
    if (instances ==null || instances.size()<=0){
        return null;
    }
    //传入自己的
    ServiceInstance serviceInstance = loadBalancer.instances(instances);
    URI uri = serviceInstance.getUri();
    return restTemplate.getForObject(uri+"/payment/lb",String.class);
}

```

## OpenFeign 实现服务调用

关于 `Feign` 的停更，目前已经使用 `OpenFeign` 进行替换

## 概述

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 也支持可插拔式的编码和解码器。Spring Cloud 对 feign 进行了封装，使其支持了 Spring MVC 标准注解和 HttpMessageConverters。Feign 可以与 Eureka 和 Ribbon 组合使用以支持负载均衡。

## Feign 的作用

Feign 旨在使编写 Java Http 客户端变得更容易。

前面在使用 Ribbon + RestTemplate 时，利用 RestTemplate 对 http 请求的封装处理，形成了一套模板化的调用方法。但是在实际开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以通常都会针对每个微服务自行封装一些客户端来包装这些依赖，所以 Feign 在这个基础上做了进一步封装，由他来帮助我们定义和实现依赖服务的接口定义。在 Feign 的实现下，我们只需要创建一个接口，并使用注解的方式来配置它（以前是 Dao 接口上面标注 Mapper 注解，现在是微服务接口上标注一个 Feign 注解），即可完成服务提供方的接口绑定，简化了使用 Spring Cloud Ribbon 时，自动封装服务调用客户端的开发量。

## Feign 集成 Ribbon

利用 Ribbon 维护了 Payment 的服务列表信息，并且通过轮询实现了客户端的负载均衡。而与 Ribbon 不同的是，通过 Feign 只需要定义服务绑定接口且声明式的方法，优雅而简单的实现了服务调用。

## Feign 和 OpenFeign 的区别

Feign	OpenFeign
Feign 是 Spring Cloud 组件中的一种轻量级 RestFul 的 HTTP 服务客户端，Feign 内置了 Ribbon，用来做客户端的负载均衡，去调用服务注册中心的服务，Feign 的使用方式是：使用 Feign 的注解定义接口，调用这个接口，就可以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 @RequestMapping 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法生产实现类，实现类中做均衡并调用其它服务
spring-cloud-starter-feign	spring-cloud-starter-openfeign

## OpenFeign 使用步骤

### 引入依赖

```
<!--openfeign-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

### 修改启动类

激活 Feign 组件

```
@SpringBootApplication
@EnableFeignClients
public class OrderFeignMain80 {
    public static void main(String[] args) {
        SpringApplication.run(OrderFeignMain80.class, args);
    }
}
```

### 添加业务逻辑接口

```
@Component
@FeignClient(value = "cloud-payment-service")
public interface PaymentFeignService {

    @GetMapping(value = "/payment/get/{id}")
    public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id);

    @GetMapping("/payment/feign/timeout")
    public String paymentFeignTimeout();
}
```

### 具体使用

```
@RestController
@Slf4j
public class OrderFeignController {
    @Resource
    private PaymentFeignService paymentFeignService;

    @GetMapping(value = "/consumer/payment/get/{id}")
    public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id){
        CommonResult<Payment> paymentById = paymentFeignService.getPaymentById(id);
        return paymentById;
    }
}
```

```
@GetMapping("/consumer/payment/feign/timeout")
public String paymentFeignTimeout(){
    //open-feign-ribbon,客户端默认等待一秒钟
    return paymentFeignService.paymentFeignTimeout();
}
```

## OpenFeign 的超时控制

服务提供者需要超过 3 秒才能返回数据，但是服务调用者默认只等待 1 秒，这就会出现超时问题。

## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Feb 29 17:17:24 CST 2020

There was an unexpected error (type=Internal Server Error, status=500).

Read timed out executing GET http://CLOUD-PAYMENT-SERVICE/payment/feign/timeout  
feign.RetryableException: Read timed out executing GET http://CLOUD-PAYMENT-SERVICE/payment/feign/timeout  
at feign.FeignException.errorExecuting(FeignException.java:213)  
at feign.SynchronousMethodHandler.executeAndDecode(SynchronousMethodHandler.java:115)  
at feign.SynchronousMethodHandler.invoke(SynchronousMethodHandler.java:80)  
at feign.ReflectiveFeign\$FeignInvocationHandler.invoke(ReflectiveFeign.java:103)

这是因为默认 Feign 客户端只等待一秒钟，但是服务端处理需要超过 3 秒钟，导致 Feign 客户端不想等待了，直接返回报错，这个时候，消费方的 OpenFeign 就需要增大超时时间

```
# 设置 feign 客户端超时时间(OpenFeign 默认支持 ribbon)
ribbon:
  # 指的是建立连接所用的时间,适用于网络状态正常的情况下,两端连接所用的时间
  ReadTimeout: 5000
  # 指的是建立连接后从服务器读取到可用资源所用的时间
  ConnectTimeout: 5000
```

## OpenFeign 日志打印功能

### 概念

Feign 提供了日志打印功能，我们可以通过配置来调整日志级别，从而了解 Feign 中 Http 请求的细节，说白了就是对 Feign 接口的调用情况进行监控和输出。

### 日志级别

- **NONE:** 默认的，不显示任何日志

- **BASIC**: 仅记录请求方法、URL、相应状态码以及执行时间
- **HEADERS**: 除了 BASIC 中定义的信息之外，还有请求和响应头的信息
- **FULL**: 除了 HEADERS 中定义的信息之外，还有请求和相应的正文及元数据

```
@Configuration
public class FeignConfig {
    /**
     * feignClient 配置日志级别
     *
     * @return
     */
    @Bean
    public Logger.Level feignLoggerLevel() {
        // 请求和响应的头信息,请求和响应的正文及元数据
        return Logger.Level.FULL;
    }
}
```

## 修改 YML 文件

```
logging:
  level:
    # feign 日志以什么级别监控哪个接口
    com.atguigu.springcloud.service.PaymentFeignService: debug
```

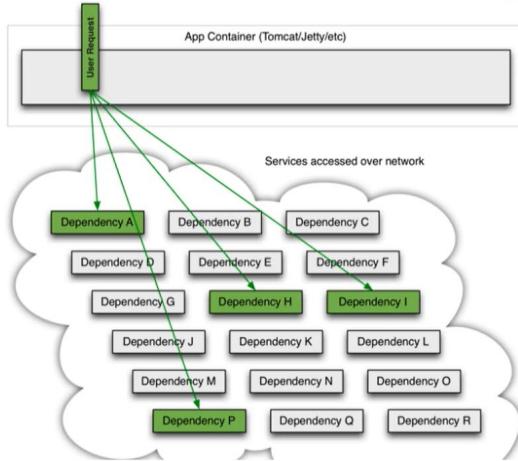
## Hystrix 断路器

Hystrix 官宣停更，官方推荐使用：resilience4j 替换，同时国内 Spring Cloud Alibaba 提出了 Sentinel 实现熔断和限流

### 概述

## 分布式面临的问题

复杂分布式体系结构中的应用程序有数十个依赖关系，每个依赖关系在某些时候将不可避免地失败（网络卡顿，网络超时）



左图中的请求需要调用A, P, H, I四个服务，如果一切顺利则没有什么问题，关键是如何I服务超时会出现什么情况呢？



## 服务雪崩

多个微服务之间调用的时候，假设微服务 A 调用微服务 B 和微服务 C，微服务 B 和微服务 C 又调用其它的微服务，这就是所谓的“扇出”。如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务 A 的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的雪崩效应

对于高流量的应用来说，单一的后端依赖可能会导致所有服务器上的所有资源都在几秒钟内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其它系统资源紧张，导致整个系统发生更多的级联故障，这些都表示需要对故障和延迟进行隔离和管理，以便单个依赖关系的失败，不能取消整个应用程序或系统。

通常当你发现一个模块下的某个实例失败后，这时候这个模块依然还会接收流量，然后这个有问题的模块还调用了其他的模块，这样就会发生级联故障，或者叫雪崩

## HyStrix 的诞生

HyStrix 是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时，异常等，HyStrix 能够保证在一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

断路器本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似于熔断保险丝），向调用方返回一个符合预期的，可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中蔓延，乃至雪崩。

## Hystrix 作用

- 服务降级
- 服务熔断
- 接近实时的监控（Hystrix Dashboard）
- . . .

## Hystrix 重要概念

### 服务降级

fallback，假设对方服务不可用了，那么至少需要返回一个兜底的解决方法，即向服务调用方返回一个符合预期的，可处理的备选响应。

例如：服务繁忙，请稍后再试，不让客户端等待并立刻返回一个友好的提示，  
fallback

#### 哪些情况会触发降级

- 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

### 服务熔断

break，类比保险丝达到了最大服务访问后，直接拒绝访问，拉闸断电，然后调用服务降级的方法并返回友好提示

一般过程：服务降级 -> 服务熔断 -> 恢复调用链路

### 服务限流

flowlimit，秒杀高并发等操作，严禁一窝蜂的过来拥挤，大家排队，一秒钟 N 个，有序进行

## Hystrix 案例

### 构建

#### 引入依赖

```
<!--hystrix-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

#### 启动类添加 Hystrix 注解

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class PaymentHystrixMain8001 {
    public static void main(String[] args) {
        SpringApplication.run(PaymentHystrixMain8001.class, args);
    }
    /**
     * 此配置是为了服务监控而配置，与服务容错本身无关，springCloud 升级之后的坑
     * ServletRegistrationBean 因为 springboot 的默认路径不是/hystrix.stream
     * 只要在自己的项目中配置上下面的 servlet 即可
     * @return
     */
    @Bean
    public ServletRegistrationBean getServlet(){
        HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();
        ServletRegistrationBean<HystrixMetricsStreamServlet> registrationBean = new ServletRegistrationBean<>(streamServlet);
        registrationBean.setLoadOnStartup(1);
        registrationBean.addUrlMappings("/hystrix.stream");
        registrationBean.setName("HystrixMetricsStreamServlet");
        return registrationBean;
    }
}
```

#### 业务类

```
@Service
public class PaymentService {
    /**
     * 正常访问
     *
     * @param id
     * @return
     */
```

```

        */
    public String paymentInfo_OK(Integer id) {
        return "线程池:" + Thread.currentThread().getName() + " payment
Info_OK,id:" + id + "\t" + "0(∩_∩)O 哈哈~";
    }

    /**
     * 超时访问
     *
     * @param id
     * @return
     */
    @HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler", commandProperties = {
        @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000")
    })
    public String paymentInfo_TimeOut(Integer id) {
        int timeNumber = 3;
        try { TimeUnit.SECONDS.sleep(timeNumber); } catch (InterruptedException e) { e.printStackTrace(); }
        return "线程池:" + Thread.currentThread().getName() + " payment
Info_TimeOut,id:" + id + "\t" +
            "0(∩_∩)O 哈哈~ 耗时(秒)";
    }

    public String paymentInfo_TimeOutHandler(Integer id){
        return "线程池:" + Thread.currentThread().getName() + " 8001 系
统繁忙请稍后再试！！ ,id:" + id + "\t"+"我哭了！！ ";
    }

    //====服务熔断，上方是降级
    /**
     * 在 10 秒窗口期中 10 次请求有 6 次是请求失败的，断路器将起作用
     * @param id
     * @return
     */
    @HystrixCommand(
        fallbackMethod = "paymentCircuitBreaker_fallback", commandProperties = {
            @HystrixProperty(name = "circuitBreaker.enabled", value = "true"),// 是否开启断路器
            @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"),// 请求次数
            @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000"),// 时间窗口期/时间范文
            @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60")// 失败率达到多少后跳闸
    }
)

```

```

    }
}
public String paymentCircuitBreaker(@PathVariable("id") Integer id) {
    if (id < 0) {
        throw new RuntimeException("*****id 不能是负数");
    }
    String serialNumber = IdUtil.simpleUUID();
    return Thread.currentThread().getName() + "\t" + "调用成功,流水号:" + serialNumber;
}

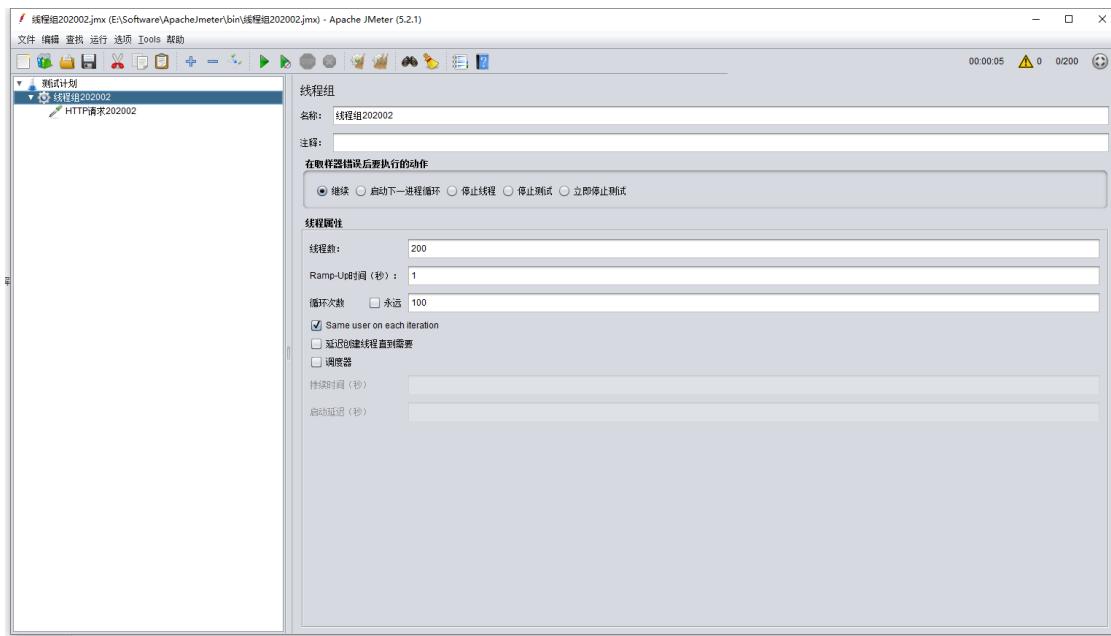
public String paymentCircuitBreaker_fallback(@PathVariable("id") Integer id) {
    return "id 不能负数,请稍后重试,o(╥﹏╥)o id:" + id;
}
}

```

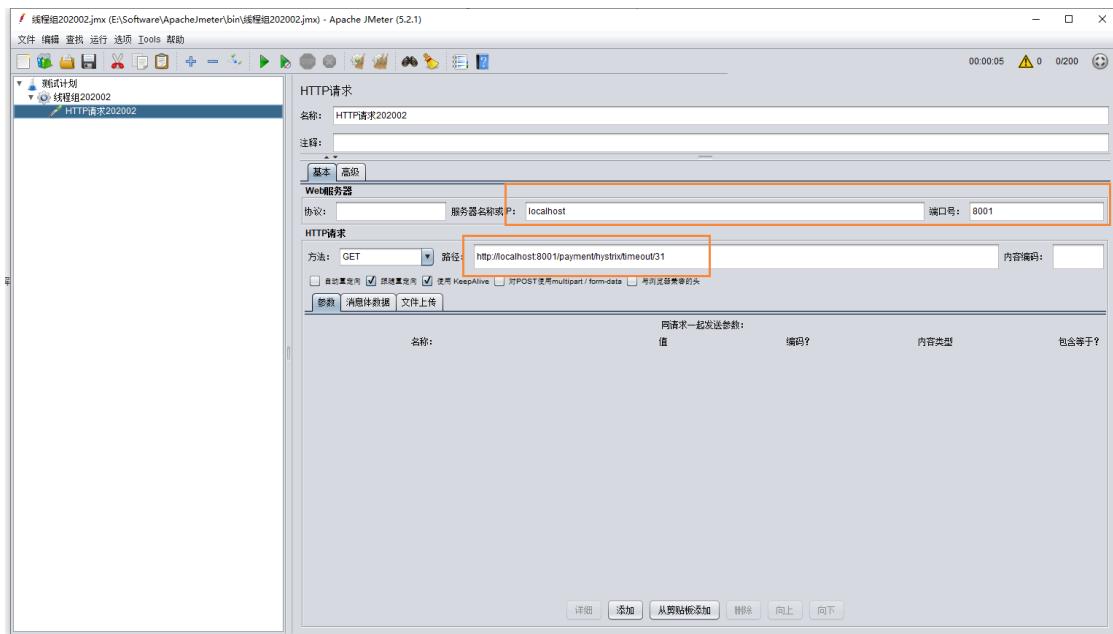
## 高并发测试

Jmeter 高并发测试

我们创建 20000 个线程去访问



访问刚刚我们写的两个 延时接口



我们会发现当线程多的时候，会直接卡死，甚至把其它正常的接口都已经拖累。这是因为我们使用 20000 个线程去访问那个延时的接口，这样会把该微服务的资源全部集中处理 延时接口，而导致正常的接口资源不够，出现卡顿的现象。

同时 tomcat 的默认工作线程数被打满，没有多余的线程来分解压力和处理。

## 服务消费者加入

刚刚我们能够看到，光是调用服务提供者就不能支持 20000 的并发量，这个时候，在使用服务消费者的引入，同时请求该接口，这个时候我们就需要在服务消费端设置服务降级了

首先启动 hystrix，`@EnableHystrix`

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
@EnableHystrix
public class OrderHystrixMain80 {
    public static void main(String[] args) {
        SpringApplication.run(OrderHystrixMain80.class, args);
    }
}
```

然后修改 yml

```

server:
  port: 80
eureka:
  client:
    register-with-eureka: false
    fetch-registry: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.
com:7002/eureka
  # 设置 feign 客户端超时时间(OpenFeign 默认支持 ribbon)
feign:
  hystrix:
    enabled: true

```

最后我们在 feign 调用上增加 fallBack

```

@Component
@FeignClient(value = "cloud-provider-hystrix-payment", fallback = PaymentFallbackService.class)
public interface PaymentHystrixService {
    /**
     * 正常访问
     *
     * @param id
     * @return
     */
    @GetMapping("/payment/hystrix/ok/{id}")
    public String paymentInfo_OK(@PathVariable("id") Integer id);
    /**
     * 超时访问
     *
     * @param id
     * @return
     */
    @GetMapping("/payment/hystrix/timeout/{id}")
    public String paymentInfo_TimeOut(@PathVariable("id") Integer id);
}

```

同时 fallback 的方法为

```

@Component
public class PaymentFallbackService implements PaymentHystrixService {
    @Override
    public String paymentInfo_OK(Integer id) {
        return "--- PaymentFallbackService fall paymentInfo_OK vack ,
/(ㄒ o ㄒ)/~~";
    }

    @Override

```

```

public String paymentInfo_TimeOut(Integer id) {
    return "--- PaymentFallbackService fall paymentInfo_TimeOut,
/(ㄒ o ㄒ)/~~";
}

```

## 解决方案

### 原因

超时导致服务器变慢，超时不再等待

出错，宕机或者程序运行出错，出错要有兜底

解决

- 对方服务 8001 超时了，调用者 80 不能一直卡死等待，必须有服务降级
- 对方服务 8001 宕机了，调用者 80 不能一直卡死，必须有服务降级
- 对方服务 8001 正常，调用者自己出故障或者有自我要求（自己的等待时间小于服务提供者），自己处理降级

## 服务降级

使用新的注解 `@HystrixCommand`

同时需要在主启动类上新增：`@EnableCircuitBreaker`

设置 8001 自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，作为服务降级 fallback

```

/**
 * 超时访问
 *
 * @param id
 * @return
 */
@HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler", commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "3000")
})
public String paymentInfo_TimeOut(Integer id) {
    int timeNumber = 3;
    try { TimeUnit.SECONDS.sleep(timeNumber); } catch (InterruptedException e) { e.printStackTrace(); }
    return "线程池：" + Thread.currentThread().getName() + " paymentInfo_TimeOut,id:" + id + "\t" +
}

```

```

        "0(∩_∩)0 哈哈~ 耗时(秒)";
    }

    /**
     * 兜底的解决方案
     * @param id
     * @return
     */
    public String paymentInfo_TimeOutHandler(Integer id){
        return "线程池:" + Thread.currentThread().getName() + " 8001 系统繁忙请稍后再试！！！，id:" + id + "\t" +"我哭了！！！";
    }
}

```

上述的方法就是当在规定的 5 秒内没有完成，那么就会触发服务降级，返回一个兜底的解决方案

同时不仅是超时，假设服务内的方法出现了异常，也同样会触发兜底的解决方法，例如下面的代码，我们制造出一个除数为 0 的异常。

```

@HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler", commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "3000")
})
public String paymentInfo_TimeOut(Integer id) {
    int timeNumber = 10 / 0;
    return "线程池:" + Thread.currentThread().getName() + " paymentInfo_TimeOut,id:" + id + "\t" +
        "0(∩_∩)0 哈哈~ 耗时(秒)";
}

```

上述说的是服务提供方的降级，服务消费者也需要设置服务降级的处理保护，也就是对客户端进行保护

也就是说服务降级，既可以放在客户端，也可以放在服务端，一般而言是放在客户端进行服务降级的

首先主启动类设置：`@EnableHystrix`

配置过的 devtool 热部署对 java 代码的改动明显，但是对`@HystrixCommand` 内属性的修改建议重启微服务

然后 yml 开启 hystrix

```

feign:
  hystrix:
    enabled: true

```

## 服务消费端降级

```
@GetMapping("/consumer/payment/hystrix/timeout/{id}")
@HystrixCommand(fallbackMethod = "paymentTimeOutFallbackMethod", commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1500")
})
public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
    return paymentHystrixService.paymentInfo_TimeOut(id);
}
```

## 目前问题

目前异常处理的方法，和业务代码耦合，这就造成耦合度比较高

解决方法就是使用统一的服务降级方法

### 方法 1：

除了个别重要核心业务有专属，其它普通的可以通过

`@DefaultProperties(defaultFallback = "")`，这样通用的和独享的各自分开，避免了代码膨胀，合理减少了代码量

```
可以在 Controller 处设置 @DefaultProperties(defaultFallback =
"payment_Global_FallbackMethod")

@RestController
@Slf4j
(DefaultProperties(defaultFallback = "payment_Global_FallbackMethod"))
public class OrderHystrixController {

    @GetMapping("/consumer/payment/hystrix/timeout/{id}")
    @HystrixCommand // 这个方法也会走全局 fallback
    public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
        int age = 10/0; //方法前挂了，跟后面挂了两种
        return paymentHystrixService.paymentInfo_TimeOut(id);
    }

    //下面是全局 fallback 方法
    public String payment_Global_FallbackMethod(){
        return "Global 异常处理信息，请稍后再试,/(ㄒoㄒ)/~~";
    }
}
```

## 方法 2:

我们现在还发现，兜底的方法和我们的业务代码耦合在一块比较混乱

我们可以在 feign 调用的时候，增加 hystrix 的服务降级处理的实现类，这样就可以进行解耦

格式：@FeignClient(fallback = PaymentFallbackService.class)

我们要面对的异常主要有

- 运行
- 超时
- 宕机

需要新建一个 FallbackService 实现类，然后通过实现类统一为 feign 接口里面的方法进行异常处理

feign 接口

```
@Component
@FeignClient(value = "cloud-provider-hystrix-payment", fallback = PaymentFallbackService.class)
public interface PaymentHystrixService {
    /**
     * 正常访问
     *
     * @param id
     * @return
     */
    @GetMapping("/payment/hystrix/ok/{id}")
    public String paymentInfo_OK(@PathVariable("id") Integer id);
    /**
     * 超时访问
     *
     * @param id
     * @return
     */
    @GetMapping("/payment/hystrix/timeout/{id}")
    public String paymentInfo_TimeOut(@PathVariable("id") Integer id);
}
```

实现类

```
@Component
public class PaymentFallbackService implements PaymentHystrixService {
    @Override
    public String paymentInfo_OK(Integer id) {
```

```
        return "--- PaymentFallbackService fall paymentInfo_OK vack ,  
/(ㄒ o ㄒ)/~~";  
    }  
  
    @Override  
    public String paymentInfo_TimeOut(Integer id) {  
        return "--- PaymentFallbackService fall paymentInfo_TimeOut,  
/(ㄒ o ㄒ)/~~";  
    }  
}
```

这个时候，如果我们将服务提供方进行关闭，但是我们在客户端做了服务降级处理，让客户端在服务端不可用时，也会获得提示信息，而不会挂起耗死服务器

## 服务熔断

服务熔断也是服务降级的一个特例

### 熔断概念

熔断机制是应对雪崩效应的一种微服务链路保护机制，当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回错误的响应状态

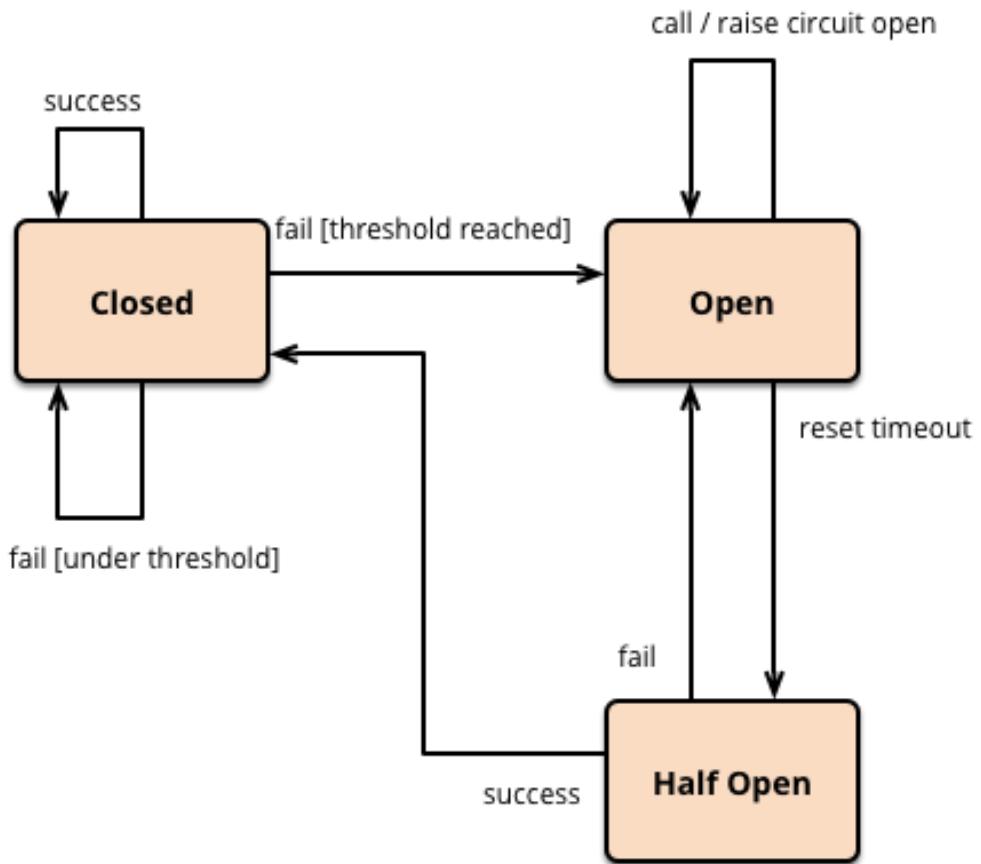
当检测到该节点微服务调用响应正常后，恢复调用链路

在 Spring Cloud 框架里，熔断机制通过 Hystrix 实现，Hystrix 会监控微服务间调用的状况，当失败的调用到一定的阈值，缺省是 5 秒内 20 次调用失败，就会启动熔断机制，熔断机制的注解还是 `@HystrixCommand`

来源，微服务提出者马丁福勒：

<https://martinfowler.com/bliki/CircuitBreaker.html>

这个简单的断路器避免了在电路打开时进行保护调用，但是当情况恢复正常时需要外部干预来重置它。对于建筑物中的断路器，这是一种合理的方法，但是对于软件断路器，我们可以让断路器本身检测底层调用是否再次工作。我们可以通过在适当的间隔之后再次尝试 `protected` 调用来实现这种自重置行为，并在断路器成功时重置它



熔断器的三种状态：打开，关闭，半开

这里提出了半开的概念，首先打开一半的，然后慢慢的进行恢复，最后在把断路器关闭

降级 -> 熔断 -> 恢复

这里我们在服务提供方 8001，增加服务熔断

这里有四个字段

```

// 是否开启断路器
@HystrixProperty(name = "circuitBreaker.enabled", value = "true"),

// 请求次数
@HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value
= "10"),

// 时间窗口期/时间范文
  
```

```

@HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000"),
// 失败率达到多少后跳闸
@HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60")

```

首先是是否开启熔断器，然后是在一个时间窗口内，有 60% 的失败，那么就启动断路器，也就是 10 次里面，6 次失败，完整代码如下：

```

/**
 * 在 10 秒窗口期中 10 次请求有 6 次是请求失败的，断路器将起作用
 * @param id
 * @return
 */
@HystrixCommand(
    fallbackMethod = "paymentCircuitBreaker_fallback", commandProperties = {
        @HystrixProperty(name = "circuitBreaker.enabled", value = "true"),// 是否开启断路器
        @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"),// 请求次数
        @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000"),// 时间窗口期/时间范文
        @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60")// 失败率达到多少后跳闸
    }
)
public String paymentCircuitBreaker(@PathVariable("id") Integer id) {
    if (id < 0) {
        throw new RuntimeException("*****id 不能是负数");
    }
    String serialNumber = IdUtil.simpleUUID();
    return Thread.currentThread().getName() + "\t" + "调用成功，流水号：" + serialNumber;
}

```

当断路器被打开的时候，即使是正确的请求，该方法也会被断路

## 总结

### 熔断类型

- 熔断打开：请求不再进行调用当前服务，内部设置时钟一般为 MTTR（平均故障处理时间），当打开时长达所设时钟则进入半熔断状态
- 熔断关闭：熔断关闭不会对服务进行熔断

- 熔断半开：部分请求根据规则调用当前服务，如果请求成功且符合规则，则认为当前服务恢复正常，关闭熔断

## 断路器启动条件

```

@HystrixCommand(
    fallbackMethod = "paymentCircuitBreaker_fallback", commandProperties = {
        @HystrixProperty(name = "circuitBreaker.enabled", value = "true"),// 是否开启断路器
        @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"),// 请求次数
        @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000"),// 时间窗口期/时间范文
        @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60")// 失败率达到多少后跳闸
    }
)

```

涉及到断路器的三个重要参数：快照时间窗，请求总阈值，错误百分比阈值

- 快照时间窗：断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，默认为最近的 10 秒。
- 请求总数阈值：在快照时间窗口内，必须满足请求数量阈值才有资格熔断。默认为 20，意味着在 10 秒内，如果 hystrix 的调用总次数不足 20 次，即使所有请求都超时或者其他原因失败，断路器都不会打开。
- 错误百分比阈值：当请求数量在快照时间窗内超过了阈值，比如发生了 30 次调用，并且有 15 次发生了超时异常，也就是超过了 50% 的错误百分比，在默认设定的 50% 阈值情况下，这时候就会将断路器打开

## 开启和关闭的条件

- 当满足一定阈值的时候（默认 10 秒内超过 20 个请求）
- 当失败率达到一定的时候（默认 10 秒内超过 50% 的请求失败）
- 到达以上阈值，断路器将会开启
- 当开启的时候，所有请求都不会进行转发
- 一段时间之后（默认是 5 秒），这个时候断路器是半开状态，会让其中一个请求进行转发。如果成功，断路器会关闭，若失败，继续开启，重复 4 和 5

## 断路器开启后

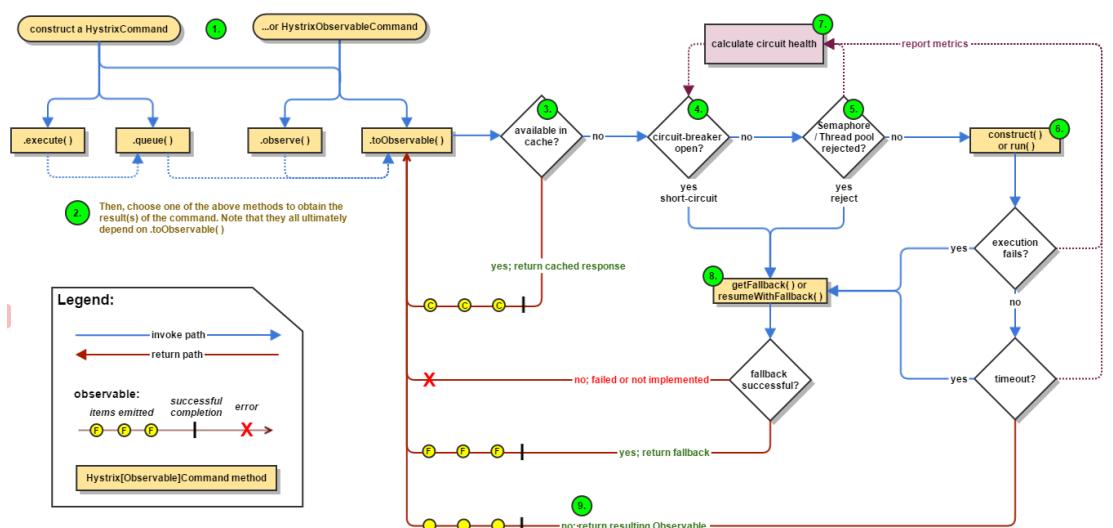
- 再有请求调用的时候，将不会调用主逻辑，而是直接调用降级 fallback，通过断路器，实现了自动的发现错误并将降级逻辑切换为主逻辑，减少相应延迟的效果。
- 原来的主逻辑如何恢复？

- 对于这个问题，Hystrix 实现了自动恢复功能，当断路器打开，对主逻辑进行熔断之后，hystrix 会启动一个休眠时间窗，在这个时间窗内，降级逻辑是临时的成为主逻辑，当休眠时间窗到期，断路器将进入半开状态，释放一次请求到原来的主逻辑上，如果此次请求正常返回，断路器将继续闭合，主逻辑恢复，如果这次请求依然有问题，断路器继续保持打开状态，休眠时间窗重新计时。

## 服务限流

后面讲解 Sentinel 的时候进行说明

## Hystrix 工作流程



蓝色：调用路径

红色：返回路径

完整的请求路线：

1. 选择一个 Hystrix 注册方式
2. 二选一即可
3. 判断缓存中是否包含需要返回的内容（如果有直接返回）
4. 断路器是否为打开状态（如果是，直接跳转到 8，返回）
5. 断路器为健康状态，判断是否有可用资源（没有，直接跳转 8）

6. 构造方法和 Run 方法
7. 将正常，超时，异常的消息发送给断路器
8. 调用 getFallback 方法，也就是服务降级
9. 直接返回正确结果

## 服务监控 HystrixDashboard

### 概述

除了隔离依赖服务的调用以外，Hystrix 还提供了准实时的调用监控（Hystrix Dashboard），Hystrix 会持续地记录所有通过 Hystrix 发起的请求的执行信息，并以统计报表和图形化的形式展示给用户，包括每秒执行多少请求，成功多少请求，失败多少，Netflix 通过 Hystrix-metrics-event-stream 项目实现了对以上指标的监控，SpringCloud 也提供了 HystrixDashboard 整合，对监控内容转化成可视化页面

### 搭建

引入依赖

```
<!--hystrix dashboard-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>

<!--监控-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

application.yml 添加端口

```
server:
  port: 9001
```

主启动类：配置注解@EnableHystrixDashboard

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardMain9001 {
    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardMain9001.class);
```

```
    }  
}
```

同时，最后我们需要注意，每个服务类想要被监控的，都需要在 pom 文件中，添加一下注解

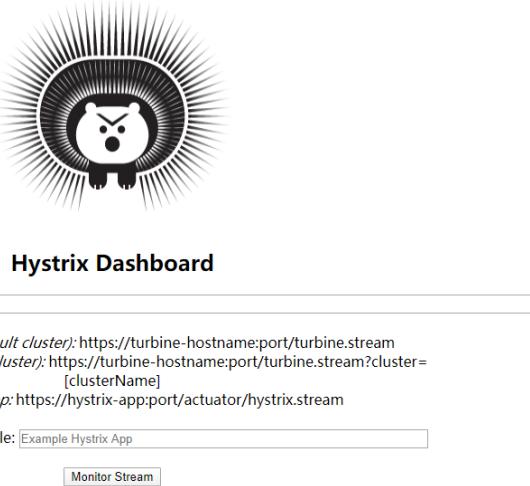
```
<!--监控-->  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>
```

同时在服务提供者的启动类上，需要添加以下的内容

```
@SpringBootApplication  
@EnableDiscoveryClient  
@EnableCircuitBreaker  
public class PaymentHystrixMain8001 {  
    public static void main(String[] args) {  
        SpringApplication.run(PaymentHystrixMain8001.class, args);  
    }  
    /**  
     * 此配置是为了服务监控而配置，与服务容错本身无关，springCloud 升级之后  
     * 的坑  
     * ServletRegistrationBean 因为 springboot 的默认路径不是/hystrix.stre  
     * am  
     * 只要在自己的项目中配置上下面的 servlet 即可  
     * @return  
     */  
    @Bean  
    public ServletRegistrationBean getServlet(){  
        HystrixMetricsStreamServlet streamServlet = new HystrixMetricsS  
treamServlet();  
        ServletRegistrationBean<HystrixMetricsStreamServlet> registrati  
onBean = new ServletRegistrationBean<>(streamServlet);  
        registrationBean.setLoadOnStartup(1);  
        registrationBean.addUrlMappings("/hystrix.stream");  
        registrationBean.setName("HystrixMetricsStreamServlet");  
        return registrationBean;  
    }  
}
```

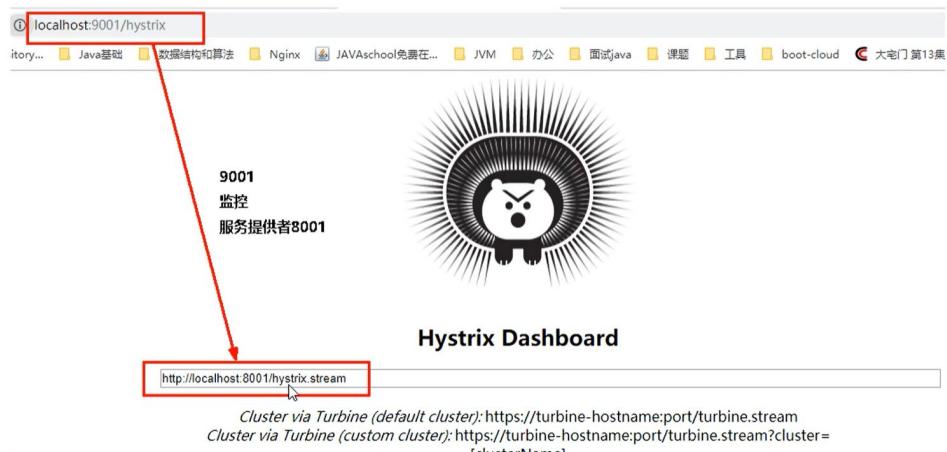
输入以下地址，进入 Hystrix 的图形化界面

<http://localhost:9001/hystrix>



## 使用监控

我们需要使用当前 hystrix 需要监控的端口号，也就是使用 9001 去监控 8001，即使用 hystrix dashboard 去监控服务提供者的端口号



然后我们运行

<http://localhost:8001/payment/circuit/31>

就能够发现 Hystrix Dashboard 能够检测到我们的请求

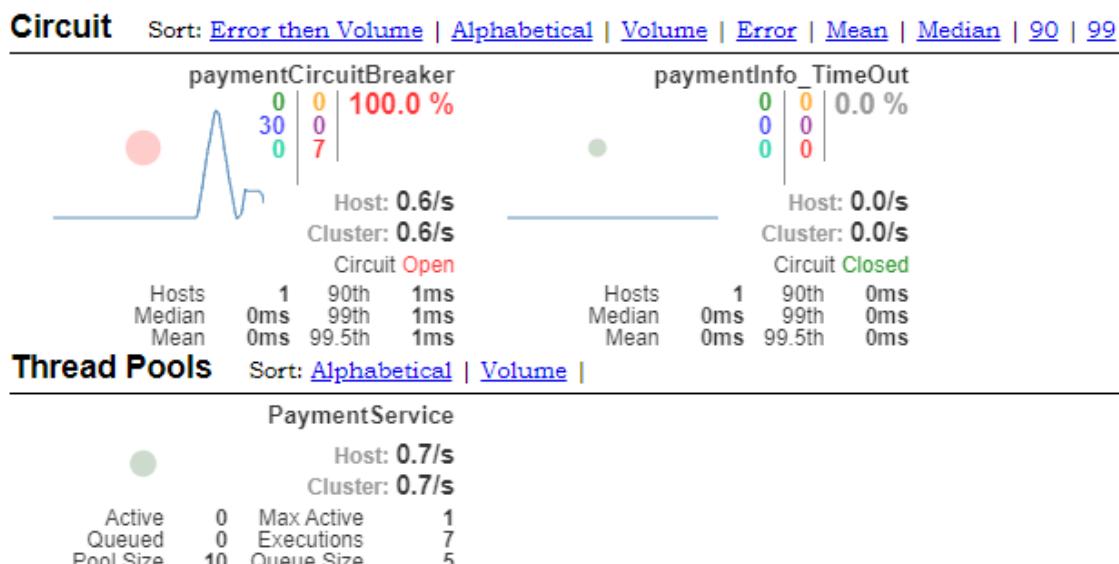
## Hystrix Stream: http://localhost:8001/hystrix.stream



假设我们访问错误的方法后

<http://localhost:8001/payment/circuit/-31>

我们能够发现，此时断路器处于开启状态，并且错误率百分 100

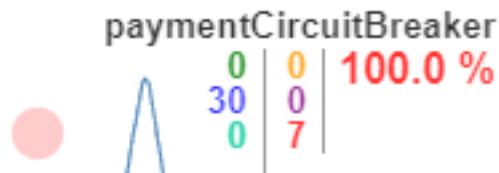


## 如何看懂图

首先是七种颜色



每个颜色都对应的一种结果



然后是里面的圆

实心圆：共有两种含义。它通过颜色的变化代表了实例的健康程度，它的健康程度从

绿色 < 黄色 < 橙色 < 红色，递减

该实心圆除了颜色变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大，所以通过该实心圆的展示，就可以快速在大量的实例中快速发现故障实例和高压力实例

曲线：用于记录 2 分钟内流量的相对变化，可以通过它来观察到流量的上升和下降趋势



## 服务网关

### 前言

zuul 目前已经出现了分歧，zuul 升级到 Zuul2 的时候出现了内部分歧，并且导致 Zuul 的核心人员的离职，导致 Zuul2 一直跳票，等了两年，目前造成的局面是 Zuul 已经没人维护，Zuul2 一直在开发中

目前主流的服务网关采用的是 Spring Cloud 社区推出了 Gateway

### 概念

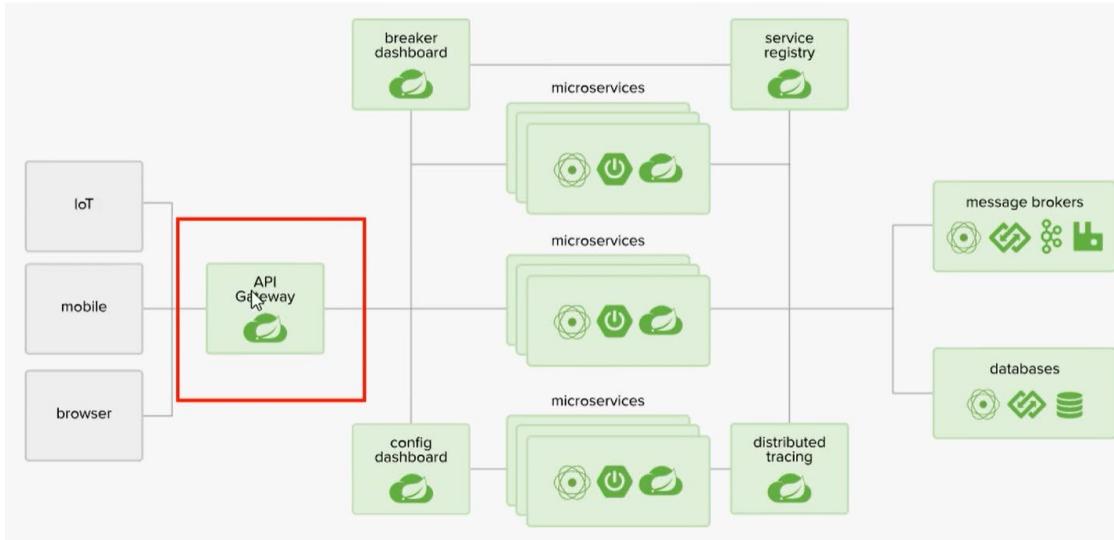
#### Zuul

官网：<https://github.com/Netflix/zuul/wiki>

Zuul 是所有来自设备和 web 站点到 Netflix 流媒体应用程序后端的请求的前门。作为一个边缘服务应用程序，Zuul 的构建是为了支持动态路由、监视、弹性和安全性。它还可以根据需要将请求路由到多个 Amazon 自动伸缩组。

#### Gateway

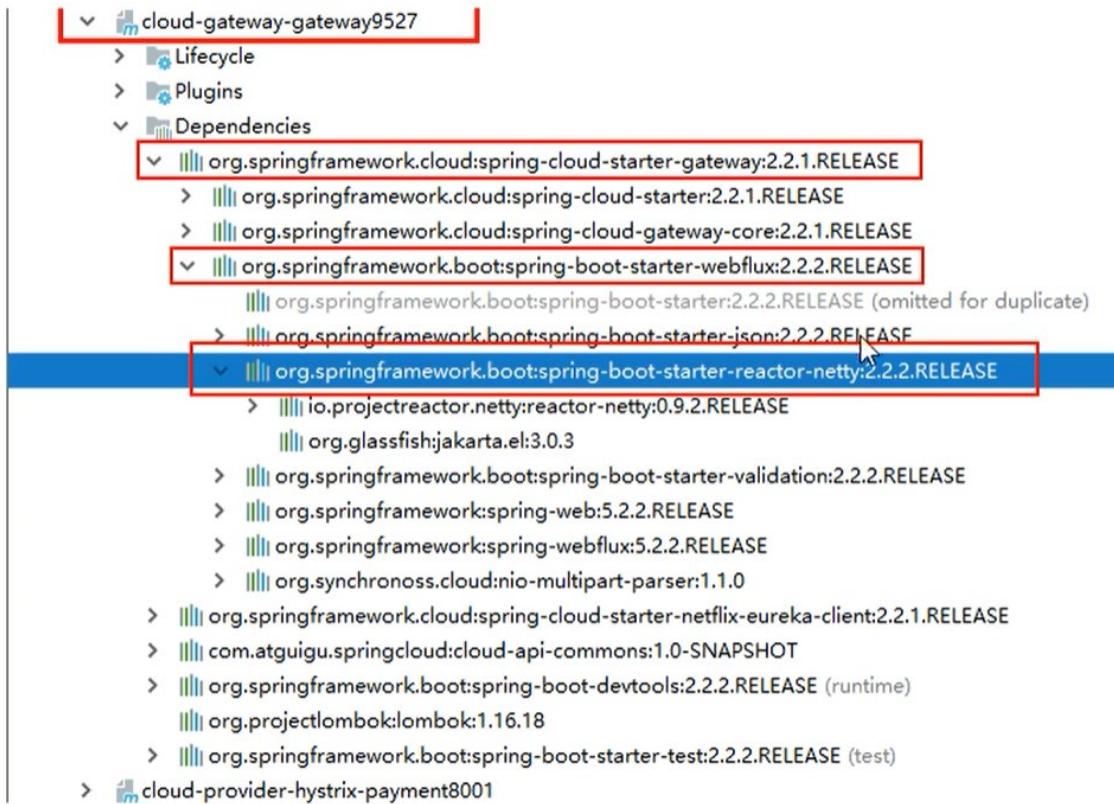
Cloud 全家桶有个很重要的组件就是网关，在 1.X 版本中都是采用 Zuul 网关，但在 2.X 版本中，zuul 的升级一直跳票，SpringCloud 最后自己研发了一个网关替代 Zuul，那就是 SpringCloudGateway，一句话 Gateway 是原来 Zuul 1.X 版本的替代品



Gateway 是在 Spring 生态系统之上构建的 API 网关服务，基于 Spring 5, Spring Boot 2 和 Project Reactor 等技术。Gateway 旨在提供一种简单而且有效的方式来对 API 进行路由，以及提供一些强大的过滤器功能，例如：熔断，限流，重试等。

Spring Cloud Gateway 是 Spring Cloud 的一个全新项目，作为 Spring Cloud 生态系统中的网关，目标是替代 Zuul，在 Spring Cloud 2.0 以上版本中，没有对新版本的 Zuul 2.0 以上最新高性能版本进行集成，仍然还是使用的 Zuul 1.X 非 Reactor 模式的老版本，而为了提高网关的性能，Spring Cloud Gateway 是基于 WebFlux 框架实现的，而 WebFlux 框架底层则使用了高性能的 Reactor 模式通信框架 Netty。

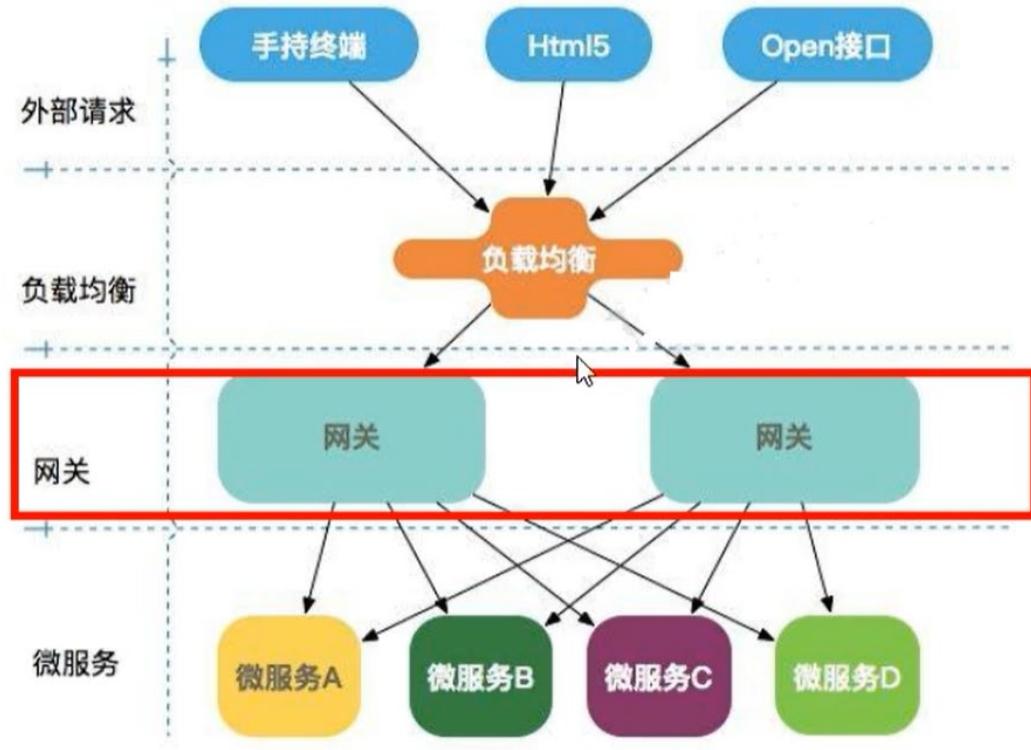
Spring Cloud Gateway 的目标提供统一的路由方式，且基于 Filter 链的方式提供了网关基本的功能，例如：安全，监控、指标和限流。



## 能做啥

- 反向代理
- 鉴权
- 流量控制
- 熔断
- 日志监控

## 使用场景



网关可以想象成是所有服务的入口

## 为什么选用 **Gateway**

目前已经有了 Zuul 了，为什么还要开发出 Gateway 呢？

一方面是因为 Zuul 1.0 已经进入了维护阶段，而且 Gateway 是 Spring Cloud 团队研发的，属于亲儿子，值得信赖，并且很多功能 Zuul 都没有用起来，同时 Gateway 也非常简单便捷

Gateway 是基于异步非阻塞模型上进行开发的，性能方面不需要担心。虽然 Netflix 早就发布了 Zuul 2.X，但是 Spring Cloud 没有整合计划，因为 NetFlix 相关组件都进入维护期，随意综合考虑 Gateway 是很理想的网关选择。

## Gateway 特性

基于 Spring Framework 5, Project Reactor 和 Spring boot 2.0 进行构建

- 动态路由，能匹配任何请求属性
- 可以对路由指定 Predicate (断言) 和 Filter (过滤器)

- 集成 Hystrix 的断路器功能
- 集成 Spring Cloud 服务发现功能
- 易于编写 Predicate 和 Filter
- 请求限流功能
- 支持路径重写

## Spring Cloud Gateway 和 Zuul 的区别

在 Spring Cloud Gateway Finchley 正式版发布之前，Spring Cloud 推荐网关是 NetFlix 提供的 Zuul

- Zuul 1.X 是一个基于阻塞 IO 的 API Gateway
- Zuul 1.x 基于 Servlet 2.5 使用阻塞架构，它不支持任何场连接，Zuul 的设计模式和 Nginx 比较像，每次 IO 操作都是从工作线程中选择一个执行，请求线程被阻塞到工作线程完成，但是差别是 Nginx 用 C++ 实现，Zuul 用 Java 实现，而 JVM 本身会有第一次加载较慢的情况，使得 Zuul 的性能较差。
- Zuul 2.X 理念更先进，想基于 Netty 非阻塞和支持长连接，但 Spring Cloud 目前还没有整合。Zuul 2.X 的性能相比于 1.X 有较大提升，在性能方面，根据官方提供的基准测试，Spring Cloud Gateway 的 RPS（每秒请求数）是 Zuul 的 1.6 倍。
- Spring Cloud Gateway 建立在 Spring 5，Spring Boot 2.X 之上，使用非阻塞 API
- Spring Cloud Gateway 还支持 WebSocket，并且与 Spring 紧密集成拥有更好的开发体验。

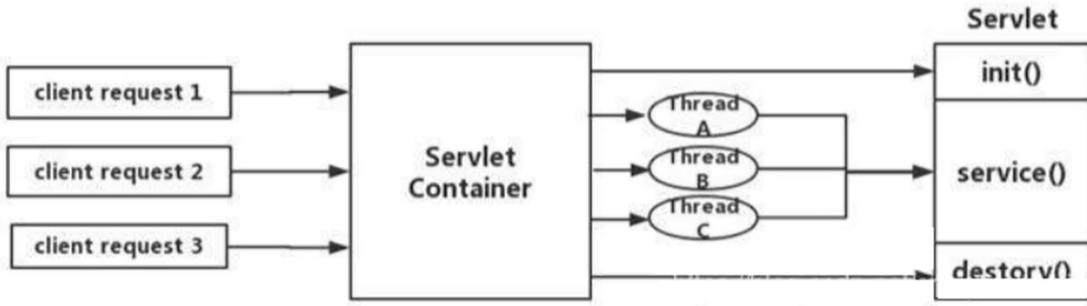
Spring Cloud 中所集成的 Zuul 版本，采用的是 Tomcat 容器，使用的还是传统的 Servlet IO 处理模型

Servlet 的生命周期中，Servlet 由 Servlet Container 进行生命周期管理。

Container 启动时构建 servlet 对象，并调用 `servlet init()` 进行初始化

Container 运行时接收请求，并为每个请求分配一个线程，（一般从线程池中获取空闲线程），然后调用 Service

container 关闭时，调用 `servlet destroy()` 销毁 servlet



上述模式的缺点：

Servlet 是一个简单的网络 IO 模型，当请求进入 Servlet container 时，Servlet container 就会为其绑定一个线程，在并发不高的场景下，这种网络模型是适用的，但是一旦高并发（Jmeter 测试），线程数就会上涨，而线程资源代价是昂贵的（上下文切换，内存消耗大），严重影响了请求的处理时间。在一些简单业务场景下，不希望为每个 Request 分配一个线程，只需要 1 个或几个线程就能应对极大并发的请求，这种业务场景下 Servlet 模型没有优势。

所以 Zuul 1.X 是基于 Servlet 之上的一种阻塞式模型，即 Spring 实现了处理所有 request 请求的 Servlet（DispatcherServlet）并由该 Servlet 阻塞式处理，因此 Zuul 1.X 无法摆脱 Servlet 模型的弊端

## WebFlux 框架

传统的 Web 框架，比如 Struts2，Spring MVC 等都是基于 Servlet API 与 Servlet 容器基础之上运行的，但是在 Servlet 3.1 之后有了异步非阻塞的支持，而 WebFlux 是一个典型的非阻塞异步的框架，它的核心是基于 Reactor 的相关 API 实现的，相对于传统的 Web 框架来说，它可以运行在如 Netty，Undertow 及支持 Servlet3.1 的容器上。非阻塞式 + 函数式编程（Spring5 必须让你使用 Java8）

Spring WebFlux 是 Spring 5.0 引入的新的响应式框架，区别与 Spring MVC，他不依赖 Servlet API，它是完全异步非阻塞的，并且基于 Reactor 来实现响应式流规范。

# 三大核心概念

## Route 路由

路由就是构建网关的基本模块，它由 ID，目标 URI，一系列的断言和过滤器组成，如果断言为 True 则匹配该路由

## Predicate 断言

参考的 Java8 的 `java.util.function.Predicate`

开发人员可以匹配 HTTP 请求中的所有内容，例如请求头和请求参数，如果请求与断言想匹配则进行路由

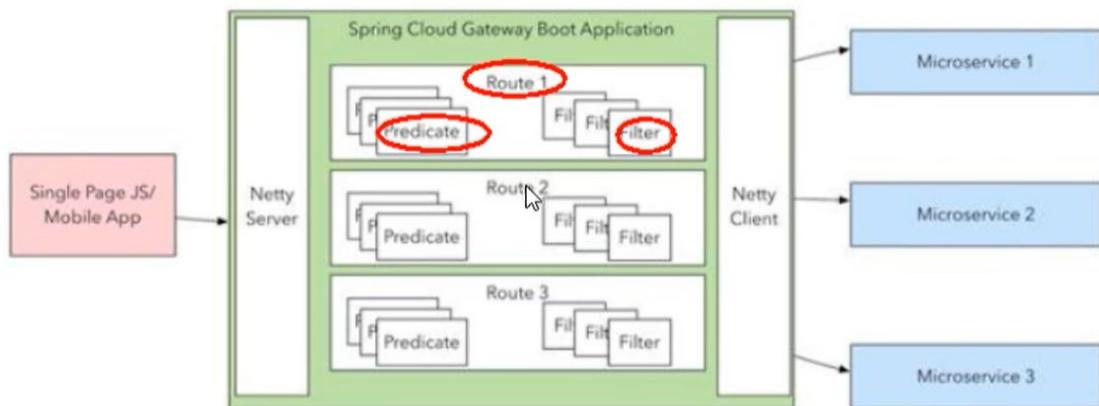
## Filter 过滤

指的是 Spring 框架中 `GatewayFilter` 的实例，使用过滤器，可以在请求被路由前或者之后对请求进行修改。

## Gateway 工作流程

Web 请求通过一些匹配条件，定位到真正的服务节点，并在这个转发过程的前后，进行了一些精细化的控制。

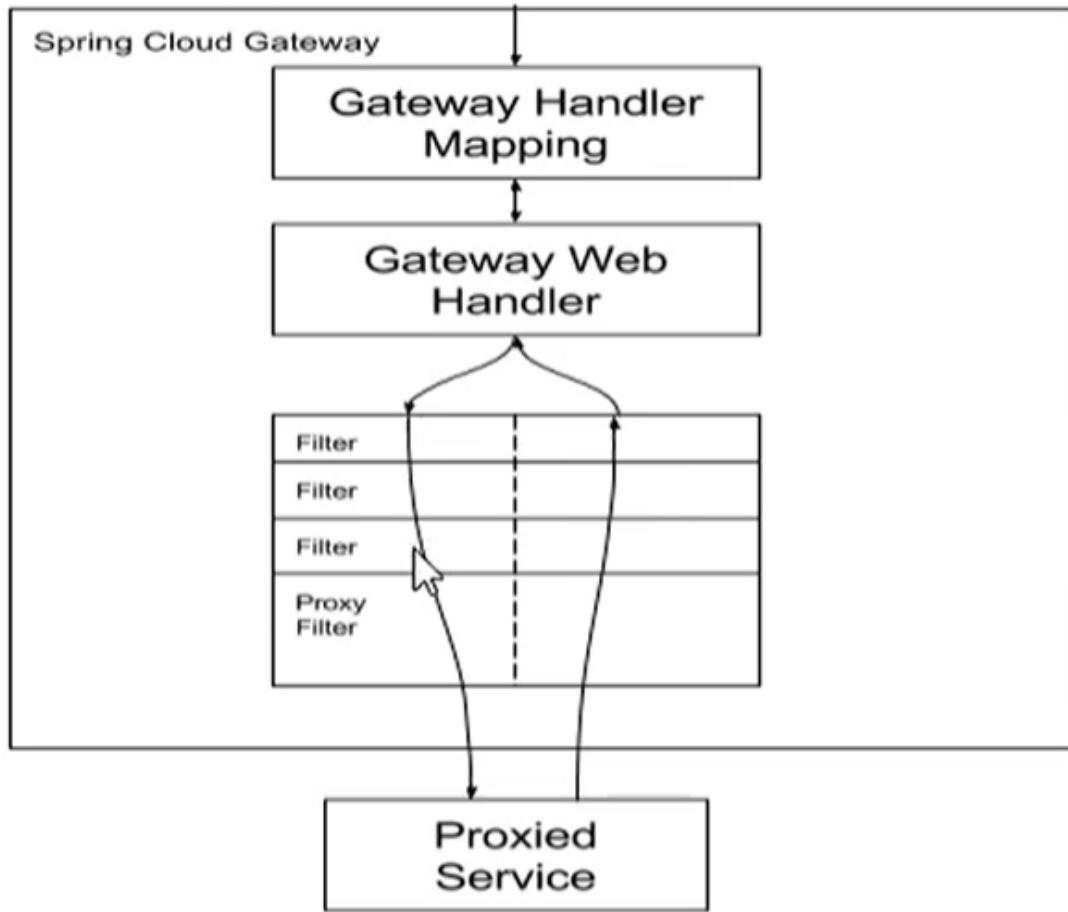
Predicate 就是我们的匹配条件，而 Filter 就可以理解为一个无所不能的拦截器，有了这两个元素，在加上目标 URL，就可以实现一个具体的路由了。



客户端向 Spring Cloud Gateway 发出请求，然后在 Gateway Handler Mapping 中找到与请求相匹配的路由，将其发送到 Gateway Web Handler。

Handler 在通过指定的过滤器链来将请求发送到我们实际的服务执行业务逻辑，然后返回。

过滤器之间用虚线分开是因为过滤器可能会在发送代理请求前（pre）或之后（post）执行业务逻辑。



Filter 在 Pre 类型的过滤器可以做参数校验，权限校验，流量监控，日志输出，协议转换等。

在 Post 类型的过滤器中可以做响应内容，响应头的修改，日志的输出，流量监控等有着非常重要的作用。

Gateway 的核心逻辑：路由转发 + 执行过滤链

## 入门配置

### 引入依赖

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
  
```

## 修改 YML

```
spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true # 开启从注册中心动态创建路由的功能，利用微服务名称
进行路由
      routes:
        - id: payment_route # 路由的 id, 没有规定规则但要求唯一, 建议配合服务
名
          #匹配后提供服务的路由地址
          #uri: http://localhost:8001
          uri: lb://CLOUD-PAYMENT-SERVICE
          predicates:
            - Path=/payment/get/** # 断言, 路径相匹配的进行路由
        - id: payment_route2
          #uri: http://localhost:8001
          uri: lb://CLOUD-PAYMENT-SERVICE
          predicates:
            - Path=/payment/lb/** #断言,路径相匹配的进行路由
            - After=2020-03-09T22:40:37.365+08:00[Asia/Shanghai]
```

## 访问

在添加网关之前，我们的访问是

<http://localhost:8001/payment/get/31>

添加网关之后，我们的访问路径是

<http://localhost:9527/payment/get/31>

这么做的好处是慢慢淡化我们真实的 IP 端口号

## 路由匹配

The screenshot shows two side-by-side code editors. On the left is the `application.yml` configuration file, and on the right is the `PaymentController.java` code.

**application.yml:**

```
server:
  port: 9527

spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
      routes:
        - id: payment_routh #payment_route
          uri: http://localhost:8001
          predicates:
            - Path=/payment/get/** # 路由的ID #匹配后缀
eureka:
  instance:
    hostname: cloud-gateway-service
    client: #服务提供者provider注册进eureka服务列表内
      service-url:
        register-with-eureka: true
```

**PaymentController.java:**

```
PaymentController > create()
{
  int result = paymentService.create(payment);
  Log.info("*****插入操作返回结果：" + result);

  if(result > 0)
  {
    return new CommonResult<Payment>(code: 200, message: "成功");
  }else{
    return new CommonResult<Payment>(code: 444, message: "失败");
  }
}

@GetMapping(value = "/payment/get/{id}")
public CommonResult<Payment> getPaymentById(@PathVariable("id") Integer id)
{
  Payment payment = paymentService.getPaymentById(id);
  Log.info("*****查询结果:{}" , payment);
  if (payment != null) {
    return new CommonResult<Payment>(code: 200, message: "成功");
  }else{
    return new CommonResult<Payment>(code: 444, message: "失败");
  }
}
```

A red box highlights the `Path=/payment/get/**` predicate in the YAML configuration, which corresponds to the `@GetMapping` annotation in the Java code.

## 路由配置的两种方式

- 在配置文件 yml 中配置
- 代码中注入 `RouteLocator` 的 Bean

```
@Configuration
public class GateWayConfig {

  // 配置了一个 id 为 route-name 的路由规则，当访问地址 http://localhost:9527/guonei 时，会自动转发到
  // 地址 http://news.baidu.com/guonei
  @Bean
  public RouteLocator customRouteLocator(RouteLocatorBuilder routeLocatorBuilder){
    RouteLocatorBuilder.Builder routes = routeLocatorBuilder.routes();
    routes.route("path route atguigu",
      r ->r.path("/guonei").uri("https://www.baidu.com")).build();
    return routes.build();
  }
}
```

## 通过微服务名实现动态路由

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。

首先需要开启从注册中心动态创建路由的功能，利用微服务名进行路由

```
spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true # 开启从注册中心动态创建路由的功能，利用微服务名称
进行路由
```

URL 换成服务名

```
uri: lb://CLOUD-PAYMENT-SERVICE
```

## Predicate 的使用

### 概念

断言，路径相匹配的进行路由

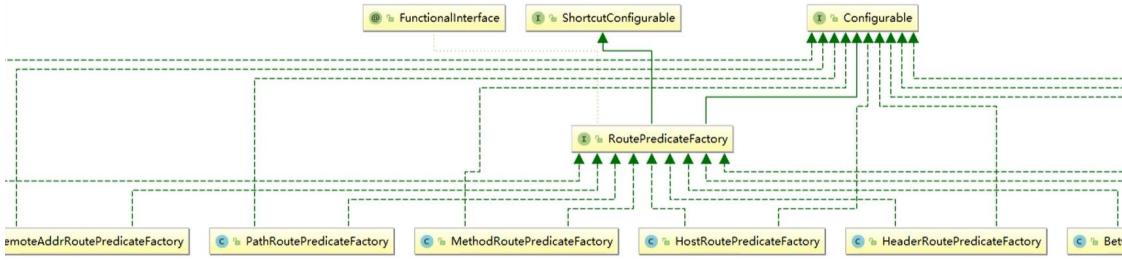
```
c.n.c.sources.URLConfigurationSource
o.s.c.g.r.RouteDefinitionRouteLocator
o.s.b.a.e.web.EndpointLinksResolver
o.s.b.d.a.OptionalLiveReloadServer
: To enable URLs as dynamic configuration sources, define System property
: Loaded RoutePredicateFactory [After]
: Loaded RoutePredicateFactory [Before]
: Loaded RoutePredicateFactory [Between]
: Loaded RoutePredicateFactory [Cookie]
: Loaded RoutePredicateFactory [Header]
: Loaded RoutePredicateFactory [Host]
: Loaded RoutePredicateFactory [Method]
: Loaded RoutePredicateFactory [Path]
: Loaded RoutePredicateFactory [Query]
: Loaded RoutePredicateFactory [ReadBodyPredicateFactory]
: Loaded RoutePredicateFactory [RemoteAddr]
: Loaded RoutePredicateFactory [Weight]
: Loaded RoutePredicateFactory [CloudFoundryRouteService]
: Exposing 2 endpoint(s) beneath base path /actuator
: Unable to start LiveReload server
```

Spring Cloud Gateway 将路由匹配作为 Spring WebFlux HandlerMapping 基础架构的一部分

Spring Cloud Gateway 包括许多内置的 Route Predicate 工厂，所有这些 Predicate 都与 Http 请求的不同属性相匹配，多个 Route Predicate 工厂可以进行组合

Spring Cloud Gateway 创建 Route 对象时，使用 RoutePredicateFactory 创建 Predicate 对象，Predicate 对象可以赋值给 Route，SpringCloudGateway 包含许多内置的 RoutePredicateFactories。

所有这些谓词都匹配 Http 请求的不同属性。多种谓词工厂可以组合，并通过逻辑 and



## 常用的 Predicate

- After Route Predicate: 在什么时间之后执行

### 113.1 After Route Predicate Factory

The After Route Predicate Factory takes one parameter, a datetime. This predicate matches requests that happen after the current datetime.

```

spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: https://example.org
          predicates:
            - After=2017-01-20T17:42:47.789-07:00[America/Denver]

```

- Before Route Predicate: 在什么时间之前执行
- Between Route Predicate: 在什么时间之间执行
- Cookie Route Predicate: Cookie 级别

常用的测试工具：

- jmeter
- postman
- curl

```
// curl 命令进行测试，携带 Cookie
curl http://localhost:9527/payment/lb --cookie "username=zzyy"
```

- Header Route Predicate: 携带请求头
- Host Route Predicate: 什么样的 URL 路径过来

- Method Route Predicate: 什么方法请求的, Post, Get
- Path Route Predicate: 请求什么路径 - Path=/api-web/\*\*
- Query Route Predicate: 带有什么参数的

## Filter 的使用

### 概念

路由过滤器可用于修改进入的 HTTP 请求和返回的 HTTP 响应，路由过滤器只能指定路由进行使用

Spring Cloud Gateway 内置了多种路由过滤器，他们都由 GatewayFilter 的工厂类来产生的

### Spring Cloud Gateway Filter

生命周期: only Two: pre, Post

种类: Only Two

- GatewayFilter
- GlobalFilter

### 自定义过滤器

主要作用:

- 全局日志记录
- 统一网关鉴权

需要实现接口: implements GlobalFilter, Ordered

全局过滤器代码如下:

```
@Component
@Slf4j
public class MyLogGateWayFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        log.info("come in global filter: {}", new Date());
    }
}
```

```

        ServerHttpRequest request = exchange.getRequest();
        String uname = request.getQueryParams().getFirst("uname");
        if (uname == null) {
            log.info("用户名为 null, 非法用户");
            exchange.getResponse().setStatusCode(HttpStatus.NOT_ACCEPTABLE);
        }
        // 放行
        return chain.filter(exchange);
    }

    /**
     * 过滤器加载的顺序 越小,优先级别越高
     *
     * @return
     */
    @Override
    public int getOrder() {
        return 0;
    }
}

```

## 分布式配置中心 SpringCloudConfig

### 概述

#### 面临的问题

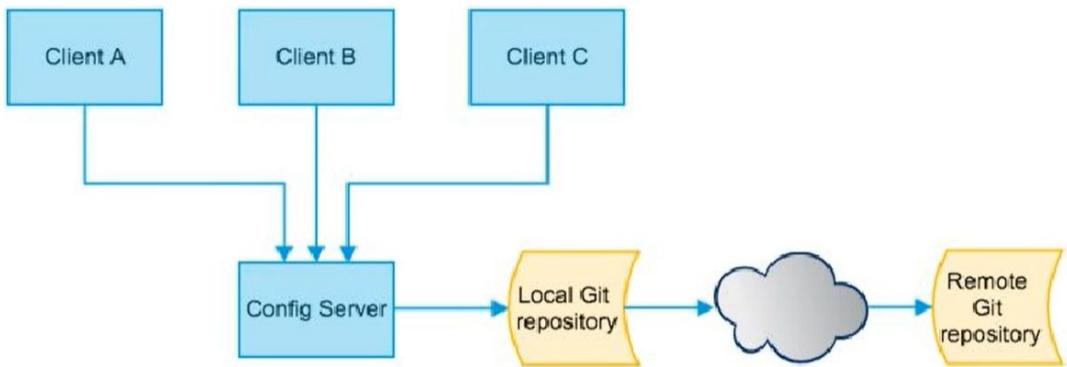
微服务意味着要将单体应用中的业务拆分成一个个子服务，每个服务的粒度相对较小，因此系统中会出现大量的服务，由于每个服务都需要必要的配置信息才能运行，所以一套集中式，动态的配置管理设施是必不可少的。

SpringCloud 提供了 ConfigServer 来解决这个问题，原来四个微服务，需要配置四个 application.yml，但需要四十个微服务，那么就需要配置 40 份配置文件，我们需要做的就是一处配置，到处生效。

所以这个时候就需要一个统一的配置管理

### 是什么

SpringCloud Config 为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用提供了一个中心化的外部配置。



## 怎么玩

服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置服务器并为客户端提供获取配置信息，加密/解密信息等访问接口。

客户端则是通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息，配置服务器默认采用 git 来存储配置信息，这样有助于对环境配置进行版本管理，并且可以通过 git 客户端工具来方便的管理和访问配置内容。

## 能做什么

- 集中管理配置文件
- 不同环境不同配置，动态化的配置更新，分布式部署，比如 dev/test/prod/beta/release
- 运行期间动态调整配置，不再需要在每个服务部署的机器上编写配置文件，服务会向配置中心统一拉取自己的信息
- 当配置发生变动时，服务不需要重启即可感知配置的变化并应用新的配置
- 将配置信息以 REST 接口的形式暴露：post, curl 命令刷新

## 与 Github 整合部署

由于 SpringCloud Config 默认使用 Git 来存储配置文件（也有其他方式，比如支持 SVN 和本地文件），但最推荐的还是 Git，而且使用的是 Http/https 访问的形式

## Config 服务端配置与测试

### 引入依赖

```
<!--添加消息总线 Rabbitmq 支持-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>

<!--config-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

### 修改 YML

```
server:
  port: 3344
spring:
  application:
    name: cloud-config-center
  cloud:
    config:
      server:
        git:
          uri: https://github.com/boytian/springcloud-config.git
          search-paths:
            - springcloud-config
      label: master
  rabbitmq: #mq 相关配置
    host: localhost
    port: 5672
    username: guest
    password: guest
eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
# rabbitmq 相关配置，暴露 bus 刷新点
management:
  endpoints: #暴露 bus 刷新配置的端点
  web:
    exposure:
      include: 'bus-refresh'
```

## 配置读取规则

- /{label}/{application}-{profile}.yml
  - http://config-3344.com:3344/master/config-dev.yml
- /{application}-{profile}.yml
  - http://config-3344.com:3344/config-dev.yml: 如果不写的话， 默认就是从 master 分支上找

## 参数总结

label: 分支， branch

name: 服务名

profiles: 环境(dev/test/prod)

## Config 客户端配置与测试

### 引入依赖

```
<!--添加消息总线 Rabbitmq 支持-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>

<!--config-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

### bootstrap.yml

application.yml: 是用户级的资源配置项

bootstrap.yml: 是系统级别的，优先级更加高

Spring Cloud 会创建一个 Bootstrap Context，作为 Spring 应用的 Application Context 的父上下文。初始化的时候，Bootstrap Context 负责从外部源加载配置属性并解析配置。这两个上下文共享一个从外部获取的 Environment。

Bootstrap 属性有高优先级，默认情况下，他们不会被本地配置覆盖，Bootstrap context 和 Application Context 有着不同的约定，所以新增了一个 bootstrap.yml 文件，保证 Bootstrap Context 和 Application Context 配置的分离。

要将客户端 Client 模块下的 Application.yml 文件改成 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

```
server:
  port: 3355
spring:
  application:
    name: config-client
  cloud:
    config:
      label: master #分支名称
      name: config #配置文件名称
      profile: dev #读取后缀文件名, 即 master 分支 config-dev.yml
      uri: http://localhost:3344 #配置中心地址

eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.
com:7002/eureka
#暴露监控端点
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

## 存在的问题

分布式配置的动态刷新问题？

- Linux 运维修改 Github 上的配置文件内容做调整
- 刷新 3344，发现 ConfigServer 配置中心立刻响应
- 刷新 3355，发现 ConfigClient 客户端没有任何响应
- 3355 没有变化除非自己重启或者重启加载
- 难道每次运维修改后，都需要重启？

相当于直接修改 Github 上的配置文件，配置不会改变，这个时候就存在了分布式配置的动态刷新问题

## Config 客户端之动态刷新

为了避免每次更新配置都要重启客户端微服务 3355

### 引入了动态刷新

引入 actuator 监控依赖

```
<!--web 启动器-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!--监控-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

然后修改 bootstrap.yml，暴露监控端点

```
#暴露监控端点
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

在业务类中，添加 @RefreshScope 标签

```
@RestController
//实现刷新功能
@RefreshScope
public class ConfigClientController {

    @Value("${config.info}")
    private String configInfo;

    @GetMapping("/configInfo")
    public String getConfigInfo(){
        return configInfo;
    }

}
```

然后让运维人员发送一个 post 请求

```
curl -X POST "http://localhost:3355/actuator/refresh"
```

然后就能够生效了，成功刷新了配置，避免了服务重启

这个方案存在问题：

- 假设有多个微服务客户端 3355 / 3366 / 3377
- 每个微服务都要执行一次 post 请求，手动刷新？
- 可否广播，一次通知，处处生效？
- ....

目前来说，暂时做不到这个，所以才用了下面的内容，即 Spring Cloud Bus 消息总线

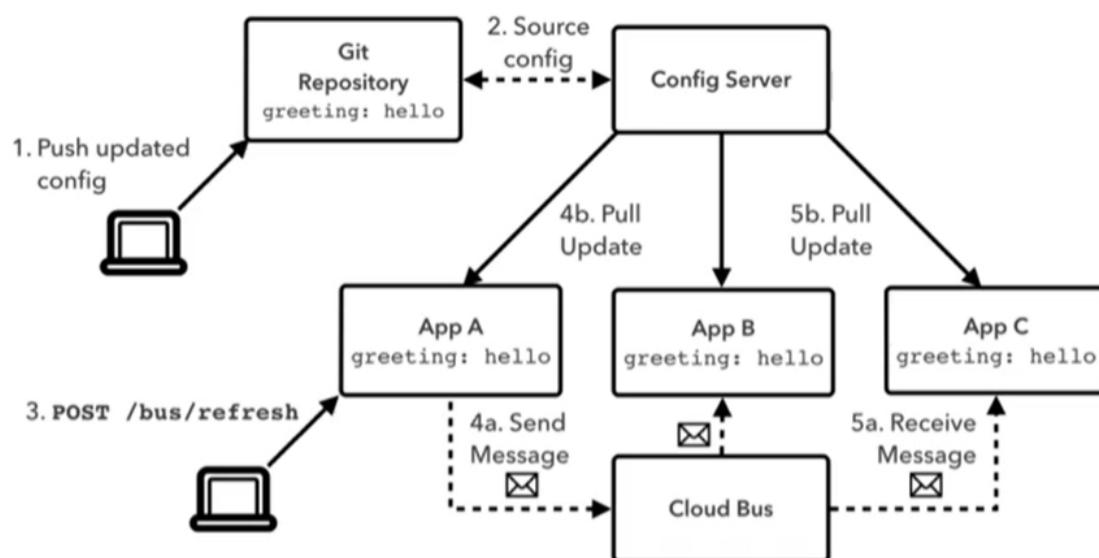
## 消息总线 SpringCloudBus

消息总线一般是配合 SpringCloudConfig 一起使用的

### 概述

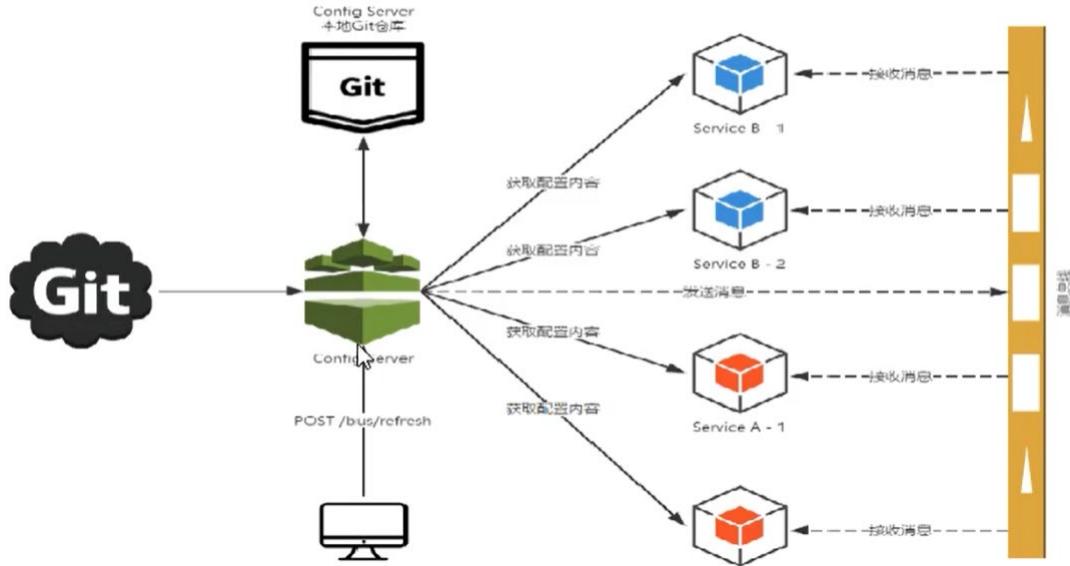
分布式自动刷新配置功能，SpringCloudBus 配合 SpringCloudConfig 使用可以实现配置的动态刷新

Bus 支持两种消息代理：RabbitMQ 和 Kafka



SpringCloudBus 是用来将分布式系统的节点与轻量级消息系统链接起来的框架，它整合了 Java 的事件处理机制和消息中间件的功能。

SpringCloudBus 能管理和传播分布式系统的消息，就像一个分布式执行器，可用于广播状态更改，事件推送等，也可以当做微服务的通信通道。



## 什么是总线

在微服务架构的系统中，通常会使用轻量级的消息代理来构建一个共用的消息主题，并让系统中所有微服务实例都连接上来。由于该主题中产生的消息会被所有实例监听和消费，所以被称为消息总线。在总线上的各个实例，都可以方便的广播一些需要让其它连接在该主题上的实例都知道的消息。

## 基本原理

ConfigClient 实例都监听 MQ 中同一个 topic（默认是 SpringCloudBus），但一个服务刷新数据的时候，它会被这个消息放到 Topic 中，这样其它监听同一个 Topic 的服务就能够得到通知，然后去更新自身的配置

The screenshot shows the RabbitMQ Management UI with the 'Exchanges' tab selected. The page title is 'Exchanges' and there is a dropdown menu 'All exchanges (9)'. Below the title, there is a 'Pagination' section with a page number '1' and a 'Filter' input field. A search bar with 'Regex ?' is also present. The main table lists nine exchanges:

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
input	topic	D			
<b>springCloudBus</b>	topic	D	0.00/s	0.00/s	

At the bottom left, there is a link 'Add a new exchange'.

通过 topic 进行广播通知

## RabbitMQ 环境配置

蘑菇博客配置 RabbitMQ

## SpringCloudBus 动态刷新全局广播

### 配置

首先需要搭建好 rabbitmq 环境

然后引入 RabbitMQ 依赖

```
<!--添加消息总线 Rabbitmq 支持-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

增加 yml 中的 rabbitmq 配置

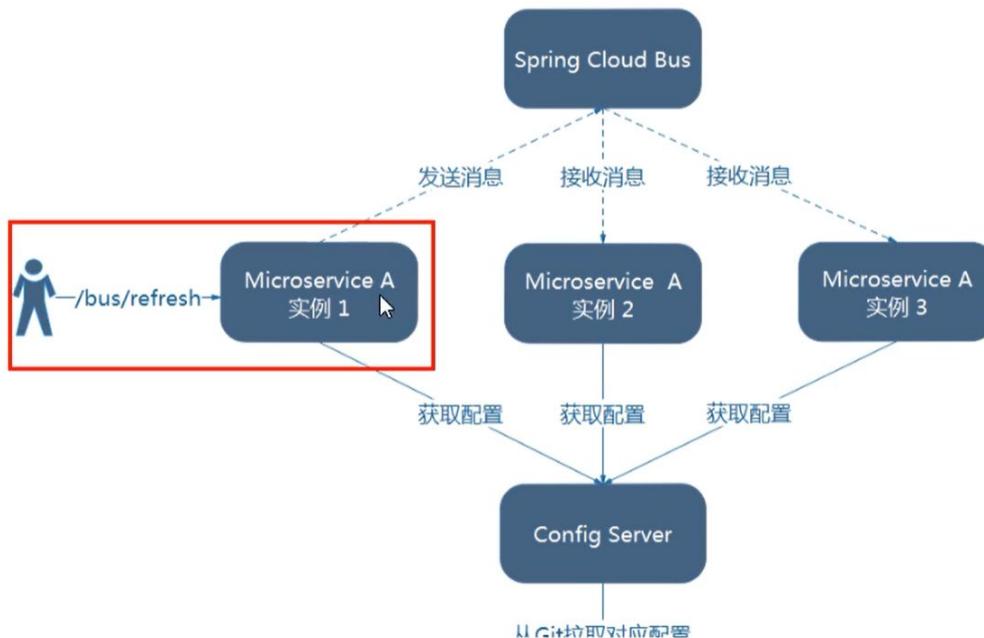
```

spring:
  application:
    name: config-client
  cloud:
    config:
      label: master #分支名称
      name: config #配置文件名称
      profile: dev #读取后缀文件名, 即 master 分支 config-dev.yml
      uri: http://localhost:3344 #配置中心地址
  rabbitmq: #mq 相关配置
    host: localhost
    port: 5672
    username: guest
    password: guest

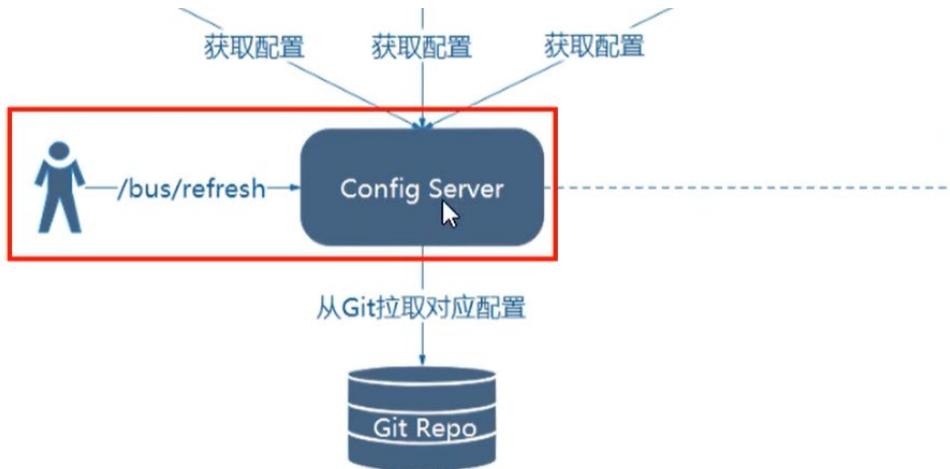
```

## 设计思想

- 利用消息总线触发一个客户端/bus/refresh，而刷新所有客户端配置



- 利用消息总线出发一个服务端 ConfigServer 的/bus/refresh 断点，而刷新所有客户端的配置



- 图二的架构更加适合，图一不适合的原因有

- 图一打破了微服务的职责单一性，因为微服务本身是业务模块，他不应该承担配置刷新之职责
- 打破了微服务各节点的对等性
- 有一定的局限性，例如，微服务在迁移时，它的网络地址常常发生变化，此时如果想要做到自动刷新，那就会增加更多的修改。

## 服务端添加消息总线

### 引入依赖

```
<!--添加消息总线 Rabbitmq 支持-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

### 增加配置

```
spring:
  application:
    name: cloud-config-center
  cloud:
    config:
      server:
        git:
          uri: https://github.com/boytian/springcloud-config.git
          search-paths:
            - springcloud-config
  label: master
```

```

rabbitmq: #mq 相关配置
  host: localhost
  port: 5672
  username: guest
  password: guest

# rabbitmq 相关配置，暴露 bus 刷新点
management:
  endpoints: #暴露 bus 刷新配置的端点
    web:
      exposure:
        include: 'bus-refresh'

```

注意，上面的 `bus-refresh` 就是 actuator 的刷新机制，相当于提供了一个 `/bus-refresh` 的 post 方法，当我们调用的时候，会刷新配置，然后一次发送，处处生效。

## 客户端引入消息总线支持

### 引入依赖

```

<!--添加消息总线 Rabbitmq 支持-->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>

```

### 修改 yml

```

spring:
  application:
    name: config-client
  cloud:
    config:
      label: master #分支名称
      name: config #配置文件名称
      profile: dev #读取后缀文件名，即 master 分支 config-dev.yml
      uri: http://localhost:3344 #配置中心地址
  rabbitmq: #mq 相关配置
    host: localhost
    port: 5672
    username: guest
    password: guest

#暴露监控端点
management:
  endpoints:
    web:
      exposure:
        include: "*"

```

这里的暴露端点和上面的不太一样

## 测试

当我们的服务端配置中心 和 客户端都增加完上述配置后，我们需要做的就是手动发送一个 POST 请求到服务端

```
curl -X POST "http://localhost:33444/actuator/bus-refresh"
```

执行完成后，配置中心会通过 BUS 消息总线，发送到所有的客户端，并完成配置的刷新操作。

完成了一次修改，广播通知，处处生效的效果

## SpringCloudBus 动态刷新定点通知

就是我想通知的目标是有差异化，有些客户端需要通知，有些不通知，也就是 10 个客户端，我只通知 1 个

简单一句话，就是指定某一个实例生效而不是全部

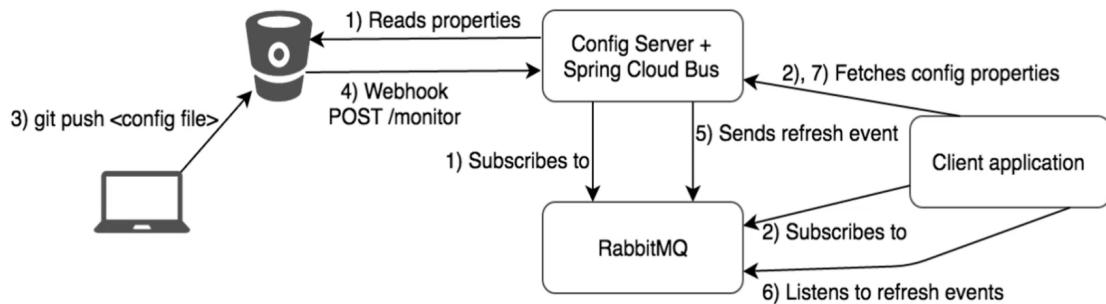
公式：`http://localhost:配置中心端口/actuator/bus-refresh/{destination}`

`/bus/refresh` 请求不再发送到具体的服务实例上，而是发送给 config server 并通过 `destination` 参数类指定需要更新配置的服务或实例。

## 案例

以刷新运行在 3355 端口上的 config-client 为例，只通知 3355，不通知 3366，可以使用下面命令

```
curl -X POST "http://localhost:33444/actuator/bus-refresh/config-client:3355"
```



## SpringCloud Stream 消息驱动

### 为什么引入消息驱动？

首先看到消息驱动，我们会想到，消息中间件

- ActiveMQ
- RabbitMQ
- RocketMQ
- Kafka



存在的问题就是，中台和后台 可能存在两种 MQ，那么他们之间的实现都是不一样的，这样会导致多种问题出现，而且上述我们也看到了，目前主流的 MQ 有四种，我们不可能每个都去学习

这个时候的痛点就是：有没有一种新的技术诞生，让我们不在关注具体 MQ 的细节，我们只需要用一种适配绑定的方式，自动的给我们在各种 MQ 内切换。

这个时候，SpringCloudStream 就运营而生，解决的痛点就是屏蔽了消息中间件的底层的细节差异，我们操作 Stream 就可以操作各种消息中间件了，从而降低开发人员的开发成本。

## 消息驱动概述

### 是什么

屏蔽底层消息中间件的差异，降低切换成本，统一消息的编程模型

这就有点像 Hibernate，它同时支持多种数据库，同时还提供了 Hibernate Session 的语法，也就是 HQL 语句，这样屏蔽了 SQL 具体实现细节，我们只需要操作 HQL 语句，就能够操作不同的数据库。

### 什么是 SpringCloudStream

官方定义 SpringCloudStream 是一个构件消息驱动微服务的框架

应用程序通过 inputs 或者 outputs 来与 SpringCloudStream 中 binder 对象（绑定器）交互。

通过我们配置来 binding(绑定)，而 SpringCloudStream 的 binder 对象负责与消息中间件交互

所以，我们只需要搞清楚如何与 SpringCloudStream 交互，就可以方便的使用消息驱动的方式。

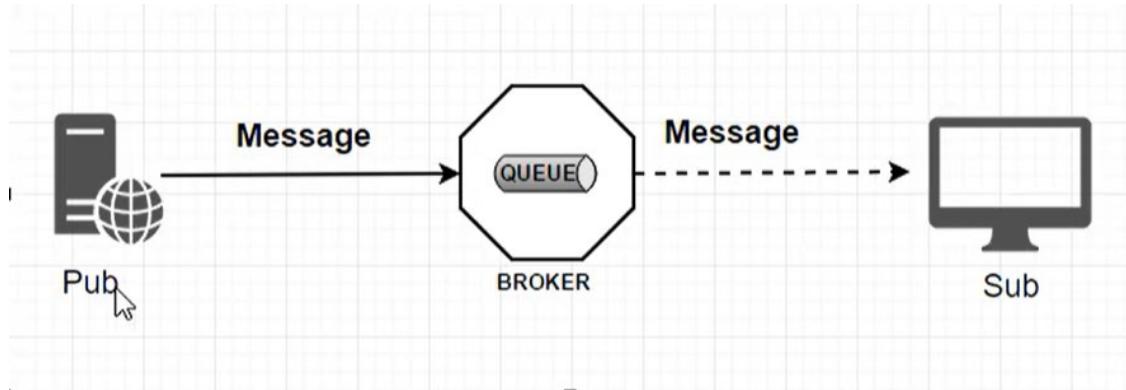
通过使用 SpringIntegration 来连接消息代理中间件以实现消息事件驱动。

SpringCloudStream 为一些供应商的消息中间件产品提供了个性化的自动化配置实现，引用了发布-订阅，消费组，分区的三个核心概念

目前仅支持 RabbitMQ 和 Kafka

## SpringCloudStream 设计思想

### 标准 MQ

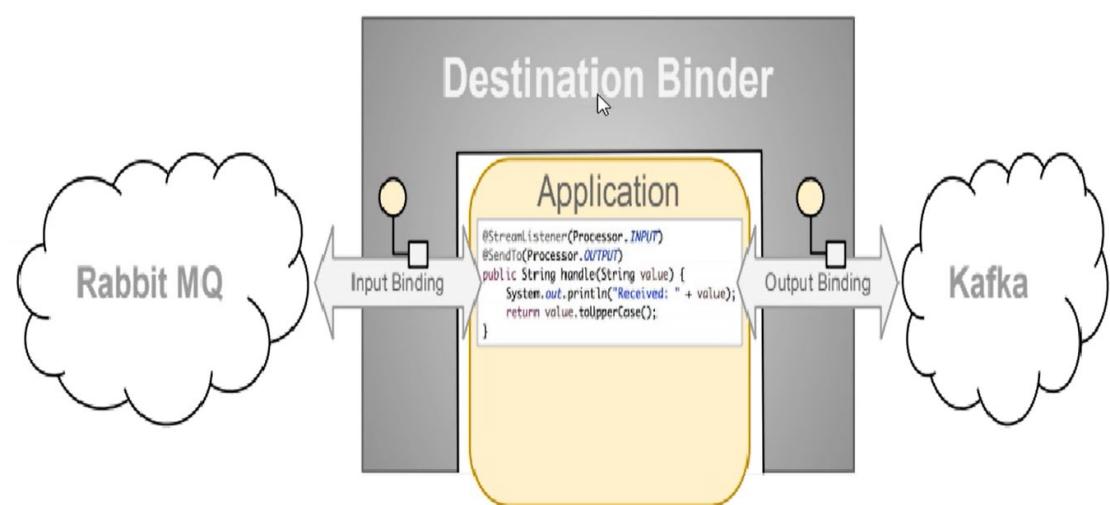


- 生产者/消费者之间靠消息媒介传递消息内容：Message
- 消息必须走特定的通道：Channel
- 消息通道里的消息如何被消费呢，谁负责收发处理
  - 消息通道 MessageChannel 的子接口 SubscribableChannel，由 MessageHandler 消息处理器所订阅

### 为什么用 SpringCloudStream

RabbitMQ 和 Kafka，由于这两个消息中间件的架构上不同

像 RabbitMQ 有 exchange，kafka 有 Topic 和 Partitions 分区



这些中间件的差异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，我们想往另外一种消息队列进行迁移，这时候无疑就是灾难性的，一大堆东西都要推到重新做，因为它跟我们的系统耦合了，这时候 SpringCloudStream 给我们提供了一种解耦的方式

这个时候，我们就需要一个绑定器，可以想成是翻译官，用于实现两种消息之间的转换

### SpringCloudStream 为什么能屏蔽底层差异

在没有绑定器这个概念的情况下，我们的 SpringBoot 应用要直接与消息中间件进行消息交互的时候，由于各消息中间件构建的初衷不同，它们的实现细节上会有较大的差异性。通过定义绑定器作为中间件，完美的实现了应用程序与消息中间件细节之间的隔离。

通过向应用程序暴露统一的 Channel 通道，使得应用程序不需要在考虑各种不同消息中间件的实现。

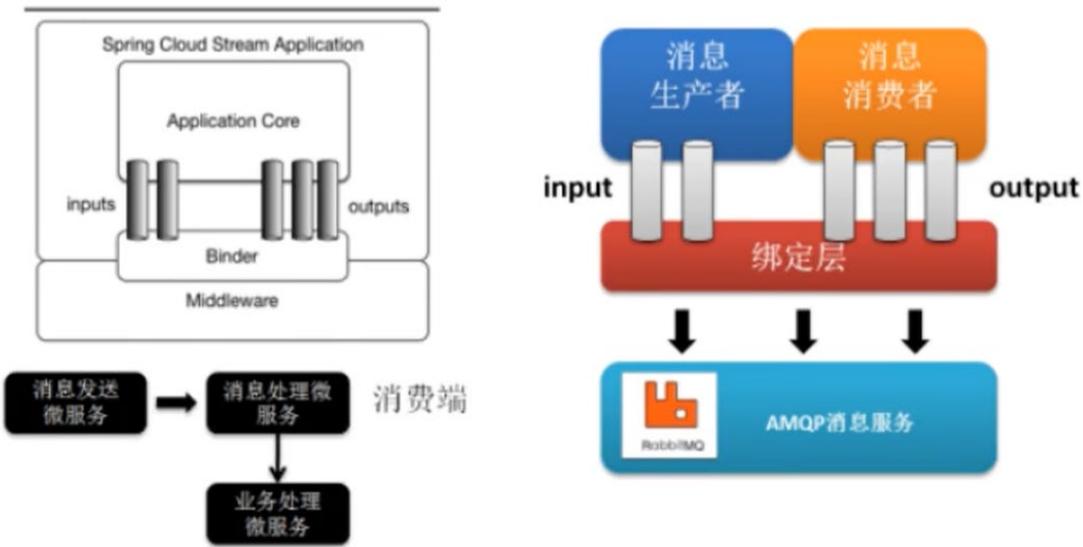
通过定义绑定器 Binder 作为中间层，实现了应用程序与消息中间件细节之间的隔离。

#### Binder

- `input`: 对应消费者
- `output`: 对应生产者

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（RabbitMQ 切换 Kafka），使得微服务开发的高度解耦，服务可以关注更多的自己的业务流程。

# SpringCloudStream处理架构

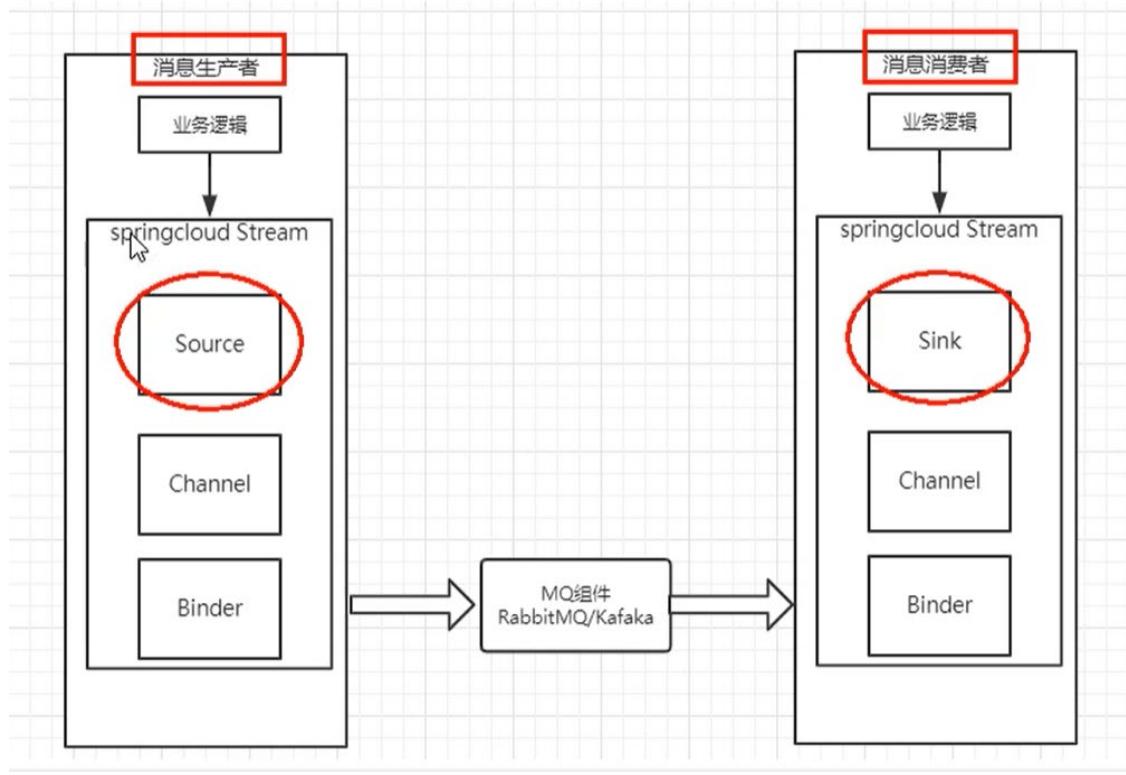


通过定义绑定器 Binder 作为中间层，实现了应用程序与消息中间件细节之间的隔离。

Stream 中的消息通信方式遵循了发布-订阅模式，Topic 主题进行广播，在 RabbitMQ 中就是 Exchange，在 Kafka 中就是 Topic

## Stream 标准流程套路

我们的消息生产者和消费者只和 Stream 交互



- **Binder:** 很方便的连接中间件，屏蔽差异
- **Channel:** 通道，是队列 Queue 的一种抽象，在消息通讯系统中就是实现存储和转发的没接，通过 Channel 对队列进行配置
- **Source 和 Sink:** 简单的可以理解为参照对象是 SpringCloudStream 自身，从 Stream 发布消息就是输出，接受消息就是输入。

### 编码中的注解

Spring Cloud Stream Application	
组成	说明
Middleware	中间件，目前只支持RabbitMQ和Kafka
Binder	Binder是应用与消息中间件之间的封装，目前实行了Kafka和RabbitMQ的Binder，通过Binder可以很方便的连接中间件，可以动态的改变消息类型(对应于Kafka的topic, RabbitMQ的exchange)，这些都可以通过配置文件来实现
@Input	注解标识输入通道，通过该输入通道接收到的消息进入应用程序
@Output	注解标识输出通道，发布的消息将通过该通道离开应用程序
@StreamListener	监听队列，用于消费者的队列的消息接收
@EnableBinding	指信道channel和exchange绑定在一起

## 案例说明

前提是已经安装好了 RabbitMQ

- cloud-stream-rabbitmq-provider8801，作为消息生产者进行发消息模块
- cloud-stream-rabbitmq-provider8802，消息接收模块
- cloud-stream-rabbitmq-provider8803，消息接收模块

## 消息驱动之生产者

### 引入依赖

```
<!--Stream-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

### 修改 yml

```
spring:
  application:
    name: cloud-stream-provider
  cloud:
    stream:
      binders: # 在此处配置要绑定的 rabbitMQ 的服务信息
        defaultRabbit: # 表示定义的名称，用于 binding 的整合
          type: rabbit # 消息中间件类型
          environment: # 设置 rabbitMQ 的相关环境配置
            spring:
              rabbitmq:
                host: localhost
                port: 5672
                username: guest
                password: guest
      bindings: # 服务的整合处理
        output: # 这个名字是一个通道的名称
          destination: studyExchange # 表示要使用的 exchange 名称定义
          content-type: application/json # 设置消息类型，本次为 json，文本
则设为 text/plain
        binder: defaultRabbit # 设置要绑定的消息服务的具体设置
```

## 业务类

### 发送消息的接口

```
public interface IMassageProvider {  
    // 定义一个发送方法  
    public String send();  
}
```

### 发送消息的接口实现类

```
// @EnableBinding: 指信道 channel 和 exchange 绑定在一起  
@EnableBinding(Source.class) //定义消息的推送管道  
public class MessageProviderImpl implements IMassageProvider {  
  
    @Resource  
    private MessageChannel output; //消息发送管道（原来这里是操作 dao，现在  
    是操作消息中间件发送消息）  
  
    @Override  
    public String send() {  
        String serial= UUID.randomUUID().toString();  
        // 消息构建器构建一个消息 MessageBuilder  
        output.send(MessageBuilder.withPayload(serial).build());  
        System.out.println("*****serial"+serial);  
        return null;  
    }  
}
```

### Controller

```
@RestController  
public class SendMessageController {  
    @Resource  
    private IMassageProvider massageProvider;  
  
    @GetMapping("/sendMessage")  
    public String sendMessage(){  
        return massageProvider.send();  
    }  
}
```

定义一个 REST 接口，调用的时候，发送一个消息

### 测试

我们进入 RabbitAdmin 页面 <http://localhost:15672>

The screenshot shows the RabbitMQ Management UI with the 'Exchanges' tab selected. The page displays a table of exchanges, with one entry, 'studyExchange', highlighted by an orange border. The table columns are: Name, Type, Features, Message rate in, and Message rate out. The 'studyExchange' row has a '+'/- button in the last column.

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
exchange.direct	direct	D			
jolokiaEndpoint-out-0	topic	D			
mogu.direct	direct	D			
mogu.fanout	fanout	D			
mogu.topic	topic	D			
springCloudBus	topic	D			
studyExchange	topic	D			

Page 1 of 1 - Filter:   Regex ?

Add a new exchange

会发现它已经成功创建了一个 studyExchange 的交换机，这个就是我们上面配置的

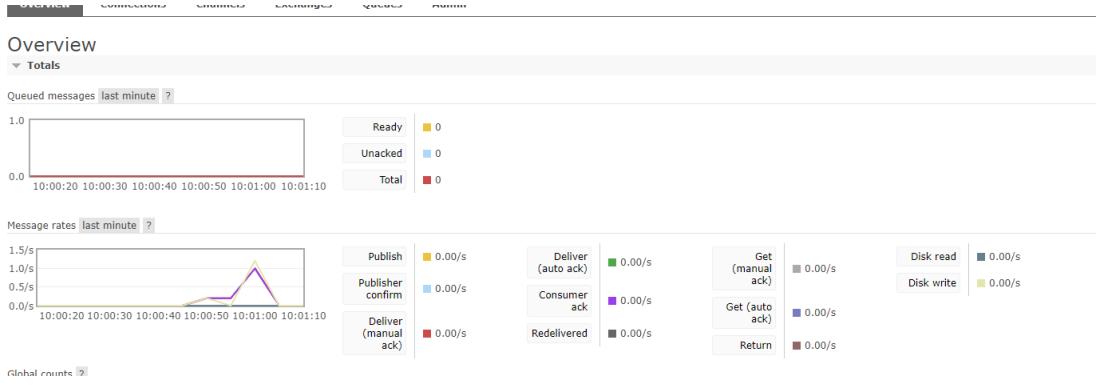
```

bindings: # 服务的整合处理
output: # 这个名字是一个通道的名称
destination: studyExchange # 表示要使用的 exchange 名称定义
content-type: application/json # 设置消息类型，本次为 json，文本
则设为 text/plain
binder: defaultRabbit # 设置要绑定的消息服务的具体设置
    
```

以后就会通过这个交换机进行消息的消费

我们运行下列代码，进行测试消息发送 <http://localhost:8801/sendMessage>

能够发现消息已经成功被 RabbitMQ 捕获，这个时候就完成了消息的发送



## 消息驱动之消费者

### 引入依赖

```
<!--Stream-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

### 修改 yml

```
spring:
  application:
    name: cloud-stream-consumer
  cloud:
    stream:
      binders: # 在此处配置要绑定的 rabbitMQ 的服务信息
        defaultRabbit: # 表示定义的名称, 用于 binding 的整合
          type: rabbit # 消息中间件类型
        environment: # 设置 rabbitMQ 的相关环境配置
          spring:
            rabbitmq:
              host: localhost
              port: 5672
              username: guest
              password: guest
      bindings: # 服务的整合处理
        input: # 这个名字是一个通道的名称
          destination: studyExchange # 表示要使用的 exchange 名称定义
          content-type: application/json # 设置消息类型, 本次为 json, 文本则设为 text/plain
          binder: defaultRabbit # 设置要绑定的消息服务的具体设置
          group: atguiguA
```

## 业务类

```
@Component
@EnableBinding(Sink.class) // 绑定通道
public class ReceiveMessageListenerController {

    @Value("${server.port}")
    private String serverPort;

    // 监听队列，用于消费者队列的消息接收
    @StreamListener(Sink.INPUT)
    public void input(Message<String> message) {
        System.out.println("消费者 1 号, 0----->接收到消息: "+message.getPayload()+"\t port:"+serverPort);
    }
}
```

## 分组消费

我们在创建一个 8803 的消费者服务，需要启动的服务

- RabbitMQ: 消息中间件
- 7001: 服务注册
- 8801: 消息生产
- 8802: 消息消费
- 8803: 消息消费

## 运行后有两个问题

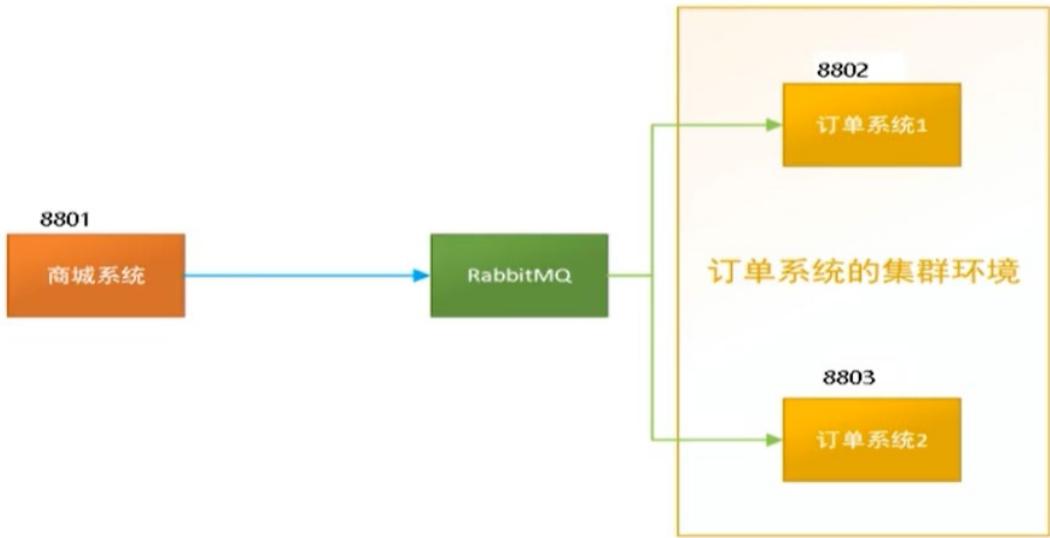
- 有重复消费问题
- 消息持久化问题

## 消费

目前 8802、8803 同时都收到了，存在重复消费的问题

如何解决：使用分组和持久化属性 group 来解决

比如在如下场景中，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，这时我们就可以使用 Stream 中的消息分组来解决。



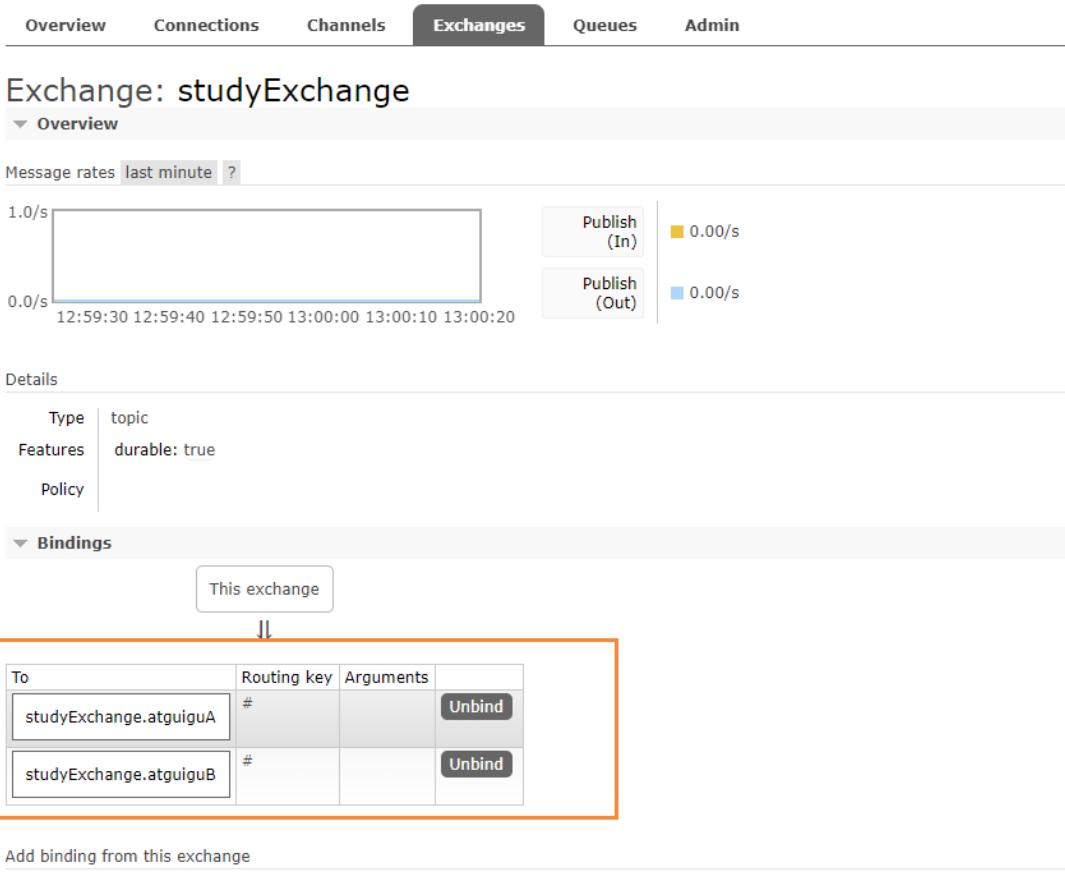
注意：在 Stream 中处于同一个 group 中的多个消费者是竞争关系，就能够保证消息只能被其中一个消费一次

不同组是可以全面消费的（重复消费）

同一组会发生竞争关系，只能其中一个可以消费

分布式微服务应用为了实现高可用和负载均衡，实际上都会部署多个实例，这里部署了 8802 8803

多数情况下，生产者发送消息给某个具体微服务时，只希望被消费一次，按照上面我们启动两个应用的例子，虽然它们同属一个应用，但是这个消息出现了被重复消费两次的情况，为了解决这个情况，在 SpringCloudStream 中，就提供了消费组的概念



## 分组

### 原理

微服务应用放置于同一个 group 中，就能够保证消息只会被其中一个应用消费一次，不同的组是可以消费的，同一组内会发生竞争关系，只有其中一个可以被消费。

我们将 8802 和 8803 划分为同一组

```
spring:
  application:
    name: cloud-stream-consumer
  cloud:
    stream:
      binders: # 在此处配置要绑定的rabbitMQ 的服务信息
        defaultRabbit: # 表示定义的名称, 用于binding 的整合
          type: rabbit # 消息中间件类型
          environment: # 设置rabbitMQ 的相关环境配置
            spring:
              rabbitmq:
```

```

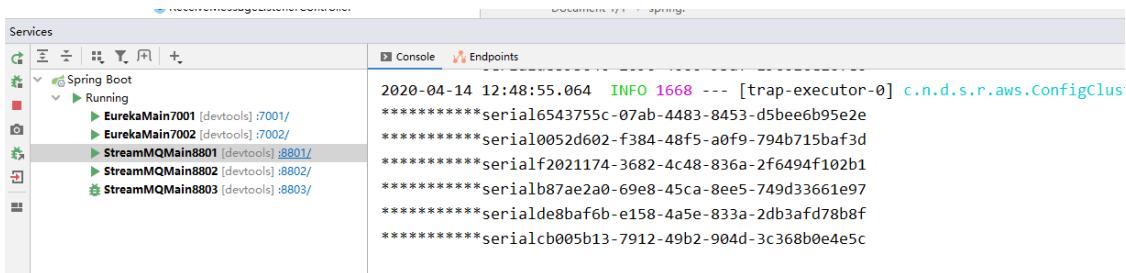
host: localhost
port: 5672
username: guest
password: guest
bindings: # 服务的整合处理
    input: # 这个名字是一个通道的名称
        destination: studyExchange # 表示要使用的exchange 名称定义
        content-type: application/json # 设置消息类型, 本次为json, 文本则设为text/plain
        binder: defaultRabbit # 设置要绑定的消息服务的具体设置
        group: atguiguA

```

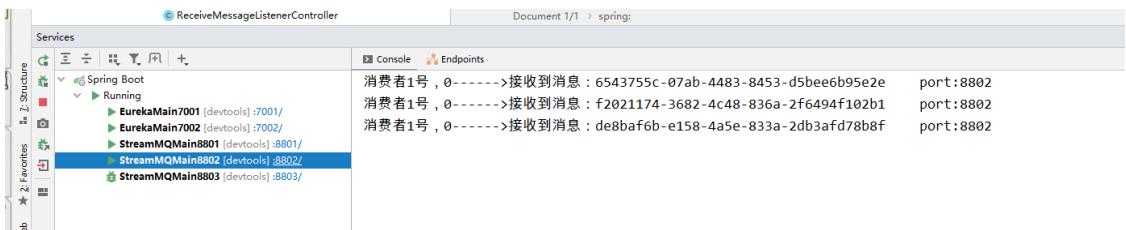
引入: group: atguiguA

然后我们执行消息发送的接口: <http://localhost:8801/sendMessage>

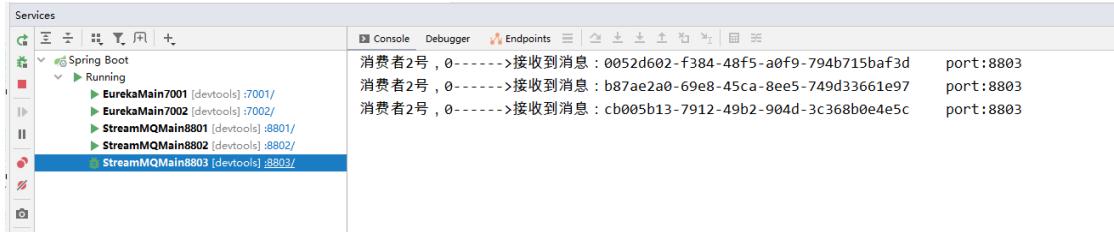
我们在 8801 服务, 同时发送了 6 条消息



然后看 8802 服务, 接收到了 3 条



8803 服务, 也接收到了 3 条



这个时候，就通过分组，避免了消息的重复消费问题

8802、8803 通过实现轮询分组，每次只有一个消费者，最后发送的消息只能够被一个接受

如果将他们的 group 变成两个不同的组，那么消息就会被重复消费

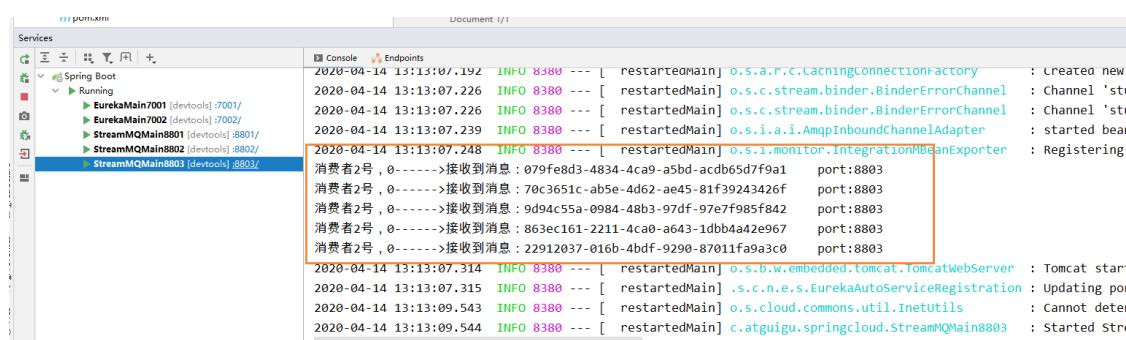
## 消息持久化

通过上面的方式，我们解决了重复消费的问题，再看看持久化

### 案例

- 停止 8802 和 8803，并移除 8802 的 group，保留 8803 的 group
- 8801 先发送 4 条消息到 RabbitMQ
- 先启动 8802，无分组属性，后台没有打出来消息
- 在启动 8803，有分组属性，后台打出来 MQ 上的消息

这就说明消息已经被持久化了，等消费者登录后，会自动从消息队列中获取消息进行消费



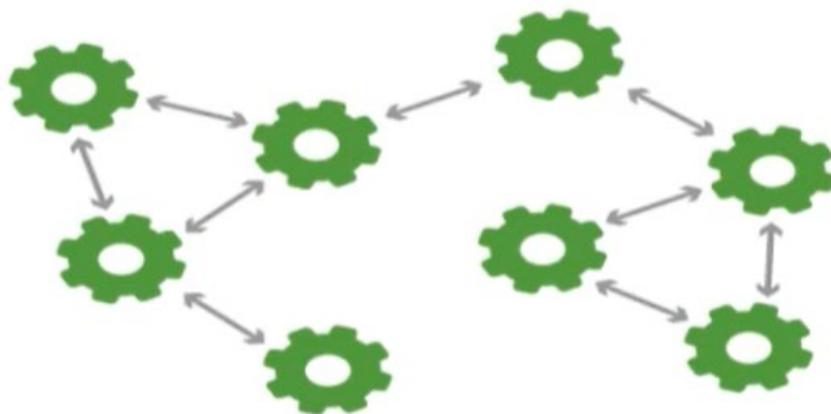
## SpringCloudSleuth 分布式请求链路跟踪

### 概述

详细可以参考：使用 Zipkin 搭建蘑菇博客链路追踪

在微服务框架中，一个由客户端发起的请求在后端系统中会经过多个不同的服务节点调用来协同产生最后的请求结果，每一个前端请求都会形成一条复杂的分布式服务调用链路，链路中的任何一环出现高延时或错误都会引起整个请求最后的失败。

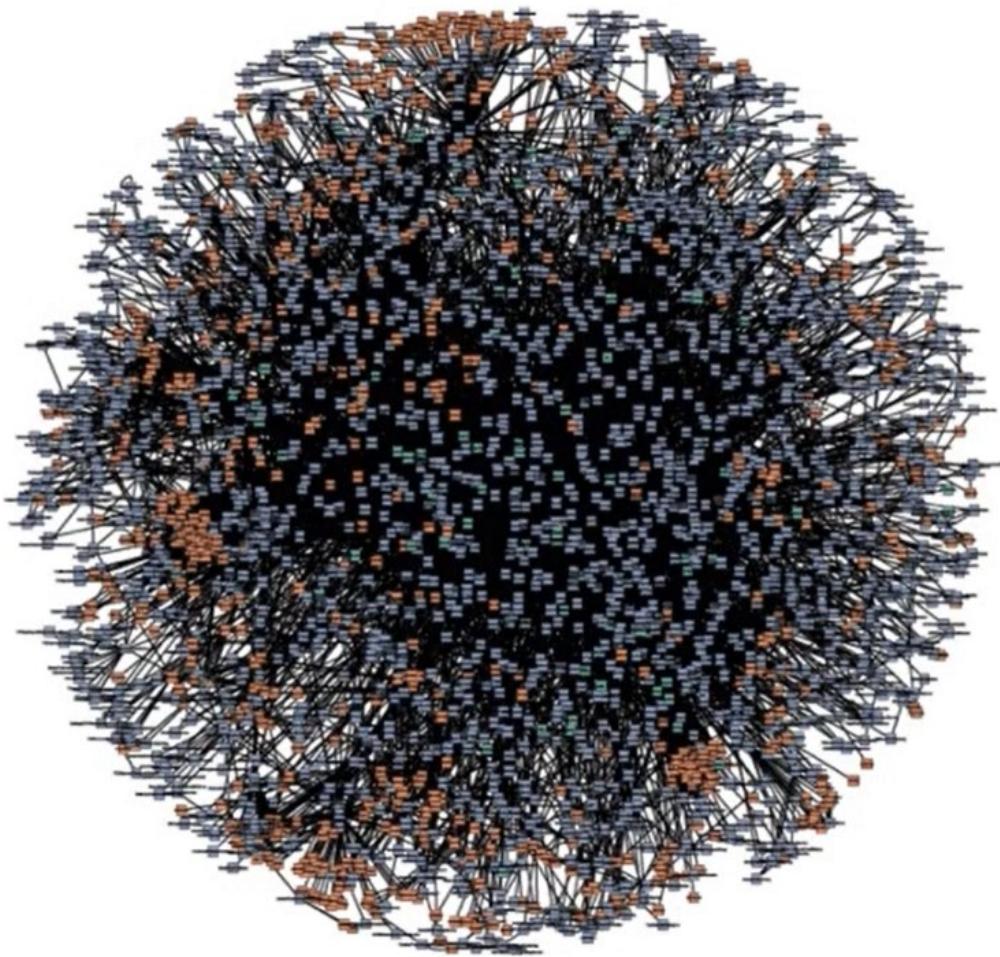
## MICROSERVICES ARCHITECTURE



---

**Microservices** are a number of independent application services delivering one single functionality in a loosely connected and self-contained fashion, communicating through light-weight messaging protocols such as HTTP, REST or Thrift API.

当链路特别多的时候



就需要有一个用于调用链路的监控和服务跟踪的解决方案

SpringCloudSleuth 提供了一套完整的服务跟踪解决方案，在分布式系统中，提供了追踪解决方案，并且兼容支持了 zipkin。

## 搭建

### zipkin

SpringCloud 从 F 版起，已经不需要自己构建 Zipkin Server 了，只需要调用 jar 包即可

### 运行

```
java -jar zipkin.jar
```

### 打开

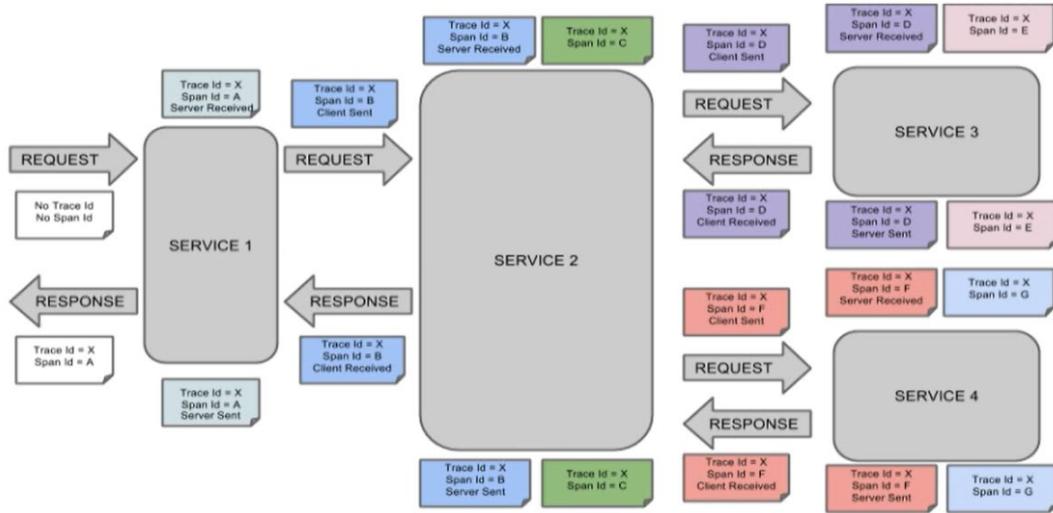
<http://localhost:9441/zipkin>

## 名词解释

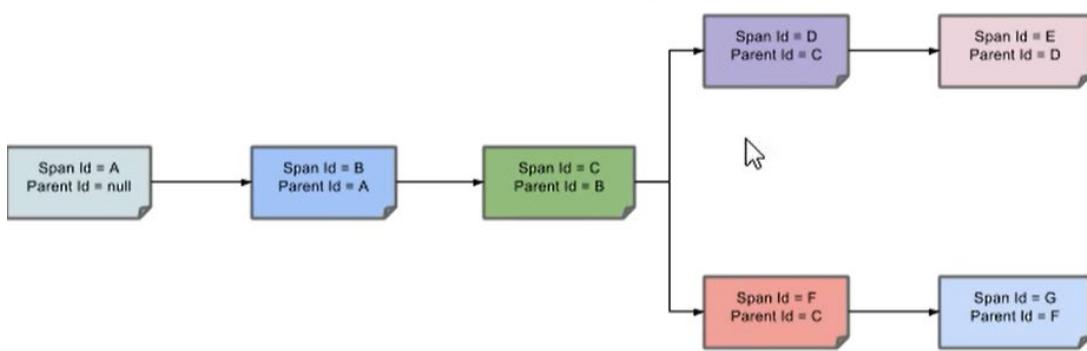
- Trace: 类似于树结构的 Span 集合, 表示一条调用链路, 存在唯一标识
- Span: 表示调用链路来源, 通俗的理解 span 就是一次请求信息

## 完整的调用链路

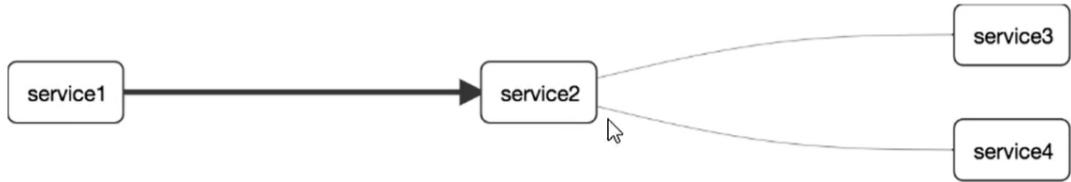
表示一请求链路, 一条链路通过 Trace ID 唯一标识, Span 标识发起请求信息, 各 span 通过 parent id 关联起来。



一条链路通过 Trace Id 唯一标识, Span 表示发起的请求信息, 各 span 通过 parent id 关联起来



整个链路的依赖关系如下:



## 引入依赖

```

<!--链路监控包含 sleuth+zipkin-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>

```

## 修改 yml

```

spring:
  application:
    name: cloud-order-service
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      # 采集率介于 0 到 1 之间, 1 表示全部采集
      probability: 1

```

## Nacos

### SpringCloud Alibaba 简介

SpringCloud Alibaba 诞生的主要原因是：因为 Spring Cloud Netflix 项目进入了维护模式

## 维护模式

将模块置为维护模式，意味着 SpringCloud 团队将不再向模块添加新功能，我们将恢复 block 级别的 bug 以及安全问题，我们也会考虑并审查社区的小型 pull request

我们打算继续支持这些模块，知道 Greenwich 版本被普遍采用至少一年

## 意味着

Spring Cloud Netflix 将不再开发新的组件，我们都知道 Spring Cloud 项目迭代算是比较快，因此出现了很多重大 issue 都还来不及 Fix，就又推出了另一个 Release。

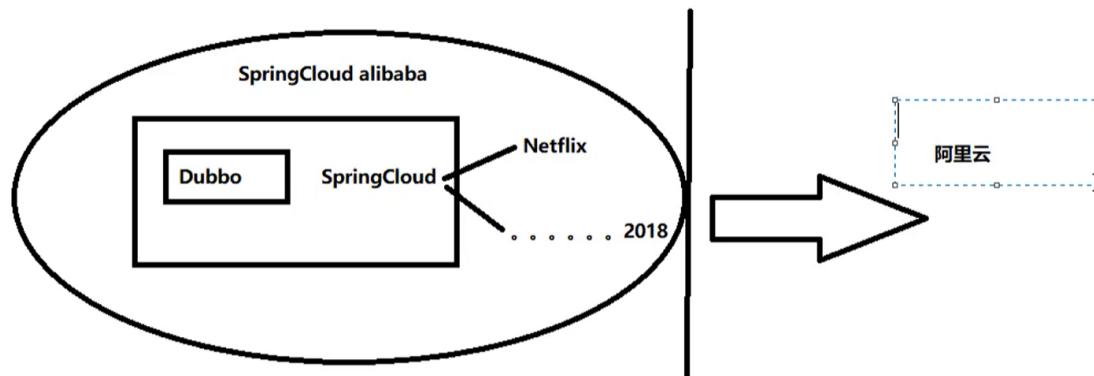
进入维护模式意思就是以后一段时间 Spring Cloud Netflix 提供的服务和功能就这么多了，不在开发新的组件和功能了，以后将以维护和 Merge 分支 Pull Request 为主，新组件将以其他替代

**Replacements**

We recommend the following as replacements for the functionality provided by these modules.

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

Look for a future blog post on Spring Cloud Loadbalancer and integration with a new Netflix project Concurrency Limits.



## 诞生

官网：[SpringCloud Alibaba](#)

2018.10.31，Spring Cloud Alibaba 正式入驻 Spring Cloud 官方孵化器，并在 Maven 仓库发布了第一个

## Spring Cloud for Alibaba 0.2.0 released



The Spring Cloud Alibaba project, consisting of Alibaba's open-source components and several Alibaba Cloud products, aims to implement and expose well known Spring Framework patterns and abstractions to bring the benefits of Spring Boot and Spring Cloud to Java developers using Alibaba products.

Spring Cloud for Alibaba，它是由一些阿里巴巴的开源组件和云产品组成的。这个项目的目的是为了让大家所熟知的 Spring 框架，其优秀的设计模式和抽象理念，以给使用阿里巴巴产品的 Java 开发者带来使用 Spring Boot 和 Spring Cloud 的更多便利。

## 能做啥

- 服务限流降级：默认支持 servlet, Feign, RestTemplate, Dubbo 和 RocketMQ 限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 Metrics 监控
- 服务注册与发现：适配 Spring Cloud 服务注册与发现标准，默认集成了 Ribbon 的支持
- 分布式配置管理：支持分布式系统中的外部化配置，配置更改时自动刷新
- 消息驱动能力：基于 Spring Cloud Stream（内部用 RocketMQ）为微服务应用构建消息驱动能力
- 阿里云对象存储：阿里云提供的海量、安全、低成本、高可靠的云存储服务，支持在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- 分布式任务调度：提供秒级，精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务，同时提供分布式的任务执行模型，如网格任务，网格任务支持海量子任务均匀分配到所有 Worker

## 引入依赖版本控制

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>2.2.0.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

## 怎么玩

- **Sentinel:** 阿里巴巴开源产品，把流量作为切入点，从流量控制，熔断降级，系统负载保护等多个维度保护系统服务的稳定性
- **Nacos:** 阿里巴巴开源产品，一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台
- **RocketMQ:** 基于 Java 的高性能，高吞吐量的分布式消息和流计算平台
- **Dubbo:** Apache Dubbo 是一款高性能 Java RPC 框架
- **Seata:** 一个易于使用的高性能微服务分布式事务解决方案
- **Alibaba Cloud OOS:** 阿里云对象存储（Object Storage Service，简称 OOS），是阿里云提供的海量，安全，低成本，高可靠的云存储服务，您可以在任何应用，任何时间，任何地点存储和访问任意类型的数据。
- **Alibaba Cloud SchedulerX:** 阿里中间件团队开发的一款分布式任务调度产品，支持周期的任务与固定时间点触发

## Nacos 简介

Nacos 服务注册和配置中心，兼顾两种

## 为什么叫 Nacos

前四个字母分别为：Naming（服务注册） 和 Configuration（配置中心） 的前两个字母，后面的 s 是 Service

## 是什么

一个更易于构建云原生应用的动态服务发现，配置管理和服务

Nacos: Dynamic Naming and Configuration Server

Nacos 就是注册中心 + 配置中心的组合

等价于：Nacos = Eureka + Config

## 能干嘛

替代 Eureka 做服务注册中心

替代 Config 做服务配置中心

## 下载

官网: <https://github.com/alibaba/nacos>

nacos 文档: <https://nacos.io/zh-cn/docs/what-is-nacos.html>

## 比较

服务注册与发现框架	CAP 模型	控制台管理	社区活跃度
Eureka	AP	支持	低 (2.x 版本闭源)
Zookeeper	CP	不支持	中
Consul	CP	支持	高
Nacos	AP	支持	高

Nacos 在阿里巴巴内部有超过 10 万的实例运行，已经过了类似双十一等各种大型流量的考验

## 安装并运行

本地需要 java8 + Maven 环境

下载: [地址](#)

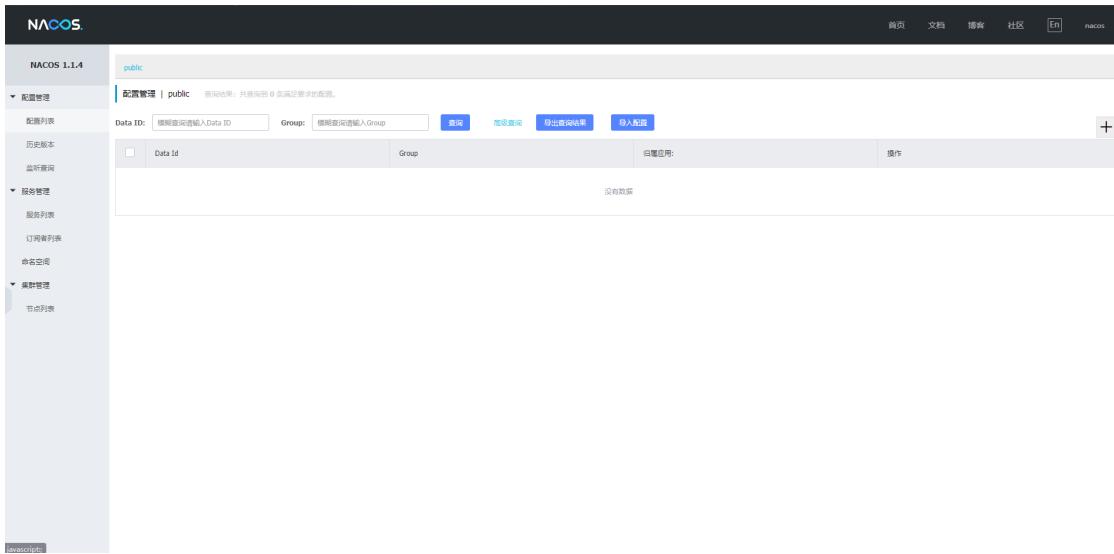
github 经常抽风，可以使用:

<https://blog.csdn.net/buyaopa/article/details/104582141>

解压后：运行 bin 目录下的：startup.cmd

打开: <http://localhost:8848/nacos>

结果页面



## Nacos 作为服务注册中心

### 服务提供者注册 Nacos

#### 引入依赖

```
<!--SpringCloud alibaba nacos-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

#### 修改 yml

```
server:
  port: 9002
spring:
  application:
    name: nacos-payment-provider
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848    # 配置 nacos 地址
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

## 主启动类

添加 @EnableDiscoveryClient 注解

```
@SpringBootApplication
@EnableDiscoveryClient
public class PaymentMain9002 {
    public static void main(String[] args) {
        SpringApplication.run(PaymentMain9002.class);
    }
}
```

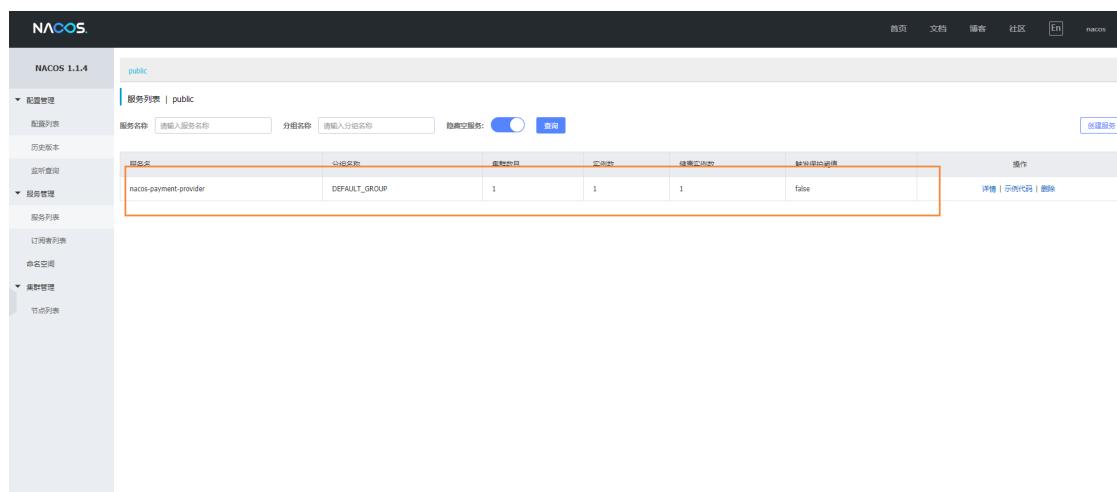
## 业务类

```
@RestController
public class PaymentController {
    @Value("${server.port}")
    private String serverPort;

    @GetMapping("/payment/nacos/{id}")
    public String getPayment(@PathVariable("id") Integer id) {
        return "nacos registry ,serverPore:"+serverPort+"\t id:"+id;
    }
}
```

## 启动

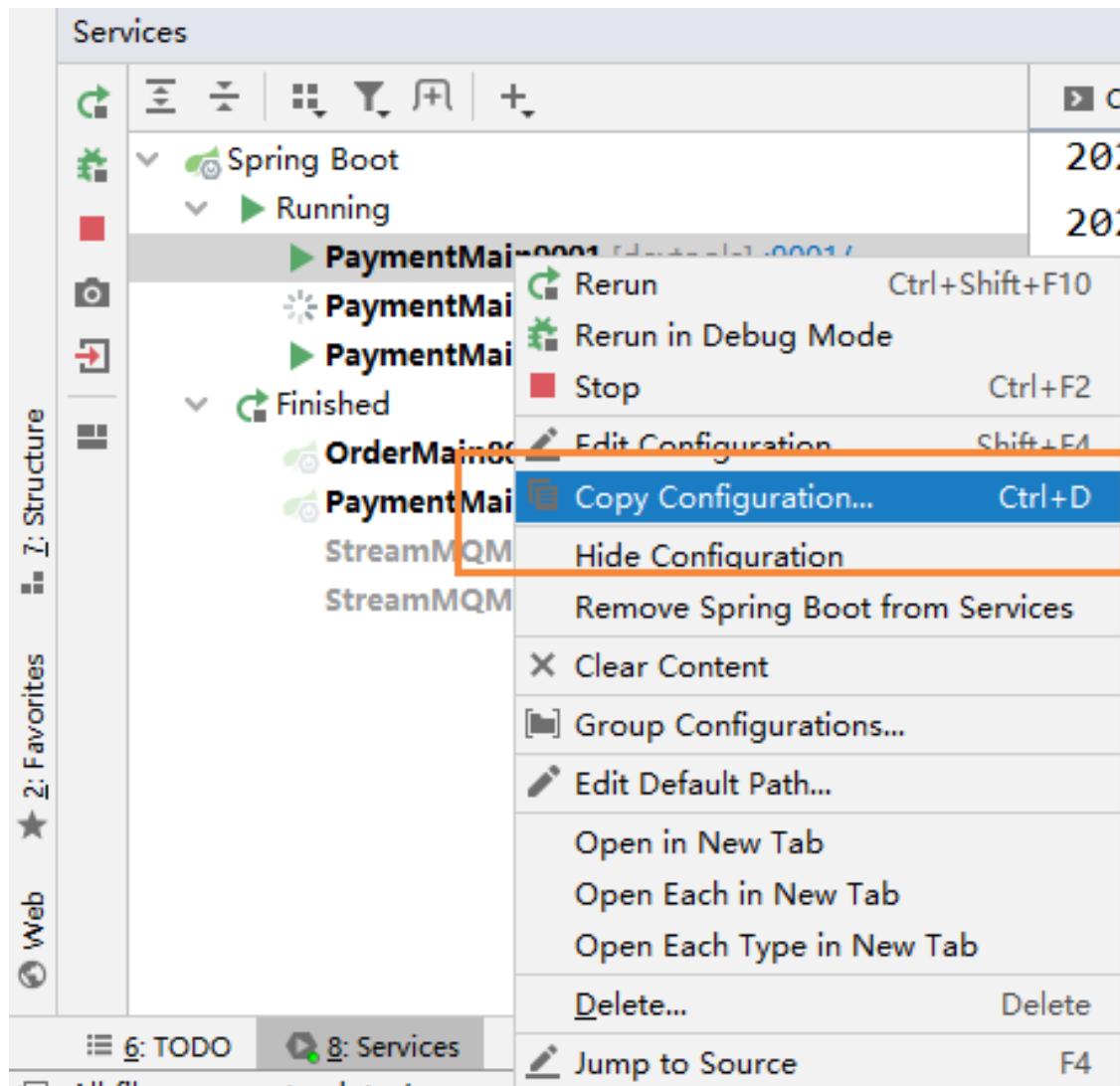
nacos-payment-provider 已经成功注册了



The screenshot shows the Nacos 1.1.4 service registration interface. On the left, there's a sidebar with navigation links: 配置管理 (Configuration Management), 监听查询 (Listener Query), 服务管理 (Service Management), 服务列表 (Service List), 订阅者列表 (Subscriber List), 命名空间 (Namespace), and 集群管理 (Cluster Management). The '服务管理' section is currently selected. In the main area, there's a search bar with '服务列表 | public' and a '模糊搜索' button. Below it is a table with columns: 服务名 (Service Name), 公组类别 (Public Group Category), 服务状态 (Service Status), 实例数 (Number of Instances), 健康实例数 (Number of Healthy Instances), 和 健康状况 (Health Status). A single row is highlighted with an orange border, representing the 'nacos-payment-provider' service. The table also has a '操作' (Operation) column with three buttons: 详情 (Details), 示例代码 (Example Code), and 图像 (Image). The top right of the interface includes links for 首页 (Home), 文档 (Documentation), 帮助 (Help), 社区 (Community), and En/nacos language switch.

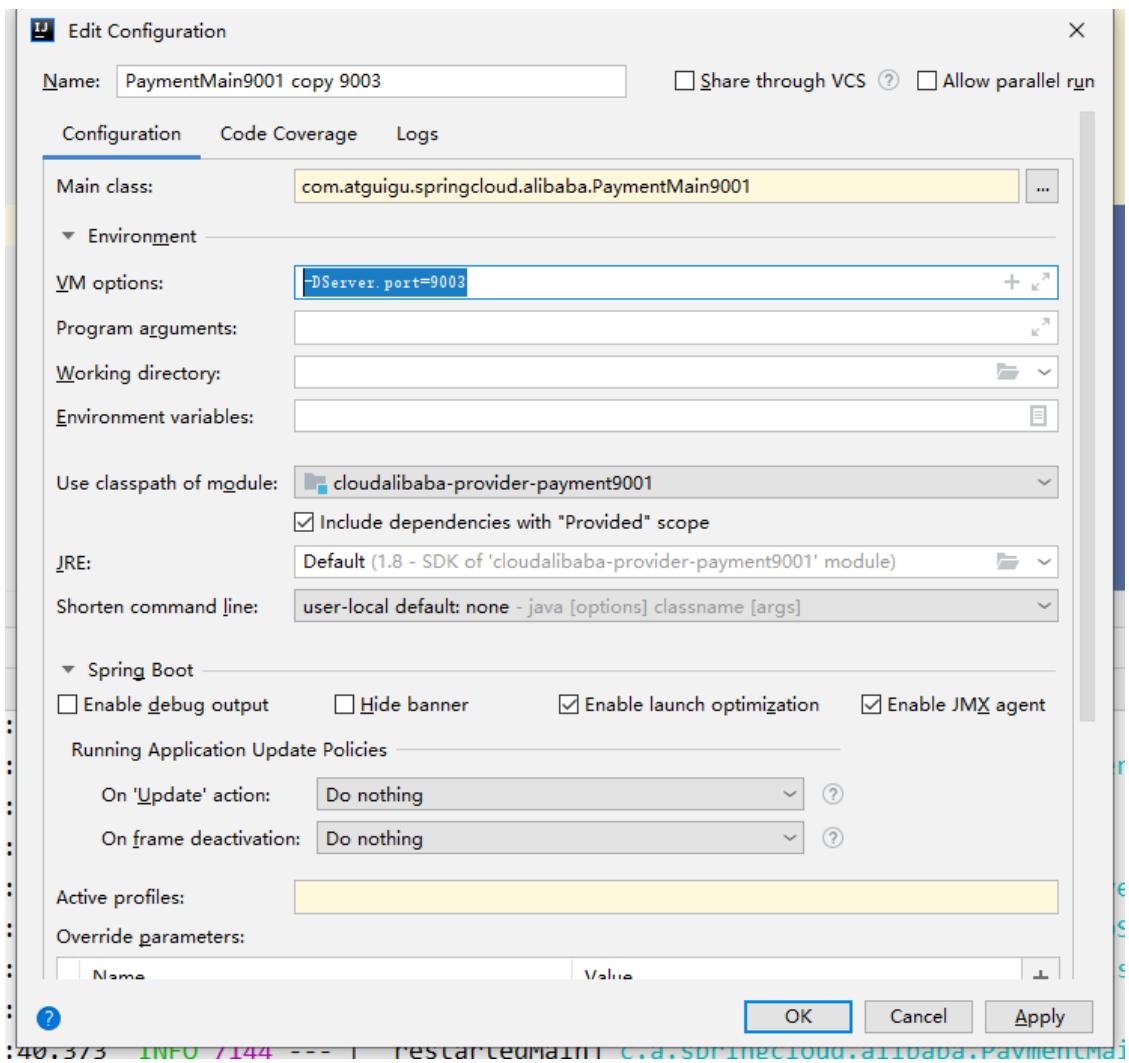
这个时候 nacos 服务注册中心 + 服务提供者 9001 都 OK 了

通过 IDEA 的拷贝映射



添加

-DServer.port=9003



最后能够看到两个实例

服务列表   public						
名称	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
cos-payment-provider	DEFAULT_GROUP	1	2	2	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">删除</a>

## 服务消费者注册到 Nacos

Nacos 天生集成了 Ribbon，因此它就具备负载均衡的能力

### 引入依赖

```
<!--SpringCloud alibaba nacos-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

### 修改 yml

```
server:
  port: 83
spring:
  application:
    name: nacos-order-consumer
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
```

# 消费者将要去访问的微服务名称（注册成功进 nacos 的微服务提供者）

```
service-url:
  nacos-user-service: http://nacos-payment-provider
```

### 增加配置类

因为 nacos 集成了 Ribbon，因此需要配置 RestTemplate，同时通过注解 @LoadBalanced 实现负载均衡，默认是轮询的方式

```
@Configuration
public class ApplicationContextConfig {
    @Bean
    @LoadBalanced
    public RestTemplate getRestTemple() {
        return new RestTemplate();
    }
}
```

## 业务类

```
@RestController
@Slf4j
public class OrderNacosController {

    @Resource
    private RestTemplate restTemplate;

    @Value("${service-url.nacos-user-service}")
    private String serverURL;

    @GetMapping(value = "/consumer/payment/nacos/{id}")
    public String paymentInfo(@PathVariable("id")Long id){
        return restTemplate.getForObject(serverURL+"/payment/nacos/" + i
d, String.class);
    }
}
```

测试

<http://localhost:83/consumer/payment/nacos/13>

得到的结果

```
nacos registry ,serverPore:9001 id:13
nacos registry ,serverPore:9002 id:13
```

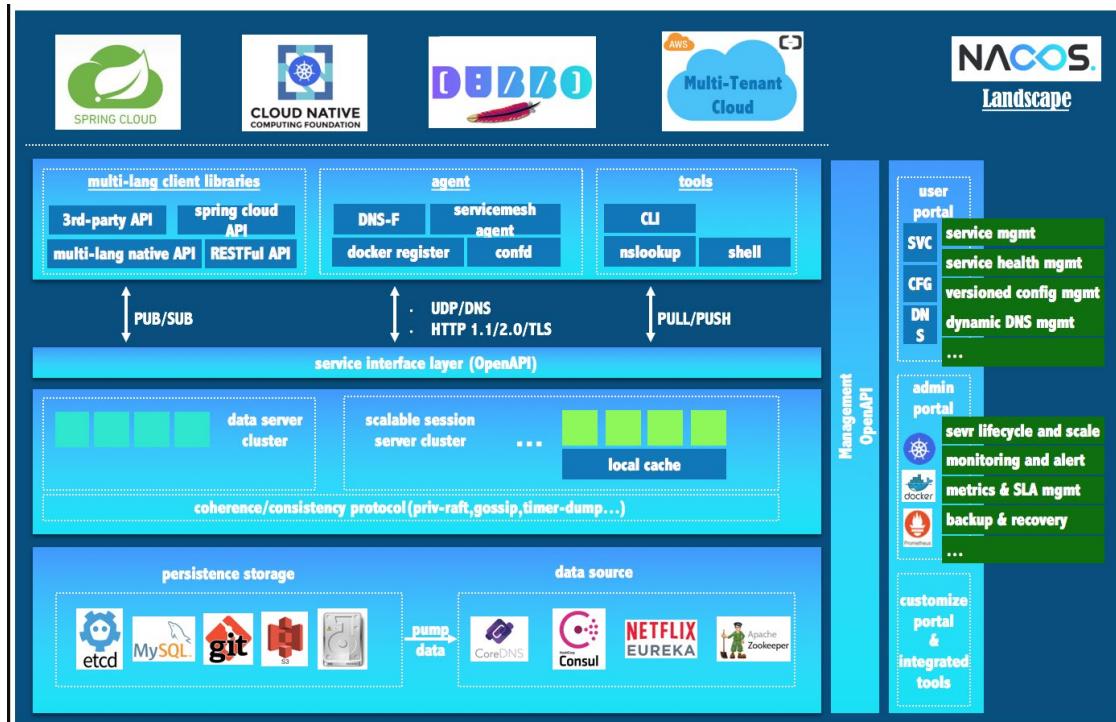
我们发现只需要配置了 nacos，就轻松实现负载均衡

## 服务中心对比

之前我们提到的注册中心对比图

服务注册与发现框架	CAP 模型	控制台管理	社区活跃度
Eureka	AP	支持	低 (2.x 版本闭源)
Zookeeper	CP	不支持	中
Consul	CP	支持	高
Nacos	AP	支持	高

但是其实 Nacos 不仅支持 AP，而且还支持 CP，它的支持模式是可以切换的，我们首先看看 Spring Cloud Alibaba 的全景图，



## Nacos 和 CAP

CAP：分别是一致性，可用性，分容容忍



我们从下图能够看到，nacos 不仅能够和 Dubbo 整合，还能和 K8s，也就是偏运维的方向

Nacos 与其他注册中心特性对比

	Nacos	Eureka	Consul	CoreDNS	ZooKeeper
一致性协议	CP+AP	AP	CP	/	CP
健康检查	TCP/HTTP/MySQL/Client Beat	Client Beat	TCP/HTTP/gRPC/Cmd	/	Client Beat
负载均衡	权重/DSL/metadata/CMDB	Ribbon	Fabio	RR	/
雪崩保护	支持	支持	不支持	不支持	不支持
自动注销实例	支持	支持	不支持	不支持	支持
访问协议	HTTP/DNS/UDP	HTTP	HTTP/DNS	DNS	TCP
监听支持	支持	支持	支持	不支持	支持
多数据中心	支持	支持	支持	不支持	不支持
跨注册中心	支持	不支持	支持	不支持	不支持
SpringCloud 集成	支持	支持	支持	不支持	不支持
Dubbo 集成	支持	不支持	不支持	不支持	支持
K8s 集成	支持	不支持	支持	支持	不支持

Nacos 服务发现实例模型

### Nacos 支持 AP 和 CP 切换

C 是指所有的节点同一时间看到的数据是一致的，而 A 的定义是所有的请求都会收到响应

合适选择何种模式？

一般来说，如果不需要存储服务级别的信息且服务实例是通过 nacos-client 注册，并能够保持心跳上报，那么就可以选择 AP 模式。当前主流的服务如 Spring Cloud 和 Dubbo 服务，都是适合 AP 模式，AP 模式为了服务的可用性而减弱了一致性，因此 AP 模式下只支持注册临时实例。

如果需要在服务级别编辑或存储配置信息，那么 CP 是必须，K8S 服务和 DNS 服务则适用于 CP 模式。

CP 模式下则支持注册持久化实例，此时则是以 Raft 协议为集群运行模式，该模式下注册实例之前必须先注册服务，如果服务不存在，则会返回错误。

## Nacos 作为服务配置中心演示

我们将我们的配置写入 Nacos，然后以 Spring Cloud Config 的方式，用于抓取配置

### Nacos 作为配置中心 - 基础配置

#### 引入依赖

```
<!--引入nacos config-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

#### 修改 YML

Nacos 同 SpringCloud Config 一样，在项目初始化时，要保证先从配置中心进行配置拉取，拉取配置之后，才能保证项目的正常运行。

SpringBoot 中配置文件的加载是存在优先级顺序的：bootstrap 优先级 高于 application

#### application.yml 配置

```
spring:
  profiles:
    #  active: dev # 开发环境
    #  active: test # 测试环境
    active: info # 开发环境
```

#### bootstrap.yml 配置

```
server:
  port: 3377
spring:
  application:
```

```
name: nacos-config-client
cloud:
  nacos:
    discovery:
      server-addr: localhost:8848 # 注册中心
    config:
      server-addr: localhost:8848 # 配置中心
      file-extension: yml # 这里指定的文件格式需要和 nacos 上新建的配置文件后缀相同，否则读不到
    group: TEST_GROUP
    namespace: 1bdf1418-3ed4-442c-97c1-f525b6a85b34

# ${spring.application.name}-${spring.profile.active}.${spring.cloud.nacos.config.file-extension}
```

## 主启动类

```
@SpringBootApplication
@EnableDiscoveryClient
public class NacosConfigClientMain3377 {
    public static void main(String[] args) {
        SpringApplication.run(NacosConfigClientMain3377.class, args);
    }
}
```

## 业务类

```
@RestController
@RefreshScope // 支持 nacos 的动态刷新
public class ConfigClientController {
    @Value("${config.info}")
    private String configInfo;

    @GetMapping("/config/info")
    public String getConfigInfo(){
        return configInfo;
    }
}
```

通过 SpringCloud 原生注解 `@RefreshScope` 实现配置自动刷新

## 在 Nacos 中添加配置信息

### Nacos 中匹配规则

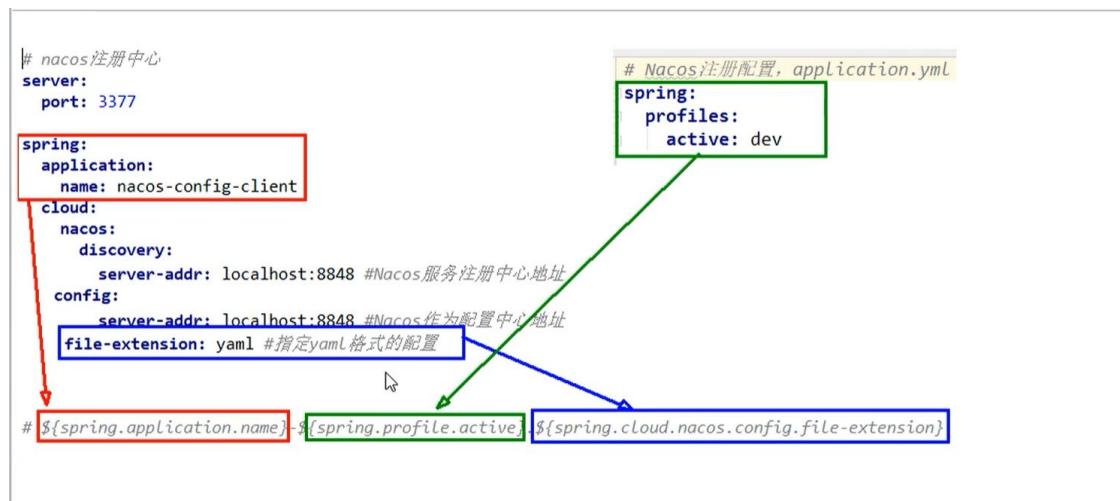
Nacos 中的 dataid 的组成格式及与 SpringBoot 配置文件中的匹配规则

```
${spring.application.name}-${spring.profile.active}.${spring.cloud.nacos.config.file-extension}
```

这样，就对应我们 Nacos 中的这样一个配置

## nacos-config-client-dev.yml

### 配置说明



### 我们在 Nacos 中添加配置

The screenshot shows the Nacos 1.1.4 UI interface. The left sidebar has a tree structure with nodes like 'NACOS 1.1.4', '配置管理' (highlighted), '历史版本', '监控查询', '服务管理', '命名空间', and '集群管理'. The main area is titled 'public' and contains a '配置管理' tab. The table lists a single configuration entry:

Data ID	Group	归属应用	操作
nacos-config-client-dev.yml	DEFAULT_GROUP		详情   示例代码   编辑   删除   复制

Buttons at the bottom include '导出为yaml文件' and '刷新'.

The screenshot shows the Nacos Configuration Center interface. On the left, there's a sidebar with sections like '配置管理' (Configuration Management), '服务管理' (Service Management), and '集群管理' (Cluster Management). The main area is titled '新建配置' (New Configuration) and shows a form for creating a new configuration. The 'Data ID' field contains 'nacos-config-client-dev.yml', and the 'Group' field is set to 'DEFAULT\_GROUP'. Below that is a '更多高级选项' (More Advanced Options) section with a '描述' (Description) input field containing 'nacos配置中心'. Under '配置格式' (Format), 'YAML' is selected. The '配置内容' (Content) field contains the following YAML code, which is highlighted with an orange box:

```
config:
  info: "config info for dev, from nacos config center"
```

这里需要注意的是，在 `config:` 的后面必须加上一个空格

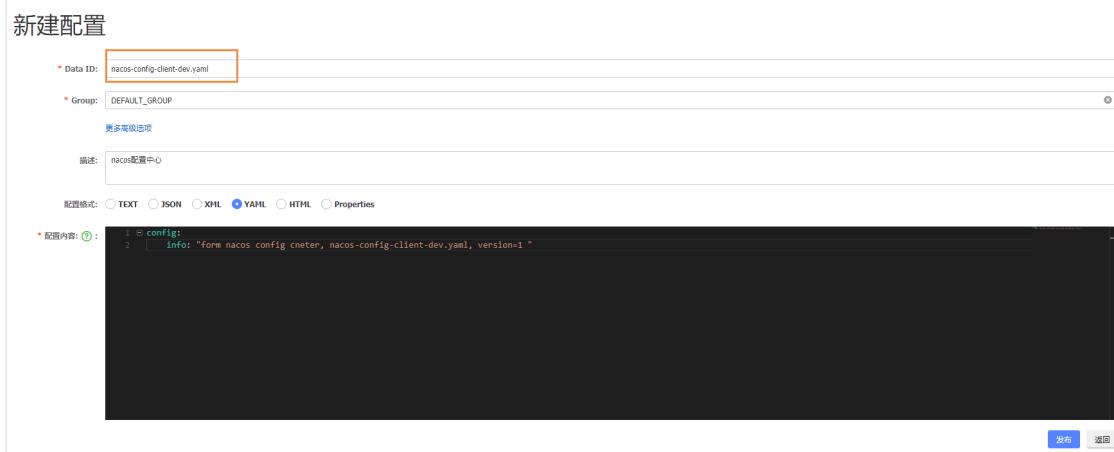
## 测试

启动前需要在 nacos 客户端-配置管理下有对应的 yml 配置文件，然后运行 `cloud-config-nacos-client:3377` 的主启动类，调用接口查看配置信息。

启动的时候出现问题

```
at org.springframework.boot.SpringApplication.run(SpringApplication.java:1215) [spring-boot-2.2.2.RELEASE.jar:2.2.2.RELEASE]
at com.atguigu.springcloud.alibaba.NacosConfigClientMain3377.main(NacosConfigClientMain3377.java:15) [classes/:na]
Caused by: java.lang.IllegalArgumentException: Could not resolve placeholder 'config.info' in value "${config.info}"
at org.springframework.util.PropertyPlaceholderHelper.parseStringValue(PropertyPlaceholderHelper.java:116) ~[spring-core-5.2.2.RELEASE]
at org.springframework.util.PropertyPlaceholderHelper.replacePlaceholders(PropertyPlaceholderHelper.java:124) ~[spring-core-5.2.2.RELEASE]
at org.springframework.core.env.AbstractPropertyResolver.doResolvePlaceholders(AbstractPropertyResolver.java:236) ~[spring-core-5.2.2.RELEASE]
at org.springframework.core.env.AbstractPropertyResolver.resolveRequiredPlaceholders(AbstractPropertyResolver.java:210) ~[spring-core-5.2.2.RELEASE]
at org.springframework.context.support.PropertySourcesPlaceholderConfigurer.lambda$processProperties$0(PropertySourcesPlaceholderConfigurer.java:90)
at org.springframework.beans.factory.support.AbstractBeanFactory.resolveEmbeddedValue(AbstractBeanFactory.java:908) ~[spring-beans-5.2.2.RELEASE]
at org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(DefaultListableBeanFactory.java:1228)
```

这是因为无法读取配置所引起的，解决方案就是我们的文件名不能用 `.yml` 而应该是 `.yaml`



我们需要删除重新建立。

## 自带动态刷新

修改 Nacos 中的 yaml 配置文件，再次查看配置的接口，就会发现配置已经刷新了

## Nacos 作为配置中心 - 分类配置

从上面的配置中心 + 动态刷新，就相当于有了 SpringCloud Config + Spring Cloud Bus 的功能

作为后起之秀的 Nacos，还具备分类配置的功能

### 问题

用于解决多环境多项目管理

在实际开发中，通常一个系统会准备

- dev 开发环境
- test 测试环境
- prod 生产环境

如何保证指定环境启动时，服务能正确读取到 Nacos 上相应环境的配置文件呢？

同时，一个大型分布式微服务系统会有很多微服务子项目，每个微服务子项目又都会有相应的开发环境，测试环境，预发环境，正式环境，那怎么对这些微服务配置进行管理呢？

## Nacos 图形化界面

配置管理:

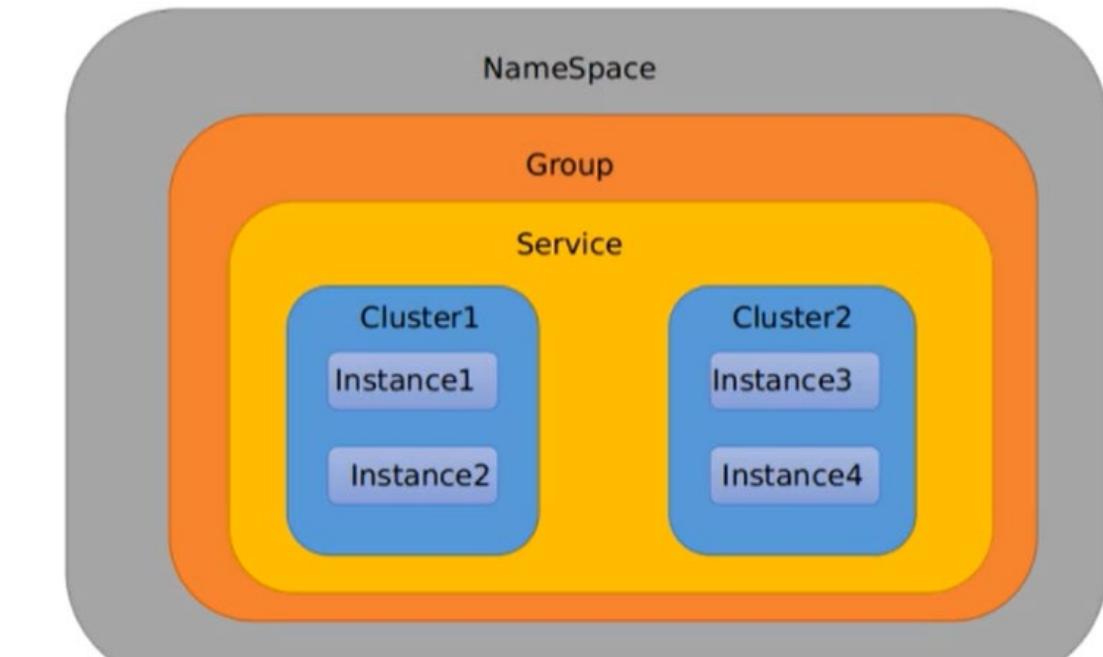
The screenshot shows the Nacos 1.4 configuration management interface. On the left sidebar, '命名空间' (Namespace) is highlighted with a red box. The main area shows a table with one entry: Data ID: nacos-config-client-dev.yaml, Group: DEFAULT\_GROUP. A red box highlights the 'public' entry in the table. A red arrow points from the '命名空间' box on the sidebar to the 'public' entry in the table.

命名空间:

The screenshot shows the Nacos 1.4 namespace management interface. On the left sidebar, '命名空间' (Namespace) is highlighted with a red box. The main area shows a table with one entry: 命名空间名称: public(保留空间). A red box highlights the 'public' entry in the table. A red arrow points from the '命名空间' box on the sidebar to the 'public' entry in the table.

### Namespace + Group + Data ID 三者关系

这种分类的设计思想，就类似于 java 里面的 package 名和类名，最外层的 namespace 是可以用于区分部署环境的，Group 和 DataID 逻辑上区分两个目标对象



默认情况：

Namespace=public, Group=DEFAULT\_GROUP, 默认 Cluster 是 DEFAULT

Nacos 默认的命名空间是 public, Namespace 主要用来实现隔离

比如说我们现在有三个环境：开发，测试，生产环境，我们就可以建立三个 Namespace，不同的 Namespace 之间是隔离的。

Group 默认是 DEFAULT\_GROUP, Group 可以把不同微服务划分到同一个分组里面去

Service 就是微服务，一个 Service 可以包含多个 Cluster（集群），Nacos 默认 Cluster 是 DEFAULT, Cluster 是对指定微服务的一个虚拟划分。比如说为了容灾，将 Service 微服务分别部署在了杭州机房，这时就可以给杭州机房的 Service 微服务起一个集群名称（HZ），给广州机房的 Service 微服务起一个集群名称，还可以尽量让同一个机房的微服务相互调用，以提升性能，最后 Instance，就是微服务的实例。

### 三种方案加载配置

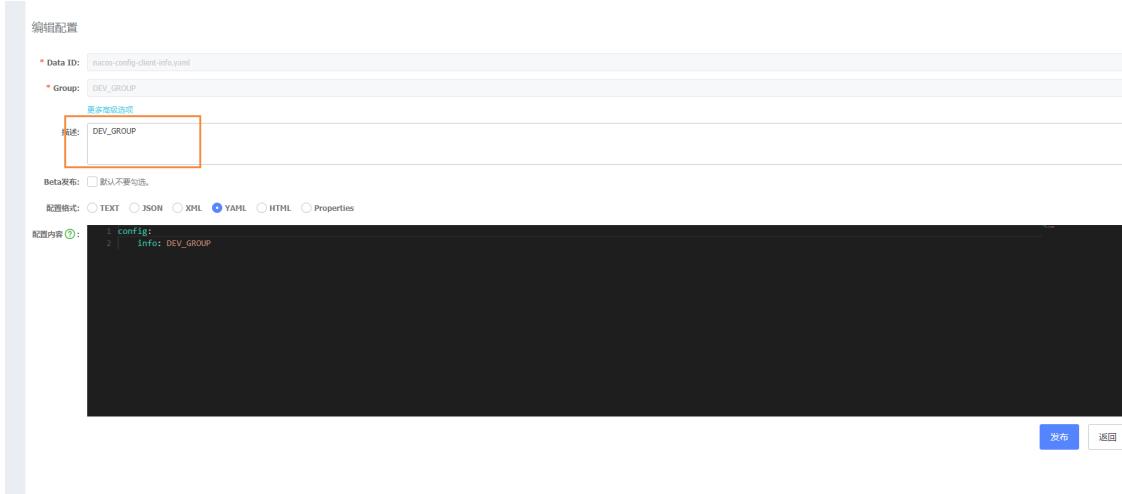
#### *DataID 方案*

- 指定 `spring.profile.active` 和 配置文件的 DataID 来使不同环境下读取不同的配置

- 默认空间 + 默认分组 + 新建 dev 和 test 两个 DataID

### Group 方案

在创建的时候，添加分组信息



然后就可以添加分组

```
server:
  port: 3377
spring:
  application:
    name: nacos-config-client
cloud:
  nacos:
    discovery:
      server-addr: localhost:8848 # 注册中心
    config:
      server-addr: localhost:8848 # 配置中心
      file-extension: yaml # 这里指定的文件格式需要和 nacos 上新建的配置
      files后缀相同，否则读不到
      group: TEST_GROUP
```

### Namspace 方案

首先我们需要新建一个命名空间

The screenshot shows the Nacos 1.1.4 web interface. On the left sidebar, under '命名空间' (Namespace), there is a sub-menu '命名空间' (Namespace) which is highlighted with an orange box. In the main content area, there is a table titled '命名空间' (Namespace) with columns: 命名空间名称 (Namespace Name), 命名空间ID (Namespace ID), 配置数 (Config Count), and 操作 (Operations). A row for 'public(保留空间)' is shown with 3/200 configurations. At the top right of the table, there is a blue button labeled '新建命名空间' (Create Namespace). A modal window titled '新建命名空间' (Create Namespace) is open in the center, also with a blue '新建命名空间' button at the bottom right. The modal has fields for '命名空间名' (Namespace Name) set to 'dev' and '描述' (Description) set to 'dev命名空间'.

新建完成后，能够看到有命名空间 id

This screenshot shows the same Nacos interface after the 'dev' namespace has been created. The table now includes a new row for 'dev' with a namespace ID of 'bbf379fb-f979-4eab-8947-2f08cfae6c0c'. The 'bbf379fb-f979-4eab-8947-2f08cfae6c0c' part of the ID is highlighted with an orange box. The rest of the interface remains the same, with the '命名空间' sub-menu still highlighted.

创建完成后，我们会发现，多出了几个命名空间切换

This screenshot shows the Nacos interface with three namespaces selected in the top navigation bar: 'public', 'test', and 'dev'. The 'dev' namespace is highlighted with an orange box. Below the navigation, the '配置管理' (Configuration Management) section is visible, showing a table of configuration items grouped by namespace. The table includes columns for Data ID, Group, 归属组 (Owner Group), and 操作 (Operations). There are three entries: 'nacos-config-client-dev.yaml' in 'DEFAULT\_GROUP', 'nacos-config-client-test.yaml' in 'DEFAULT\_GROUP', and 'nacos-config-client-info.yaml' in 'DEV\_GROUP'. Each entry has a small orange box highlighting its namespace group.

同时，我们到服务列表，发现也多了命名空间的切换

Nacos 1.1.4

public | test | dev

服务列表 | public

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
nacos-config-client	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">删除</a>

下面我们就可以通过引入 namespace，来创建到指定的命名空间下

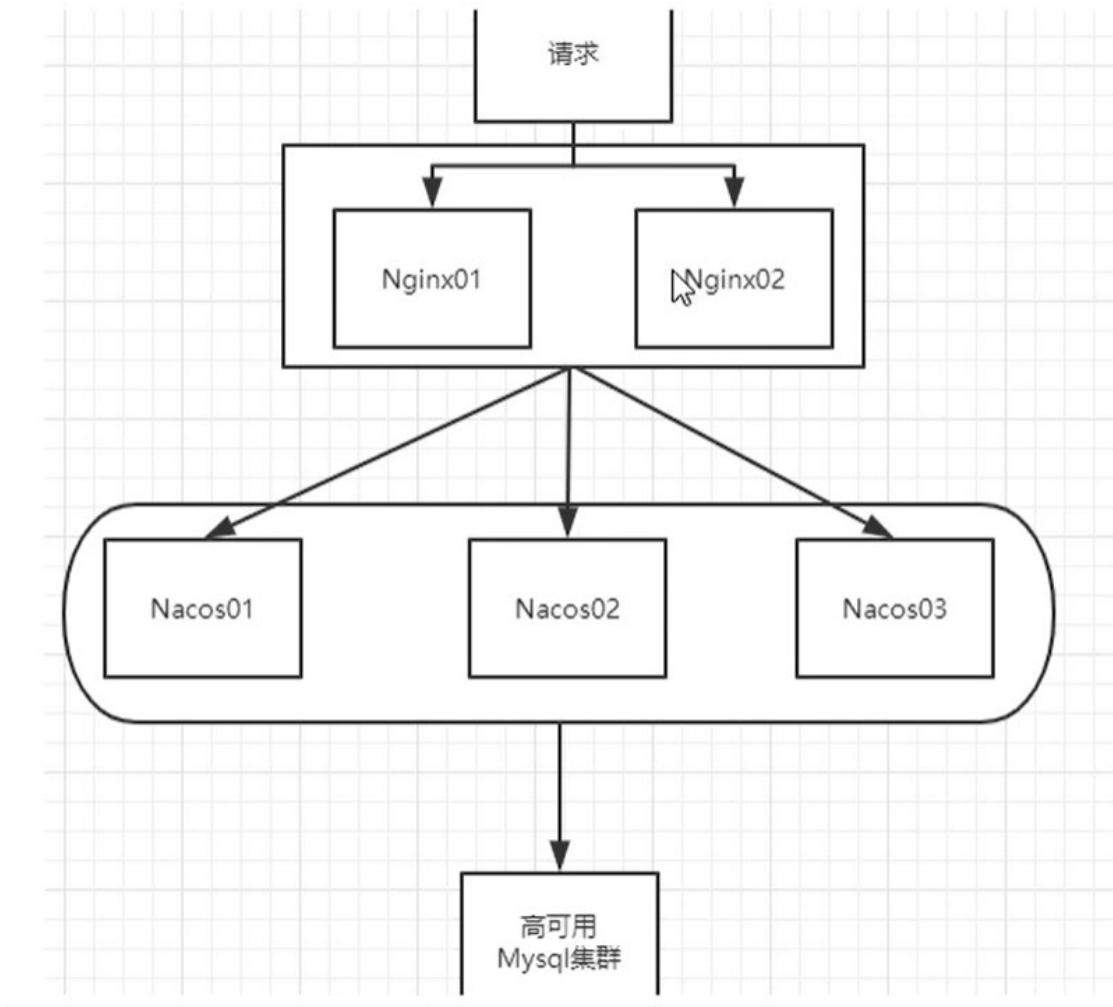
```
server:
  port: 3377
spring:
  application:
    name: nacos-config-client
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 # 注册中心
      config:
        server-addr: localhost:8848 # 配置中心
        file-extension: yaml # 这里指定的文件格式需要和 nacos 上新建的配置
        文件后缀相同，否则读不到
        group: DEV_GROUP
        namespace: bbf379fb-f979-4eab-8947-2f38cfaf6c0c
```

最后通过 namespace + group + DataID 形成三级分类

## Nacos 集群和持久化配置

### 官网说明

用于部署生产中的集群模式



默认 Nacos 使用嵌入数据库实现数据的存储，所以，如果启动多个默认配置下的 Nacos 节点，数据存储是存在一致性问题的。为了解决这个问题，Nacos 采用了集中式存储的方式来支持集群化部署，目前只支持 MySQL 的存储。

Nacos 支持三种部署模式

- **单机模式：**用于测试和单机使用
- **集群模式：**用于生产环境，确保高可用
- **多集群模式：**用于多数据中心场景

### 单机模式支持 mysql

在 0.7 版本之前，在单机模式下 nacos 使用嵌入式数据库实现数据的存储，不方便观察数据存储的基本情况。0.7 版本增加了支持 mysql 数据源能力，具体的操作流程：

- 安装数据库，版本要求：5.6.5 +
- 初始化数据库，数据库初始化文件：nacos-mysql.sql
- 修改 conf/application.properties 文件，增加 mysql 数据源配置，目前仅支持 mysql，添加 mysql 数据源的 url，用户名和密码

```
spring.datasource.platform=mysql

db.num=1
db.url.0=jdbc:mysql://11.162.196.16:3306/nacos_devtest?characterEncoding=utf8&connectTimeout=1000
db.user=nacos_devtest
db.password=youdontknow
```

再次以单机模式启动 nacos，nacos 所有写嵌入式数据库的数据都写到了 mysql 中。

## Nacos 持久化配置解释

Nacos 默认自带的是嵌入式数据库 derby

因此我们需要完成 derby 到 mysql 切换配置步骤

- 在 nacos\conf 目录下，找到 SQL 脚本

名称	修改日期	类型	大小
application.properties	2019/11/4 10:26	PROPERTIES 文件	2 KB
application.properties.example	2019/10/11 14:12	EXAMPLE 文件	1 KB
cluster.conf.example	2019/10/11 14:09	EXAMPLE 文件	1 KB
nacos-logback.xml	2019/11/4 10:26	XML 文档	20 KB
<b>nacos-mysql.sql</b>	<b>2019/10/11 14:12</b>	<b>SQL 文件</b>	<b>10 KB</b>
schema.sql	2019/10/11 14:12	SQL 文件	8 KB

然后执行 SQL 脚本，同时修改 application.properties 目录

[官网地址](#)

```
spring.datasource.platform=mysql

db.num=1
db.url.0=jdbc:mysql://127.0.0.1:3306/nacos_devtest?characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true
db.user=root
db.password=root
```

修改完成后，启动 nacos，可以看到是一个全新的空记录页面，以前是记录进 derby

## Linux 版 Nacos + Mysql 生产环境配置

### 配置

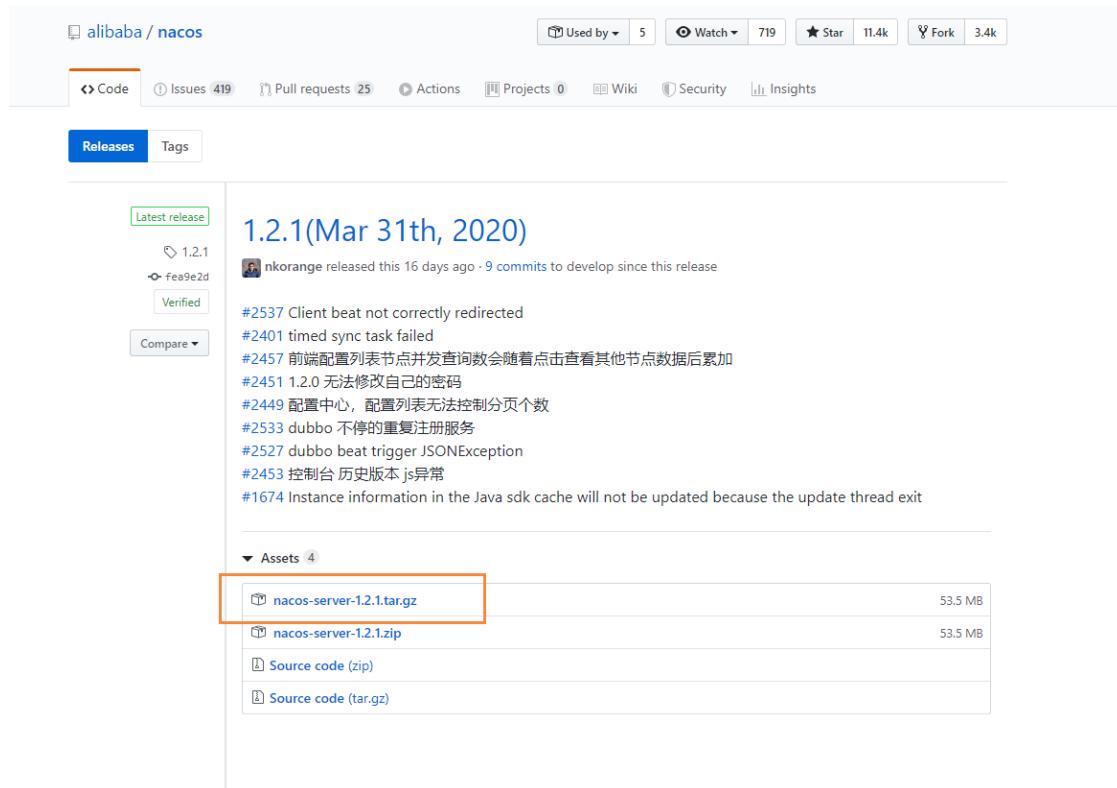
预计需要：1 个 Nginx + 3 个 nacos 注册中心 + 1 个 mysql

所有的请求过来，首先先打到 nginx 上

### Nacos 下载 Linux 版本

在 nacos github 下载：<https://github.com/alibaba/nacos/releases>

选择 Linux 版本下载



## 集群配置

如果是一个 nacos: 启动 8848 即可

如果是多个 nacos: 3333,4444,5555

那么就需要修改 `startup.sh` 里面的，传入端口号

步骤：

- Linux 服务器上 mysql 数据库配置
- `application.properties` 配置
- Linux 服务器上 nacos 的集群配置 `cluster.conf`
  - 梳理出 3 台 nacos 集群的不同服务端口号
  - 复制出 `cluster.conf`（备份）
  - 修改

```

# it is ip
#example
#10. 10. 109. 214
#11. 16. 128. 34
#11. 16. 128. 36

192. 168. 111. 144: 3333
192. 168. 111. 144: 4444
192. 168. 111. 144: 5555

```

- 编辑 Nacos 的启动脚本 startup.sh，使它能够接受不同的启动端口
  - /nacos/bin 目录下有 startup.sh
  - 平时单机版的启动，直接./startup.sh
  - 但是集群启动时，我们希望可以类似其它软件的 shell 命令，传递不同的端口号启动不同的 nacos 实例，命令：./startup.sh -p 3333 表示启动端口号为 3333 的 nacos 服务器实例，和上一步的 cluster.conf 配置一样。

修改启动脚本，添加 P，这样能够明确 nacos 启动的什么脚本

修改前	修改后
<pre> while getopts ":m:f:s:" opt do   case \$opt in     m)       MODE=\$OPTARG; ;     f)       FUNCTION_MODE=\$OPTARG; ;     s)       SERVER=\$OPTARG; ;     ?)       echo "Unknown parameter"       exit 1; ;   esac done </pre>	<pre> while getopts ":m:f:s:p:" opt do   case \$opt in     m)       MODE=\$OPTARG; ;     f)       FUNCTION_MODE=\$OPTARG; ;     s)       SERVER=\$OPTARG; ;     p)       PORT=\$OPTARG; ;     ?)       echo "Unknown parameter"       exit 1; ;   esac done </pre>
<pre> # start echo "\$JAVA \$JAVA_OPT" &gt; \${BASE_DIR}/logs/start.out 2&gt;&amp;1 &amp; nohup \$JAVA \$JAVA_OPT nacos.nacos &gt;&gt; \${BASE_DIR}/logs/start.out 2&gt;&amp;1 &amp; echo "nacos is starting, you can check the \${BASE_DIR}/logs/start.out" </pre>	<pre> # start echo "\$JAVA \$JAVA_OPT" &gt; \${BASE_DIR}/logs/start.out 2&gt;&amp;1 &amp; nohup \$JAVA -Dserver.port=\${PORT} \$JAVA_OPT nacos.nacos &gt;&gt; \${BASE_DIR}/logs/start.out 2&gt;&amp;1 &amp; echo "nacos is starting, you can check the \${BASE_DIR}/logs/start.out" </pre>

修改完成后，就能够使用下列命令启动集群了

```
./startup.sh -p 3333  
./startup.sh -p 4444  
./startup.sh -p 5555
```

- Nginx 的配置，由它作为负载均衡器
  - 修改 nginx 的配置文件

```
#gzip on;  
  
upstream cluster{  
    server 127.0.0.1:3333;  
    server 127.0.0.1:4444;  
    server 127.0.0.1:5555;  
}  
  
server {  
    listen 1111;  
    server_name localhost;  
  
    #charset koi8-r;  
  
    #access_log logs/host.access.log main;  
  
    location / {  
        #root html;  
        #index index.html index.htm;  
        proxy_pass http://cluster;  
    }  
  
    #error_page 404 /404.html;  
}
```

作为负载均衡分流，同时 upstream 支持 weight

通过 nginx 访问 nacos 节点: <http://192.168.111.144:1111/nacos/#/login>

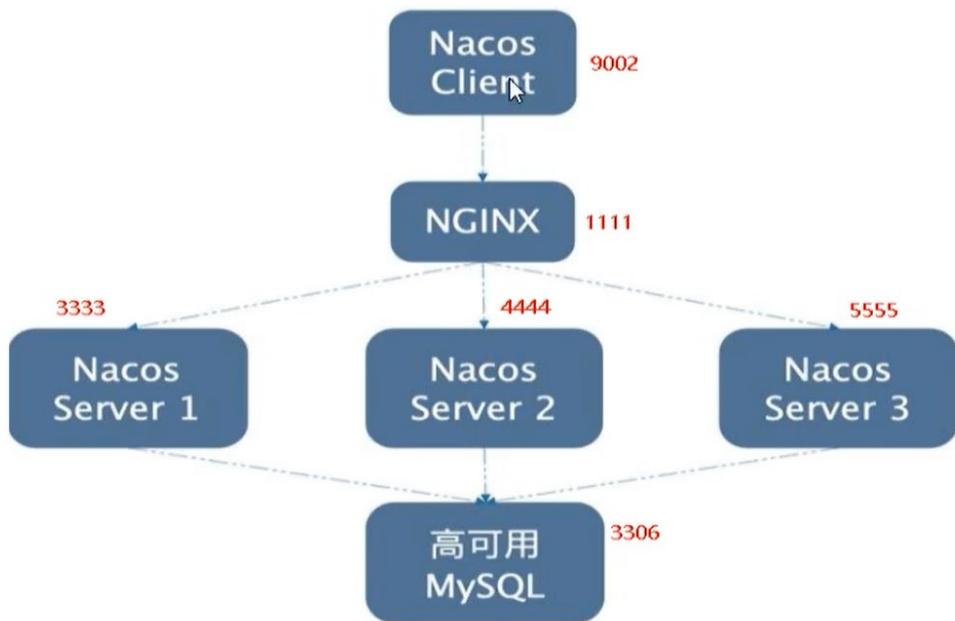
微服务注册进集群中

```
server:  
  port: 9002  
spring:  
  application:  
    name: nacos-payment-provider  
cloud:
```

```
nacos:  
  discovery:  
    server-addr: 192.168.111.144:1111 # 换成 nginx 的 1111 端口，做负载均衡  
  management:  
    endpoints:  
      web:  
        exposure:  
          include: '*'
```

## 总结

Nginx + 3 个 Nacos + mysql 的集群化配置



## SpringCloudAlibabaSentinel 实现熔断和限流

### Sentinel

#### 官网

Github: <https://github.com/alibaba/Sentinel>

Sentinel: 分布式系统的流量防卫兵，相当于 Hystrix

Hystrix 存在的问题

- 需要我们程序员自己手工搭建监控平台
- 没有一套 web 界面可以给我们进行更加细粒度化的配置，流量控制，速率控制，服务熔断，服务降级。

这个时候 Sentinel 运营而生

- 单独一个组件，可以独立出来
- 直接界面化的细粒度统一配置

约定 > 配置 > 编码，都可以写在代码里，但是尽量使用注解和配置代替编码

## 是什么

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。Sentinel 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保护服务的稳定性。

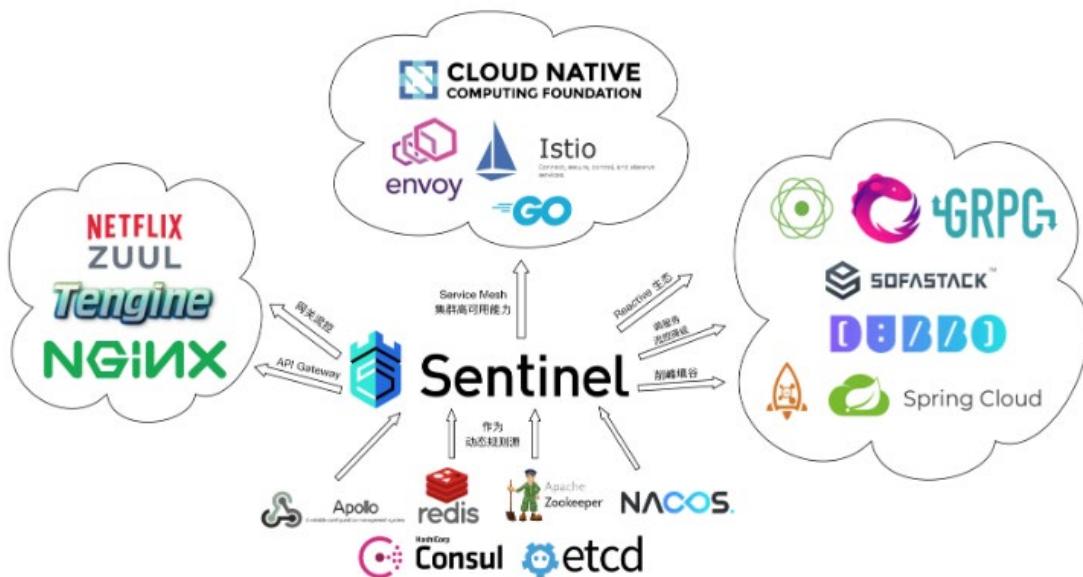
Sentinel 具有以下特征：

- **丰富的应用场景：** Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息削峰填谷、集群流量控制、实时熔断下游不可用应用等。
- **完备的实时监控：** Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器秒级数据，甚至 500 台以下规模的集群的汇总运行情况。
- **广泛的开源生态：** Sentinel 提供开箱即用的与其它开源框架/库的整合模块，例如与 Spring Cloud、Dubbo、gRPC 的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入 Sentinel。
- **完善的 SPI 扩展点：** Sentinel 提供简单易用、完善的 SPI 扩展接口。您可以通过对实现扩展接口来快速地定制逻辑。例如定制规则管理、适配动态数据源等。

## 主要特征



## 生态圈



## 下载

Github: <https://github.com/alibaba/Sentinel/releases>

- Force modifyRule command handler to fail if an incompatible old fastjson found ([#1377](#))
- Calculate both process and system CPU usage to support single process running in container environment ([#1204](#))
- Improve the logs when the heartbeat response indicates failure ([#1303](#))

### Bug fixes

- Fix NPE bug in Tracer when context size exceeds the limit ([#1293](#))
- Fix the parsing issue in large post request for sentinel-transport-simple-http ([#1255](#))
- Fix the bug that context was not released when blocked in Spring Web adapter ([#1353](#))
- Fix timezone problem of sentinel-block.log

### Dashboard

- Improve the compatibility on the Content-Type header of POST request ([#1260](#))
- Enhance reliability and performance of InMemoryMetricsRepository ([#1319](#))
- Support setting value pattern for client IP and host in gateway flow rule dialog ([#1325](#))
- Hide advanced options in flow rule dialog when cluster mode is selected ([#1367](#))
- Fix NoNodeException problem of FlowRuleZookeeperProvider sample ([#1352](#))

Thanks for the contributors: @cdfive, @echoymxq, @jasonjoo2010, @jy2156121, @linlinisme, @mantuliu, @olof-nord, @sczyh30, @tianhaowhu, @wavesZh, @WongTheo, @xue8, @zhaoyuguang

▼ Assets 4		
	<a href="#">sentinel-dashboard-1.7.2.jar</a>	20.2 MB
	<a href="#">sentinel-envoy-rls-token-server-1.7.2.jar</a>	12.1 MB
	<a href="#">Source code (zip)</a>	
	<a href="#">Source code (tar.gz)</a>	

## 安装 Sentinel 控制台

sentinel 组件由两部分组成，后台和前台 8080

Sentinel 分为两部分

- 核心库（Java 客户端）不依赖任何框架/库，能够运行在所有 Java 运行时环境，同时对 Dubbo、SpringCloud 等框架也有较好的支持。
- 控制台（Dashboard）基于 SpringBoot 开发，打包后可以直接运行，不需要额外的 Tomcat 等应用容器

使用 `java -jar` 启动，同时 Sentinel 默认的端口号是 8080，因此不能被占用

注意，下载时候，由于 Github 经常抽风，因此可以使用 Gitee 进行下，首先先去 Gitee 下载源码

Sentinel 是什么 随着微服务的流行，服务和服务之间的稳定性变得越来越重要  
<https://www.oschina.net/p/sentinel>

561 次提交	77 个分支	19 个标签	0 个发行版	97 位贡献者
1.7.2	文件	挂件	克隆/下载	
<b>Eric Zhao</b> 最后提交于 8 天前 Bump version to 1.7.2				
.circleci Disable spell checking in Circle CI lint temporary (need more config later) 1年前 .github Update documentation and issue template 1年前 doc doc: Update README.md 9天前 sentinel-adapter Bump version to 1.7.2 8天前 sentinel-benchmark Bump version to 1.7.2 8天前 sentinel-cluster Bump version to 1.7.2 8天前 sentinel-core Bump version to 1.7.2 8天前 sentinel-dashboard Bump version to 1.7.2 8天前 sentinel-demo Bump version to 1.7.2 8天前 sentinel-extension Bump version to 1.7.2 8天前 sentinel-logging Bump version to 1.7.2 8天前 sentinel-transport Bump version to 1.7.2 8天前 .codecov.yml Update codecov conf file 1年前				

然后执行 `mvn package` 进行构建，本博客同级目录下了，已经有个已经下载好的，欢迎自取

## 初始化演示工程

启动 Nacos8848 成功

### 引入依赖

```
<!--SpringCloud alibaba sentinel-datasource-nacos 后续做持久化用到-->
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>

<!--SpringCloud alibaba sentinel -->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
```

```
<artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

## 修改 YML

```
server:
  port: 8401

spring:
  application:
    name: cloudalibaba-sentinel-service
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #Nacos 服务注册中心地址
    sentinel:
      transport:
        dashboard: localhost:8080 #配置 Sentinel dashboard 地址
        port: 8719
```

## 增加业务类

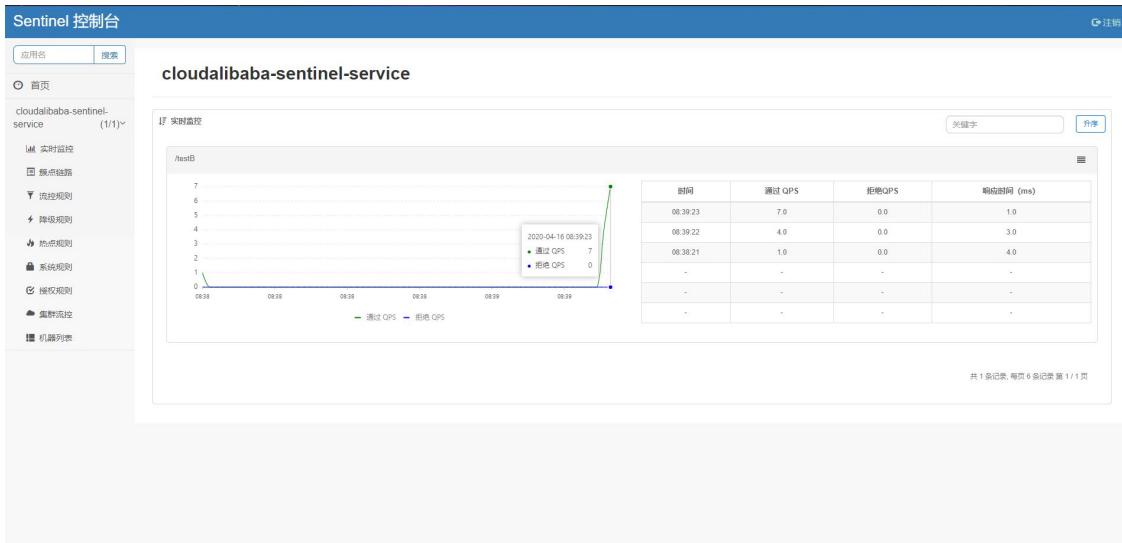
```
@RestController
@Slf4j
public class FlowLimitController
{
    @GetMapping("/testA")
    public String testA()
    {
        return "-----testA";
    }

    @GetMapping("/testB")
    public String testB()
    {
        log.info(Thread.currentThread().getName()+"\t"+"...testB");
        return "-----testB";
    }
}
```

启动 8401 微服务，查看 Sentinel 控制台

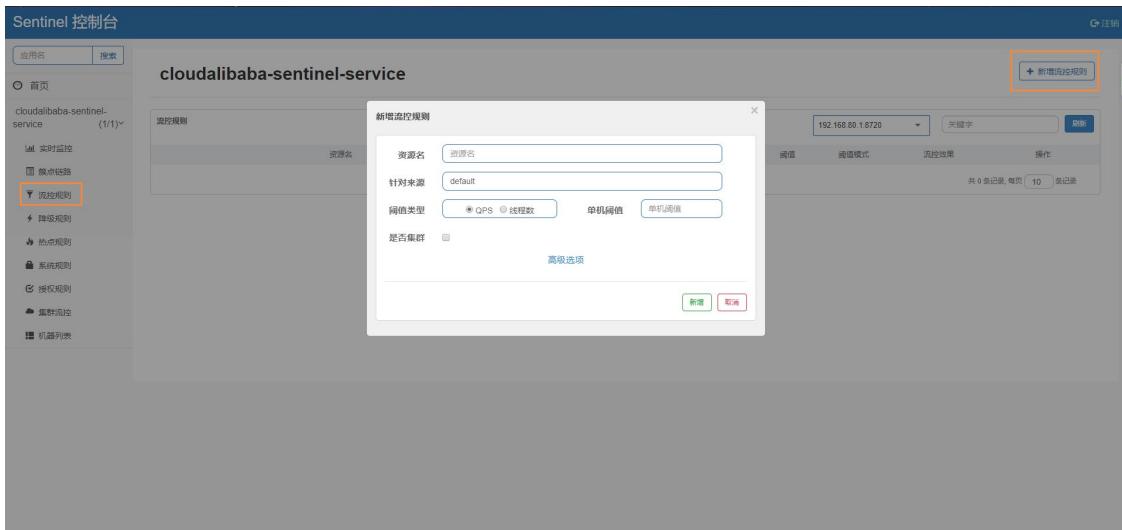
我们会发现 Sentinel 里面空空如也，什么也没有，这是因为 Sentinel 采用的懒加载

执行一下访问即可：<http://localhost:8401/testA>  
<http://localhost:8401/testB>



## 流控规则

### 基本介绍



### 字段说明

- 资源名: 唯一名称, 默认请求路径
- 针对来源: Sentinel 可以针对调用者进行限流, 填写微服务名, 默认 default (不区分来源)

- 阈值类型 / 单机阈值
  - QPS: (每秒钟的请求数量) : 但调用该 API 的 QPS 达到阈值的时候, 进行限流
  - 线程数: 当调用该 API 的线程数达到阈值的时候, 进行限流
- 是否集群: 不需要集群
- 流控模式
  - 直接: api 都达到限流条件时, 直接限流
  - 关联: 当关联的资源达到阈值, 就限流自己
  - 链路: 只记录指定链路上的流量 (指定资源从入口资源进来的流量, 如果达到阈值, 就进行限流) 【API 级别的针对来源】
- 流控效果
  - 快速失败: 直接失败, 抛异常
  - Warm UP: 根据 codeFactory (冷加载因子, 默认 3), 从阈值 /CodeFactor, 经过预热时长, 才达到设置的 QPS 阈值
  - 排队等待: 匀速排队, 让请求以匀速的速度通过, 阈值类型必须设置 QPS, 否则无效

## 流控模式

### 直接 (默认)

我们给 testA 增加流控

The screenshot shows the Sentinel Control Panel interface. On the left, there's a sidebar with navigation items like '实时监控' (Real-time Monitoring), '限流规则' (Flow Control Rules), '降级规则' (Degradation Rules), '热点规则' (Hotspot Rules), '授权规则' (Authorization Rules), and '集群状态' (Cluster Status). The main area is titled 'cloudalibaba-sentinel-service' and contains a table for '节点限流' (Node Flow Control). The table has columns: 资源名 (Resource Name), 通过QPS (Throughput QPS), 拒绝QPS (Rejected QPS), 线程数 (Threads), 平均RTT (Average RTT), 分钟通过 (Throughput per minute), 分钟拒绝 (Rejected per minute), and 操作 (Operations). There are two rows: 'favicon.ico' and 'testA'. The 'testA' row has its '资源名' field highlighted with a red box. In the '操作' column for 'testA', there are four buttons: '+ 流控' (Flow Control), '+ 降级' (Degradation), '+ 热点' (Hotspot), and '+ 授权' (Authorization). The '+ 流控' button is highlighted with a blue box.

Sentinel 控制台

cloudalibaba-sentinel-service

新增流控规则

资源名: testA  
针对来源: default  
阈值类型:  QPS  线程数  
单机阈值: 1  
是否集群:   
流控模式:  直接  关联  链路  
流控效果:  快速失败  Warm Up  排队等待  
关闭高级选项  
当QPS大于了单机阈值，马上报错

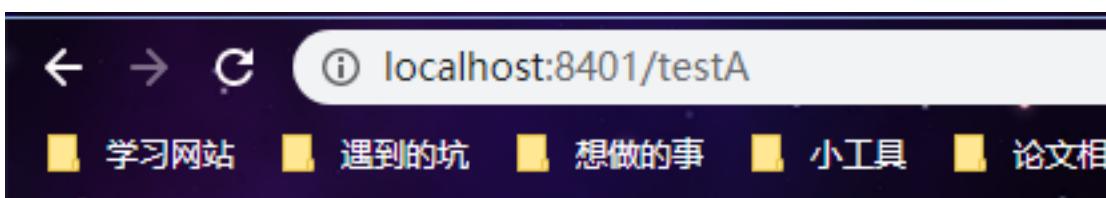
操作: + 流控 + 升级 + 热点 + 授权  
+ 流控 + 升级 + 热点 + 授权  
+ 流控 + 升级 + 热点 + 授权

共 3 条记录, 每页 16 条记录

表示1秒钟内查询1次就是OK，若超过次数1，就直接-快速失败，报默认错误



然后我们请求 <http://localhost:8401/testA>，就会出现失败，被限流，快速失败



思考：

直接调用的是默认报错信息，能否有我们的后续处理，比如更加友好的提示，类似有 hystrix 的 fallback 方法

线程数

这里的线程数表示一次只有一个线程进行业务请求，当前出现请求无法响应的时候，会直接报错，例如，在方法的内部增加一个睡眠，那么后面来的就会失败

```
@GetMapping("/testD")
public String testD()
{
    try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e) { e.printStackTrace(); }
    return "-----testD";
}
```

## 关联

当关联的资源达到阈值时，就限流自己

当与 A 关联的资源 B 达到阈值后，就限流 A 自己，B 惹事，A 挂了

场景：支付接口达到阈值后，就限流下订单的接口

设置：

当关联资源 /testB 的 QPS 达到阈值超过 1 时，就限流/testA 的 Rest 访问地址，  
当关联资源达到阈值后，限制配置好的资源名



这个使用我们利用 postman 模拟并发密集访问 testB

首先我们需要使用 postman，创建一个请求

New Report Runner

My Workspace Invite

No Environment Examples (0)

Collection2020.4 1 request

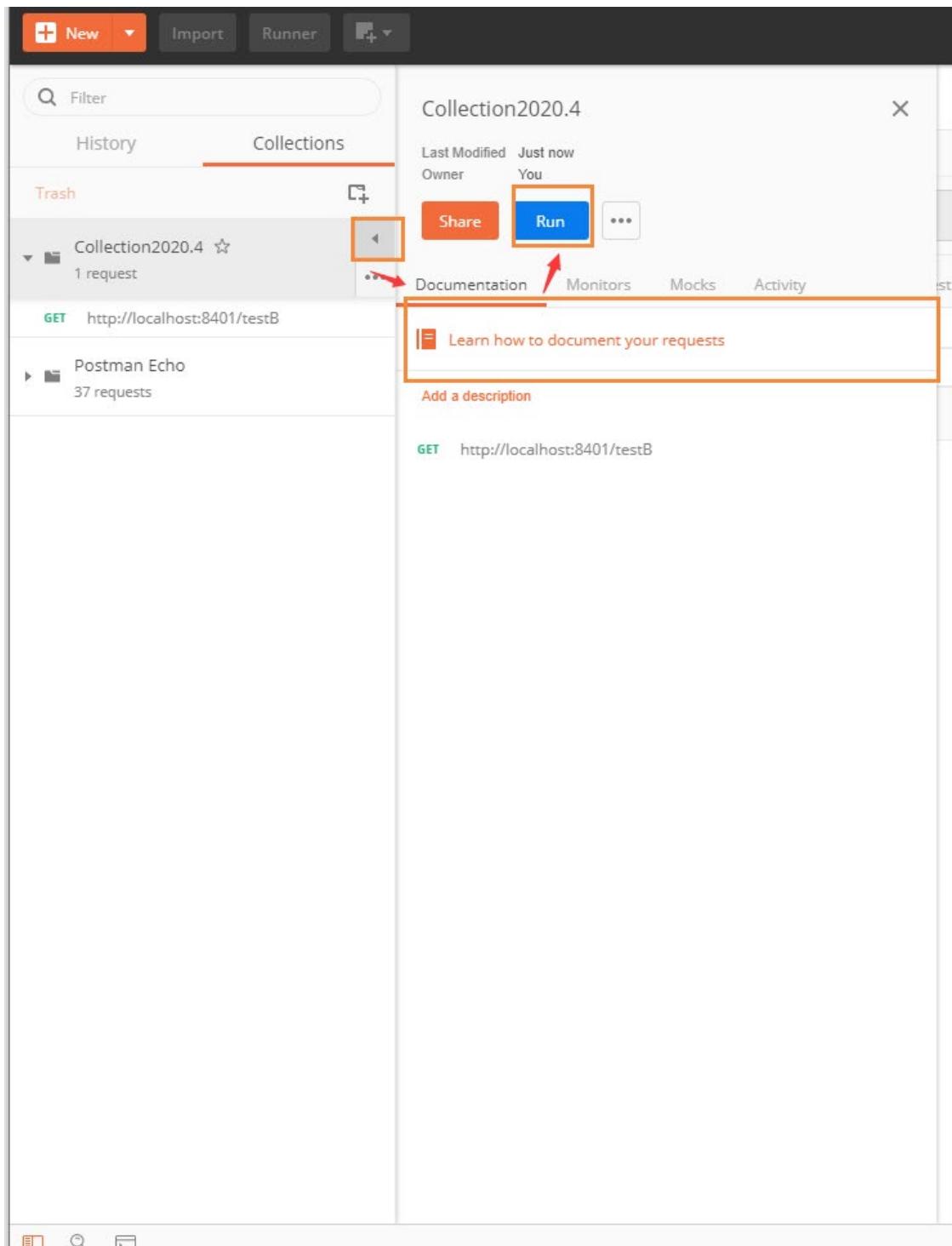
POST http://localhost:8401/testB

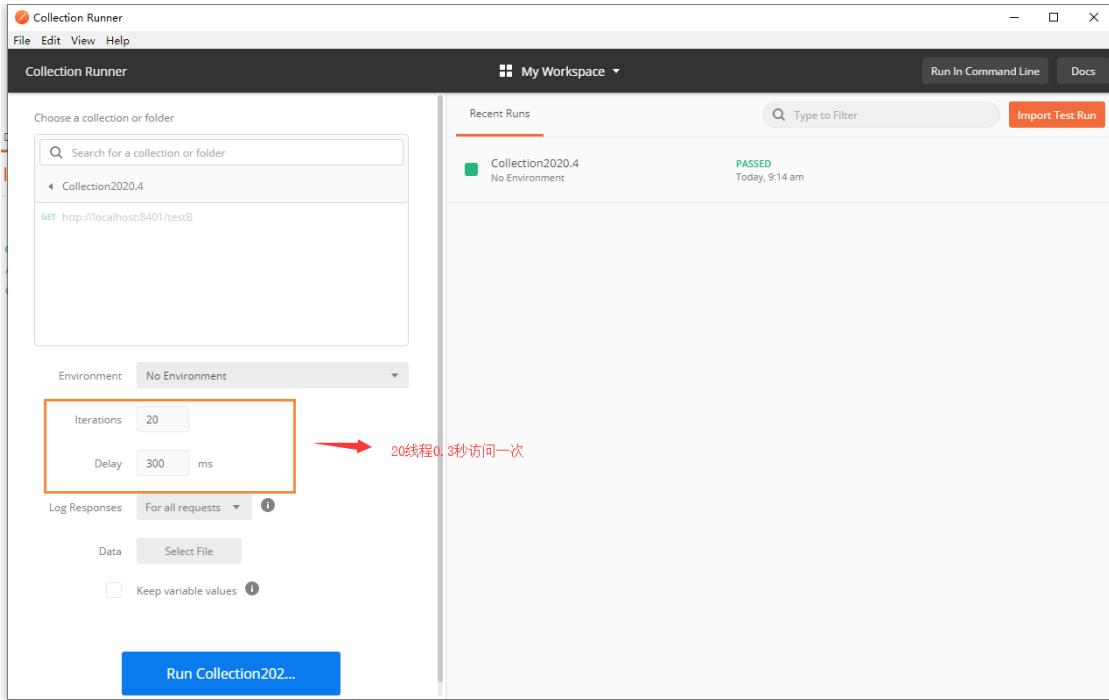
Send Save

Hit the Send button to get a response.

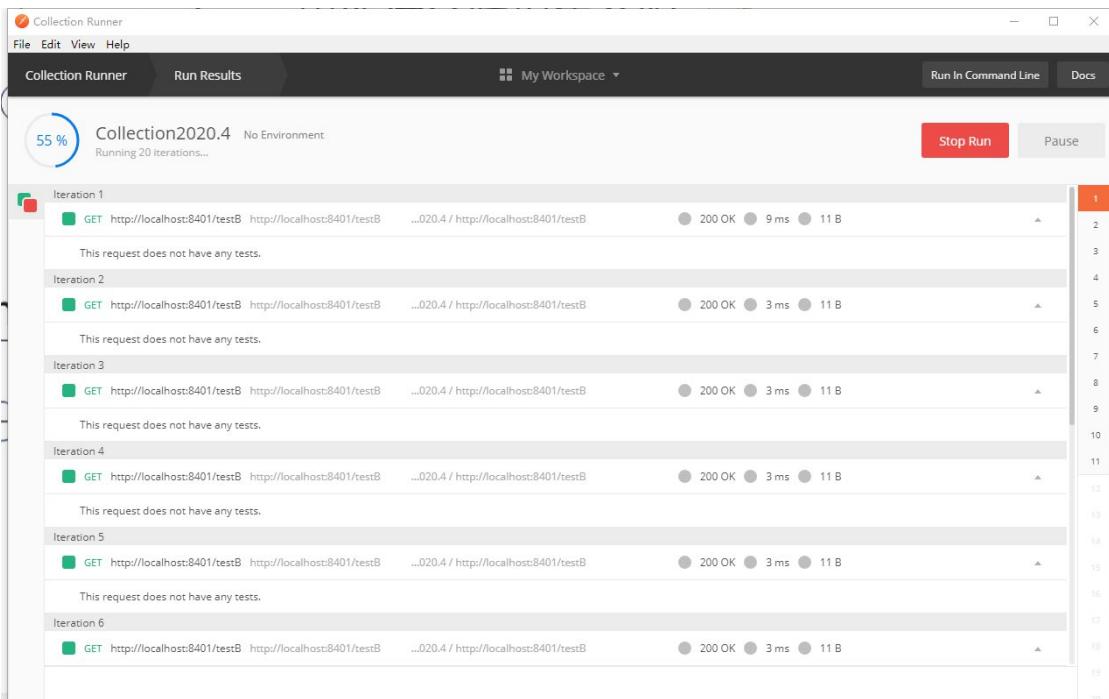
同时将请求保存在 Collection 中

然后点击箭头，选中接口，选择 run

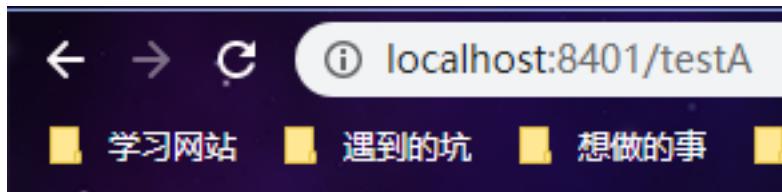




点击运行，大批量线程高并发访问 B，导致 A 失效了，同时我们点击访问 <http://localhost:8401/testA>，结果发现，我们的 A 已经挂了



在测试 A 接口



Blocked by Sentinel (flow limiting)

这就是我们的关联限流

## 链路

多个请求调用了同一个微服务

## 流控效果

### 直接

快速失败，默认的流控处理

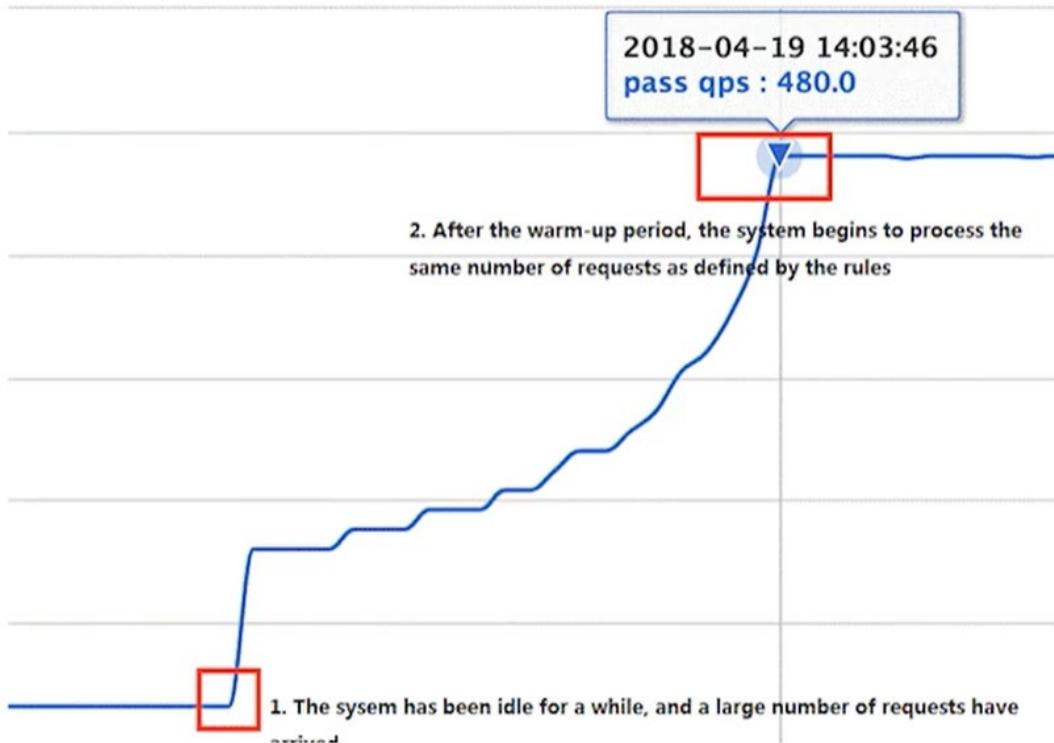
- 直接失败，抛出异常：Blocked by Sentinel (Flow limiting)

### 预热

系统最怕的就是出现，平时访问是 0，然后突然一瞬间来了 10W 的 QPS

公式：阈值 除以 coldFactor（默认值为 3），经过预热时长后，才会达到阈值

Warm Up 方式，即预热/冷启动方式，当系统长期处于低水位的情况下，当流量突然增加时，直接把系统拉升到高水位可能会瞬间把系统压垮。通过冷启动，让通过的流量缓慢增加，在一定时间内逐渐增加到阈值，给冷系统一个预热的时间，避免冷系统被压垮。通常冷启动的过程系统允许的 QPS 曲线如下图所示



默认 cloudFactor 为 3，即请求 QPS 从 threshold / 3 开始，经预热时长逐渐提升至设定的 QPS 阈值

编辑流控规则

资源名	/testA
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
单机阈值	10
是否集群	<input type="checkbox"/>
流控模式	<input checked="" type="radio"/> 直接 <input type="radio"/> 关联 <input type="radio"/> 链路
关联资源	/testB
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待
预热时长	5

[关闭高级选项](#)

[保存](#) [取消](#)

假设这个系统的 QPS 是 10，那么最开始系统能够接受的  $QPS = 10 / 3 = 3$ ，然后从 3 逐渐在 5 秒内提升到 10

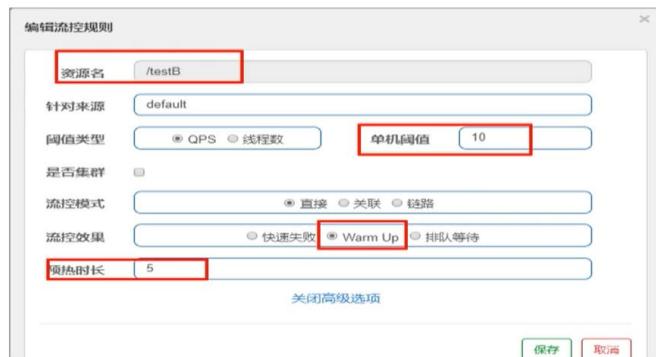
应用场景：

秒杀系统在开启的瞬间，会有很多流量上来，很可能把系统打死，预热的方式就是为了保护系统，可能慢慢的把流量放进来，慢慢的把阈值增长到设置的阈值。

默认 coldFactor 为 3，即请求QPS从(threshold / 3)开始，经多少预热时长才逐渐升至设定的 QPS 阈值。

案例，阀值为10+预热时长设置5秒。

系统初始化的阀值为 $10 / 3$  约等于3,即阀值刚开始为3；然后过了5秒后阀值才慢慢升高恢复到10

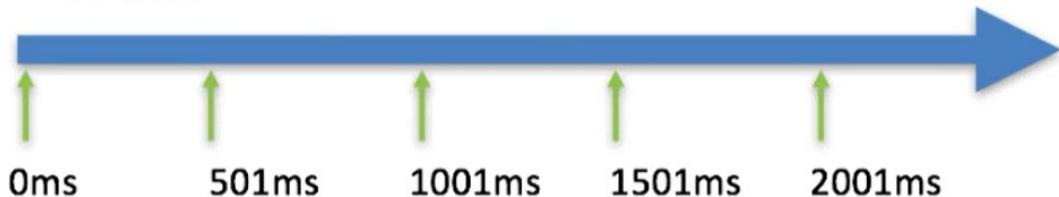


## 排队等待

大家均速排队，让请求以均匀的速度通过，阈值类型必须设置成 QPS，否则无效

均速排队方式必须严格控制请求通过的间隔时间，也即让请求以匀速的速度通过，对应的是漏桶算法。

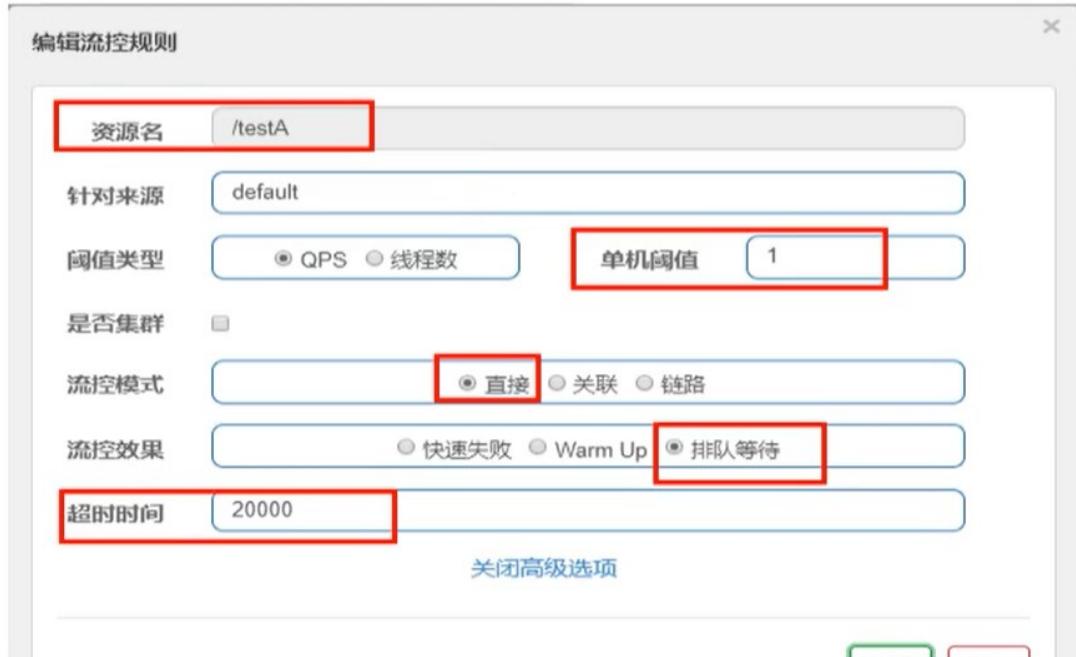
## 时间轴



阈值QPS=2时，每隔500ms才允许通过下一个请求

这种方式主要用于处理间隔性突发的流量，例如消息队列，想象一下这样的场景，在某一秒有大量的请求到来，而接下来的几秒处于空闲状态，我们系统能够接下來的空闲期间逐渐处理这些请求，而不是在第一秒直接拒绝多余的请求。

设置含义：/testA 每秒 1 次请求，超过的话，就排队等待，等待时间超过 20000 毫秒



## 降级规则

### 名词介绍



- RT (平均响应时间, 秒级)

- 平均响应时间，超过阈值 且 时间窗口内通过的请求  $\geq 5$ ，两个条件同时满足后出发降级
  - 窗口期过后，关闭断路器
  - RT 最大 4900 (更大的需要通过 -Dcsp.sentinel.staticstic.max.rt=XXXXX 才能生效)
- 异常比例 (秒级)
    - QPA  $\geq 5$  且 异常比例 (秒级) 超过阈值时，触发降级；时间窗口结束后，关闭降级
  - 异常数 (分钟级)
    - 异常数 (分钟统计) 超过阈值时，触发降级，时间窗口结束后，关闭降级

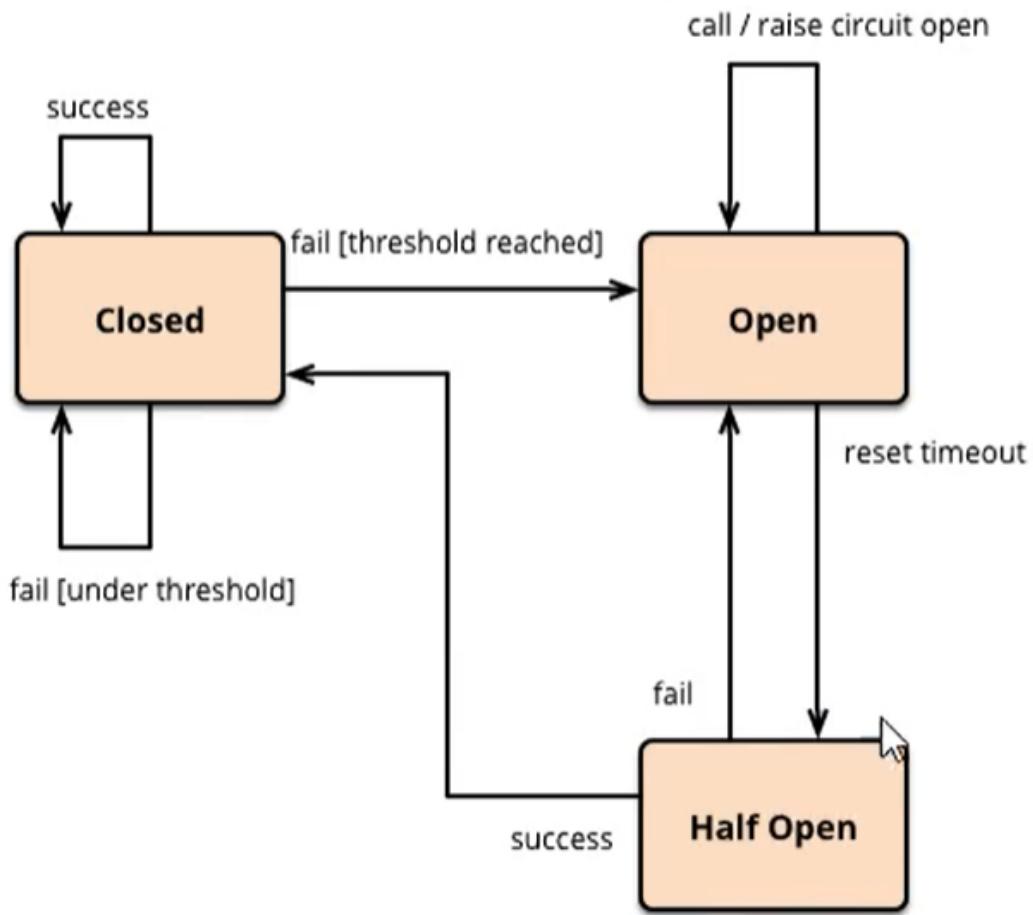
## 概念

Sentinel 熔断降级会在调用链路中某个资源出现不稳定状态时（例如调用超时或异常异常比例升高），对这个资源的调用进行限制，让请求快速失败，避免影响到其它的资源而导致级联错误。

当资源被降级后，在接下来的降级时间窗口之内，对该资源的调用都进行自动熔断（默认行为是抛出 `DegradeException`）

Sentinel 的断路器是没有半开状态

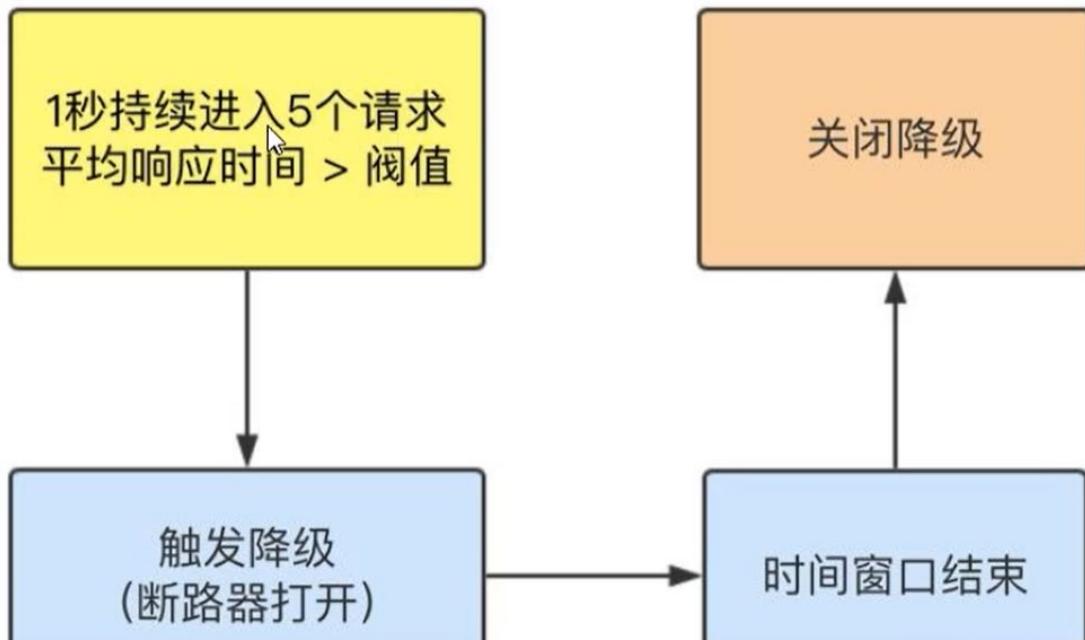
半开的状态，系统自动去检测是否请求有异常，没有异常就关闭断路器恢复使用，有异常则继续打开断路器不可用，具体可以参考 `hystrix`



## 降级策略实战

### RT

平均响应时间 (DEGRADE\_GRADE\_RT): 当 1s 内持续进入 N 个请求，对应时刻的平均响应时间（秒级）均超过阈值（count，以 ms 为单位），那么在接下的时间窗口（DegradeRule 中的 timeWindow，以 s 为单位）之内，对这个方法的调用都会自动地熔断（抛出 DegradeException）。注意 Sentinel 默认统计的 RT 上限是 4900 ms，超出此阈值的都会算作 **4900 ms**，若需要变更此上限可以通过启动配置项 - Dcsp.sentinel.statistic.max.rt=xxx 来配置。

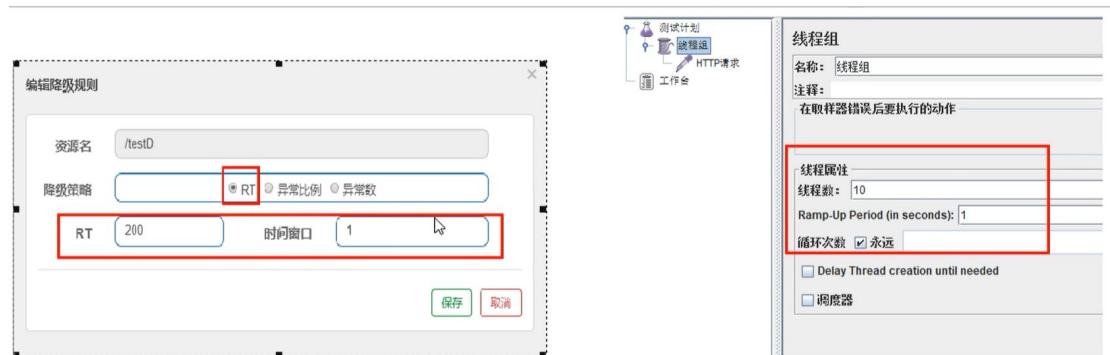


### 代码测试

```

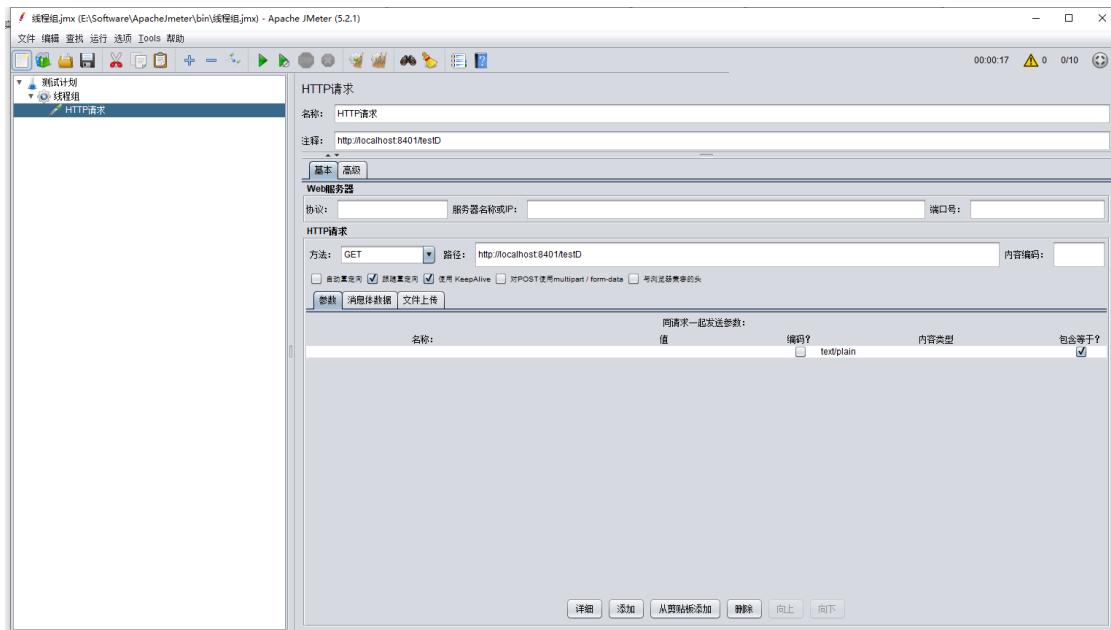
@GetMapping("/testD")
public String testD()
{
    try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e) { e.printStackTrace(); }
    log.info("testD 异常比例");
    return "-----testD";
}
  
```

然后使用 Jmeter 压力测试工具进行测试



按照上述操作，永远 1 秒种打进来 10 个线程，大于 5 个了，调用 `tesetD`，我们希望 200 毫秒内处理完本次任务，如果 200 毫秒没有处理完，在未来的 1 秒的时间窗口内，断路器打开（保险丝跳闸）微服务不可用，保险丝跳闸断电

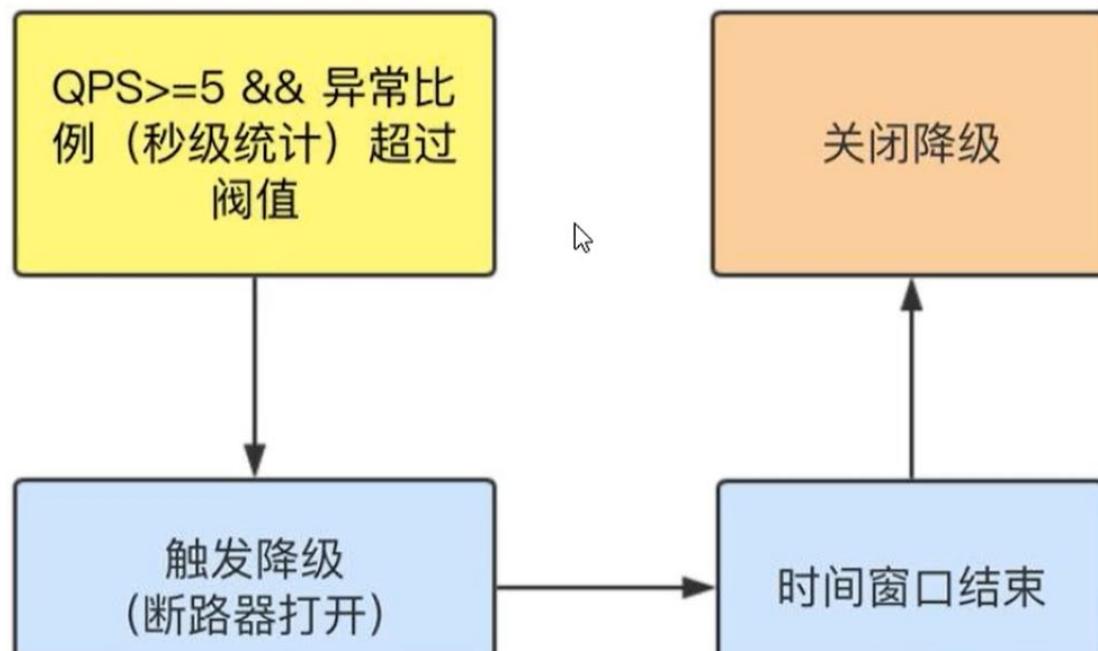
### Blocked by Sentinel (flow limiting)



后续我们停止使用 jmeter，没有那么大的访问量了，断路器关闭（保险丝恢复），微服务恢复 OK

### 异常比例

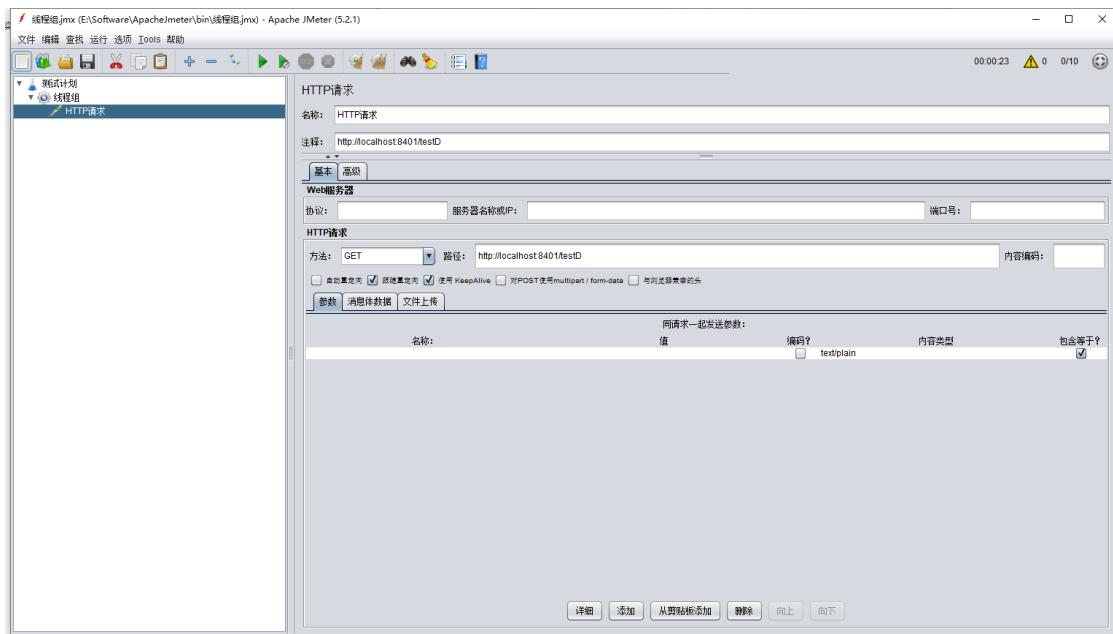
异常比例 (DEGRADE\_GRADE\_EXCEPTION\_RATIO): 当资源的每秒请求量  $\geq N$  (可配置)，并且每秒异常总数占通过量的比值超过阈值 (DegradeRule 中的 count) 之后，资源进入降级状态，即在接下的时间窗口 (DegradeRule 中的 timeWindow，以 s 为单位) 之内，对这个方法的调用都会自动地返回。异常比率的阈值范围是 [0.0, 1.0]，代表 0% - 100%。



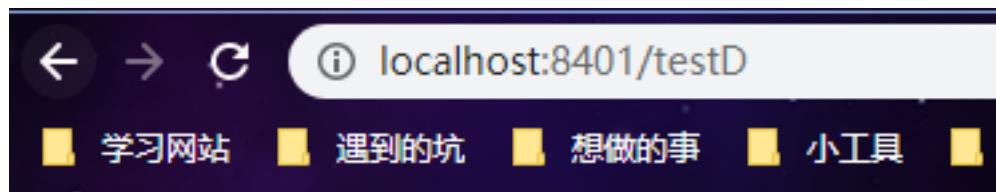
单独访问一次，必然来一次报错一次，开启 jmeter 后，直接高并发发送请求，多次调用达到我们的配置条件了，断路器开启（保险丝跳闸），微服务不可用，不在报错，而是服务降级了



设置 3 秒内，如果请求百分 50 出错，那么就会熔断



我们用 jmeter 每秒发送 10 次请求，3 秒后，再次调用 `localhost:8401/testD` 出现服务降级



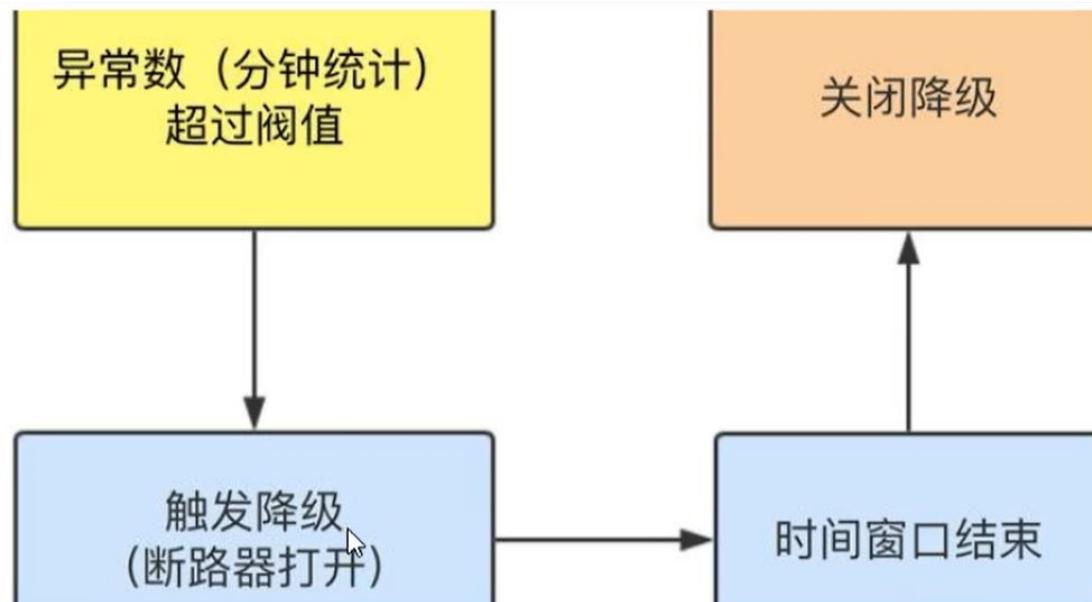
Blocked by Sentinel (flow limiting)

## 异常数

异常数 (DEGRADE\_GRADE\_EXCEPTION\_COUNT): 当资源近 1 分钟的异常数目超过阈值之后会进行熔断。注意由于统计时间窗口是分钟级别的，若 `timeWindow` 小于 60s，则结束熔断状态后仍可能再进入熔断状态

时间窗口一定要大于等于 60 秒

异常数是按分钟来统计的



下面设置是，一分钟内出现 5 次，则熔断



首先我们再次访问 <http://localhost:8401/testE>，第一次访问绝对报错，因为除数不能为 0，我们看到 error 窗口，但是达到 5 次报错后，进入熔断后的降级

## Sentinel 热点规则

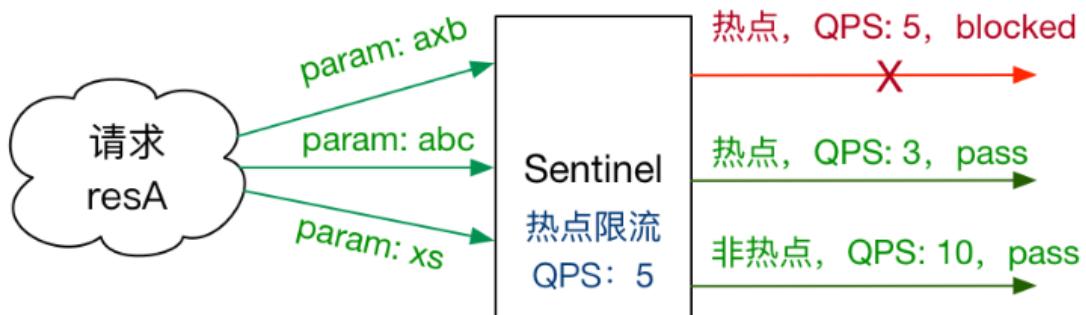
### 什么是热点数据

[Github 文档传送门](#)

何为热点？热点即经常访问的数据。很多时候我们希望统计某个热点数据中访问频次最高的 Top K 数据，并对其进行限制。比如：

- 商品 ID 为参数，统计一段时间内最常购买的商品 ID 并进行限制
- 用户 ID 为参数，针对一段时间内频繁访问的用户 ID 进行限制

热点参数限流会统计传入参数中的热点参数，并根据配置的限流阈值与模式，对包含热点参数的资源调用进行限流。热点参数限流可以看做是一种特殊的流量控制，仅对包含热点参数的资源调用生效。



Sentinel 利用 LRU 策略统计最近最常访问的热点参数，结合令牌桶算法来进行参数级别的流控。热点参数限流支持集群模式。

### 兜底的方法

分为系统默认的和客户自定义的，两种，之前的 case 中，限流出现问题了，都用 sentinel 系统默认的提示：Blocked By Sentinel，我们能不能自定义，类似于 hystrix，某个方法出现问题了，就找到对应的兜底降级方法。

从 `@HystrixCommand` 到 `@SentinelResource`

### 配置

`@SentinelResource` 的 value，就是我们的资源名，也就是对哪个方法配置热点规则

```
@GetMapping("/testHotKey")
@SentinelResource(value = "testHotKey",blockHandler = "deal_testHot
```

```

Key")
    public String testHotKey(@RequestParam(value = "p1", required = false) String p1,
    @RequestParam(value = "p2", required = false) String p2)
    {
        //int age = 10/0;
        return "-----testHotKey";
    }

    // 和上面的参数一样，不错需要加入 BlockException
    public String deal_testHotKey (String p1, String p2, BlockException exception)
    {
        return "-----deal_testHotKey,o(╥﹏╥)o"; // 兜底的方法
    }
}

```

我们对参数 0，设置热点 key 进行限流

The screenshot shows the Sentinel Control Console interface. On the left, there's a sidebar with navigation links like '实时监控', '热点白名单', '流控规则', '降级规则', '热点规则', '系统规则', '授权规则', '集群监控', and '机器列表'. The main area is titled 'cloudalibaba-sentinel-service' and shows a '热点参数限流规则' (Hotkey Parameter Flow Control Rule) configuration dialog. The dialog has fields for '资源名' (Resource Name) set to 'testHotKey', '限流模式' (Flow Control Mode) set to 'QPS 模式' (QPS Mode), '参数索引' (Parameter Index) set to '0' (highlighted with a red box and labeled '第一个参数' - first parameter), '单机阈值' (Single Machine Threshold) set to '1', and '统计窗口时长' (Statistical Window Duration) set to '1 秒' (1 second). A red arrow points to the '资源名' field. The background shows a list of rules with one entry: 'testHotKey'.

配置完成后

This screenshot shows the same 'cloudalibaba-sentinel-service' interface after configuration. The '热点参数限流规则' (Hotkey Parameter Flow Control Rule) section now lists the rule we just created: 'testHotKey'. The table includes columns for '资源名' (Resource Name), '参数索引' (Parameter Index), '流控模式' (Flow Control Mode), '阈值' (Threshold), '是否集群' (Is Clustered), '例外项数目' (Number of Exception Items), and '操作' (Operations). The '操作' column for this row contains '编辑' (Edit) and '删除' (Delete) buttons. A red arrow points to the '操作' column of the 'testHotKey' row.

当我们不断的请求时候，也就是以第一个参数为目标，请求接口，我们会发现多次请求后

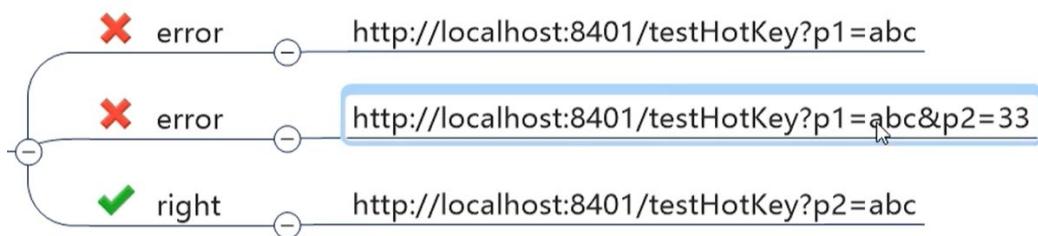
`http://localhost:8401/testHotKey?p1=a`

就会出现以下的兜底错误

`-----deal_testHotKey,o(╥_╥)o`

这是因为我们针对第一个参数进行了限制，当我们 QPS 超过 1 的时候，就会触发兜底的错误

假设我们请求的接口是：`http://localhost:8401/testHotKey?p2=a`，我们会发现他就没有进行限流



## 参数例外项

上述案例演示了第一个参数 `p1`，当 QPS 超过 1 秒 1 次点击狗，马上被限流

- 普通：超过一秒 1 个后，达到阈值 1 后马上被限流
- 我们期望 `p1` 参数当它达到某个特殊值时，它的限流值和平时不一样
- 特例：假设当 `p1` 的值等于 5 时，它的阈值可以达到 200
- 一句话说：当 key 为特殊值的时候，不被限制

平时的时候，参数 1 的 QPS 是 1，超过的时候被限流，但是有特殊值，比如 5，那么它的阈值就是 200

我们通过 `http://localhost:8401/testHotKey?p1=5` 一直刷新，发现不会触发兜底的方法，这就是参数例外项

热点参数的注意点，参数必须是基本类型或者 String

## 结语

`@SentinelResource` 处理的是 Sentinel 控制台配置的违规情况，有 `blockHandler` 方法配置的兜底处理

`RuntimeException`, 如 `int a = 10/0;` 这个是 java 运行时抛出的异常，`RuntimeException`, `@RentinelResource` 不管

也就是说：`@SentinelResource` 主管配置出错，运行出错不管。

如果想要有配置出错，和运行出错的话，那么可以设置 `fallback`

```

@GetMapping("/testHotKey")
@SentinelResource(value = "testHotKey",blockHandler = "deal_testHot
Key", fallback = "fallBack")
    public String testHotKey(@RequestParam(value = "p1",required = fals
e) String p1,
                           @RequestParam(value = "p2",required = fals
e) String p2)
{
    //int age = 10/0;
    return "-----testHotKey";
}

```

## Sentinel 系统配置

Sentinel 系统自适应限流从整体维度对应用入口流量进行控制，结合应用的 Load、CPU 使用率、总体平均 RT、入口 QPS 和并发线程数等几个维度的监控指标，通过自适应的流控策略，让系统的入口流量和系统的负载达到一个平衡，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

系统保护规则是从应用级别的入口流量进行控制，从单台机器的 load、CPU 使用率、平均 RT、入口 QPS 和并发线程数等几个维度监控应用指标，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

系统保护规则是应用整体维度的，而不是资源维度的，并且仅对入口流量生效。入口流量指的是进入应用的流量（`EntryType.IN`），比如 Web 服务或 Dubbo 服务端接收的请求，都属于入口流量。

系统规则支持以下的模式：

- **Load 自适应**（仅对 Linux/Unix-like 机器生效）：系统的 `load1` 作为启发指标，进行自适应系统保护。当系统 `load1` 超过设定的启发值，且系统当前的并发线程数超过估算的系统容量时才会触发系统保护（BBR 阶段）。系统容量由系统的 `maxQps * minRt` 估算得出。设定参考值一般是 CPU cores \* 2.5。
- **CPU usage**（1.5.0+ 版本）：当系统 CPU 使用率超过阈值即触发系统保护（取值范围 0.0-1.0），比较灵敏。
- **平均 RT**：当单台机器上所有入口流量的平均 RT 达到阈值即触发系统保护，单位是毫秒。
- **并发线程数**：当单台机器上所有入口流量的并发线程数达到阈值即触发系统保护。
- **入口 QPS**：当单台机器上所有入口流量的 QPS 达到阈值即触发系统保护。



这样相当于设置了全局的 QPS 过滤

## @SentinelResource 注解

- 按资源名称限流 + 后续处理
- 按 URL 地址限流 + 后续处理

## 问题

- 系统默认的，没有体现我们自己的业务要求
- 依照现有条件，我们自定义的处理方法又和业务代码耦合在一块，不直观
- 每个业务方法都添加一个兜底方法，那代码膨胀加剧
- 全局统一的处理方法没有体现
- 关闭 8401，发现流控规则已经消失，说明这个是没有持久化

## 客户自定义限流处理逻辑

创建 CustomerBlockHandler 类用于自定义限流处理逻辑

```
public class CustomerBlockHandler
{
    public static CommonResult handlerException(BlockException exception)
    {
        return new CommonResult(4444,"按客户自定义,global handlerException----1");
    }
    public static CommonResult handlerException2(BlockException exception)
    {
    }
```

```

        return new CommonResult(4444,"按客户自定义,global handlerExceptionOn----2");
    }
}

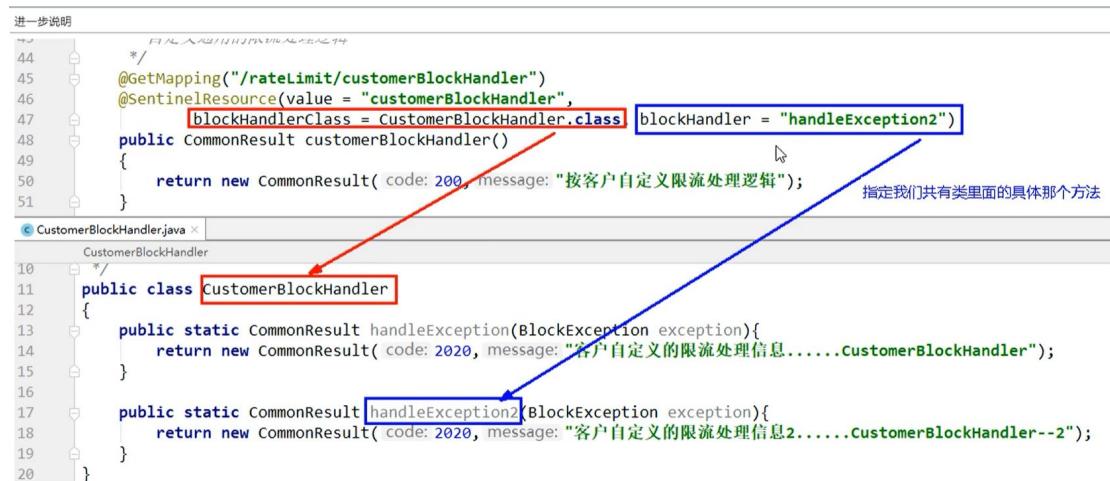
```

那么我们在使用的时候，就可以首先指定是哪个类，哪个方法

```

@GetMapping("/rateLimit/customerBlockHandler")
@SentinelResource(value = "customerBlockHandler",
    blockHandlerClass = CustomerBlockHandler.class,
    blockHandler = "handleException2")
public CommonResult customerBlockHandler()
{
    return new CommonResult(200,"按客户自定义",new Payment(2020L,"serial003"));
}

```



## 更多注解属性说明

所有的代码都要用 try - catch - finally 进行处理

sentinel 主要有三个核心 API

- Sphu 定义资源
- Tracer 定义统计
- ContextUtil 定义了上下文

## 服务熔断

sentinel 整合 Ribbon + openFeign + fallback

搭建 9003 和 9004 服务提供者

### 不设置任何参数

然后在使用 84 作为服务消费者，当我们值使用 `@SentinelResource` 注解时，不添加任何参数，那么如果出错的话，是直接返回一个 error 页面，对前端用户非常不友好，因此我们需要配置一个兜底的方法

```
@RequestMapping("/consumer/fallback/{id}")
@SentinelResource(value = "fallback") //没有配置
public CommonResult<Payment> fallback(@PathVariable Long id)
{
    CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL +
        "/paymentSQL/" + id, CommonResult.class, id);

    if (id == 4) {
        throw new IllegalArgumentException ("IllegalArgumentException,非法参数异常....");
    }else if (result.getData() == null) {
        throw new NullPointerException ("NullPointerException,该 ID
没有对应记录,空指针异常");
    }

    return result;
}
```

### 设置 fallback

```
@RequestMapping("/consumer/fallback/{id}")
@SentinelResource(value = "fallback", fallback = "handlerFallback")
//fallback 只负责业务异常
public CommonResult<Payment> fallback(@PathVariable Long id)
{
    CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL +
        "/paymentSQL/" + id, CommonResult.class, id);

    if (id == 4) {
        throw new IllegalArgumentException ("IllegalArgumentException,非法参数异常....");
    }else if (result.getData() == null) {
        throw new NullPointerException ("NullPointerException,该 ID
没有对应记录,空指针异常");
    }
}
```

```

        return result;
    }

    //本例是 fallback
    public CommonResult handlerFallback(@PathVariable Long id, Throwable e) {
        Payment payment = new Payment(id, "null");
        return new CommonResult<>(444, "兜底异常 handlerFallback,exception 内容 "+e.getMessage(),payment);
    }

```

加入 fallback 后，当我们程序运行出错时，我们会有一个兜底的异常执行，但是服务限流和熔断的异常还是出现默认的

## 设置 blockHandler

```

@RequestMapping("/consumer/fallback/{id}")
@SentinelResource(value = "fallback",blockHandler = "blockHandler" ,
fallback = "handlerFallback") //blockHandler 只负责 sentinel 控制台配置违规
public CommonResult<Payment> fallback(@PathVariable Long id)
{
    CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL +
"/paymentSQL/"+id,CommonResult.class,id);

    if (id == 4) {
        throw new IllegalArgumentException ("IllegalArgumentException,非法参数异常....");
    }else if (result.getData() == null) {
        throw new NullPointerException ("NullPointerException,该 ID 没有对应记录,空指针异常");
    }

    return result;
}

//本例是 blockHandler
public CommonResult blockHandler(@PathVariable Long id,BlockException blockException) {
    Payment payment = new Payment(id, "null");
    return new CommonResult<>(445,"blockHandler-sentinel 限流,无此流水: blockException "+blockException.getMessage(),payment);
}

```

## blockHandler 和 fallback 一起配置

```

@RequestMapping("/consumer/fallback/{id}")
@SentinelResource(value = "fallback",blockHandler = "blockHandler")
//blockHandler 只负责 sentinel 控制台配置违规

```

```

public CommonResult<Payment> fallback(@PathVariable Long id)
{
    CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL + "/paymentSQL/" + id, CommonResult.class, id);

    if (id == 4) {
        throw new IllegalArgumentException ("IllegalArgumentException,非法参数异常....");
    }else if (result.getData() == null) {
        throw new NullPointerException ("NullPointerException,该ID没有对应记录,空指针异常");
    }
    return result;
}

```

若 blockHandler 和 fallback 都进行了配置，则被限流降级而抛出 BlockException 时，只会进入 blockHandler 处理逻辑

## 异常忽略

```

@RequestMapping("/consumer/fallback/{id}")
@SentinelResource(value = "fallback", fallback = "handlerFallback", blockHandler = "blockHandler",
    exceptionsToIgnore = {IllegalArgumentException.class})
public CommonResult<Payment> fallback(@PathVariable Long id)
{
    CommonResult<Payment> result = restTemplate.getForObject(url: SERVICE_URL + "/paymentsSQL/" + id, CommonResult.class, id);
    if (id == 4) {
        throw new IllegalArgumentException ("非法参数异常....");
    }else if (result.getData() == null) {
        throw new NullPointerException ("NullPointerException,该ID没有对应记录");
    }
    return result;
}

public CommonResult handlerFallback(@PathVariable Long id, Throwable e) {
    Payment payment = new Payment(id, serial: "null");
    return new CommonResult<>{ code: 444, message: "fallback,无此流水,exception " + e.getMessage(), payment};
}

public CommonResult blockHandler(@PathVariable Long id, BlockException blockException) {
    Payment payment = new Payment(id, serial: "null");
    return new CommonResult<>{ code: 445, message: "blockHandler-sentinel限流,无此流水: blockException " + blockException.getMessage()};
}

```

## Feign 系列

### 引入依赖

```

<!--SpringCloud openfeign -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>

```

## 修改 YML

```
server:
  port: 84

spring:
  application:
    name: nacos-order-consumer
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
  sentinel:
    transport:
      #配置 Sentinel dashboard 地址
      dashboard: localhost:8080
      #默认 8719 端口, 假如被占用会自动从 8719 开始依次+1 扫描, 直至找到未被
      占用的端口
      port: 8719
```

#消费者将要去访问的微服务名称(注册成功进 nacos 的微服务提供者)

```
service-url:
  nacos-user-service: http://nacos-payment-provider
```

# 激活 Sentinel 对 Feign 的支持

```
feign:
  sentinel:
    enabled: true
```

## 启动类激活 Feign

```
@EnableDiscoveryClient
@SpringBootApplication
@EnableFeignClients
public class OrderNacosMain84
{
    public static void main(String[] args) {
        SpringApplication.run(OrderNacosMain84.class, args);
    }
}
```

## 引入 Feign 接口

```
@FeignClient(value = "nacos-payment-provider", fallback = PaymentFallbackService.class)
public interface PaymentService
{
    @GetMapping(value = "/paymentSQL/{id}")
    public CommonResult<Payment> paymentSQL(@PathVariable("id") Long id);
}
```

## 加入 fallback 兜底方法实现

```
@Component
public class PaymentFallbackService implements PaymentService
{
    @Override
    public CommonResult<Payment> paymentSQL(Long id)
    {
        return new CommonResult<>(44444,"服务降级返回,---PaymentFallback Service",new Payment(id,"errorSerial"));
    }
}
```

## 测试

请求接口: <http://localhost:84/consumer/paymentSQL/1>

测试 84 调用 9003, 此时故意关闭 9003 微服务提供者, 看 84 消费侧自动降级

我们发现过了一段时间后, 会触发服务降级, 返回失败的方法

## 熔断框架对比

	Sentinel	Hystrix	resilience4j
隔离策略	信号量隔离 (并发线程数限流)	线程池隔离/信号量隔离	信号量隔离
熔断降级策略	基于响应时间、异常比率、异常数	基于异常比率	基于异常比率、响应时间
实时统计实现	滑动窗口 (LeapArray)	滑动窗口 (基于 RxJava)	Ring Bit Buffer
动态规则配置	支持多种数据源	支持多种数据源	有限支持
扩展性	多个扩展点	插件的形式	接口的形式
基于注解的支持	支持	支持	支持
限流	基于 QPS, 支持基于调用关系的限流	有限的支持	Rate Limiter

## Sentinel 规则持久化

### 是什么

一旦我们重启应用，sentinel 规则将会消失，生产环境需要将规则进行持久化

### 怎么玩

将限流配置规则持久化进 Nacos 保存，只要刷新 8401 某个 rest 地址，sentinel 控制台的流控规则就能看到，只要 Nacos 里面的配置不删除，针对 8401 上的流控规则持续有效

### 解决方法

使用 nacos 持久化保存

#### 引入依赖

```
<!--SpringCloud alibaba sentinel-datasource-nacos 后续做持久化用到-->
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

#### 修改 yml

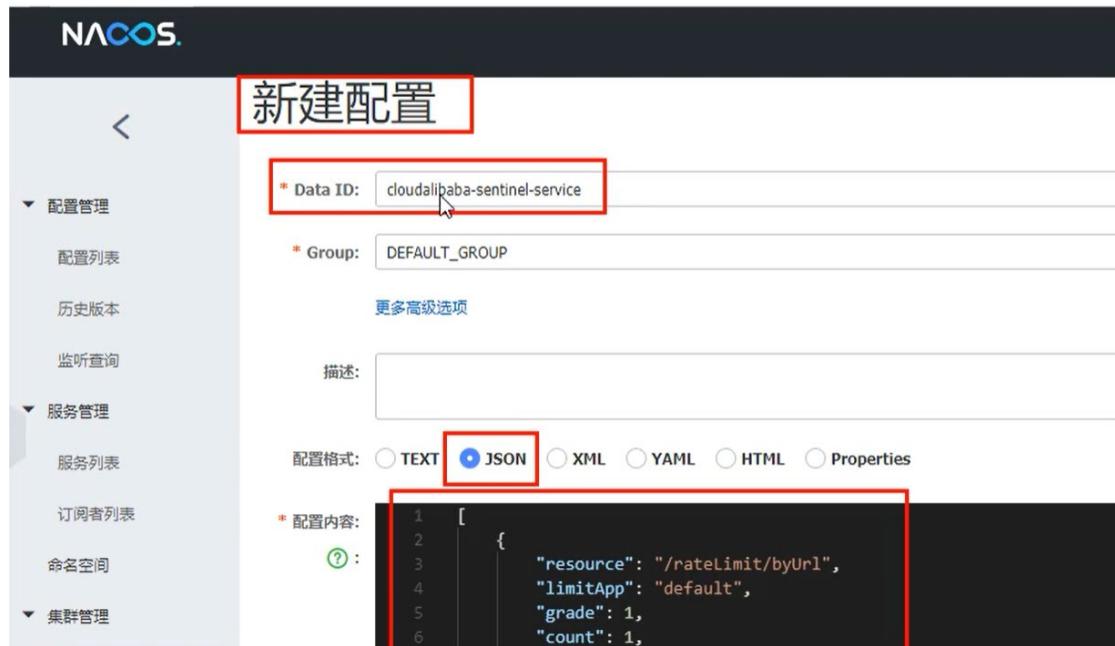
```
server:
  port: 8401

spring:
  application:
    name: cloudalibaba-sentinel-service
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #Nacos 服务注册中心地址
    sentinel:
      transport:
        dashboard: localhost:8080 #配置 Sentinel dashboard 地址
        port: 8719
    datasource:
      ds1:
        nacos:
          server-addr: localhost:8848
          dataId: cloudalibaba-sentinel-service
          groupId: DEFAULT_GROUP
          data-type: json
          rule-type: flow
```

```
management:
```

```
endpoints:  
  web:  
    exposure:  
      include: '*'  
  
feign:  
  sentinel:  
    enabled: true # 激活 Sentinel 对 Feign 的支持
```

### 添加 nacos 配置



The screenshot shows the Nacos configuration interface. On the left, there's a sidebar with navigation options: 配置管理 (Configuration Management), 历史版本 (History Version), 监听查询 (Listener Query), 服务管理 (Service Management), 服务列表 (Service List), 订阅者列表 (Subscriber List), 命名空间 (Namespace), and 集群管理 (Cluster Management). The main area is titled "新建配置" (New Configuration). It has fields for "Data ID" (set to "cloudalibaba-sentinel-service") and "Group" (set to "DEFAULT\_GROUP"). Below these are "更多高级选项" (More Advanced Options) and a "描述" (Description) field. Under "配置格式" (Configuration Format), "JSON" is selected. The "配置内容" (Configuration Content) field contains the following JSON code:

```
1 [  
2   {  
3     "resource": "/rateLimit/byUrl",  
4     "limitApp": "default",  
5     "grade": 1,  
6     "count": 1,
```

### 内容解析

```
[  
  {  
    "resource": "/rateLimit/byUrl",  
    "limitApp": "default",  
    "grade": 1,  
    "count": 1,  
    "strategy": 0,  
    "controlBehavior": 0,  
    "clusterMode": false  
  }  
]
```

- resource: 资源名称
- limitApp: 来源应用
- grade: 阈值类型, 0 表示线程数, 1 表示 QPS
- count: 单机阈值
- strategy: 流控模式, 0 表示直接, 1 表示关联, 2 表示链路
- controlBehavior: 流控效果, 0 表示快速失败, 1 表示 Warm, 2 表示排队等待

- clusterMode: 是否集群

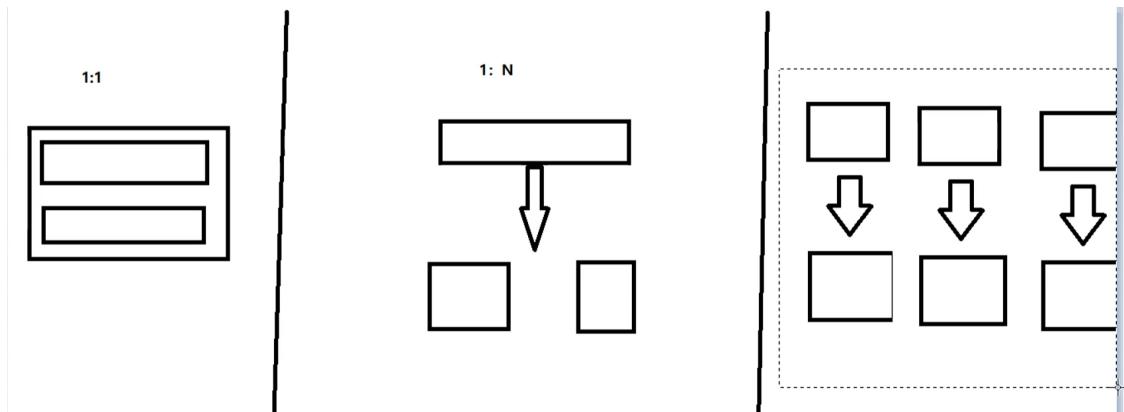
这样启动的时候，调用一下接口，我们的限流规则就会重新出现~

## SpringCloudAlibabaSeata 处理分布式事务

基于分布式的事务管理

### 分布式事务

分布式之前，单机单库没有这个问题，从 1:1 -> 1:N -> N:N



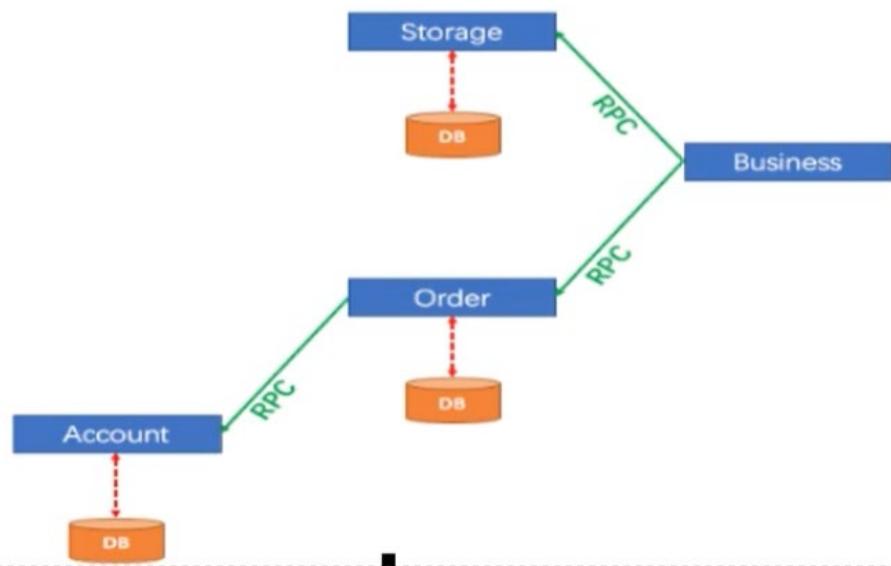
跨数据库，多数据源的统一调度，就会遇到分布式事务问题

如下图，单体应用被拆分成微服务应用，原来的三个模板被拆分成三个独立的应用，分别使用三个独立的数据源，业务操作需要调用三个服务来完成。此时每个服务内部的数据一致性由本地事务来保证，但是全局的数据一致性问题没法保证。

用户购买商品的业务逻辑。整个业务逻辑由3个微服务提供支持：

- 仓储服务：对给定的商品扣除仓储数量。
- 订单服务：根据采购需求创建订单。
- 帐户服务：从用户帐户中扣除余额。

## 架构图



## Seata 简介

官方文档：[点我传送](#)

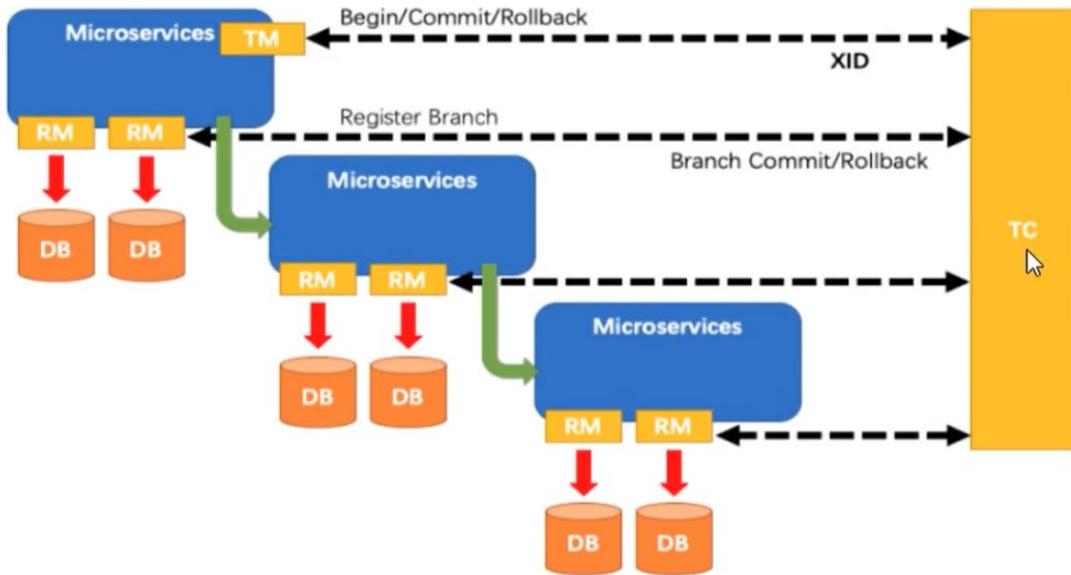
Seata 是一款开源的分布式事务解决方案，致力于提供高性能和简单易用的分布式事务服务。Seata 将为用户提供了 AT、TCC、SAGA 和 XA 事务模式，为用户打造一站式的分布式解决方案。

分布式事务处理过程的一致性 ID + 三组件模型

- Transaction ID XID：全局唯一的事务 ID
- 三组件的概念
  - Transaction Coordinator (TC)：事务协调器，维护全局事务，驱动全局事务提交或者回滚

- Transaction Manager (TM) : 事务管理器，控制全局事务的范围，开始全局事务提交或回滚全局事务
- Resource Manager (RM) : 资源管理器，控制分支事务，负责分支注册分支事务和报告

## 处理过程



- TM 向 TC 申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的 XID
- XID 在微服务调用链路的上下文中传播
- RM 向 TC 注册分支事务，将其纳入 XID 对应全局事务的管辖
- TM 向 TC 发起针对 XID 的全局提交或回滚决议
- TM 调度 XID 下管辖的全部分支事务完成提交或回滚请求

## 下载

地址: <https://github.com/seata/seata/releases>

下载 1.1 版本完成后，修改 conf 目录下的 file.conf 配置文件

### 修改 file.conf

首先我们需要备份原始的 file.conf 文件

主要修改，自定义事务组名称 + 事务日志存储模式为 db + 数据库连接信息，也就是修改存储的数据库

### 修改 service 模块

修改服务模块中的分组

```
service {  
  
    vgroup_mapping.my_test_tx_group = "fsp_tx_group"  
  
    default.grouplist = "127.0.0.1:8091"  
    enableDegrade = false  
    disable = false  
    max.commit.retry.timeout = "-1"  
    max.rollback.retry.timeout = "-1"  
}
```

### 修改 store 模块

修改存储模块

```
## transaction log store
store {
    ## store mode: file、db
    mode = "db"

    ## file store
    file {
        dir = "sessionStore"

        # branch session size , if exceeded first try compress lockkey, still exceeded throws exceptions
        max-branch-session-size = 16384
        # globe session size , if exceeded throws exceptions
        max-global-session-size = 512
        # file buffer size , if exceeded allocate new buffer
        file-write-buffer-cache-size = 16384
        # when recover batch read size
        session.reload.read_size = 100
        # async, sync
        flush-disk-mode = async
```

```
}

## database store
db {
    ## the implement of javax.sql.DataSource, such as DruidDataSource(druid)/BasicDataSource(dbcp) etc.
    datasource = "dbcp"
    ## mysql/oracle/h2/oceanbase etc.
    db-type = "mysql"
    driver-class-name = "com.mysql.jdbc.Driver"
    url = "jdbc:mysql://127.0.0.1:3306/seata"
    user = "root"
    password = "你自己密码"
    min-conn = 1
    max-conn = 3
    global.table = "global_table"
    branch.table = "branch_table"
    lock-table = "lock_table"
    query-limit = 100
}
```

## 创建一个 seata 数据库

在 seata 数据库中建表，建表语句在 seata/conf 目录下的 db\_store.sql

## 修改 seata-server 的 registry.conf 配置文件

---

```
registry {  
    # file、nacos、eureka、redis、zk、consul、etcd3、sofa  
    type = "nacos"  
  
    nacos {  
        serverAddr = "localhost:8848"  
        namespace = ""  
        cluster = "default"  
    }  
}
```

目的是：指明注册中心为 nacos，及修改 nacos 连接信息

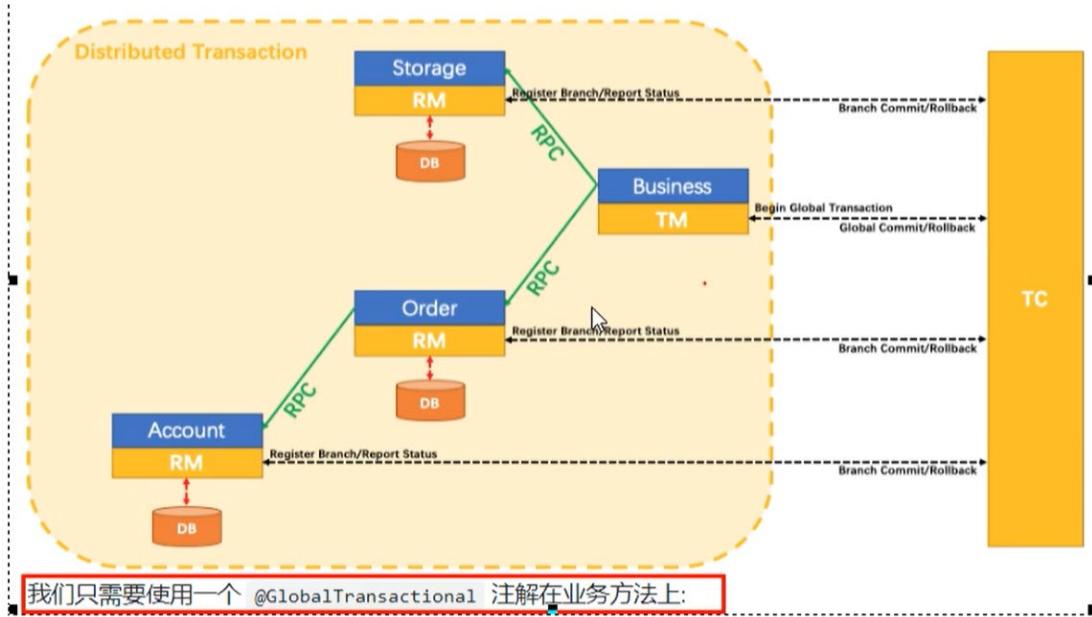
然后启动 nacos 和 seata-server

## 怎么玩

- 本地：@Transaction
- 全局：@GlobalTransaction

Spring 自带的是 @Transaction 控制本地事务

而 @GlobalTransaction 控制的是全局事务



我们只需要在需要支持分布式事务的业务类上，使用该注解即可

## 订单/库存/账户业务微服务准备

在这之前首先需要先启动 Nacos，然后启动 Seata，保证两个都 OK

### 分布式事务的业务说明

这里我们会创建三个微服务，一个订单服务，一个库存服务，一个账户服务。

当用户下单时，会在订单服务中创建一个订单，然后通过远程调用库存服务来扣减下单商品的库存，在通过远程调用账户服务来扣减用户账户里面的金额，最后在订单服务修改订单状态为已完成

该操作跨越了三个数据库，有两次远程调用，很明显会有分布式事务的问题。

一句话：下订单 -> 扣库存 -> 减余额

### 创建数据库

- seata\_order: 存储订单的数据库
- seata\_storage: 存储库存的数据库
- seata\_account: 存储账户信息的数据库

建库 SQL

```
create database seata_order;
create database seata_storage;
create database seata_account;
```

## 建立业务表

- seataorder 库下建立 torder 表
- seatastorage 库下建 tstorage 表
- seataaccount 库下建 taccount 表

```
DROP TABLE IF EXISTS `t_order`;
CREATE TABLE `t_order` (
  `int` bigint(11) NOT NULL AUTO_INCREMENT,
  `user_id` bigint(20) DEFAULT NULL COMMENT '用户 id',
  `product_id` bigint(11) DEFAULT NULL COMMENT '产品 id',
  `count` int(11) DEFAULT NULL COMMENT '数量',
  `money` decimal(11, 0) DEFAULT NULL COMMENT '金额',
  `status` int(1) DEFAULT NULL COMMENT '订单状态: 0:创建中 1:已完结',
  PRIMARY KEY (`int`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci COMMENT
T = '订单表' ROW_FORMAT = Dynamic;
```

```
DROP TABLE IF EXISTS `t_storage`;
CREATE TABLE `t_storage` (
  `int` bigint(11) NOT NULL AUTO_INCREMENT,
  `product_id` bigint(11) DEFAULT NULL COMMENT '产品 id',
  `total` int(11) DEFAULT NULL COMMENT '总库存',
  `used` int(11) DEFAULT NULL COMMENT '已用库存',
  `residue` int(11) DEFAULT NULL COMMENT '剩余库存',
  PRIMARY KEY (`int`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci COMMENT
T = '库存' ROW_FORMAT = Dynamic;
INSERT INTO `t_storage` VALUES (1, 1, 100, 0, 100);
```

```
CREATE TABLE `t_account` (
  `id` bigint(11) NOT NULL COMMENT 'id',
  `user_id` bigint(11) DEFAULT NULL COMMENT '用户 id',
  `total` decimal(10, 0) DEFAULT NULL COMMENT '总额度',
  `used` decimal(10, 0) DEFAULT NULL COMMENT '已用余额',
  `residue` decimal(10, 0) DEFAULT NULL COMMENT '剩余可用额度',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci COMMENT
T = '账户表' ROW_FORMAT = Dynamic;
```

```
INSERT INTO `t_account` VALUES (1, 1, 1000, 0, 1000);
```

## 创建回滚日志表

订单 - 库存 - 账户 3 个库都需要建各自的回滚日志表，目录在 dbundolog.sql

```
-- the table to store seata xid data
-- 0.7.0+ add context
-- you must to init this sql for your business database. the seata server not need it.
-- 此脚本必须初始化在你当前的业务数据库中，用于 AT 模式 XID 记录。与 server 端无关（注：业务数据库）
-- 注意此处 0.3.0+ 增加唯一索引 ux_undo_log
DROP TABLE `undo_log`;
CREATE TABLE `undo_log` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
  `branch_id` BIGINT(20) NOT NULL,
  `xid` VARCHAR(100) NOT NULL,
  `context` VARCHAR(128) NOT NULL,
  `rollback_info` LONGBLOB NOT NULL,
  `log_status` INT(11) NOT NULL,
  `log_created` DATETIME NOT NULL,
  `log_modified` DATETIME NOT NULL,
  `ext` VARCHAR(100) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`, `branch_id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

## 订单/库存/账户业务微服务准备

### 业务需求

下订单 -> 减库存 -> 扣余额 -> 改（订单）状态

### 新建 Order-Module 表

#### 约定

entity, domain: 相当于实体类层

vo: view object, value object

dto: 前台传到后台的数据传输类

### 新建 module2001

#### 引入 POM

```
<!--seata-->
<dependency>
```

```

<groupId>com.alibaba.cloud</groupId>
<artifactId>spring-cloud-starter-alibaba-seata</artifactId>
<exclusions>
    <exclusion>
        <artifactId>seata-all</artifactId>
        <groupId>io.seata</groupId>
    </exclusion>
</exclusions>
</dependency>
<dependency>
    <groupId>io.seata</groupId>
    <artifactId>seata-all</artifactId>
    <version>0.9.0</version>
</dependency>

```

## 修改 yml

```

server:
  port: 2001

spring:
  application:
    name: seata-order-service
  cloud:
    alibaba:
      seata:
        #自定义事务组名称需要与 seata-server 中的对应
        tx-service-group: fsp_tx_group
  nacos:
    discovery:
      server-addr: localhost:8848
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/seata_order
    username: root
    password: 123456

  feign:
    hystrix:
      enabled: false

  logging:
    level:
      io:
        seata: info

  mybatis:
    mapperLocations: classpath:mapper/*.xml

```

## 增加 file.conf

在 resources 目录下，创建 file.conf 文件

```
transport {
    # tcp udt unix-domain-socket
    type = "TCP"
    #NIO NATIVE
    server = "NIO"
    #enable heartbeat
    heartbeat = true
    #thread factory for netty
    thread-factory {
        boss-thread-prefix = "NettyBoss"
        worker-thread-prefix = "NettyServerNIOWorker"
        server-executor-thread-prefix = "NettyServerBizHandler"
        share-boss-worker = false
        client-selector-thread-prefix = "NettyClientSelector"
        client-selector-thread-size = 1
        client-worker-thread-prefix = "NettyClientWorkerThread"
        # netty boss thread size,will not be used for UDT
        boss-thread-size = 1
        #auto default pin or 8
        worker-thread-size = 8
    }
    shutdown {
        # when destroy server, wait seconds
        wait = 3
    }
    serialization = "seata"
    compressor = "none"
}

service {

    vgroup_mapping.fsp_tx_group = "default" #修改自定义事务组名称

    default.grouplist = "127.0.0.1:8091"
    enableDegrade = false
    disable = false
    max.commit.retry.timeout = "-1"
    max.rollback.retry.timeout = "-1"
    disableGlobalTransaction = false
}

client {
    async.commit.buffer.limit = 10000
    lock {
```

```

    retry.internal = 10
    retry.times = 30
}
report.retry.count = 5
tm.commit.retry.count = 1
tm.rollback.retry.count = 1
}

## transaction log store
store {
    ## store mode: file、db
    mode = "db"

    ## file store
    file {
        dir = "sessionStore"

        # branch session size , if exceeded first try compress lockkey, sti
        ll exceeded throws exceptions
        max-branch-session-size = 16384
        # globe session size , if exceeded throws exceptions
        max-global-session-size = 512
        # file buffer size , if exceeded allocate new buffer
        file-write-buffer-cache-size = 16384
        # when recover batch read size
        session.reload.read_size = 100
        # async, sync
        flush-disk-mode = async
    }
}

## database store
db {
    ## the implement of javax.sql.DataSource, such as DruidDataSource(d
    ruid)/BasicDataSource(dbcp) etc.
    datasource = "dbcp"
    ## mysql/oracle/h2/oceanbase etc.
    db-type = "mysql"
    driver-class-name = "com.mysql.jdbc.Driver"
    url = "jdbc:mysql://127.0.0.1:3306/seata"
    user = "root"
    password = "123456"
    min-conn = 1
    max-conn = 3
    global.table = "global_table"
    branch.table = "branch_table"
    lock-table = "lock_table"
    query-limit = 100
}
}

```

```

lock {
    ## the lock store mode: local、remote
    mode = "remote"

    local {
        ## store locks in user's database
    }

    remote {
        ## store locks in the seata's server
    }
}

recovery {
    #schedule committing retry period in milliseconds
    committing-retry-period = 1000
    #schedule asyn committing retry period in milliseconds
    asyn-committing-retry-period = 1000
    #schedule rollbacks retry period in milliseconds
    rollbacks-retry-period = 1000
    #schedule timeout retry period in milliseconds
    timeout-retry-period = 1000
}

transaction {
    undo.data.validation = true
    undo.log.serialization = "jackson"
    undo.log.save.days = 7
    #schedule delete expired undo_log in milliseconds
    undo.log.delete.period = 86400000
    undo.log.table = "undo_log"
}

## metrics settings
metrics {
    enabled = false
    registry-type = "compact"
    # multi exporters use comma divided
    exporter-list = "prometheus"
    exporter-prometheus-port = 9898
}

support {
    ## spring
    spring {
        # auto proxy the DataSource bean
        datasource.autoproxy = false
    }
}

```

## registry.conf 注册器

```
registry {
    # file 、 nacos 、 eureka、 redis、 zk、 consul、 etcd3、 sofa
    type = "nacos"

    nacos {
        serverAddr = "localhost:8848"
        namespace = ""
        cluster = "default"
    }
    eureka {
        serviceUrl = "http://localhost:8761/eureka"
        application = "default"
        weight = "1"
    }
    redis {
        serverAddr = "localhost:6379"
        db = "0"
    }
    zk {
        cluster = "default"
        serverAddr = "127.0.0.1:2181"
        session.timeout = 6000
        connect.timeout = 2000
    }
    consul {
        cluster = "default"
        serverAddr = "127.0.0.1:8500"
    }
    etcd3 {
        cluster = "default"
        serverAddr = "http://localhost:2379"
    }
    sofa {
        serverAddr = "127.0.0.1:9603"
        application = "default"
        region = "DEFAULT_ZONE"
        datacenter = "DefaultDataCenter"
        cluster = "default"
        group = "SEATA_GROUP"
        addressWaitTime = "3000"
    }
    file {
        name = "file.conf"
    }
}

config {
    # file、 nacos 、 apollo、 zk、 consul、 etcd3
```

```

type = "file"

nacos {
    serverAddr = "localhost"
    namespace = ""
}
consul {
    serverAddr = "127.0.0.1:8500"
}
apollo {
    app.id = "seata-server"
    apollo.meta = "http://192.168.1.204:8801"
}
zk {
    serverAddr = "127.0.0.1:2181"
    session.timeout = 6000
    connect.timeout = 2000
}
etcd3 {
    serverAddr = "http://localhost:2379"
}
file {
    name = "file.conf"
}
}

```

### domain

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class CommonResult<T>
{
    private Integer code;
    private String message;
    private T data;

    public CommonResult(Integer code, String message)
    {
        this(code,message,null);
    }
}

```

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Order
{
    private Long id;

```

```

private Long userId;

private Long productId;

private Integer count;

private BigDecimal money;

private Integer status; //订单状态: 0: 创建中; 1: 已完结
}

```

## Dao 接口及实现

```

@Mapper
public interface OrderDao
{
    //1 新建订单
    void create(Order order);

    //2 修改订单状态, 从零改为1
    void update(@Param("userId") Long userId,@Param("status") Integer s
tus);
}

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://my
batis.org/dtd/mybatis-3-mapper.dtd" >

<mapper namespace="com.atguigu.springcloud.alibaba.dao.OrderDao">

    <resultMap id="BaseResultMap" type="com.atguigu.springcloud.alibaba.
domain.Order">
        <id column="id" property="id" jdbcType="BIGINT"/>
        <result column="user_id" property="userId" jdbcType="BIGINT"/>
        <result column="product_id" property="productId" jdbcType="BIGI
NT"/>
        <result column="count" property="count" jdbcType="INTEGER"/>
        <result column="money" property="money" jdbcType="DECIMAL"/>
        <result column="status" property="status" jdbcType="INTEGER"/>
    </resultMap>

    <insert id="create">
        insert into t_order (id,user_id,product_id,count,money,status)
        values (null,#{userId},#{productId},#{count},#{money},0);
    </insert>

    <update id="update">
        update t_order set status = 1
    </update>

```

```
        where user_id=#{userId} and status = #{status};  
    </update>  
  
</mapper>
```

## Service 实现类

OrderService 接口

```
public interface OrderService  
{  
    void create(Order order);  
}
```

StorageService 的 Feign 接口，

```
@FeignClient(value = "seata-storage-service")  
public interface StorageService  
{  
    @PostMapping(value = "/storage/decrease")  
    CommonResult decrease(@RequestParam("productId") Long productId, @R  
equestParam("count") Integer count);  
}
```

AccountService 的 Feign 接口，账户接口

```
@FeignClient(value = "seata-account-service")  
public interface AccountService  
{  
    @PostMapping(value = "/account/decrease")  
    CommonResult decrease(@RequestParam("userId") Long userId, @Request  
Param("money") BigDecimal money);  
}
```

## OrderServiceImpl 实现类

```
@Service  
@Slf4j  
public class OrderServiceImpl implements OrderService  
{  
    @Resource  
    private OrderDao orderDao;  
    @Resource  
    private StorageService storageService;  
    @Resource  
    private AccountService accountService;  
  
    /**  
     * 创建订单->调用库存服务扣减库存->调用账户服务扣减账户余额->修改订单状态
```

```

    * 简单说：下订单->扣库存->减余额->改状态
    */
@Override
@GlobalTransactional(name = "fsp-create-order", rollbackFor = Exception.class)
public void create(Order order)
{
    log.info("----->开始新建订单");
    //1 新建订单
    orderDao.create(order);

    //2 扣减库存
    log.info("----->订单微服务开始调用库存， 做扣减 Count");
    storageService.decrease(order.getProductId(),order.getCount());
    log.info("----->订单微服务开始调用库存， 做扣减 end");

    //3 扣减账户
    log.info("----->订单微服务开始调用账户， 做扣减 Money");
    accountService.decrease(order.getUserId(),order.getMoney());
    log.info("----->订单微服务开始调用账户， 做扣减 end");

    //4 修改订单状态， 从零到 1,1 代表已经完成
    log.info("----->修改订单状态开始");
    orderDao.update(order.getUserId(),0);
    log.info("----->修改订单状态结束");

    log.info("----->下订单结束了， O(n_n)O 哈哈~");
}

}

```

## 业务类

```

@RestController
public class OrderController
{
    @Resource
    private OrderService orderService;

    @GetMapping("/order/create")
    public CommonResult create(Order order)
    {
        orderService.create(order);
        return new CommonResult(200,"订单创建成功");
    }
}

```

## Config 配置

Mybatis DataSourceProxyConfig 配置，这里是使用 Seata 对数据源进行代理

```
@Configuration
public class DataSourceProxyConfig {

    @Value("${mybatis.mapperLocations}")
    private String mapperLocations;

    @Bean
    @ConfigurationProperties(prefix = "spring.datasource")
    public DataSource druidDataSource(){
        return new DruidDataSource();
    }

    @Bean
    public DataSourceProxy dataSourceProxy(DataSource dataSource) {
        return new DataSourceProxy(dataSource);
    }

    @Bean
    public SqlSessionFactory sqlSessionFactoryBean(DataSourceProxy data
SourceProxy) throws Exception {
        SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFac
toryBean();
        sqlSessionFactoryBean.setDataSource(dataSourceProxy);
        sqlSessionFactoryBean.setMapperLocations(new PathMatchingResour
cePatternResolver().getResources(mapperLocations));
        sqlSessionFactoryBean.setTransactionFactory(new SpringManagedTr
ansactionFactory());
        return sqlSessionFactoryBean.getObject();
    }

}
```

Mybatis 配置

```
@Configuration
@MapperScan({"com.atguigu.springcloud.alibaba.dao"})
public class MyBatisConfig {
```

## 启动类

```
@EnableDiscoveryClient
@EnableFeignClients
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)//取
消数据源的自动创建
public class SeataOrderMainApp2001
```

```
{  
    public static void main(String[] args)  
    {  
        SpringApplication.run(SeataOrderMainApp2001.class, args);  
    }  
}
```

## 新建 Storage-Module

参考项目：seata-storage-service2002

## 新建账户 Account-Module

参考项目：seata-account-service2003

## 测试

### 数据库初始情况

```
SELECT * FROM `seata_order`.`t_order`
```

1 结果						2 个配置文件	3 信息	4 表数据	5 信息	
id	user_id	product_id	count	money	status					

```
SELECT * FROM `seata_storage`.`t_storage`
```

1 结果					2 个配置文件	3 信息	4 表数据	5 信息	
id	product_id	total	used	residue					
1	1	100	0	100					

```
SELECT * FROM `seata_account`.`t_account`;
```

```
8     SELECT * FROM `seata_account`.`t_account`;
```

1 结果					2 个配置文件	3 信息	4 表数据	5 信息	
id	user_id	total	used	residue					
1	1	1000	0	1000					

正常下单

访问

<http://localhost:2001/order/create?userId=1&productId=1&count=10&money=100>

9    SELECT \* FROM seata\_order.`t\_order`;

1 结果 2 个配置文件 3 信息 4 表数据 5 信息  
(只读) | (只读) | (只读) | (只读) | (只读) | (只读) | (只读)  
id user\_id product\_id count money status  
51 1 1 10 100 1

8    SELECT \* FROM `seata\_storage`.'t\_storage`

1 结果 2 个配置文件 3 信息 4 表数据 5 信息  
(只读) | (只读) | (只读) | (只读) | (只读) | (只读) | (只读)  
id product\_id total used residue  
1 1 100 10 90

SELECT \* FROM `seata\_account`.'t\_account`;

1 结果 2 个配置文件 3 信息 4 表数据 5 信息  
(只读) | (只读) | (只读) | (只读) | (只读) | (只读) | (只读)  
id user\_id total used residue  
1 1 1000 100 900

超时异常，没加@GlobalTransaction

我们在 account-module 模块，添加睡眠时间 20 秒，因为 openFeign 默认时间是 1 秒

```
/**  
 * 扣减账户余额  
 */  
@Override  
public void decrease(Long userId, BigDecimal money) {  
    LOGGER.info("----->account-service中扣减账户余额开始");  
    //模拟超时异常，全局事务回滚  
    //暂停几秒钟线程  
    try { TimeUnit.SECONDS.sleep( timeout: 20); } catch (InterruptedException e) { e.printStackTrace(); }  
    accountDao.decrease(userId,money);  
    LOGGER.info("----->account-service中扣减账户余额结束");  
}
```

出现了数据不一致的问题

故障情况

- 当库存和账户金额扣减后，订单状态并没有设置成已经完成，没有从零改成 1
- 而且由于 Feign 的重试机制，账户余额还有可能被多次扣除

超时异常，添加@GlobalTransaction

```
@GlobalTransactional(name = "fsp-create-order", rollbackFor = Exception.class)
```

rollbackFor 表示，什么错误就会回滚

添加这个后，发现下单后的数据库并没有改变，记录都添加不进来

## 一部分补充

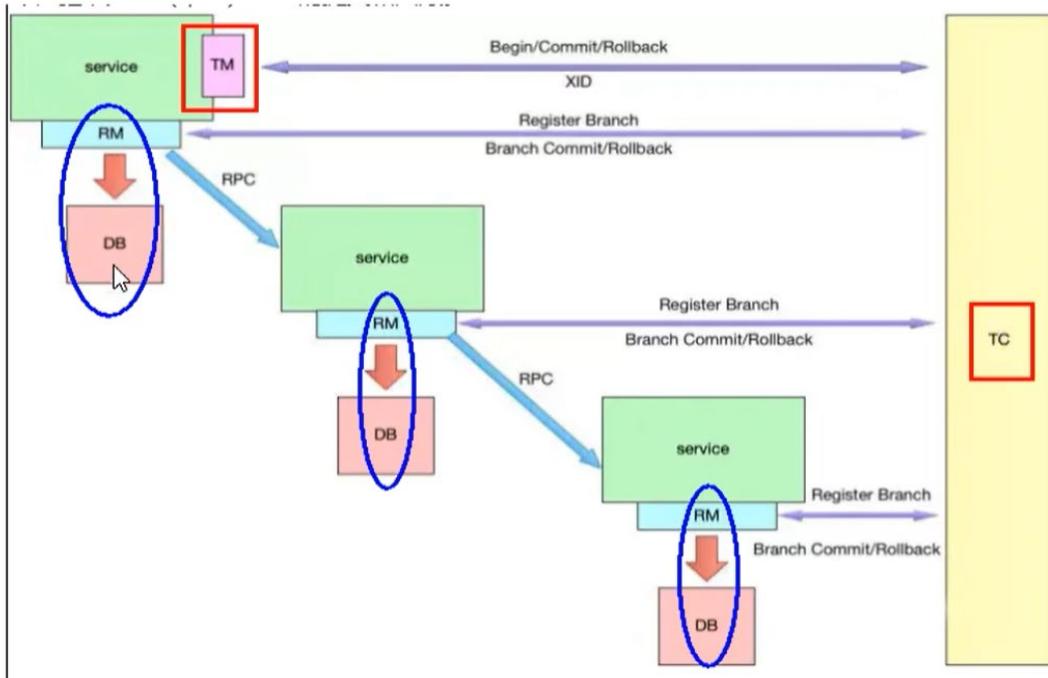
### Seata

2019 年 1 月份，蚂蚁金服和阿里巴巴共同开源的分布式事务解决方案

Seata: Simple Extensible Autonomous Transaction Architecture，简单可扩展自治事务框架

2020 起始，参加工作以后用 1.0 以后的版本。

## 再看 TC/TM/RM 三大组件

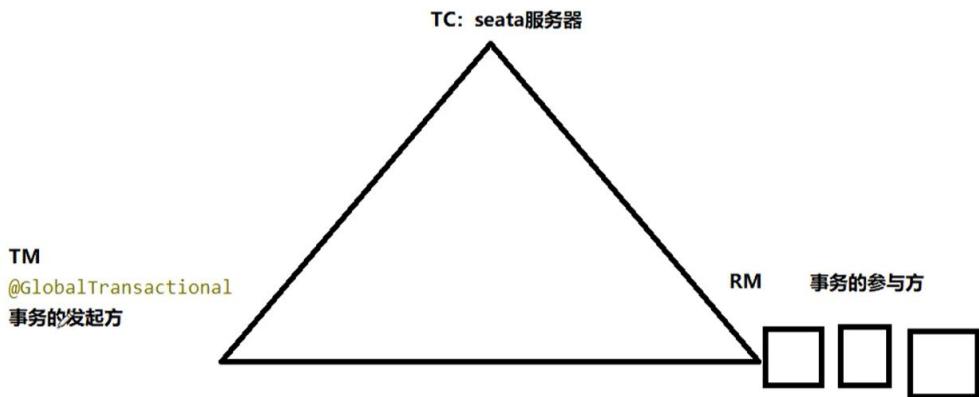


什么是 TC, TM, RM

TC: seata 服务器

TM: 带有@GlobalTransactional 注解的方法

RM: 数据库, 也就是事务参与方



## 分布式事务的执行流程

- TM 开启分布式事务（TM 向 TC 注册全局事务记录），相当于注解 `@GlobalTransaction` 注解
- 按业务场景，编排数据库，服务等事务内部资源（RM 向 TC 汇报资源准备状态）
- TM 结束分布式事务，事务一阶段结束（TM 通知 TC 提交、回滚分布式事务）
- TC 汇总事务信息，决定分布式事务是提交还是回滚
- TC 通知所有 RM 提交、回滚资源，事务二阶段结束

## AT 模式如何做到对业务的无侵入

默认 AT 模式，阿里云 GTS

### AT 模式

#### 前提

- 基于支持本地 ACID 事务的关系型数据库
- Java 应用，通过 JDBC 访问数据库

#### 整体机制

两阶段提交协议的演变

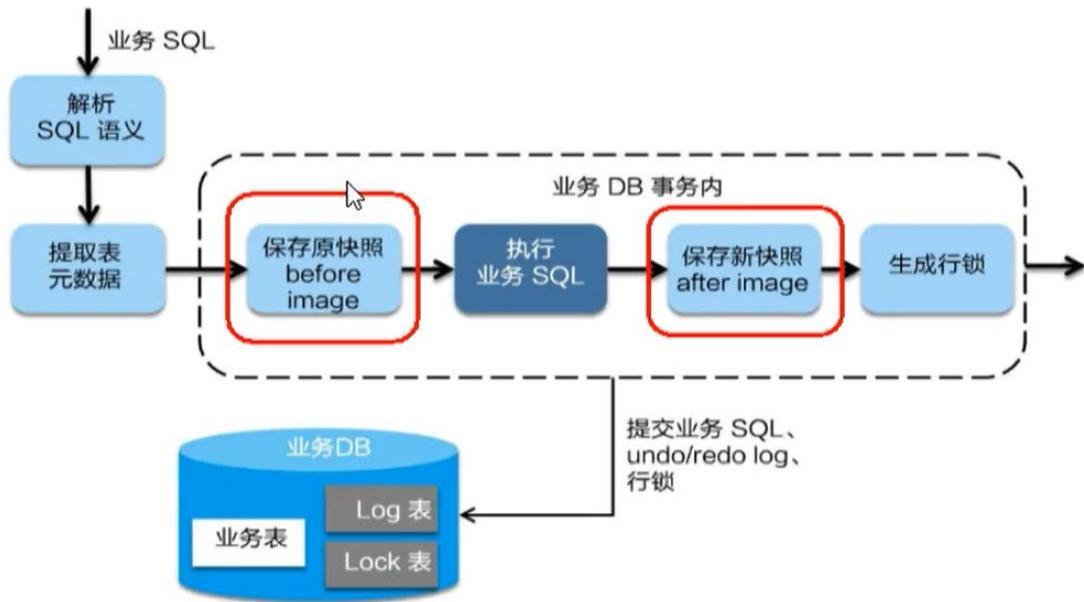
- 一阶段：业务数据和回滚日志记录在同一个本地事务中提交，释放本地锁和连接资源
- 二阶段
  - 提交异步化，非常快速的完成
  - 回滚通过一阶段的回滚日志进行反向补偿

#### 一阶段加载

在一阶段，Seata 会拦截业务 SQL

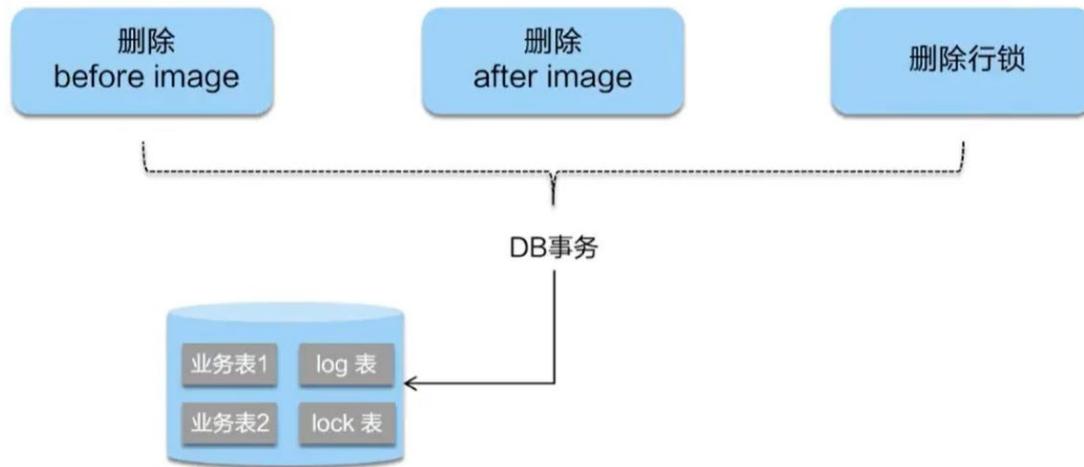
- 解析 SQL 语义，找到业务 SQL，要更新的业务数据，在业务数据被更新前，将其保存成 `before image`（前置镜像）
- 执行业务 SQL 更新业务数据，在业务数据更新之后
- 将其保存成 `after image`，最后生成行锁

以上操作全部在一个数据库事务内完成，这样保证了一阶段操作的原子性



## 二阶段提交

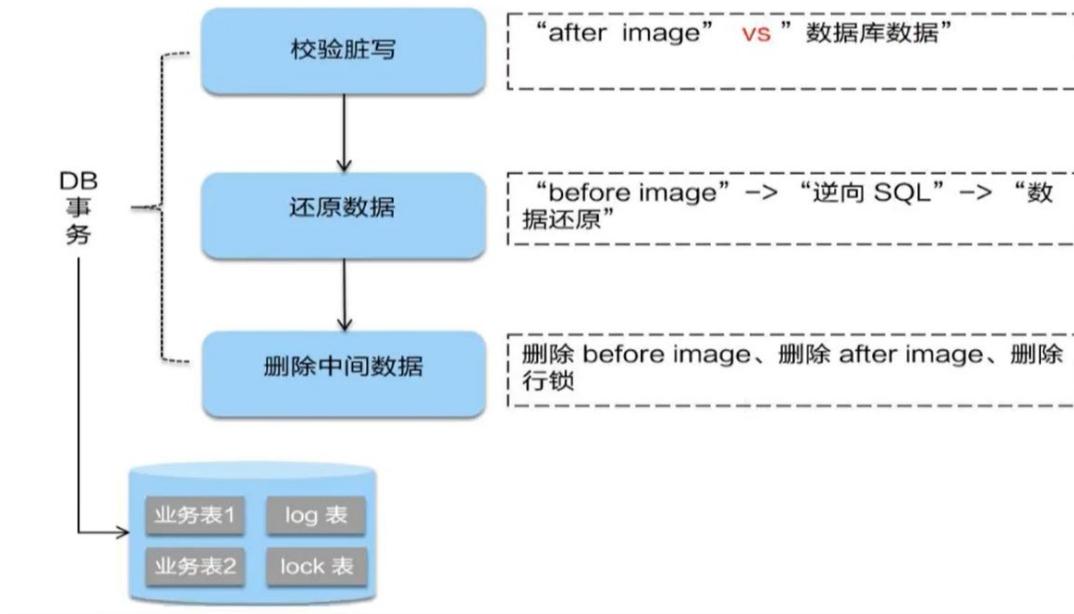
二阶段如果顺利提交的话，因为业务 SQL 在一阶段已经提交至数据库，所以 Seata 框架只需将一阶段保存的快照和行锁删除掉，完成数据清理即可



## 二阶段回滚

二阶段如果回滚的话，Seata 就需要回滚到一阶段已经执行的业务 SQL，还原业务数据

回滚方式便是用 before image 还原业务数据，但是在还原前要首先校验脏写，对比数据库当前业务数据和 after image，如果两份数据完全一致，没有脏写，可以还原业务数据，如果不一致说明有脏读，出现脏读就需要转人工处理



## 总结

