官网：

<https://github.com/alibaba/spring-cloud-alibaba/blob/master/README-zh.md>

Spring Cloud Alibaba 致力于提供微服务开发的一站式解决方案。此项目包含开发分布式应用微服务的必需组件，方便开发者通过 Spring Cloud 编程模型轻松使用这些组件来开发分布式应用服务。

依托 Spring Cloud Alibaba，您只需要添加一些注解和少量配置，就可以将 Spring Cloud 应用接入阿里微服务解决方案，通过阿里中间件来迅速搭建分布式应用系统。

## 主要功能

* **服务限流降级**：默认支持 WebServlet、WebFlux, OpenFeign、RestTemplate、Spring Cloud Gateway, Zuul, 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 为微服务应用构建消息驱动能力。
* **分布式事务**：使用 @GlobalTransactional 注解， 高效并且对业务零侵入地解决分布式事务问题。。
* **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（schedulerx-client）上执行。
* **阿里云短信服务**：覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

[**Sentinel**](https://github.com/alibaba/Sentinel)：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服务的稳定性。

[**Nacos**](https://github.com/alibaba/Nacos)：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。

[**Seata**](https://github.com/seata/seata)：阿里巴巴开源产品，一个易于使用的高性能微服务分布式事务解决方案。

## Nacos

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。  
Nacos：Dynamic Naming and Configuration Service

Nacos就是 注册中心 + 配置中心的组合

Nacos = Eureka + Config + Bus

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

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

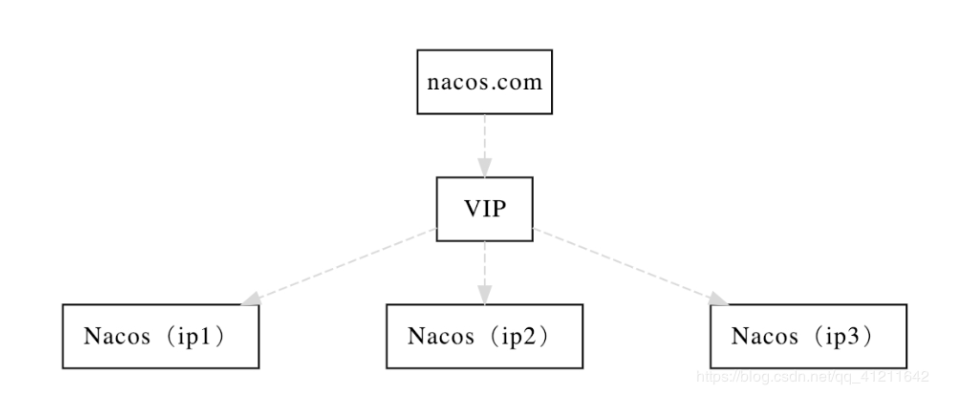


### 安装并运行Nacos

1. 本地Java8 + Maven环境已经OK
2. 先从官网下载Nacos
3. 解压安装包，直接运行bin目录下的startup.cmd
4. 运行成功后直接访问http://localhost:8848/nacos默认账号密码都是nacos

### 集群部署

部署架构图



默认Nacos使用嵌入式数据库（derby）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。

derby到mysql切换配置步骤:

1. nacos-server-1.1.4\nacos\conf 目录下执行sql脚本 nacos-mysql.sql
2. nacos-server-1.1.4\nacos\conf 目录下找到application.properties，添加配置：

|  |
| --- |
| spring.datasource.platform=mysql  db.num=1  db.url.0=jdbc:mysql://127.0.0.1:3306/nacos\_config?characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true  db.user=xxx  db.password=xxx |

### Nacos作为服务注册中心

#### 服务提供者：

新建工程：cloudalibaba-provider-payment9001

父工程pom.xml

|  |
| --- |
| *<!--spring cloud alibaba 2.1.0.RELEASE-->* <**dependency**>  <**groupId**>com.alibaba.cloud</**groupId**>  <**artifactId**>spring-cloud-alibaba-dependencies</**artifactId**>  <**version**>2.1.0.RELEASE</**version**>  <**type**>pom</**type**>  <**scope**>import</**scope**> </**dependency**> |

本工程pom.xml

|  |
| --- |
| *<!--springcloud alibaba nacos-->* <**dependency**>  <**groupId**>com.alibaba.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-alibaba-nacos-discovery</**artifactId**> </**dependency**> |

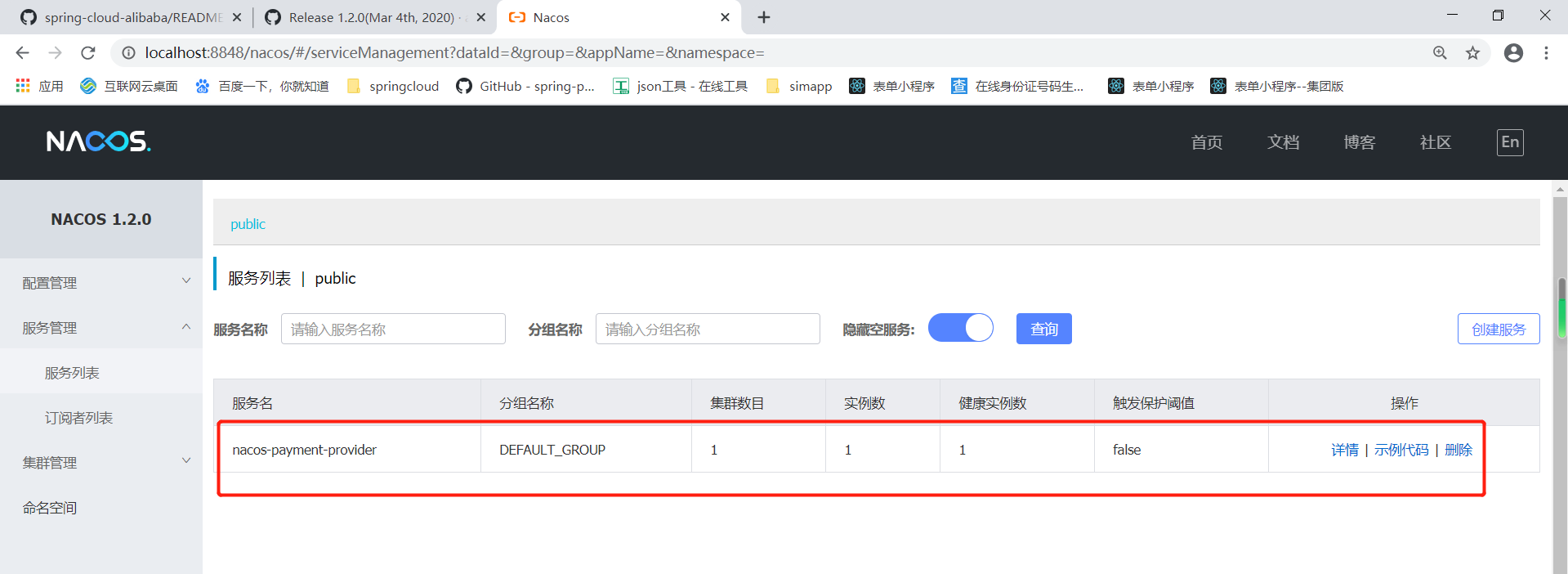
配置文件：application.yml

|  |
| --- |
| **server:  port:** 9001  **spring:  application:  name:** nacos-payment-provider  **cloud:  nacos:  discovery:  server-addr:** localhost:8848 *#配置Nacos地址* **management:  endpoints:  web:  exposure:  include: '\*'** *#监控* |

主配置类添加注解：

|  |
| --- |
| @SpringBootApplication @EnableDiscoveryClient |

启动成功，服务注册进nacos



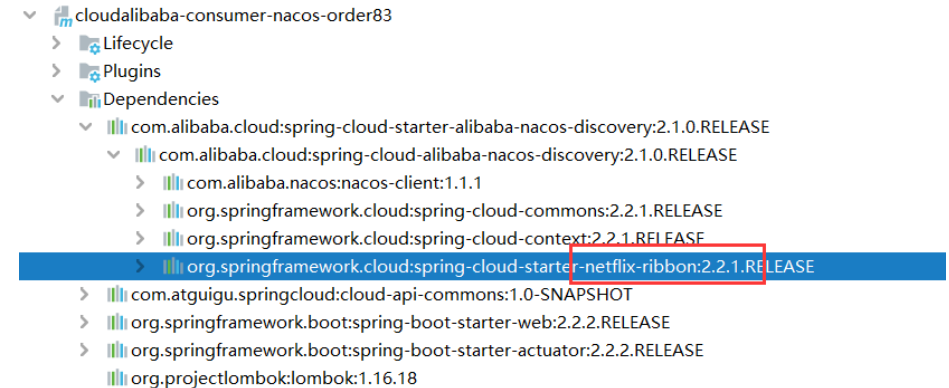
#### 服务消费者

新建工程：cloudalibaba-consumer-nacos-order83

其余依赖跟生产者一致

|  |
| --- |
| **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（成功注册进nacos的微服务提供者），在这配置了访问的服务，业务类就不用在定义常量了* **service-url:  nacos-user-service:** http://nacos-payment-provider |

Nacos自带负载均衡：



业务类引入RestTemplate

|  |
| --- |
| **package** com.demo.springcloud.config;  **import** org.springframework.cloud.client.loadbalancer.LoadBalanced; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** org.springframework.web.client.RestTemplate;  *//Nacos自带Ribbon，引入RestTemplate* @Configuration **public class** ApplicationContextConfig {   @Bean  @LoadBalanced *//RestTemplate结合Ribbon做负载均衡一定要加@LoadBalanced* **public** RestTemplate getRestTemplate() {  **return new** RestTemplate();  }  } |

controller层

|  |
| --- |
| @RestController **public class** OrderNacosController {   */\*  因为在yml中配置了service-url.nacos-user-service，  这里不需要再定义要访问微服务名常量，而是通过boot直接读出来  \*/* @Value(**"${service-url.nacos-user-service}"**)  **private** String **serverURL**;   @Resource  **private** RestTemplate **restTemplate**;   @GetMapping(**"/consumer/payment/nacos/{id}"**)  **public** String paymentInfo(@PathVariable(**"id"**) Long id){  **return restTemplate**.getForObject(**serverURL**+**"/payment/nacos/"**+id,String.**class**);  }  } |

#### 各个注册中心对比



Nacos 支持AP 和 CP模式的转换

C是所有节点在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受到响应。

何时选择使用何种模式？

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SpringCloud和Dubbo服务，都适用与AP模式，AP模式为了服务的可能性而减弱了一致性，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，如果服务不出存在，则会返回错误。

#### CP和AP模式切换

用下面命令切换

curl -X PUT '$NACOS\_SERVER:8848/nacos/v1/ns/operator/switches?entry=serverMode&value=CP'

### Nacos作为服务配置中心

**官网：Nacos Spring Cloud 快速开始**

<https://nacos.io/zh-cn/docs/quick-start-spring-cloud.html>

#### 工程搭建

1. 新建module cloudalibaba-config-nacos-client3377

2、pom.xml

|  |
| --- |
| <**dependency**>  <**groupId**>com.alibaba.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-alibaba-nacos-config</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>com.alibaba.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-alibaba-nacos-discovery</**artifactId**> </**dependency**> |

3、bootstrap.yml 和 application.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，拉取配置后，才能保证项目的正常启动。

SpringCloud中配置文件的加载是存在优先级顺序的，bootstrap优先级高于application

bootstrap.yml

|  |
| --- |
| *#bootstrap.xml* **server:  port:** 3377  **spring:  application:  name:** nacos-config-client  **cloud:  nacos:  discovery:  server-addr:** localhost:8848 *#Nacos服务注册中心地址* **config:  server-addr:** localhost:8848 *#Nacos作为配置中心地址* **file-extension:** yml *#指定yaml格式的配置 # group: TEST\_GROUP* **group:** DEFAULT\_GROUP  **namespace:** 58a0f6a1-dba6-403c-a953-0ce0a6d6200d  *# ${spring.application.name}-${spring.profile.active}.${spring.cloud.nacos.config.file.extension} # nacos-config-client-dev.yml* |

application.yml

|  |
| --- |
| **spring:  profiles:  active:** info *# active: dev #开发环境 # active: test #测试环境* |

4、主启动类添加注解

|  |
| --- |
| @SpringBootApplication @EnableDiscoveryClient |

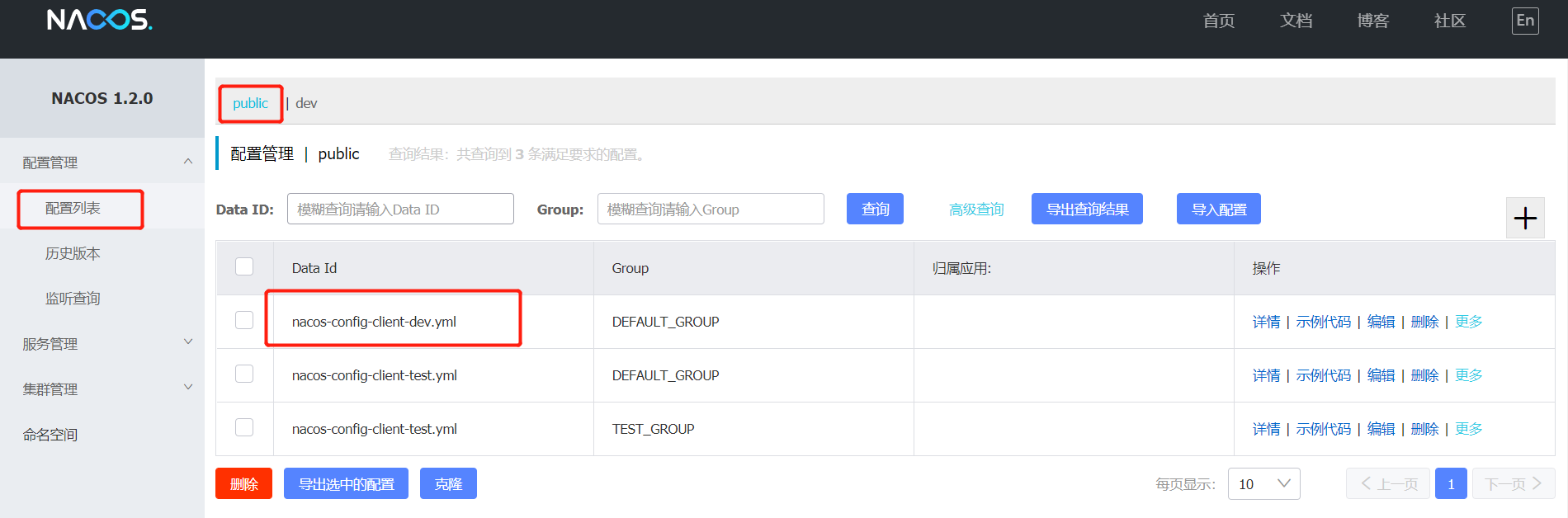
5、业务类

|  |
| --- |
| @RestController @RefreshScope *//SpringCloud原生注解 支持Nacos的动态刷新功能* **public class** ConfigClientController {   @Value(**"${config.info}"**) *//从配置中心读取*  **private** String **configInfo**;   @GetMapping(**"/config/info"**)  **public** String getConfigInfo() {  **return configInfo**;  }  } |

6、在Nacos中添加配置信息

nacos-config-client-dev.yml

|  |
| --- |
| config:  info: nacos-config center,version=2,test |



#### dataid组成规则：

在 Nacos Spring Cloud 中，dataId 的完整格式如下：

|  |
| --- |
| ${prefix}-${spring.profile.active}.${file-extension}  eg:  nacos-config-client-dev.yml  nacos-config-client-test.yml  nacos-config-client-info.yml |

* prefix 默认为 spring.application.name 的值，也可以通过配置项 spring.cloud.nacos.config.prefix来配置。
* spring.profile.active 即为当前环境对应的 profile，详情可以参考 [Spring Boot文档](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-profiles.html#boot-features-profiles)。 **注意：当 spring.profile.active 为空时，对应的连接符 - 也将不存在，dataId 的拼接格式变成 ${prefix}.${file-extension}**
* file-exetension 为配置内容的数据格式，可以通过配置项 spring.cloud.nacos.config.file-extension 来配置。目前只支持 properties 和 yaml 类型。

#### Namespace + Group + Data ID三者关系

是什么？

类似Java中的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

比方说我们现在有三个环境：开发、测试、生产环境，我们就可以创建三个Namespace，不同的Namespace之间是隔离的。

Group默认是DEFAULT\_GROUP，Group可以把不同的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

比方说为了容灾，将Service微服务分别部署在了杭州机房和广州机房，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（GZ），还可以尽量让同一个机房的微服务互相调用，以提升性能。

最后是Instance，就是微服务的实例。

#### 三种方案加载配置

1.DataID方案配置：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

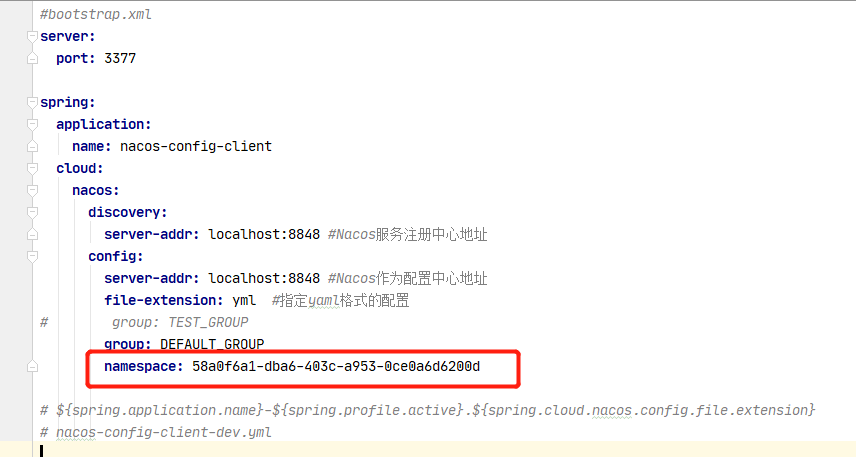
默认空间 + 默认分组 + 新建dev和test两个DataID

通过spring.profile.active属性就能进行多环境下配置文件的读取

2.Group方案：通过Group实现环境区分



3.Namespace方案：  
新建dev/test的Namespace



## Sentinel

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

下载地址：<https://github.com/alibaba/Sentinel/releases>

Sentinel分为两个部分：

* 核心库（Java客户端）不依赖任何框架/库，能够运行于所有Java运行时环境，同时对Dubbo/Spring Cloud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。

### 安装步骤：

下载到本地sentinel-dashboard-1.7.0.jar

运行命令：java -jar sentinel-dashboard-1.7.0.jar

指定端口运行：java -jar sentinel-dashboard-1.7.0.jar --server.port=8085

登录名密码都为 sentinel

sentinel控制台访问地址：<http://localhost:8080/>

### 使用说明：

1. 启动nacos8848
2. 新建module
3. 建工程cloudalibaba-sentinel-service8401
4. pom

|  |
| --- |
| *<!--springcloud alibaba nacos-->* <**dependency**>  <**groupId**>com.alibaba.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-alibaba-nacos-discovery</**artifactId**> </**dependency**> *<!--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**> <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-openfeign</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**> </**dependency**> |

1. yml

|  |
| --- |
| *#yml配置* **server:  port:** 8401  **spring:  application:  name:** cloudalibaba-sentinel-service  **cloud:  nacos:  discovery:** *#Nacos服务注册中心地址* **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  **management:  endpoints:  web:  exposure:  include: '\*'** |

1. 主启动类

|  |
| --- |
| @SpringBootApplication @EnableDiscoveryClient **public class** SentinelMain8401 {   **public static void** main(String[] args) {  SpringApplication.*run*(SentinelMain8401.**class**, args);  } } |

1. 业务类FlowLimitController

|  |
| --- |
| @RestController **public class** FlowLimitController {   @GetMapping(**"/testA"**)  **public** String testA() {  **return "--------testA"**;  }   @GetMapping(**"/testB"**)  **public** String testB() {  **return "--------testB"**;  } } |

1. 启动Sentinel8080
2. 启动微服务8401
3. 启动8401微服务后查询sentinel控制台

### 流控规则



* 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：

1. QPS（每秒钟的请求数量）：当调用该api的QPS达到阈值的时候，进行限流
2. 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：

1. 直接：api达到限流条件时，直接限流
2. 关联：当关联的资源达到阈值时，就限流自己
3. 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）

* 流控效果：

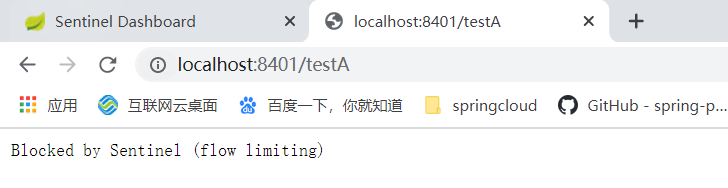
1. 快速失败：直接失败，抛异常
2. Warm Up：根据codeFactor（冷加载因子，默认3）的值，从阈值/codeFactor，经过预热时长，才达到设置的QPS阈值。
3. 排队等待：匀速排队，让请求以匀速的速度通过，阈值类型必须设置为QPS。否则无效。

#### 直接 快速失败



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

失败效果



#### 关联 快速失败

**适合做应用的让步**

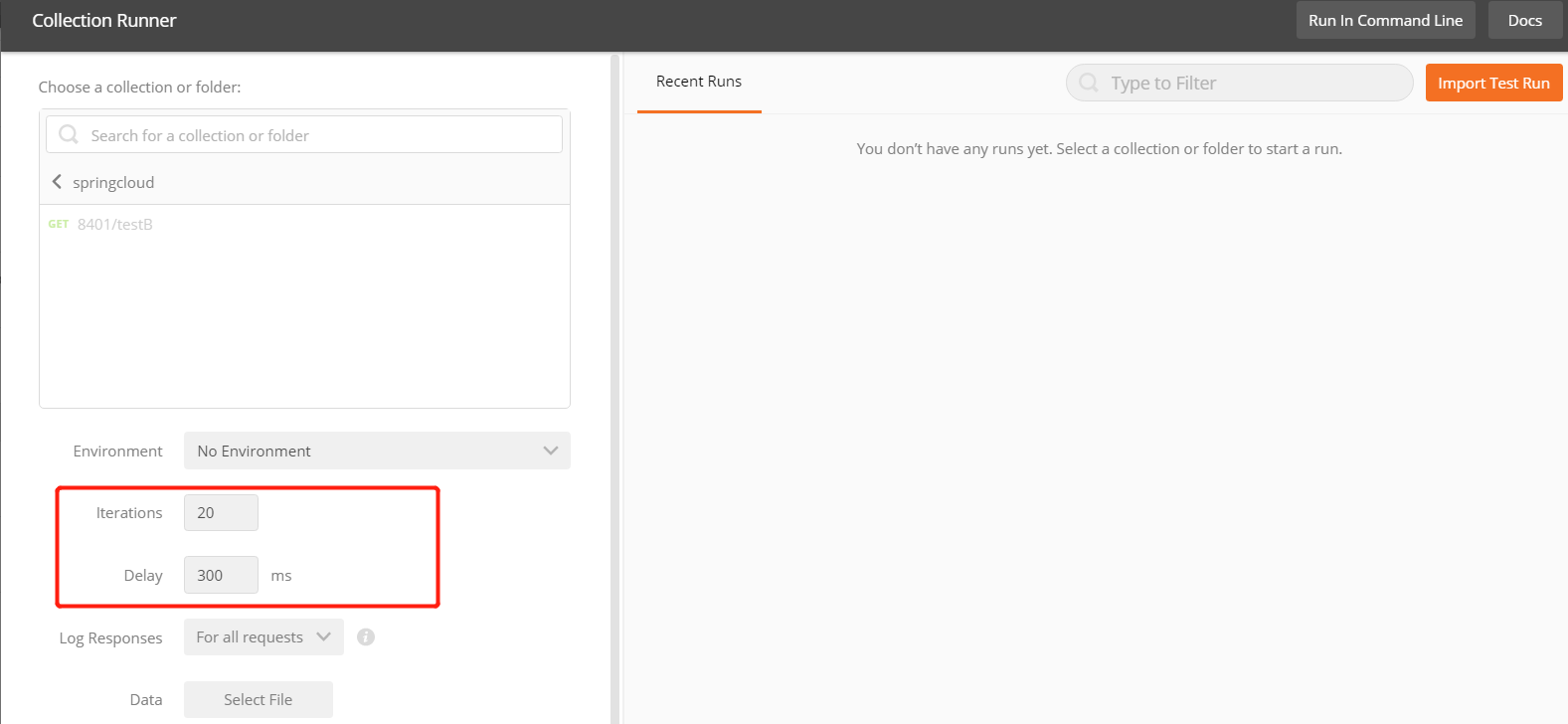
* 当关联的资源达到阈值时，就限流自己
* 当与A关联的资源B达到阀值后，就限流A自己

**设置关联资源/testB**

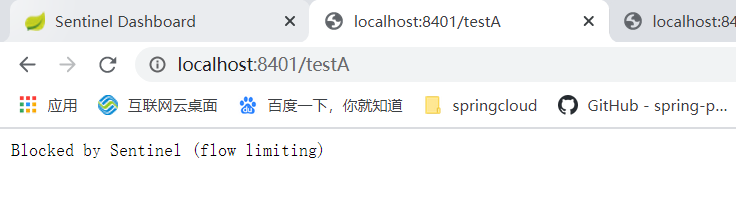


**postman测试**

这里我们起20个线程每隔0.3s访问一次

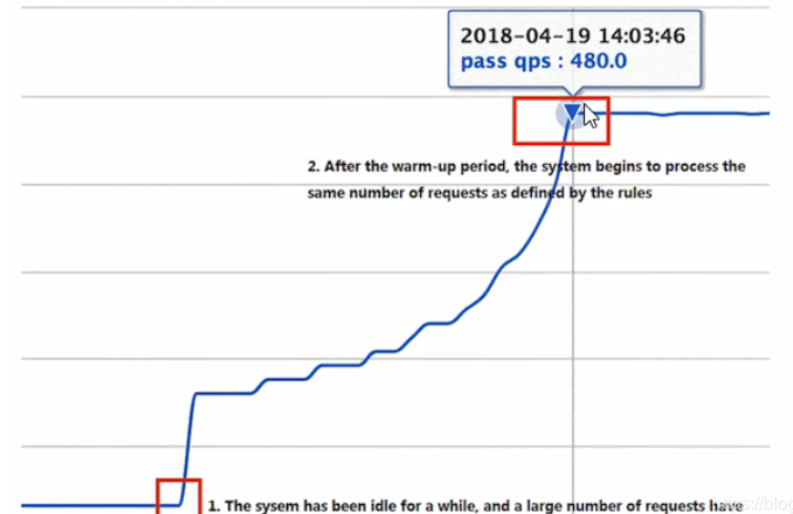


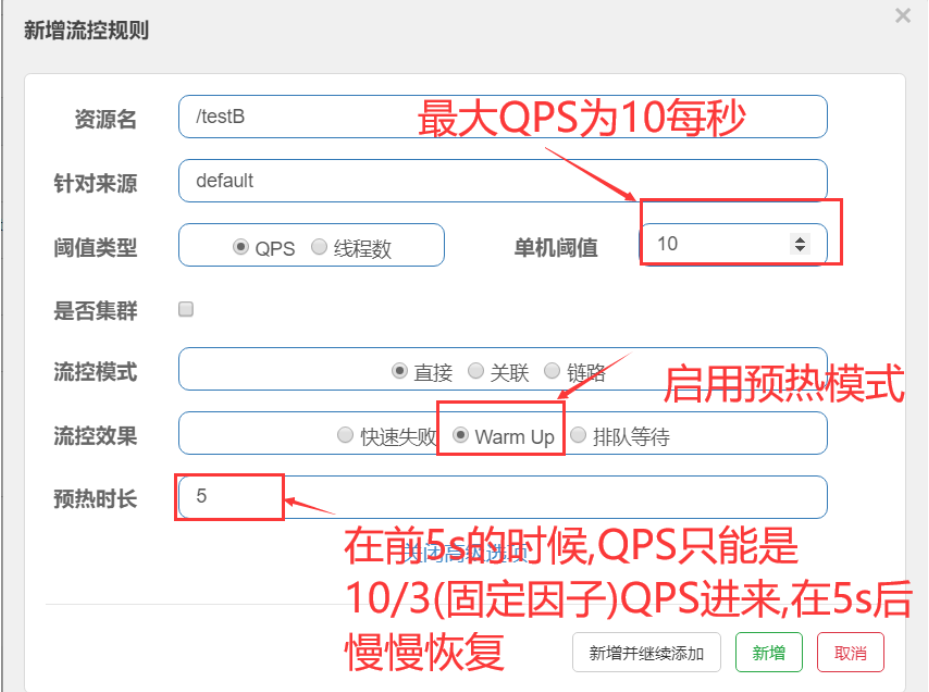
**可以看到/testB访问满了后,A服务直接挂了**



#### 流控效果——Warm Up

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上限，给冷系统一个预热的时间，避免冷系统被压垮。



默认**coldFactor**为**3**，即请求QPS从threshold / 3开始，经预热时长逐渐升至设定的QPS阈值。  
秒杀系统在开启的瞬间，会有很多流量进来，很有可能把系统打死，预热方式就是为了保护系统，可慢慢的把流量放进来，慢慢的把阈值增长到设置的阈值。

#### 流控效果——排队等待

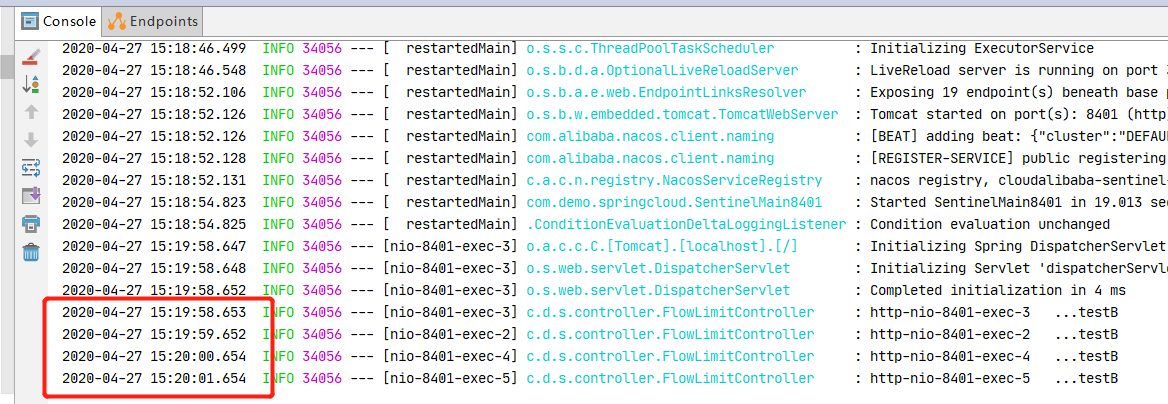
匀速排队,让请求以均匀的速度通过,阀值类型必须设成QPS,否则无效。

设置含义: /testA每秒1次请求，超过的话就排队等待，等待的超时时间为20000毫秒。

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，而接下来的几秒则处于空闲状态，我们希望系统能够在接下来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。

修改我们testB方法,打印当前线程日志





日志效果：每秒只有一个线程成功

### 降级规则



#### RT（平均响应时间，秒级）

平均响应时间超出阈值且在时间窗口内通过的请求>=5，两个条件同时满足后触发降级

窗口期过后关闭断路器

RT最大4900（更大的需要通过-Dcsp.sentinel.statistic.max.rt=XXX才能生效）

eg：

controller层新增方法

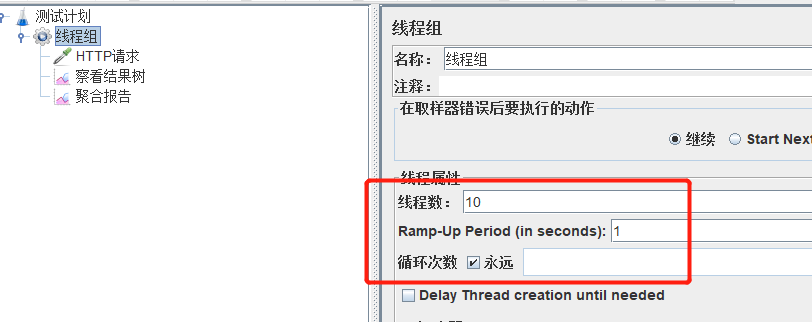
|  |
| --- |
| @GetMapping(**"/testD"**) **public** String testD() {  **try** {  TimeUnit.***SECONDS***.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return "----- testD"**; } |

新增降级规则：**返回时间要是超过200毫秒,就会在1s内降级**

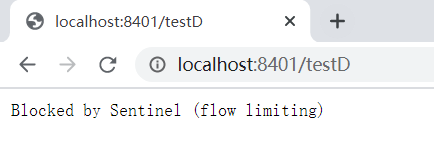


测试

jmeter开启每秒10个线程



再次访问，服务已经报错



结论：

永远一秒钟打进来10个线程(大于5个了)调用testD, 我们希望200毫秒处理完本次任务

如果超过200毫秒还没处理完，在未来1秒钟的时间窗口内，断路器打开(保险丝跳闸)微服务不可用，保险丝跳闸断电了后续我停止jmeter,没有这么大的访问量了，断路器关闭(保险丝恢复),微服务恢复OK

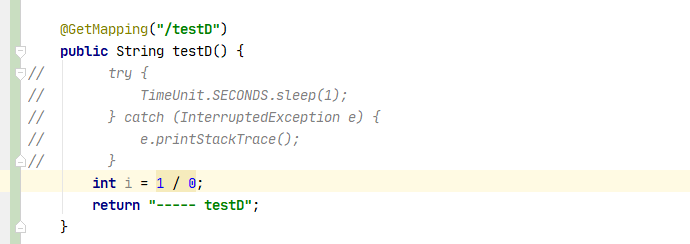
#### 异常比例（秒级）

**QPS>=5**且异常比例（秒级统计）超过阈值时，触发降级；时间窗口期结束后，关闭降级

eg：

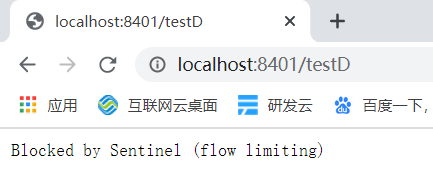
**首先资源量要>=5**

如果返回比例30%以上都是抛异常的话，那么就会在1秒内触发降级规则





继续采用jmeter压测，然后再访问，服务报错了



#### 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

* 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，对该资源的调用都自动熔断(默认行为是抛出DegradeException)。

eg：

1分钟内错误数达到6个，断路器开启，微服务降级。需要等待61秒窗口期后才可以。



### 热点参数限流规则

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，并对其访问进行限制。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。

#### 普通配置

eg：

controller层新增方法

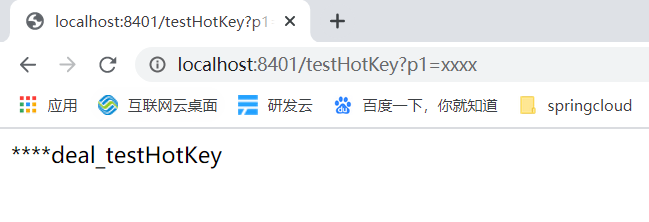
|  |
| --- |
| *//热点限流* @GetMapping(**"/testHotKey"**) @SentinelResource(value = **"testHotKey"**, blockHandler = **"deal\_testHotKey"**) **public** String testHotKey(@RequestParam(value = **"p1"**, required = **false**) String p1,  @RequestParam(value = **"p2"**, required = **false**) String p2) {  **return "\*\*\*\*\*testHotKey"**; }  **public** String deal\_testHotKey(String p1, String p2, BlockException exception) {  **return "\*\*\*\*deal\_testHotKey"**; } |

sentinel增加热点规则

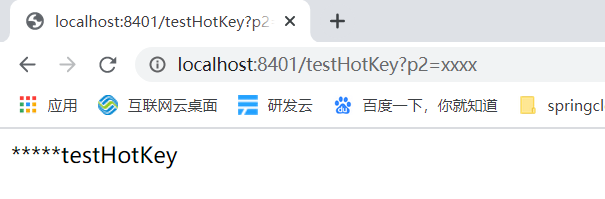
配置：索引为第0个的参数（p1）阈值为1



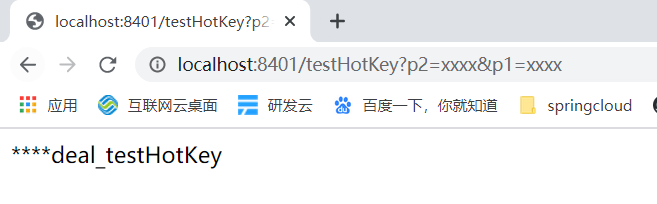
携带p1参数快速多次访问：超过1秒一次的限制，触发blockHandler方法。

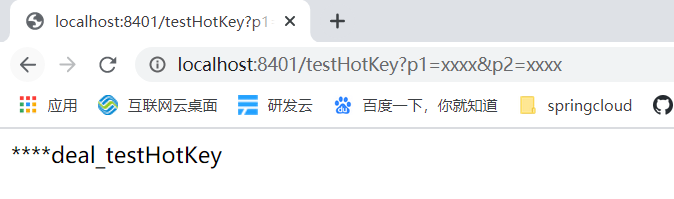


只携带p2参数快速访问多次：一切正常。



同时携带p1和p2参数快速访问：交换url参数的位置效果，也一样，触发blockHandler方法。





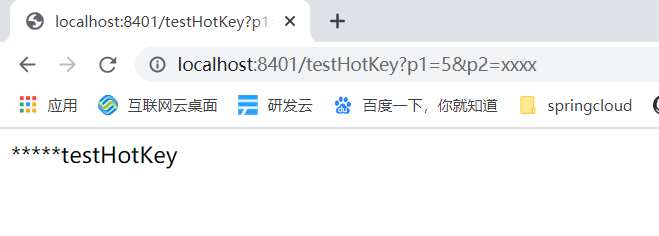
#### 参数例外项

上述案例在访问超过1秒1个后马上被限流。但是，我们希望p1参数当它时某个特殊值时，它的限流值和平时不一样。

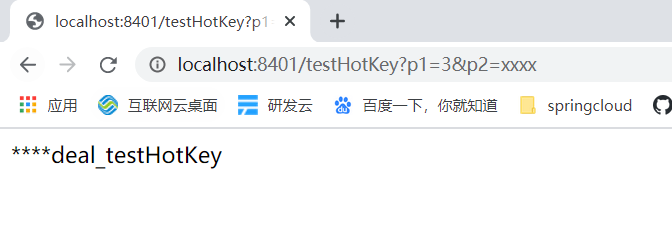
比如：当p1=5时，它的阈值可以达到200；其它值仍然为1



携带p1=5快速多次访问：正常：



携带p1=3快速多次访问：触发blockHandler方法：



**注意：**

* @SentinelResource处理的是Sentinel控制台配置的违规情况，由blockHandler方法配置的兜底处理；
* 如果逻辑代码中出现其它RuntimeException，@SentinelResource是不管的。