

1 微服务下客户端负载均衡

1.1 Ribbon负载均衡概述

Spring Cloud Ribbon是基于Netflix Ribbon 实现的一套客户端的负载均衡工具

通过Load Balancer(LB)获取到服务提供的所有机器实例，Ribbon会自动基于某种规则(轮询、随机等)去调用这些服务。

Ribbon也可以实现我们自己的负载均衡算法。

客户端负载均衡：就是进程内的LB,他是一个类库集成到消费端，通过消费端进行获取提供者的地址。

服务端负载均衡：请求调用负载均衡器，由负载均衡器算法解析决定调用哪一个服务。

Ribbon有两种使用方式：

一种是和RestTemplate结合

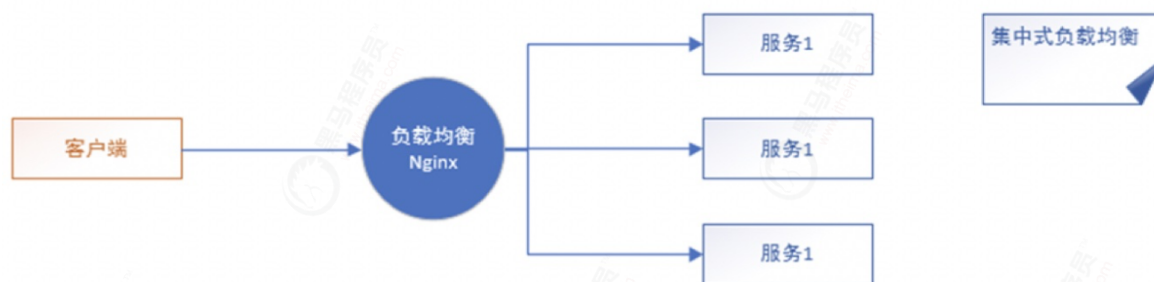
一种是和Feign(已集成) 相结合

1.2. 负载均衡分类及工作原理

业界主流的负载均衡解决方案有

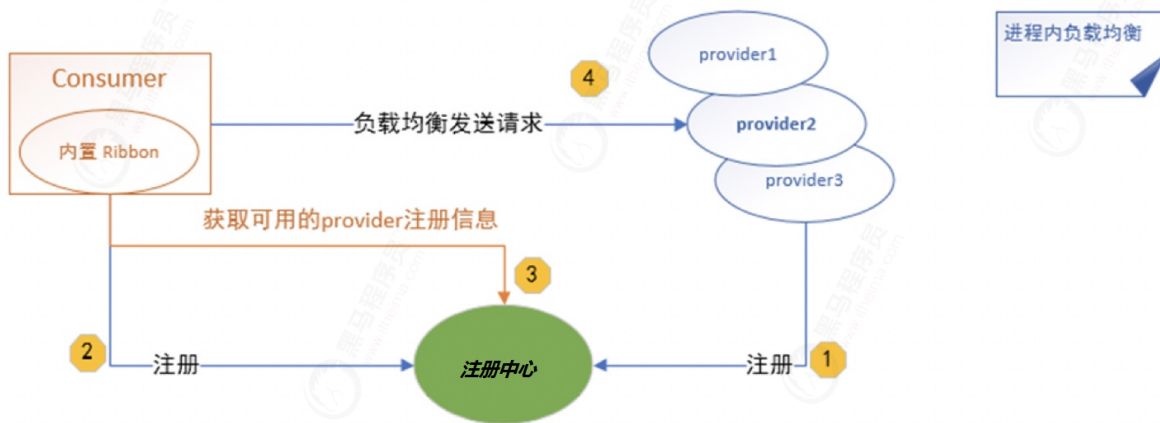
1、集中式负载均衡

即在客户端和服务端之间使用独立的负载均衡设施(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端。



2、进程内负载均衡

将负载均衡逻辑集成到客户端组件中，客户端组件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端发起请求。Ribbon就是一个进程内的负载均衡实现。



tips

只有调用方（消费者）加入负载均衡器即可

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

<https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers#components-of-load-balancer>

客户端负载均衡到底做了哪些事情？

我们再来看看ribbon官方是怎么定义的

Components of load balancer

- **Rule** - a logic component to determine which server to return from a list
- Ping - a component running in background to ensure liveness of servers
- ServerList - this can be static or dynamic. If it is dynamic (as used by `DynamicServerListLoadBalancer`), a background thread will refresh and filter the list at certain interval

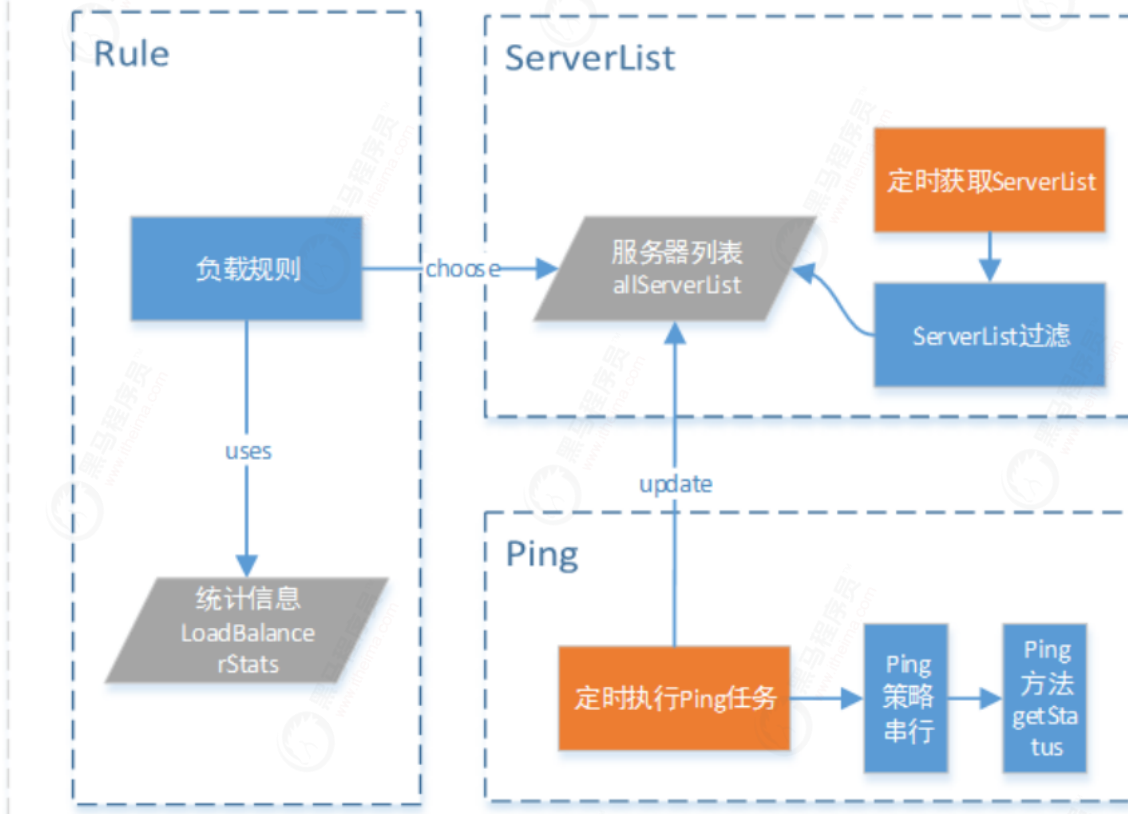
These components can be either set programmatically or be part of client configuration properties and created via reflection. These are the related properties names (should be prefixed by `<clientName>.<namespace>` in the property file):

负载均衡器有三大组件：

1. 负载规则，从服务器列表中决定用哪个服务器
 2. ping任务，后台运行的任务，用来验证服务器是否可用
 3. 服务器列表，可以是静态也可以是动态，如果是动态
- 我们微服务都是基于服务发现，服务器列表肯定都是动态增减的

结合上面的三个角色，我们可以抽象出下面的这张图

负载均衡器 LoadBalancer



1.3 负载均衡策略

id	策略名称	策略对应的类名	实现原理
1	轮询策略（默认）	RoundRobinRule	轮询策略表示每次都顺序取下一个provider，比如一共有5个provider，第1次取第1个，第2次取第2个，第3次取第3个，以此类推
2	权重轮询策略（常用）	WeightedResponseTimeRule	1.根据每个provider的响应时间分配一个权重，响应时间越长，权重越小，被选中的可能性越低。 2.原理：一开始为轮询策略，并开启一个计时器，每30秒收集一次每个provider的平均响应时间，当信息足够时，给每个provider附上一个权重，并按权重随机选择provider，高权重重的provider会被高概率选中。
3	随机策略（不推荐）	RandomRule	从provider列表中随机选择一个provider
4	最少并发数策略（应用在硬件软件环境一致的情况下）	BestAvailableRule	选择正在请求中的并发数最小的provider，除非这个provider在熔断中。
5	在“选定的负载均衡策略”基础上进行重试机制	RetryRule	1.“选定的负载均衡策略”这个策略是轮询策略RoundRobinRule 2.该重试策略先设定一个阈值时间段，如果在这个阈值时间段内当选择provider不成功，则一直尝试采用“选定的负载均衡策略：轮询策略”最后选择一个可用的provider
6	可用性敏感策略（一般在同区域内服务集群环境中使用）	AvailabilityFilteringRule	过滤性能差的provider,有2种： 第一种：过滤掉在eureka中处于一直连接失败provider 第二种：过滤掉高并发的provider
7	区域敏感性策略（应用在大型的，物理隔离分布式环境中）	ZoneAvoidanceRule	1.以一个区域为单位考察可用性，对于不可用的区域整个丢弃，从剩下区域中选可用的provider 2.如果这个ip区域内有一个或多个实例不可达或响应变慢，都会降低该ip区域内其他ip被选中的权重。

1.4 多策略负载均衡实战

本章节场景：

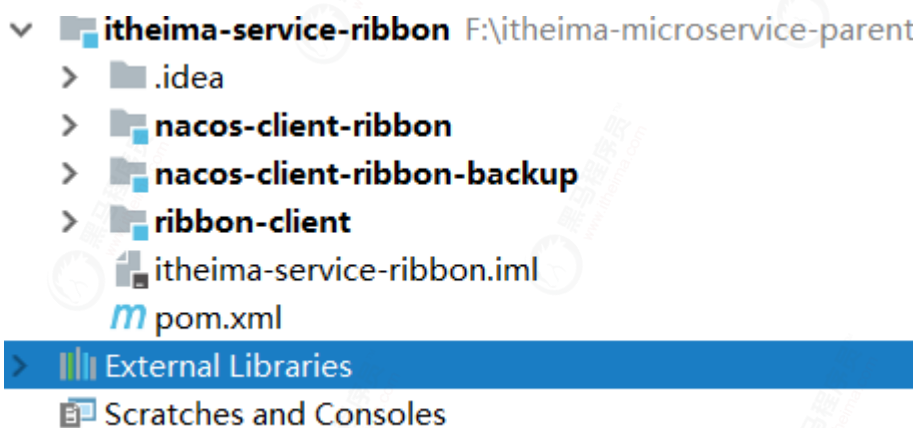
我们将建立3个Module，父项目是itheima-service-ribbon，里面没有任何代码，它用来管理我们的聚合工程。

```
<modules>
  <module>ribbon-client</module>
  <module>nacos-client-ribbon</module>
  <module>nacos-client-ribbon-backup</module>
</modules>
```

具体场景：

将nacos-client-ribbon、nacos-client-ribbon-backup注册到NACOS

然后通过ribbon-client调用nacos-client-ribbon、nacos-client-ribbon-backup上的api，实现负载均衡



5.3.1 创建负载均衡

创建负载均衡客户端模块ribbon-client

1. 引入起步依赖

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

2. 修改application.yml文件


```

spring:
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.23.139:8848
    application:
      name: ribbon-client
  server:
    port: 8764

```

3. 增加配置类RibbonConfig

```

@Configuration
public class RibbonConfig {

    /**
     * 在此类中为IoC容器中注入一个RestTemplate的Bean,并在这个Bean 上
     * 加上@LoadBalanced 注解,此时RestTemplate 就结合了Ribbon 开启了负载均衡功能
     */
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

```

新建一个RibbonConfig加上@Configuration注解,标识此类为配置类,当前配置类的作用主要是定义RestTemplate的负载均衡的能力

RestTemplate是Spring Web模块提供的一个基于Rest规范提供Http请求的工具。

Ribbon是Spring Cloud中客户端负载均衡的组件。我们在微服务架构中,往往通过RestTemplate发送RPC请求,然后通过Ribbon做客户端负载均衡,通过硬编码的形式在方法上加入元注解

@LoadBalanced, 当一个被@LoadBalanced注解修饰的RestTemplate对象向外发起HTTP请求时,就赋予了负载均衡的能力

4. 增加RibbonService

```

package com.itheima.ribbon.client.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class RibbonService {

    @Autowired
    RestTemplate restTemplate;

    /**
     * 在该类的方法用restTemplate调用eureka-client的API接口 uri 上不需要使用硬编码(比如IP),只需要写服务名nacos-client即可
     * 程序会根据服务名称 nacos-client到Nacos注册中心去自动获取IP和端口信息。
     */
}

```

```

    * @param name
    * @return
    */
    public String hello(String name) {
        return restTemplate.getForObject("http://nacos-client/hello?name=" + name,
String.class);
    }
}

```

5. 增加RibbonController

```

package com.itheima.ribbon.client.web;

import com.itheima.ribbon.client.service.RibbonService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

/**
 * 加上@RestController 注解，开启RestController 的功能
 */
@RestController
public class RibbonController {

    @Autowired
    RibbonService ribbonService;

    @GetMapping("/hello")
    public String hello(@RequestParam(required = false, defaultValue = "itheima")
String name) {
        return ribbonService.hello(name);
    }

    @Autowired
    private LoadBalancerClient loadBalancer;

    /**
     * 通过LoadBalancerClient 去选择一个nacos的服务实例的信息， 并将该信息返回
     */
    @GetMapping("/testribbon")
    public String testRibbon() {
        ServiceInstance instance = loadBalancer.choose("nacos-client");
        return instance.getHost() + ":" + instance.getPort();
    }
}

```

指定负载均衡策略

代码：

初始化到RibbonConfig类中

```

/*
 * @Description: 轮询
 * @Method: microServiceRule
 * @Param: []
 * @Update:
 * @since: 1.0.0
 * @Return: com.netflix.loadbalancer.IRule
 *
 */
@Bean
public IRule microServiceRule() {
    return new RoundRobinRule();
}

```

配置文件配置

```

nacos-client:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule
#nacos-client为当前服务调用别的的服务的应用名称，即nacos-client

```

6. 启动

在访问前依次启动ribbon-client、nacos-client-ribbon、nacos-client-ribbon-backup

NACOS 1.3.1

public

服务列表 | public

服务名称: 请输入服务名称 分组名称: 请输入分组名称 隐藏空服务: ☒ 查询 [创建服务](#)

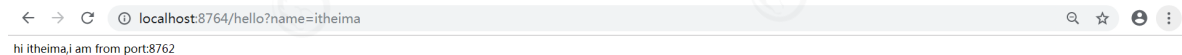
服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
ribbon-client	DEFAULT_GROUP	1	1	1	false	详情 示例代码 删除
nacos-client	DEFAULT_GROUP	1	2	2	false	详情 示例代码 删除

7. 访问（优先级演示、默认负载演示）

在浏览器输入

```
http://localhost:8764/hello?name=itheima
```

如下图：



输出显示如下：

```
hi itheima,i am from port:8762
```

```
hi itheima,i am from port:8763
```

由此可见，网页上的数据会轮流显示，说明我们的ribbon-client代码达到了负载均衡的效果（默认轮询策略）

##

LoadBalancerClient 选择服务实例

通过LoadBalancerClient 去选择一个nacos的服务实例的信息，并将该信息返回

```
http://localhost:8764/testribbon
```

← → ↻ localhost:8764/testRibbon

172.16.43.175:8763

负载策略：随机演示

```
@Bean
public IRule microServiceRule() {
//    return new RoundRobinRule();
    return new RandomRule();
}
```

配置文件为轮询、代码为随机，优先级比较

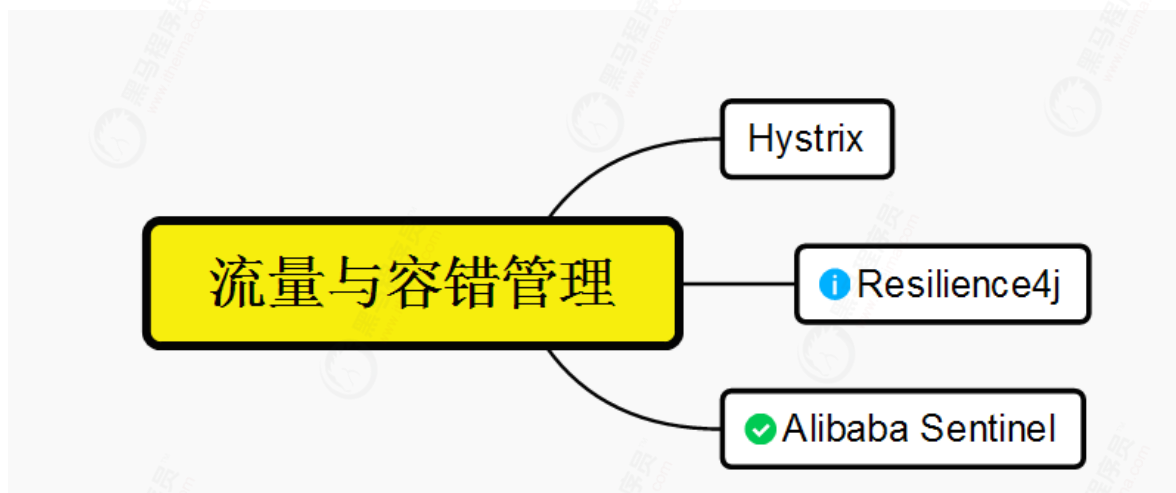
```
nacos-client:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule
```

优先级

配置类>配置文件>默认配置

2 微服务下高可用流量管理与轻量级容错

2.1.1 关于容错组件的停更/升级/替换



服务降级:

Hystrix: 官网不极力推荐, 但是中国企业中还在大规模使用, 对于限流和熔断降级虽然在1.5版本官方还支持 (版本稳定), 但现在官方已经开始推荐大家使用Resilience4j

Resilience4j: 官网推荐使用, 但是国内很少用这个。

Sentinel: 来自于Spring Cloud Alibaba, 在中国企业替换Hystrix的组件, 国内强烈建议使用

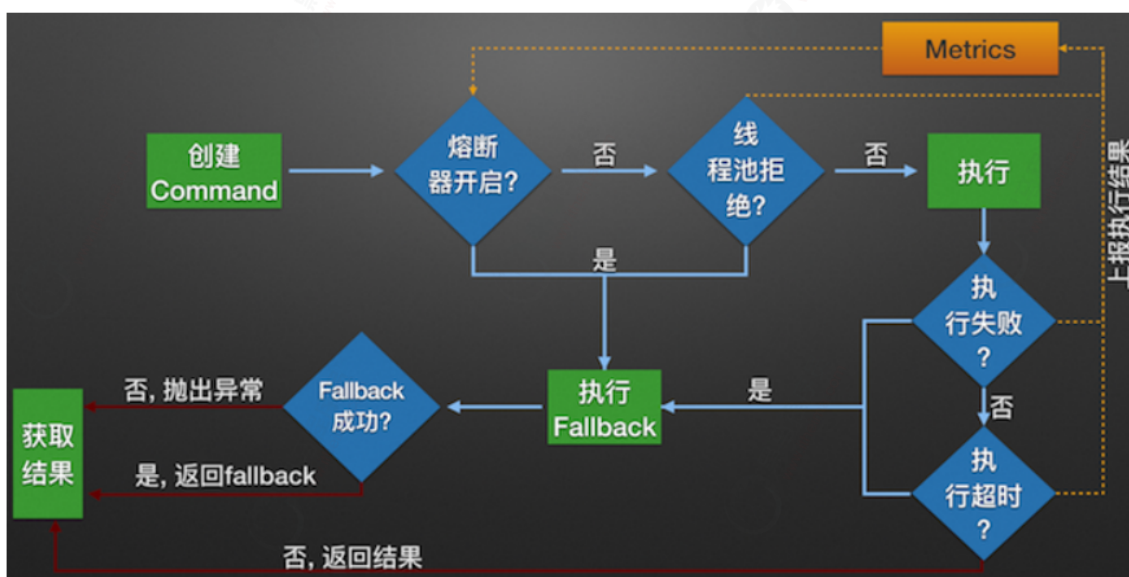
2.1 流量与容错管理之Hystrix

2.1.1 Hystrix概述

1. Hystrix简介

Hystrix是Netflix公司的一个开源项目, 在分布式环境中, 许多服务依赖项中的一些必然会失败, Hystrix主要作用是通过控制那些访问远程系统、服务和第三方库的节点, 从而对延迟和故障提供更强大的容错能力, 而有效减少微服务中雪崩发生, 提高系统的整体性能。

2. Hystrix工作原理



Hystrix Metrics解释:

Metrics中保存了当前服务的健康状况,

包括服务调用总次数和服务调用失败次数等。根据Metrics的计数,熔断器从而能计算出当前服务的调用失败率,用来和设定的阈值比较从而决定熔断器的状态和切换逻辑..

3. Hystrix业务能力

熔断

- 当应用(B)无法提供服务时, 对上游请求熔断

A---call--B,B不可用, 直接熔断B

B不可用、qps、失败率

降级

- 降级与熔断紧密相关, 当边缘业务不再提供服务的时候; 直接下线, 通过fallback响应; 即为服务降级

隔离

- 实现请求隔离算法是线程池或信号量

Hystrix在用户请求和服务之间加入了线程池, 如果线程池已满调用将被立即拒绝(降级后进入fallback)

信号隔离也可以用于限制并发访问(熔断进入fallback)

限流

- 当对服务进行限流时, 超过的流量将直接 Fallback, 即熔断。

这里的限流是针对熔断的具体描述

熔断VS降级

相同点:

目标一致 都是从可用性和可靠性出发, 为了防止系统崩溃;

熔断与降级都需要fallback

不同点:

触发原因不同 服务熔断一般是某个服务(下游服务)故障引起, 而服务降级一般是从整体负荷考虑

2.1.2 雪崩效应介绍

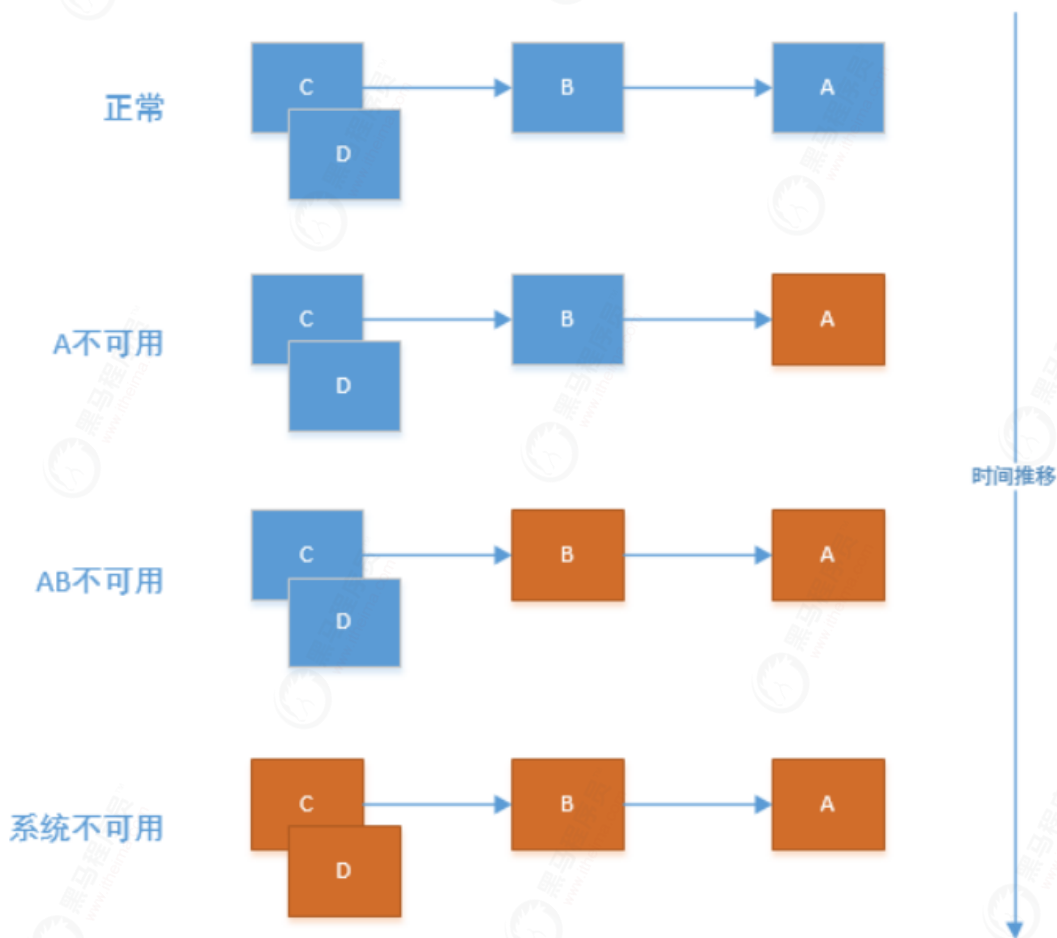
1. 下图主要描述了产生雪崩效应的流程图

(AB不可用)

B服务调用A服务, A服务不可用, 导致B服务不停重试和阻塞, 最后B服务变为不可用

(系统不可用)

C服务调用B服务，B服务不可用，导致C服务不停重试和阻塞，最后C服务变为不可用
至此，所有服务均不可用，产生了服务雪崩效应



雪崩形成大致可以分成三个阶段（造成雪崩的原因）：

服务提供者不可用

用户大量请求：在秒杀和大促开始前,如果准备不充分,用户发起大量请求也会造成服务提供者的不可用.

重试加大流量

用户重试：在服务提供者不可用后, 用户由于忍受不了界面上长时间的等待,而不断刷新页面甚至提交表单.

代码逻辑重试：服务调用端的会存在大量服务异常后的重试逻辑.

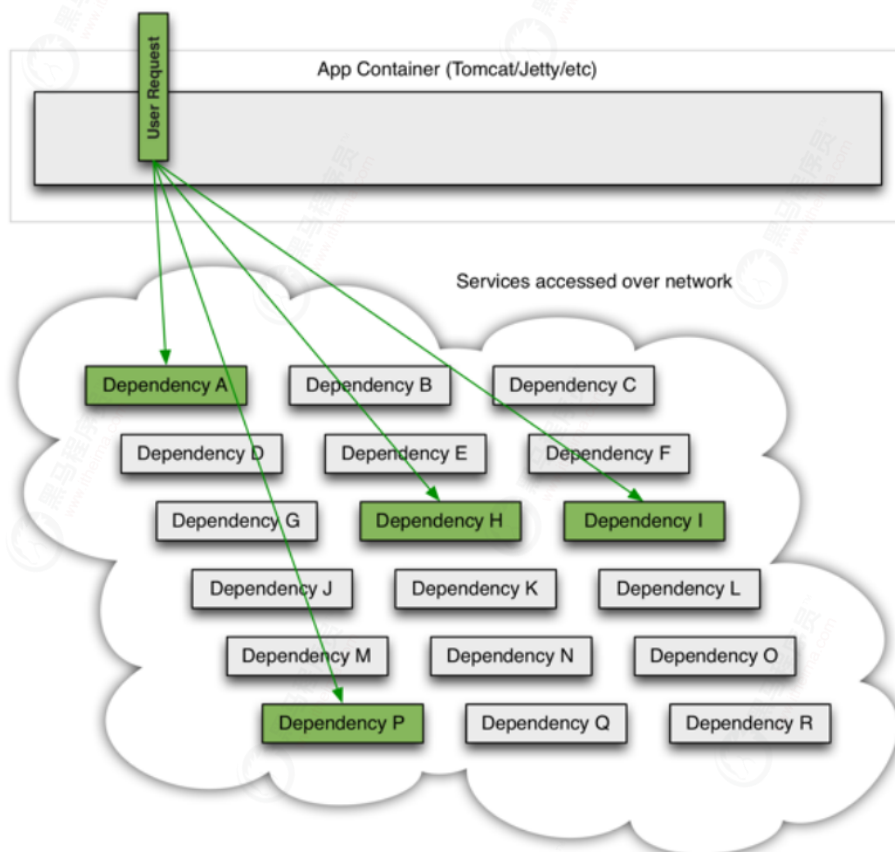
服务调用者不可用

同步等待造成的资源耗尽：当服务调用者使用 同步调用 时, 会产生大量的等待线程占用系统资源. 一旦线程资源被耗尽,服务调用者提供的服务也将处于不可用状态, 于是服务雪崩效应产生了.

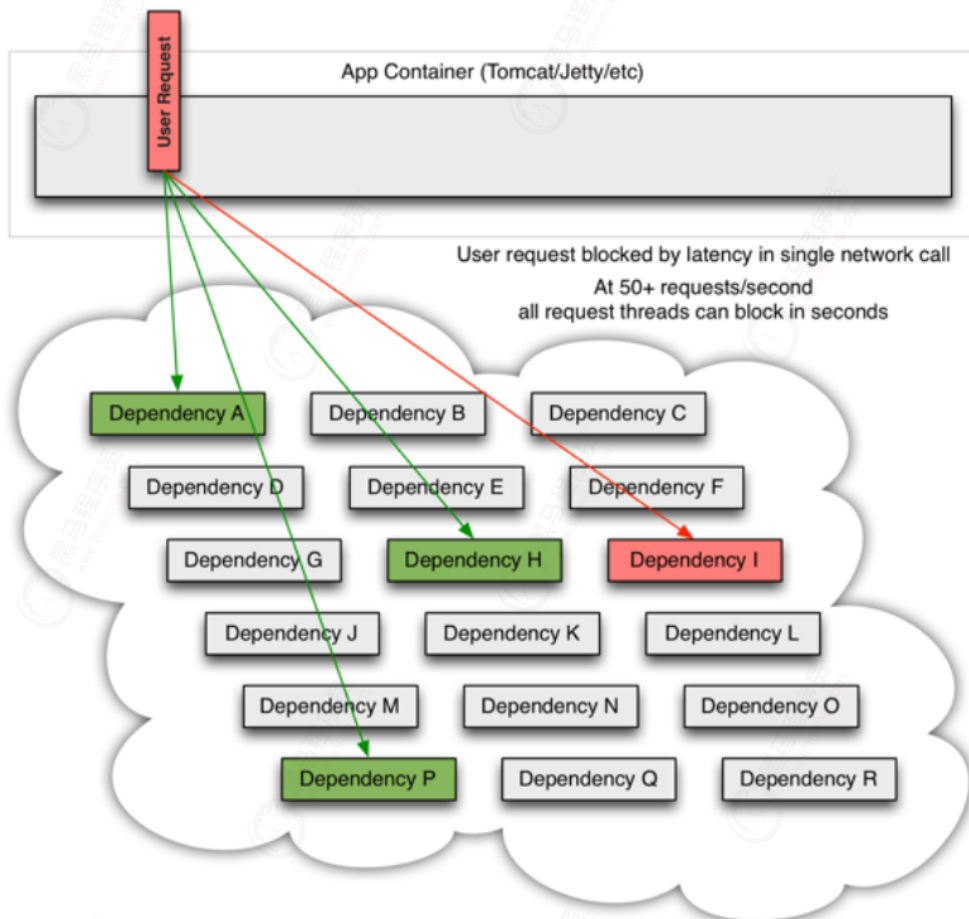
2. 雪崩场景

在微服务系统中，一个用户请求可能需要调用几个服务才能完成。

如下图，在所有的服务都处于可用状态时，一个用户请求需要调用A、H、I和P服务。



当某一个服务，例如服务I，出现网络延迟或者故障时，即使服务A、H和P可用，由于服务I的不可用，整个用户请求会处于阻塞状态，并等待服务I的响应（如下图）。



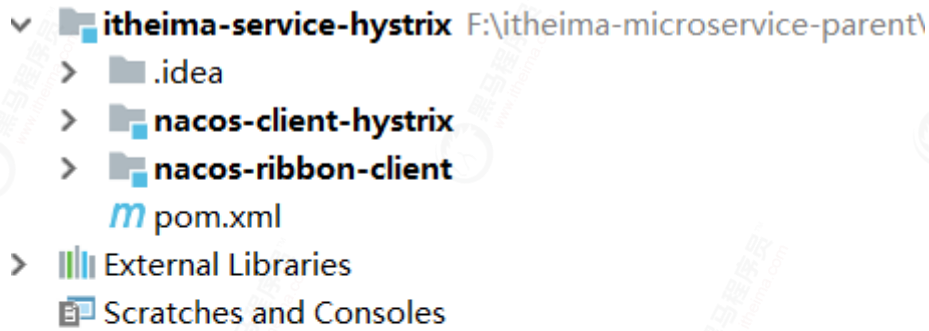
2.1.3 Hystrix降级处理

所谓降级，一般是从整体负荷考虑。就是当某个服务熔断之后，服务器不再被调用。此时客户端可以自己准备一个本地的fallback回调，返回一个缺省值
也就是说我们的整体资源快不够了，忍痛将某些服务先关掉，待渡过难关，再开启回来。

降级应用场景：

人工降级：秒杀、双11的时候主动降级（确保核心业务、边缘业务降级）

超时



1. 创建模块

创建nacos-ribbon-client 模块，使用Ribbon+RestTemplate中使用Hystrix

2. 引入起步依赖

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

3. 修改application.yml

```
spring:
  application:
    name: nacos-ribbon-client
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.23.139:8848
  server:
    port: 8764
```

4. 启动熔断器

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableHystrix
public class RibbonClientApplication {

    public static void main(String[] args) {

        SpringApplication.run(RibbonClientApplication.class, args);
    }
}
```

5. 创建RibbonConfig配置类

```

/**
 * 加入@Configuration表明当前类为配置类
 */
@Configuration
public class RibbonConfig {

    /**
     * 为IoC 容器中注入一个RestTemplate 的Bean
     * 并在这个Bean 上加上@LoadBalanced 注解，此时RestTemplate 就结合了
     * Ribbon 开启了负载均衡功能。
     *
     * @return
     */
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

```

6. 创建RibbonController类

- 1、超时引发的降级（try里面）
- 2、积分服务停止也可以引发降级

```

// @HystrixCommand(fallbackMethod = "fallback")
@GetMapping("/order")
@HystrixCommand(fallbackMethod = "fallback", commandProperties = {
    @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value = "3000")})

    public String order(@RequestParam(required = false, defaultValue = "itheima")
String name) {
    // try {
    //     TimeUnit.MILLISECONDS.sleep(5000);
    // } catch (InterruptedException e) {
    //     e.printStackTrace();
    // }
    // 订单服务调用积分服务，主动停掉积分(降级)
    return restTemplate.getForObject("http://nacos-client-hystrix/integral?
name=" + name, String.class);
}

/*
 * @Description:其他处理
 * @Method: fallback
 * @Param: [name]
 * @Update:
 * @since: 1.0.0
 * @Return: java.lang.String
 */

```



```

*/
public String fallback(String name) {
    return "请稍后再试...";
}

```

7、积分服务代码

```

@GetMapping("/integral")
public String integral(@RequestParam String name) {
    System.out.println("开始调用积分服务...");
    try {
        TimeUnit.MILLISECONDS.sleep(200);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    //    int s= 1/0;
    return "积分服务调用成功 " + name + ",i am from port:" + port;
}

```

8. 启动

分别启动 `nacos-client-hystrix`、`nacos-ribbon-client`，如下图

NACOS 1.3.1

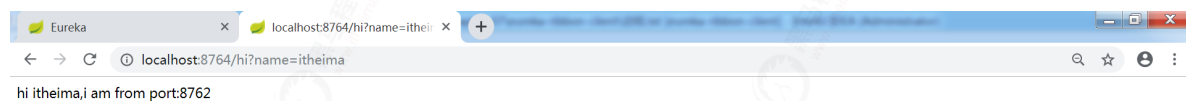
配置管理 | 服务列表 | public

服务名称: 请输入服务名称 分组名称: 请输入分组名称 隐藏空服务: ☒ 查询 创建服务

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-client-hystrix	DEFAULT_GROUP	1	1	1	false	详情 示例代码 删除
nacos-ribbon-client	DEFAULT_GROUP	1	1	1	false	详情 示例代码 删除

9. 访问

`http://localhost:8764/order?name=itheima`



请稍后再试...

2.1.4 Hystrix熔断处理

熔断，是分布式系统中非常重要的一个概念。一个典型的场景就是在多次发现某个接口或者某处逻辑频繁出现超时或者其他异常情况时，消费者调用提供者的接口耗时过长，超过了某一个既定阈值，就阻断这个接口或者服务的调用。以此来避免“雪崩”出现

典型的场景

- 1、请求量激增（单位时间内qps达到峰值）
- 2、出现异常
- 3、调用的失败率过大（动态计算）

1) 场景：并发15流量限制

场景

通过coreSize设置接口的并发数据为15；如果并发流量为20个，那么其他5个并发则进入降级

修改上面的 例子

com.itheima.hystrix.service.RibbonService增加方法

修改HystrixCommand注解

```
@HystrixCommand(  
    fallbackMethod = "orderError",  
    threadPoolKey = "order",  
    threadPoolProperties = {  
        @HystrixProperty(name = "coreSize", value = "15")  
    }  
)  
  
// @HystrixCommand(fallbackMethod = "orderError")  
public String order(String name) {  
    try {  
        TimeUnit.MILLISECONDS.sleep(500); //熔断新增代码  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    return restTemplate.getForObject("http://nacos-client-hystrix/hi?name=" +  
name, String.class);  
}
```

jmeter设置

设置并发20

继续

Start Next Thread Loop

停止线程

停止测试

Stop Test Now

线程属性

线程数: 20

Ramp-Up Period (in seconds):

循环次数: ☐ 永远 1

☐ Delay Thread creation until needed

☐ 调度器

修改端口

HTTP请求默认值

名称: HTTP请求默认值

注释:

Web服务器

服务器名称或IP: 127.0.0.1

端口号: 8764

Timeouts (milliseconds)
Connect: Response:

HTTP请求

Implementation: 协议: Content encoding: UTF-8

路径:

Parameters

同请求一起发送参数:

名称:	值	编码?	包含等于?
-----	---	-----	-------

清泉路径

HTTP请求

名称: 支付接口测试

注释:

Web服务器

服务器名称或IP: 端口号:

Timeouts (milliseconds)
Connect: Response:

HTTP请求

Implementation: 协议: 方法: GET Content encoding:

路径: /order?name=itheima

☐ 自动重定向 ☒ 跟随重定向 ☒ Use KeepAlive ☐ Use multipart/form-data for POST ☐ Browser-compatible headers

Parameters Body Data

同请求一起发送参数:

名称:	值	编码?	包含等于?
-----	---	-----	-------

Detail

添加

Add from Clipboard

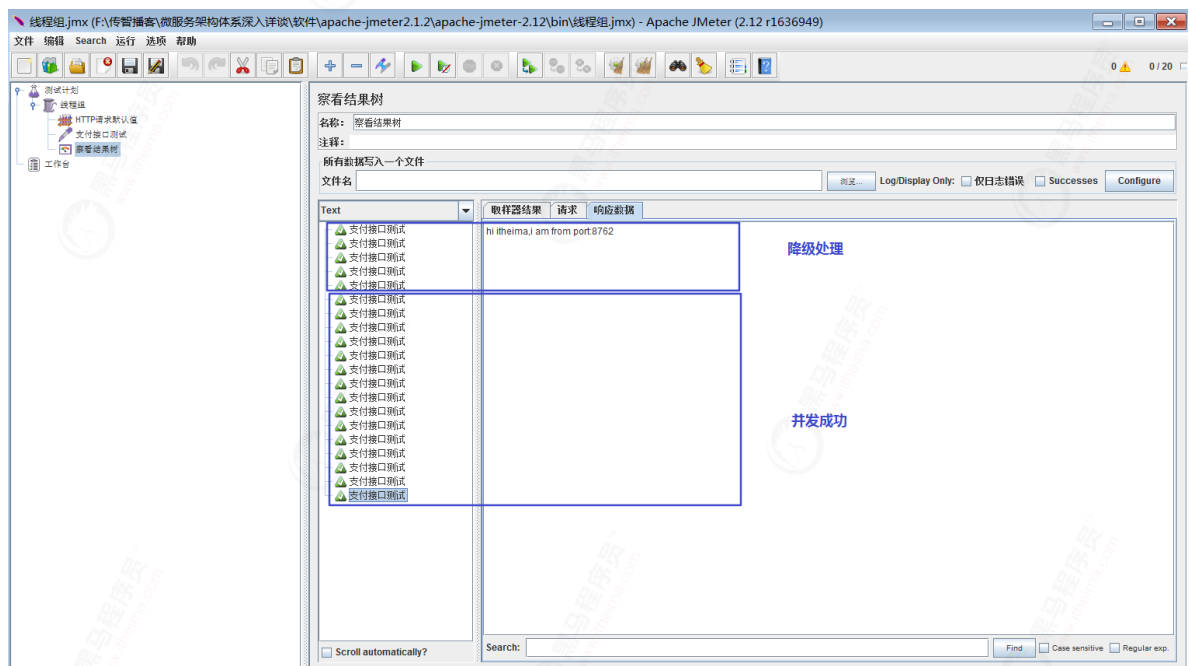
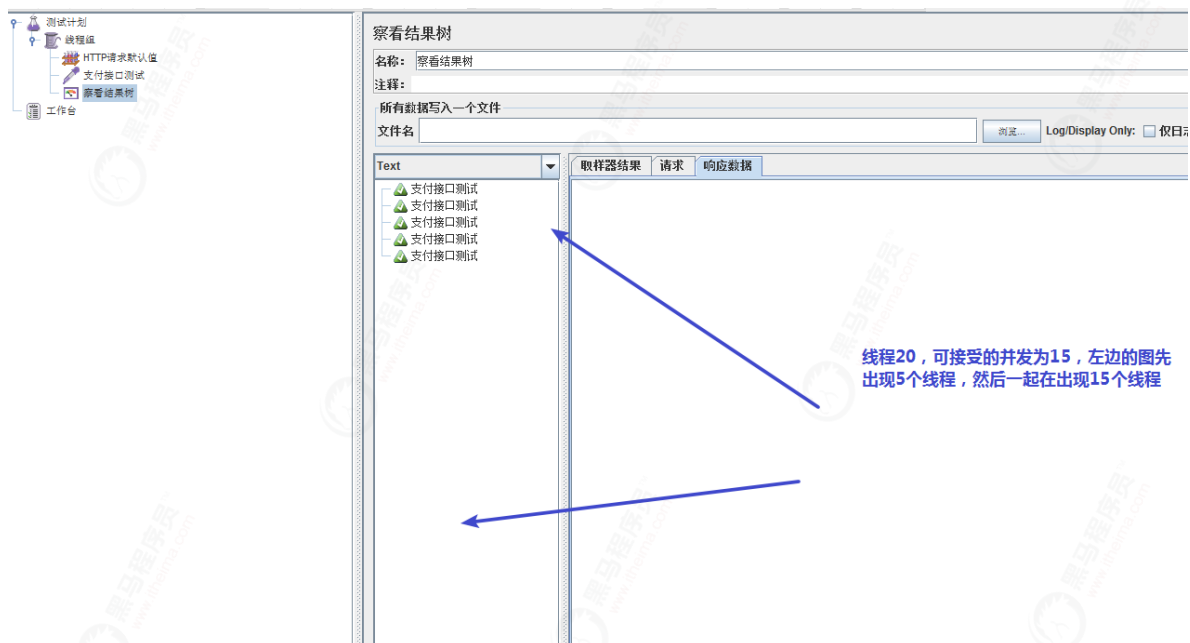
删除

Up

Down

/order?name=itheima

开始执行



思考：

如果并为5呢？

2.2 流量与容错管理之Sentinel

2.2.1 Sentinel 概述



Sentinel

概述

Sentinel是阿里开源的项目，提供了流量控制、熔断降级、系统负载保护等多个维度来保障服务之间的稳定性。

Sentinel的流控操作起来非常简单，在控制台进行配置即可看见效，所见即所得

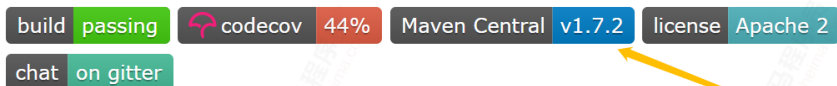
官网

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

中文

<https://github.com/alibaba/Sentinel/wiki/%E4%BB%8B%E7%BB%8D>

<https://sentinelguard.io/zh-cn/docs/introduction.html>



Sentinel 的使用可以分为两个部分：

核心库（Java 客户端）：不依赖任何框架/库，能够运行于 Java 7 及以上的版本的运行时环境，同时对 Dubbo / Spring Cloud 等框架也有较好的支持。

控制台（Dashboard）：控制台主要负责管理推送规则、监控、集群限流分配管理、机器发现等

差异化

	Sentinel	Hystrix
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
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
实时统计实现	滑动窗口（LeapArray）	滑动窗口（基于 RxJava）
动态规则配置	支持多种数据源	支持多种数据源
扩展性	多个扩展点	插件的形式
基于注解的支持	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
流量整形	支持预热模式、匀速器模式、预热排队模式	不支持
系统自适应保护	支持	不支持
控制台	提供开箱即用的控制台，可配置规则、查看秒级监控、机器发现等	简单的监控查看

2.2.2 构建Sentinel控制台

Sentinel构建方式：

- 1、可以从 [release 页面](#) 下载最新版本的控制台 jar 包。
- 2、也可以从最新版本的源码自行构建 Sentinel 控制台：
 - 下载 [控制台](#) 工程
 - 使用以下命令将代码打包成一个 fat jar: `mvn clean package`

1、Sentinel使用Jar构建

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

2、Sentinel启动

```
java -Dserver.port=8787 -Dcsp.sentinel.dashboard.server=192.168.23.139:8787 -Dproject.name=sentinel-dashboard -jar /home/sentinel/sentinel-dashboard-1.7.2.jar
```

- `Dserver.port`
指定控制台的端口
- `-Dcsp.sentinel.dashboard.server`
指定控制台的地址，相当于自己注册自己，这样启动后就能看到自身的信息，向sentinel-dashboard控制台发送心跳包的sentinel-dashboard控制台地址
- `-Dproject.name`
指定接入的应用名称

从 Sentinel 1.6.0 起，Sentinel 控制台引入基本的登录功能，默认用户名和密码都是 `sentinel`

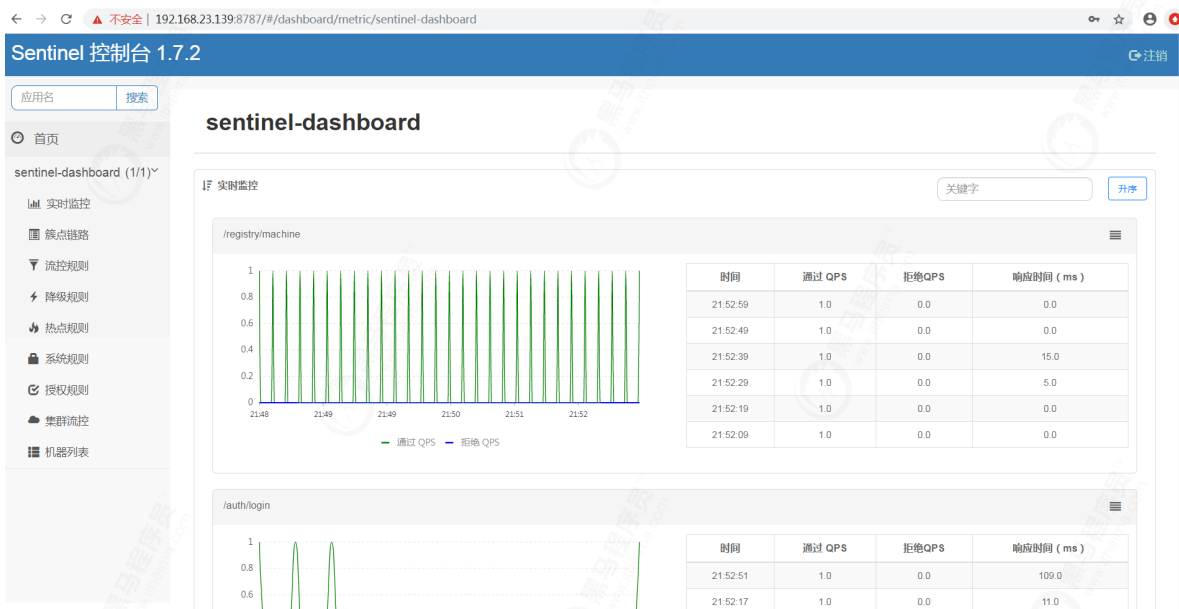
注意：启动 Sentinel 控制台需要 JDK 版本为 1.8 及以上版本

3、访问

<http://192.168.23.139:8787/#/login>



默认用户名密码为sentinel sentinel



2.2.3 SpringCloud集成Sentinel

1、引入起步依赖

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

2、配置控制台

```
server.port=8888
spring.application.name=itheima-sentinel
#####以下部分为Sentinel#####
spring.cloud.sentinel.transport.port=8719
spring.cloud.sentinel.transport.dashboard=192.168.23.139:8787
spring.cloud.nacos.discovery.server-addr=192.168.23.139:8848
management.endpoints.web.exposure.include=*
#####持久化#####
```

`spring.cloud.sentinel.transport.dashboard`:配置sentinel dashboard的访问地址

`spring.cloud.sentinel.transport.port` 应用与Sentinel控制台交互的端口

3、新增控制器

Sentinel加载项目，需要项目访问api发送心跳才能加载

此处先写一个简单的API

```
@GetMapping(value = "/sn/test")
public String hello() {
    return "发送心跳包..";
}
```

执行

http://localhost:8888/sn/test

4、打开sentinel

http://192.168.23.139:8787/#/dashboard/metric/sentinel-dashboard

5、常见问题

监控客户端报错

服务端fetch拉取埋点信息不通

```
2020-07-30 16:53:15.671 ERROR 8852 --- [pool-2-thread-1] c.a.c.s.dashboard.metric.MetricFetcher : fetch metric http://172.16.43.175:8719/metric?startTime=1596099187000&endTime=1596099193000&refresh=false error
java.net.SocketException: Network is unreachable
    at sun.nio.ch.Net.connect0(Native Method) ~[na:1.8.0_252]
    at sun.nio.ch.Net.connect(Net.java:454) ~[na:1.8.0_252]
    at sun.nio.ch.Net.connect(Net.java:446) ~[na:1.8.0_252]
    at sun.nio.ch.SocketChannelImpl.connect(SocketChannelImpl.java:645) ~[na:1.8.0_252]
    at org.apache.http.impl.nio.reactor.DefaultConnectingIOReactor.processSessionRequests(DefaultConnectingIOReactor.java:273) [httpcore-nio-4.4.6.jar!/:4.4.6]
    at org.apache.http.impl.nio.reactor.AbstractMultiworkerIOReactor.execute(AbstractMultiworkerIOReactor.java:348) [httpcore-nio-4.4.6.jar!/:4.4.6]
    at org.apache.http.impl.nio.conn.PoolingNHttpClientConnectionManager.execute(PoolingNHttpClientConnectionManager.java:194) [httpasyncclient-4.1.3.jar!/:4.1.3]
    at org.apache.http.impl.nio.client.CloseableHttpAsyncClientBase$1.run(CloseableHttpAsyncClientBase.java:64) [httpasyncclient-4.1.3.jar!/:4.1.3]
    at java.lang.Thread.run(Thread.java:748) [na:1.8.0_252]
```

虚拟机ping不同宿主机

```
? MobaXterm 20.1 ?
(SSH client, X-server and networking tools)

> SSH session to root@192.168.23.139
? SSH compression : ✓
? SSH-browser : ✓
? X11-forwarding : ✓ (remote display is forwarded through SSH)
? DISPLAY : ✓ (automatically set on remote server)

> For more info, ctrl+click on help or visit our website

Last login: Thu Jul 30 17:10:52 2020 from 192.168.23.1
root@localhost ~]# ping 172.16.43.175
connect: Network is unreachable
root@localhost ~]#
```

宿主机可以ping通虚拟机

```
C:\Users\My>ping 192.168.23.139

正在 Ping 192.168.23.139 具有 32 字节的数据:
来自 192.168.23.139 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.23.139 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.23.139 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.23.139 的回复: 字节=32 时间<1ms TTL=64

192.168.23.139 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间(以毫秒为单位):
        最短 = 0ms, 最长 = 0ms, 平均 = 0ms

C:\Users\My>
```

出现问题的原因

由于虚拟机ping不通宿主机，所以在 获取sentinel客户端指标数据【fetch metric】的时候一直报错

解决方案如下，直到虚拟机可以ping通为止

```
TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=static
IPADDR=192.168.23.129
NETMASK=255.255.255.0
GATEWAY=192.168.23.2
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=ens33
UUID=854d5035-ea3f-4a30-a0e8-364492fbb724
DEVICE=ens33
ONBOOT=true
```

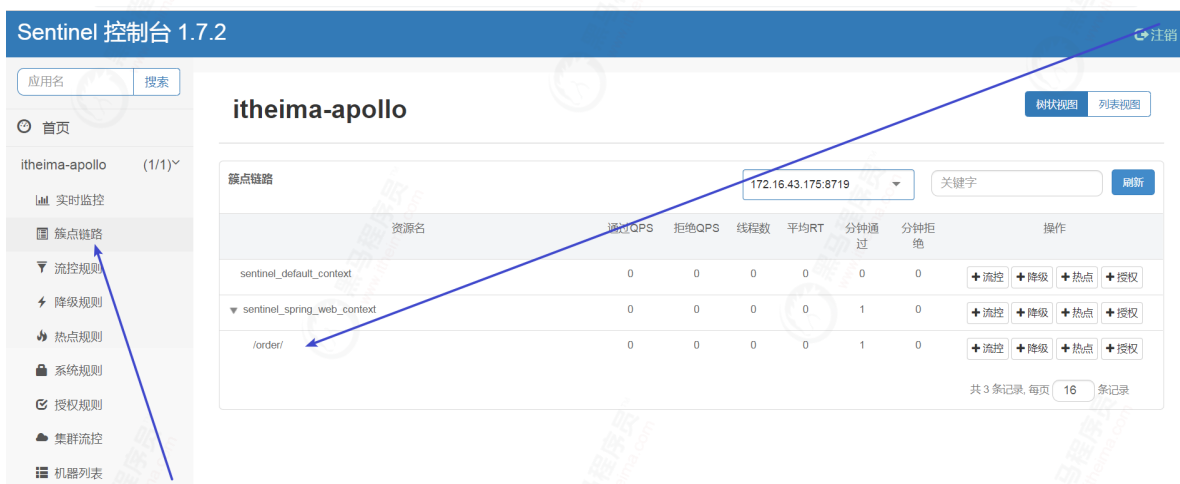
实时监控数据为空

Sentinel簇点链路能找到,但是实时监控一直都是空白状态,一般是sentinel服务器和项目服务器时间不一致

解决方法:把sentinel服务器和项目服务器改为一直,重新启动项目访问接口即可正常



vmware与服务同步后



2.2.4 流量与过载保护规则

1) 流控规则案例

场景：演示下流控规则比较典型、企业级常用的几个组合

点击【新增流控规则】

新增流控规则

资源名

/order

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

新增

取消

流控规则总体介绍

1、资源名： 唯一名称，默认请求路径，表示对该资源进行流控

2、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）

3、阈值类型/单击阈值：

QPS（每秒钟的请求数量）：当调用该api的QPS达到阈值时，进行限流

线程数：当调用该线程数达到阈值的时候，进行限流

4.是否集群： 不需要集群

5.流控模式：

直接： api达到限流条件时，直接限流

关联： 当关联的资源达到阈值时，就限流自己

流控模式

☐ 直接 ☒ 关联 ☐ 链路

关联资源

关联资源

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】

流控模式

☐ 直接 ☐ 关联 ☒ 链路

入口资源

入口资源

6.流控效果:

编辑流控规则

资源名

/hello/{name}

针对来源

default

阈值类型

☒ QPS ☐ 线程数

单机阈值

6

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

预热时长

3

关闭高级选项

保存

取消

快速失败: 直接失败, 抛异常

如果qps=1

这个时候如果在一秒内快速点击/am, 就会发现直接调用了默认的报错信息

Warm Up: 根据codeFactor (冷加载因子, 默认3) 的值, 经过预热时长, 才达到设置的QPS阈值

根据公式可以得到预热前后的qps

(开始阈值) $6/3(\text{预热时长})=2$, 也就是说1秒2次, 预热3秒后, 达到qps=6/秒

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

编辑流控规则

资源名	/hello/{name}		
针对来源	default		
阈值类型	☒ QPS ☐ 线程数	单机阈值	1
是否集群	<input data-bbox="416 465 435 492" type="checkbox"/>		
流控模式	☒ 直接 ☐ 关联 ☐ 链路		
流控效果	☐ 快速失败 ☐ Warm Up ☒ 排队等待		
超时时间	5000		

[关闭高级选项](#)

上面的表示：qps=1次/秒，如果超过了设置的【超时时间】，就排队等待

1、流控模式---直接

tips

创建规则后等待几秒后就可以使用，不用重启

关于流量全部配置到Sentinel服务端

新增流控规则

资源名

/order

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

新增

取消

```
@GetMapping("/order")
public String order() {
    return "订单创建成功...";
}
```

效果：访问/order，1秒内只能访问1次，如果超过【单机阈值】，直接快速失败

这个时候如果在一秒内快速点击/order，就会发现直接调用了默认的报错信息

http://localhost:8888/order

← → ↺ @ localhost:8888/order

Blocked by Sentinel (flow limiting)

Blocked by Sentinel (flow limiting) 是sentinel提供的默认提示语，表示已经被限流了

2、关联

关联：当关联的资源达到阈值时，就限流自己

编辑流控规则

资源名

/order

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

关联资源

/payment

流控效果

☒ 快速失败 ☐ Warm Up ☐ 排队等待

关闭高级选项

保存

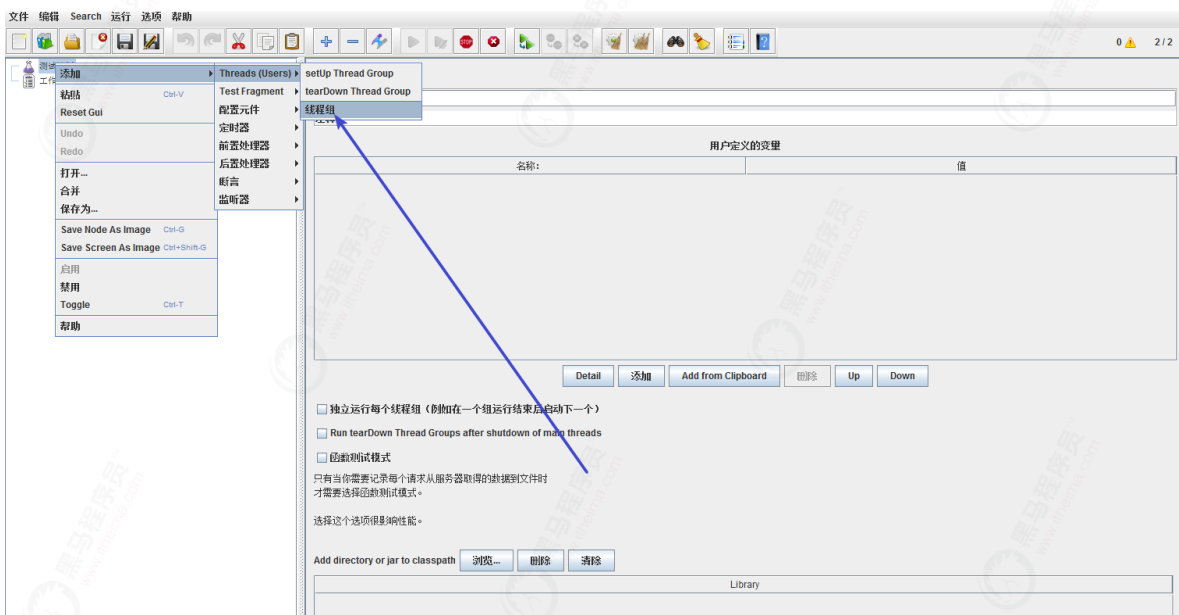
取消

当/payment关联资源访问超过阈值后，则创建订单/order限流

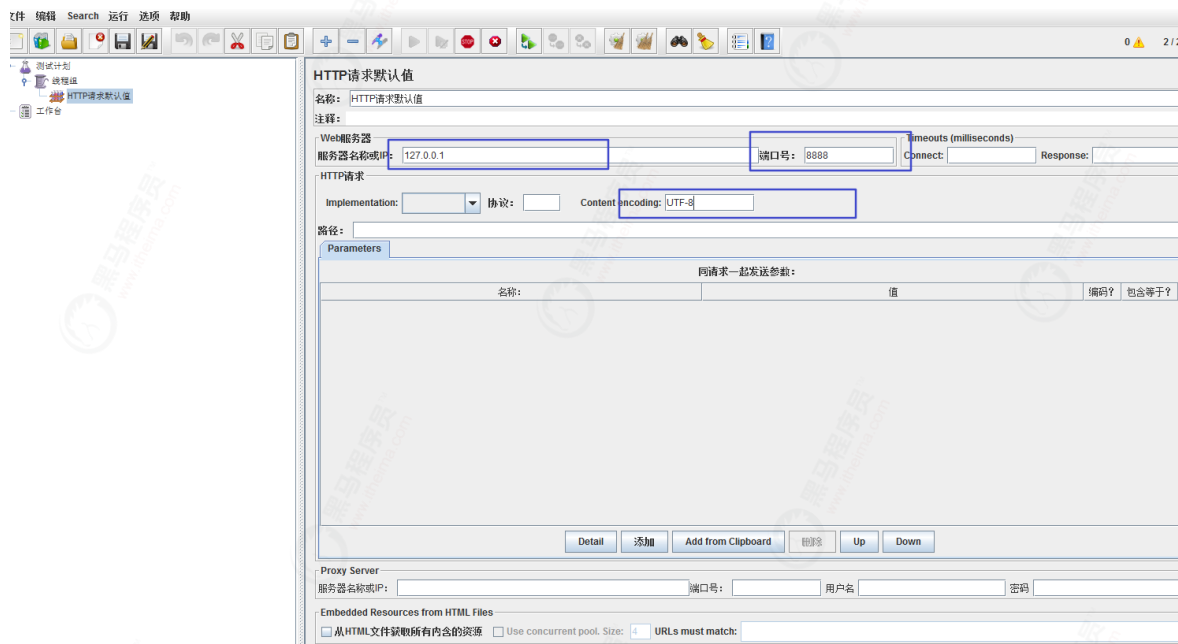
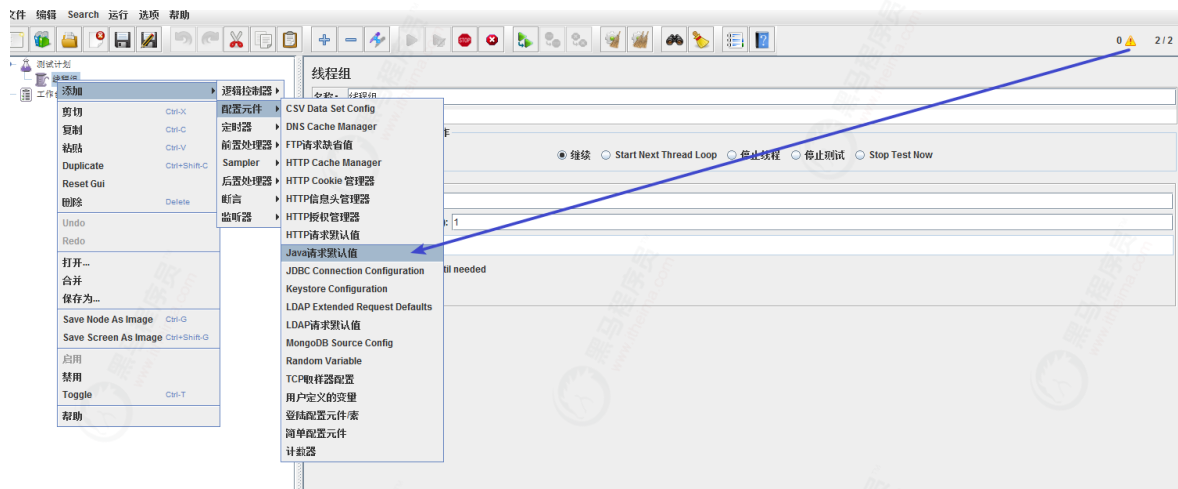
```
@GetMapping("/payment")
public String payment() {
    return "支付成功...";
}
```

Jmeter配置

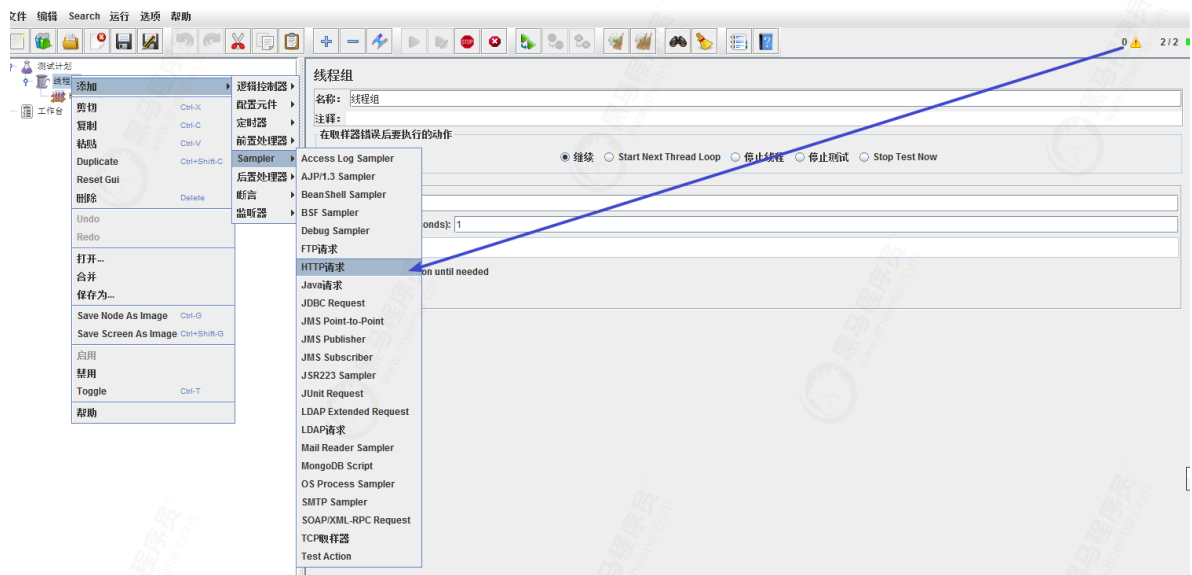
1.创建线程组



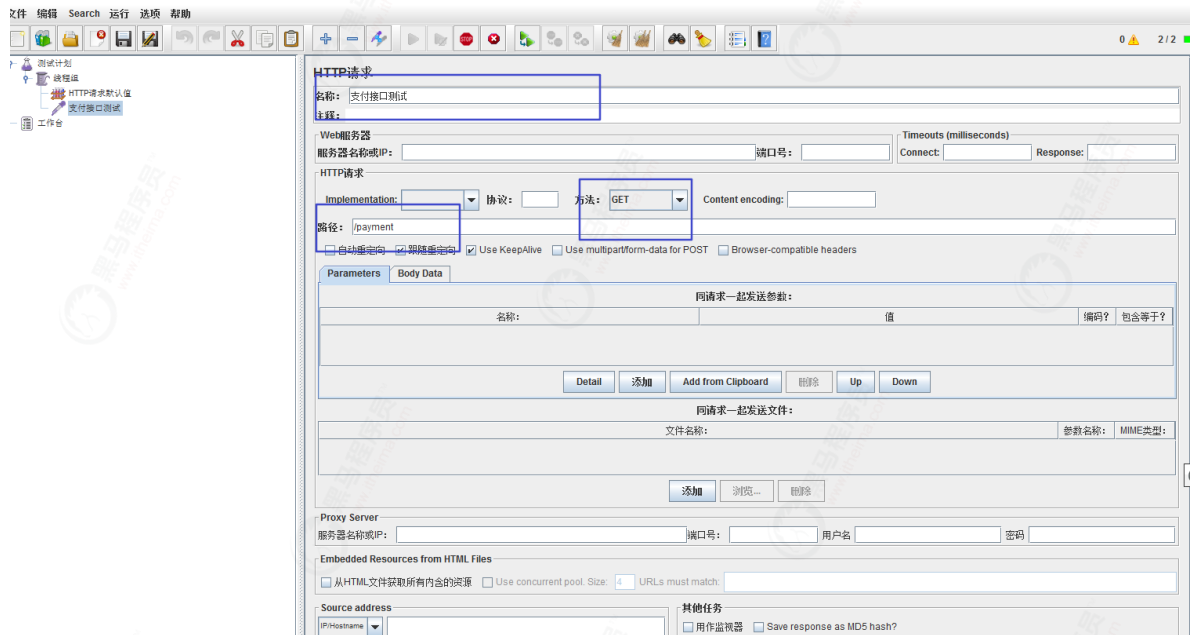
2、创建http请求默认值



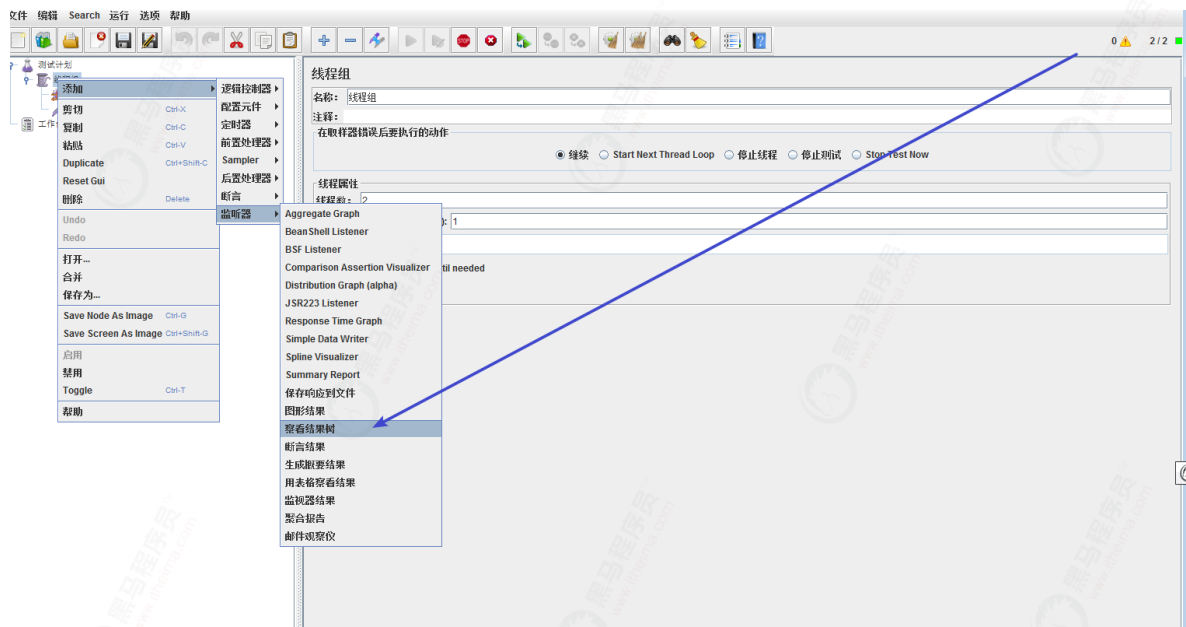
3、创建http请求



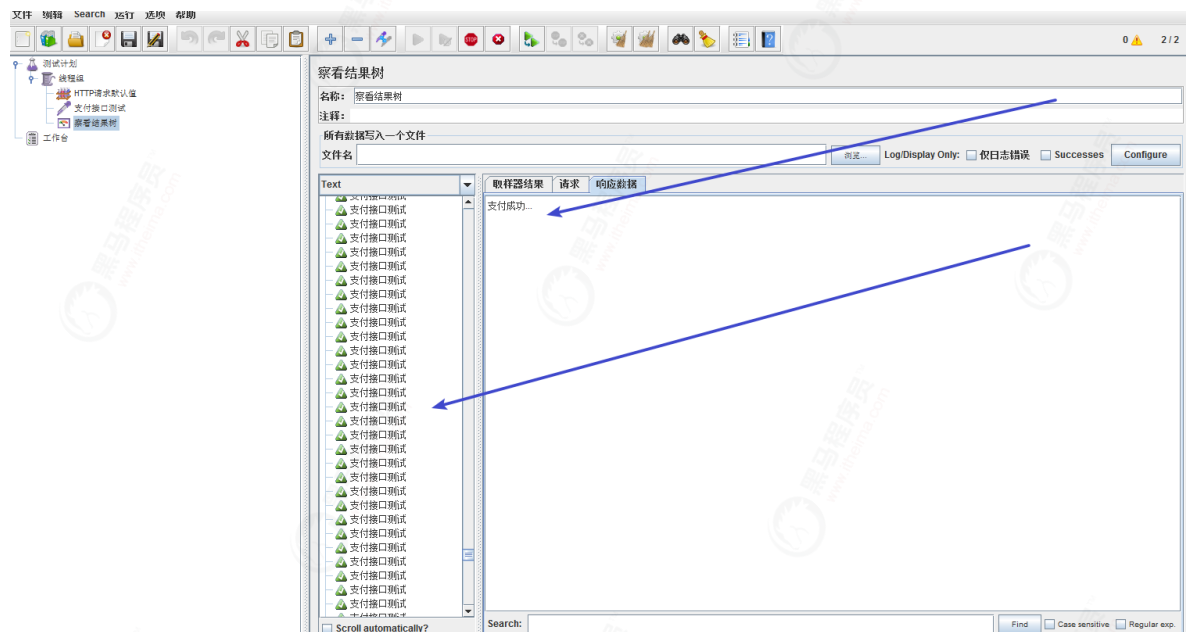
/payment



4、查看结果树



点击【执行】，第四步的【查看结果树】就可以查看到数据了。



这个时候我们在访问【创建订单】

<http://localhost:8888/order>

← → localhost:8888/order

Blocked by Sentinel (flow limiting)

Blocked by Sentinel (flow limiting) 是sentinel提供的默认提示语，表示已经被限流了

2) 降级规则案例

1、RT（平均响应时间）

平均响应时间 (DEGRADE_GRADE_RT): 当 1s 内持续进入 N 个请求，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



Sentinel配置

要触发这个RT服务降级得满足两个条件：

- 1、1秒内进入至少5个请求
- 2、持续进入的5个请求的平均响应时间大于阈值

降级配置

新增降级规则

资源名	/stock		
降级策略	☒ RT ☐ 异常比例 ☐ 异常数		
RT	200	时间窗口	5

```
/*
 * @Description: 降级测试（RT+时间窗口）
 * @Method: stock
 * @Param: []
 * @Update:
 * @since: 1.0.0
 * @Return: java.lang.String
 *
 */
@GetMapping("/stock")
public String stock() {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "库存扣减成功...";
}
```

访问

http://localhost:8888/stock

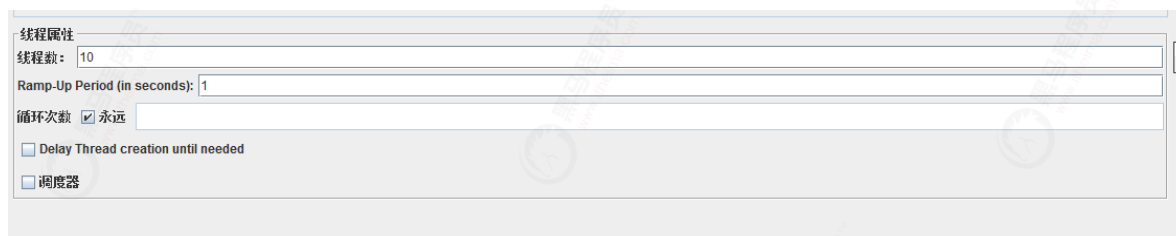
← → ↻ ⓘ localhost:8888/stock 🔍 ☆ ⓘ 🔴

库存扣减成功...

上图，没有使用jmeter前可以正常访问

jmeter设置

每秒进入10个请求



访问

http://localhost:8888/stock

← → ↻ ⓘ localhost:8888/stock 🔍 ☆ ⓘ 🔴

Blocked by Sentinel (flow limiting)

停止jmeter 5秒之后

← → ↻ localhost:8888/stock

库存扣减成功...

改造上面的代码

我们使用 `@SentinelResource(value = "stock")` 注解的方式定义降级资源

```
/*
 * @Description: 降级测试 (RT+时间窗口)
 * @Method: stock
 * @Param: []
 * @Update:
 * @since: 1.0.0
 * @Return: java.lang.String
 *
 */
@GetMapping("/stock")
@SentinelResource(value = "stock")
public String stock() {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "库存扣减成功...";
}
```

编辑降级规则

资源名

stock

注意：没有了/

降级策略

☒ RT

☐ 异常比例

☐ 异常数

RT

200

时间窗口

5

保存

取消

再次执行

http://localhost:8888/stock

Whitelabel Error Page

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

Fri Jul 31 13:41:53 CST 2020

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

No message available

可以看到，上面显示了错误页，不是上面的那一串英文了

原因是因为我们通过SentinelResource注解定义了降级资源

我们没有配置 blockHandler（流控设置）、fallback（异常设置），

刚才在限流降级时才会将 BlockException 直接抛出到了页面

接下来，我们来看如何定义blockHandler（流控设置）、fallback（异常设置）

2、异常比例

异常比例(秒级)：当资源的请求每秒请求量 ≥ 5 ，并且每秒异常总数占通过的比例超过阈值的时候，开始进入降级状态

在下面的时间窗口内，对这个方法的调用都会被限流中。，异常比例的的阈值范围（0.0-1.0）代表0%--100%

新增降级规则

资源名	ex		
降级策略	☐ RT ☒ 异常比例 ☐ 异常数		
异常比例	0.2	时间窗口	3

新增取消

Sentinel的限流、服务降级功能，但是只是限制后，返回不可控的结果肯定是不行的，例如上面那两个

1、显示英文

2、显示异常信息

我们还要保证调用者在调用那些被限制的服务时候，不管是不是被限制，都要让他们拿到一个合理的结果，而不是扔回去一个异常就完事了

Sentinel提供了这样的功能，让我们可以另外定义一个方法来代替被限制或异常服务返回数据，这就是fallback和blockHandler。

fallback: 若本接口出现未知异常，则调用**fallback**指定的接口。

blockHandler: 若本次访问被限流或服务降级，则调用**blockHandler**指定的接口。

fallback 函数位置是有要求的

1、原方法在同一个类中

2、@SentinelResource提供了**通过fallbackClass指定对应的类的Class对象**，也就是说可以不在一个类中

先看在一个类中如何定义

```
/*
 * @Description: 降级规则（异常比例）
 * @Method: exceptionPercent
 * @Param: []
 * @Update:
 * @since: 1.0.0
 * @Return: java.lang.String
 *
 */
@GetMapping("/ex")
@SentinelResource(value = "ex", fallback = "testFallback", blockHandler =
"testBlockHandler")
public String exceptionPercent() {
    int age = 10 / 0;
    return "异常比例达到阈值....";
}

/*
 * @Description: 出现异常调用当前方法
 * @Method: testFallback
 * @Param: [e]
 * @Update:
 * @since: 1.0.0
 * @Return: java.lang.String
 *
 */
public String testFallback(Throwable e) {
    if (e instanceof RuntimeException) {
        log.error(e.getMessage());
    }

    return "异常降级....";
}

/*
 * @Description: 达到流控阈值调用当前方法
 * @Method:
 * @Param:
 * @Update:
 * @since: 1.0.0
 * @Return:
 *
 */
public String testBlockHandler(BlockException e) {
```

```
    return "流控降级（限流）";  
}
```

http://localhost:8888/ex

快速点击，首先出来异常降级，也就是调用了fallback方法，因为代码出现了异常信息

← → ↻ localhost:8888/ex

异常降级....

快速点击后，达到了限流降级的阈值（根据配置，3秒内有20%失败则限流）

← → ↻ localhost:8888/ex

流控降级（限流）

总结

fallback是针对方法出现异常了，则会进入fallback方法。

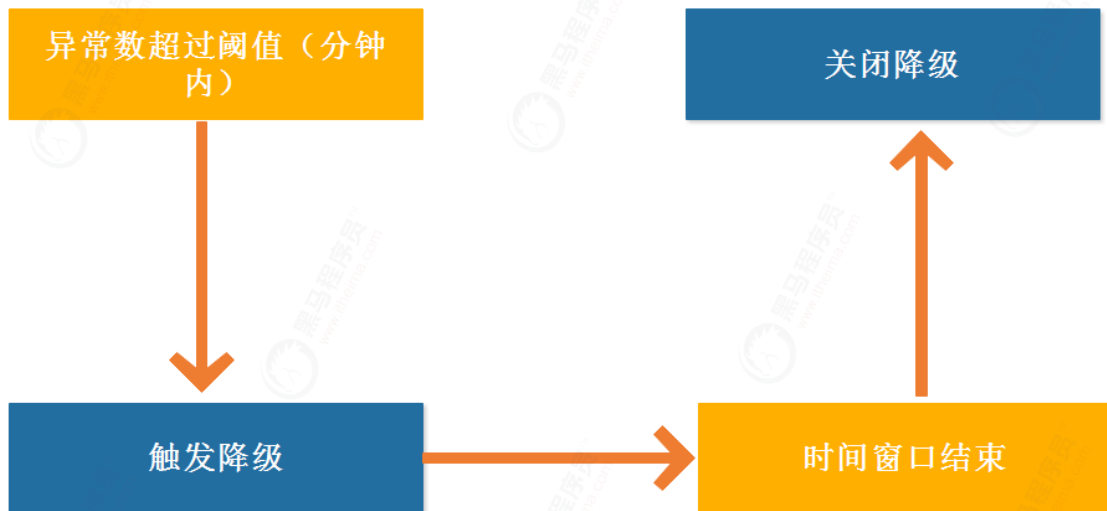
blockhandler是针对流控设置，超出规则，则会进入blockhandler方法。

3、异常数

当资源近1分钟的异常数超过阈值之后会进行降级。

提示：

1 近一分钟的异常数可以理解为，在1分钟内点击第6次的时候会触发降级。如果前5次在第一个60秒内，第6次点击的时候是在第61秒，就不会发生熔断了，异常数只是统计60秒内的累计，下一个60秒从第一个开始累计



例：一分钟内，异常数大于5个，就服务熔断降级，时间窗口61秒内访问，都服务降级，时间窗口过去后，才能正常访问

编辑降级规则

资源名	ex		
降级策略	☐ RT ☐ 异常比例 ☒ 异常数		
异常数	5	时间窗口	61

保存 取消

测试结论：在1分钟以内，点击5次后，再次点击就会返回降级页面，之前5次返回的是错误页面，进入熔断，61s后，断路器关闭，微服务恢复

http://localhost:8888/ex

点击第一次

← → ↻ localhost:8888/ex

异常降级...

点击第6次发生限流降级

← → ↻ localhost:8888/ex

流控降级（限流）

等待61秒时间窗口过去后，再访问：接口又能访问了



通过簇点链路可以查看通过和拒绝情况



上图的【分钟通过】表示正常访问

这里需要注意的是【分钟通过】数据并不是累加的，下一个分钟开始，当前的【分钟通过】清零

上图的【分钟后拒绝】表示受保护的状态，也就是提示Blocked by Sentinel (flow limiting)的那个页面

根据官方wiki文档，sentinel控制台的实时监控数据，默认仅存储 5 分钟以内的数据。如需持久化，需要定制实现相关接口

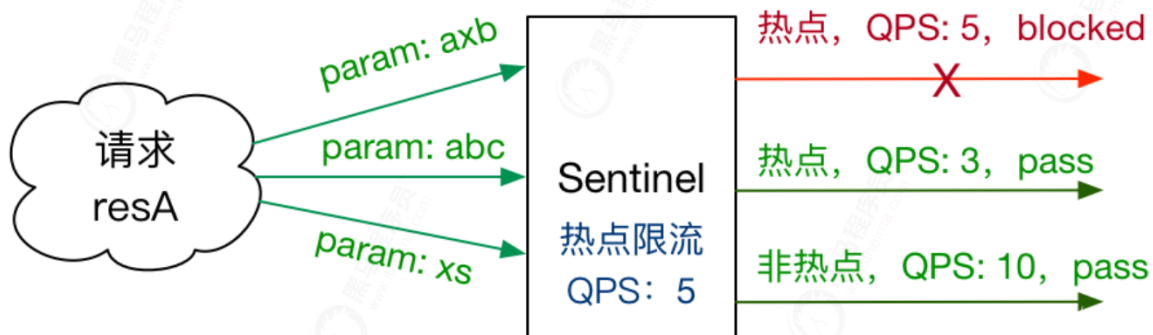
3) 热点规则案例

热点：即经常访问的数据

热点规则的限流模式只有一种 **QPS** 模式

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



编辑热点规则

参数索引：表示对下标第0个参数进行限流

单机阈值：这里限流的qps为1

统计窗口时长：统计的时间长度（时间窗口内发生了多少qps）

编辑热点规则

资源名

hot

限流模式

QPS 模式

参数索引

q

单机阈值

1

统计窗口时长

3

秒

是否集群

☐

参数例外项

参数类型

参数值

例外项参数值

限流阈值

限流阈值

+ 添加

参数值	参数类型	限流阈值	操作
czbk	java.lang.String	10	删除

关闭高级选项

保存

取消

参数类型：表示参数的数据类型

参数值。接口参数的实际值

预留阈值: qps阈值

上图的配置表示:

第0个参数的值为czbk、数据类型为String的的qps不能超过10, 否则限流, 而当前规则不受上面qps为1的限制

如果第0个参数的值是czbk以外的, 受qps=1的限制

tips

需要注意的是参数例外性只支持7种类型

资源名称此处没有【/】符号, 需要特别注意

控制器

```
* @Description: 热点规则
* @Method: hot
* @Param: [id]
* @Update:
* @since: 1.0.0
* @Return: java.lang.String
*
*/
@GetMapping("/{name}")
public String hot(@PathVariable String name) {
    return sentinelService.hot(name);
}
```

service

```
@Service
public class SentinelServiceImpl implements SentinelService {
    static final Logger logger =
        LoggerFactory.getLogger(SentinelServiceImpl.class);

    @Override
    @SentinelResource(value = "hot", fallback = "testFallback", fallbackClass =
        ExceptionUtils.class, blockHandler = "testBlockHandler", blockHandlerClass =
        ExceptionUtils.class)
    public String hot(String name) {
        System.out.println("-----" + name);
        return name;
    }
}
```

异常处理类，此处有好多坑

```
public class ExceptionUtils {  
  
    /**  
     * @Description: 降级方法  
     * 注意（坑）：  
     * 1、name参数必须携带过来否则无法调用  
     * 2、方法必须为static，否则无法调用  
     * @Method: testFallback  
     * @Param: [name]  
     * @Update:  
     * @since: 1.0.0  
     * @Return: java.lang.String  
     */  
    public static String testFallback(String name, Throwable e) {  
        return "异常降级处理...." + name;  
    }  
  
    /**  
     * @Description: 限流方法  
     * 注意（坑）：  
     * 1、name参数必须携带过来否则无法调用testBlockHandler方法  
     * 2、参数必须设置BlockException e参数，否则，流控降级的时候会进入到异常降级方法(前提是异常方法加入了Throwable，当前方法没有加BlockException)  
     * 3、方法必须为static，否则无法调用  
     * @Method: testBlockHandler  
     * @Param: [name]  
     * @Update:  
     * @since: 1.0.0  
     * @Return: java.lang.String  
     */  
    public static String testBlockHandler(String name, BlockException e) {  
        return "流控降级（限流）" + name;  
    }  
}
```

<http://localhost:8888/czbk1>

使用czbk1访问

3秒内超过1个qps发生熔断

http://localhost:8888/czbk

使用参数例外 czbk访问可以访问10次，10次之后在降级

← → ↻ ⓘ localhost:8888/czbk

流控降级（限流）czbk

注意事项

- 当设置热点规则的时候，如果满足了参数例外项，那么触发降级。
- 不满足参数例外项，但却满足简单配置项，也会触发降级。
- 参数例外项优先级高于简单配置项

4) 系统规则案例



Load是负载，只有在linux机器上才会生效。根据当前系统的负载来决定是不是触发保护。

RT：这个应用上所有的流量的平均的响应时间，也就是平均响应时间超过一个值，那么停止接收新的请求，

线程数：所有服务访问的线程数加起来

入口qps：所有服务的qps加起来达到一个值

cpu使用率：cpu的使用率超过一个百分比，

注意

发生以上这些情况的时候把你整个应用给你断掉。所有服务都不提供了，所有请求都被挡住了。

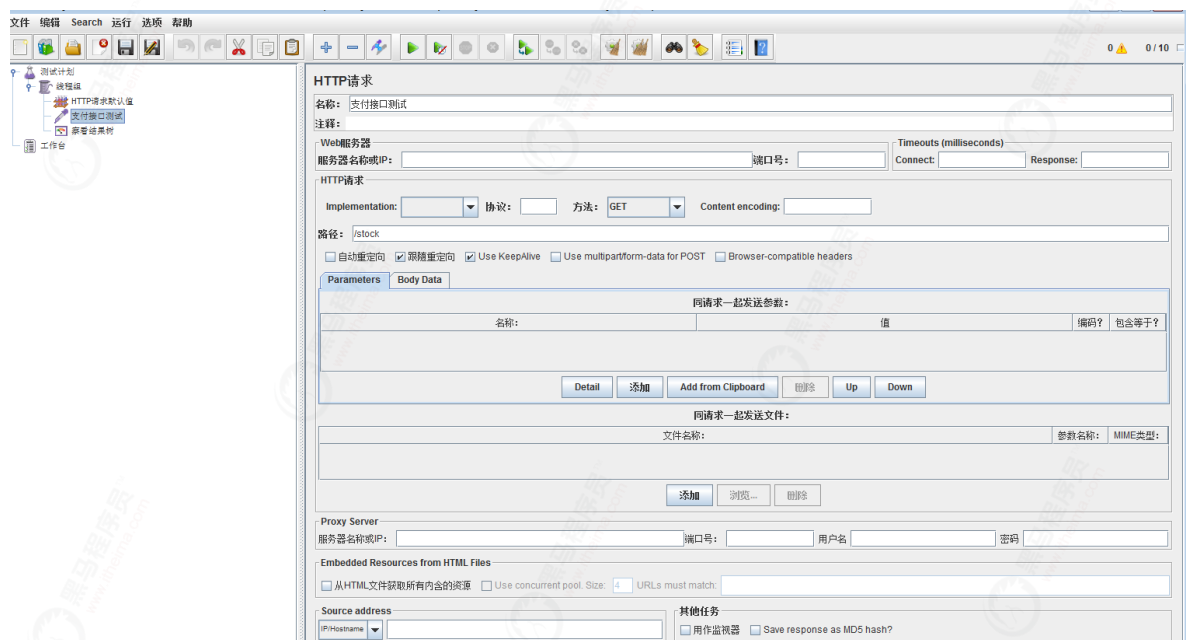
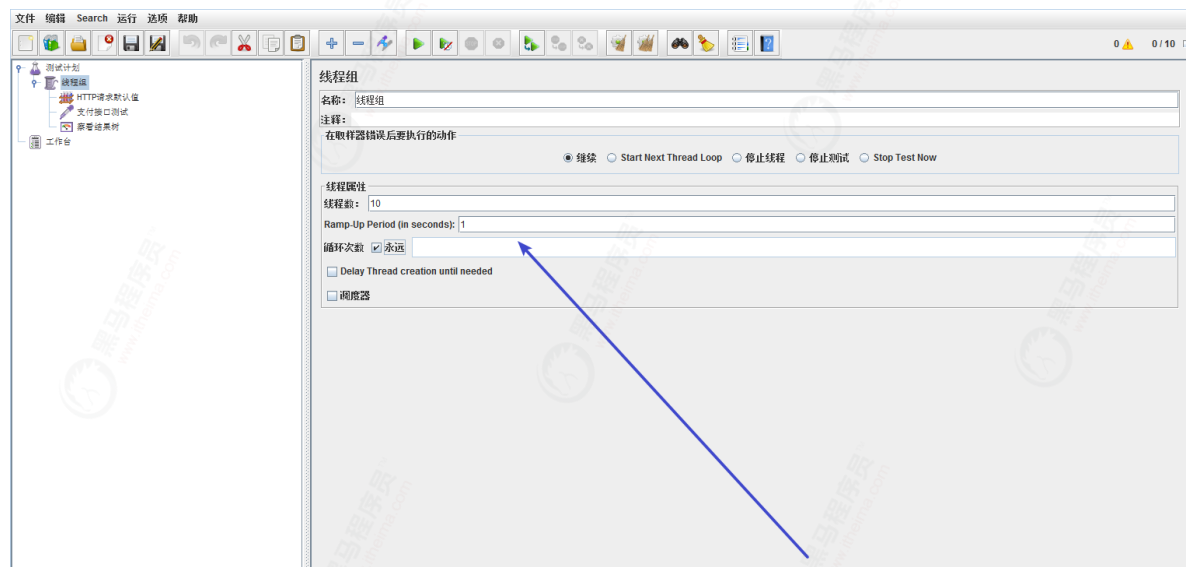
这个设置很少用，因为它太粗了。

举例

线程数（并发数）为5



jmeter设置



访问 (hot)

```
http://localhost:8888/czbk
```

由于并发数超过了5个，所有系统api都无法使用

全局设置不常用

验证

```
http://localhost:8888/order
```

```
http://localhost:8888/czbk
```

5) 授权规则案例

新增授权规则

资源名	资源名称
流控应用	指调用方，多个调用方名称用半角英文逗号(,)分隔
授权类型	☒ 白名单 ☐ 黑名单

黑白名单规则 (AuthorityRule) 非常简单，主要有以下配置项：

- resource：资源名，即限流规则的作用对象
- 流控应用：对应的黑名单/白名单，不同 origin 用 , 分隔，如 appA,appB, default，代表不区分调用来源

流控应用 这个位置要填写的是来源标识，Sentinel提供了 RequestOriginParser 接口来处理来源。只要Sentinel保护的接口资源被访问，Sentinel就会调用 RequestOriginParser 的实现类去解析访问来源

- 授权类型：黑白名单

编辑授权规则

资源名

流控应用

授权类型 ☒ 白名单 ☐ 黑名单

编写控制器

```
/*
 * @Description: 授权规则测试
 * @Method:
 * @Param:
 * @Update:
 * @since: 1.0.0
 * @Return:
 *
 */

@GetMapping("/auth")
public String auth(String auth) {
    return auth;
}
```

增加依赖，否则找不到WebCallbackManager类

```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-web-servlet</artifactId>
</dependency>
```

编写配置类，将自定义规则增加到应用中

```

public class SentinelConfig {

    @PostConstruct
    public void init(){
        //黑白名单

        webCallbackManager.setRequestOriginParser((com.alibaba.csp.sentinel.adapter.servlet.callback.RequestOriginParser) new RequestOriginParserDefinition());
    }

}

```

编写规则，调用来源的ip限制

```

@Component
public class RequestOriginParserDefinition implements RequestOriginParser {
    @Override
    public String parseOrigin(HttpServletRequest httpServletRequest) {
        // 当前 流控应用 放在了请求参数里面，可以放到的地方有很多，比如 参数/请求头/session/等
        //      return httpServletRequest.getParameter("sourceName");
        System.out.println("=====返回调用来源"+httpServletRequest.getRemoteAddr());
        //      return "app";
        return httpServletRequest.getRemoteAddr();
    }
}

```

开始验证

```
http://172.16.43.175:8888/auth?auth=app1
```

172.16.43.175可以正常访问



```
http://127.0.0.1:8888/auth?auth=app1
```

127.0.0.1访问受保护了，因为ip不在白名单

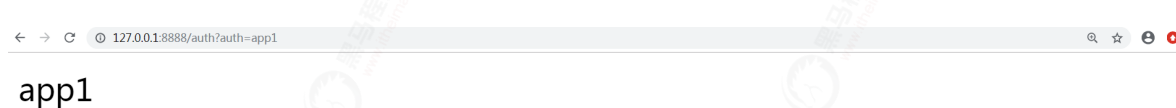


如果将127.0.0.1加入到白名单，就可以访问了

编辑授权规则

资源名	/auth
流控应用	172.16.43.175,127.0.0.1
授权类型	☒ 白名单 ☐ 黑名单

如下



2.2.5 Sentinel规则持久化

Sentinel持久化到Nacos

Sentinel Dashboard中添加的规则是存储在内存中的，只要项目一重启规则就丢失了

此处将规则持久化到nacos中，在nacos中添加规则，然后同步到dashboard中；

1、导入起步依赖

```
<dependency>
  <groupId>com.alibaba.csp</groupId>
  <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

2.在application.properties中配置sentinel-nacos信息

```
spring.cloud.sentinel.datasource.ds1.nacos.server-addr=localhost:8848
spring.cloud.sentinel.datasource.ds1.nacos.data-id=sentinel-nacos-config-stream
spring.cloud.sentinel.datasource.ds1.nacos.group-id=DEFAULT_GROUP
spring.cloud.sentinel.datasource.ds1.nacos.data-type=json
spring.cloud.sentinel.datasource.ds1.nacos.rule-type=flow
```

- `spring.cloud.sentinel.datasource.ds1.nacos.server-addr`: nacos的访问地址, , 根据上面准备工作中启动的实例配置
- `spring.cloud.sentinel.datasource.ds1.nacos.groupId`: nacos中存储规则的groupId
- `spring.cloud.sentinel.datasource.ds1.nacos.data-id`: nacos中存储规则的数据id
- `spring.cloud.sentinel.datasource.ds1.nacos.rule-type`: 该参数是spring cloud alibaba升级到0.2.2之后增加的配置, 用来定义存储的规则类型。所有的规则类型可查看枚举类:
`org.springframework.cloud.alibaba.sentinel.datasource.RuleType`, 每种规则的定义格式可以通过各枚举值中定义的规则对象来查看, 比如限流规则可查看:
`com.alibaba.csp.sentinel.slots.block.flow.FlowRule`

com.alibaba.cloud.sentinel.datasource.RuleType

```
/**
 * flow.
 */
FLOW("flow", FlowRule.class),
/**
 * degrade.
 */
DEGRADE("degrade", DegradeRule.class),
/**
 * param flow.
 */
PARAM_FLOW("param-flow", ParamFlowRule.class),
/**
 * system.
 */
SYSTEM("system", SystemRule.class),
/**
 * authority.
 */
AUTHORITY("authority", AuthorityRule.class),
```

3、nacos增加配置

sentinel-nacos-config-stream

NACOS

首页文档博客社区Ennacos

<

配置管理

配置列表

历史版本

监听查询

服务管理

服务列表

订阅者列表

命名空间

集群管理

节点列表

新建配置

* Data ID: sentinel-nacos-config-stream

* Group: DEFAULT_GROUP

[更多高级选项](#)

描述:

配置格式: ☐ TEXT ☒ JSON ☐ XML ☐ YAML ☐ HTML ☐ Properties

* 配置内容:

```
1 [{
2   {
3     "resource": "/order",
4     "limitApp": "default",
5     "grade": 1,
6     "count": 5,
7     "strategy": 0,
8     "controlBehavior": 0,
9     "clusterMode": false
10  }
11  ]
```

```
[
  {
    "resource": "/order",
    "limitApp": "default",
    "grade": 1,
    "count": 5,
    "strategy": 0,
    "controlBehavior": 0,
    "clusterMode": false
  }
]
```

resource: 资源名称

limitApp: 针对来源。default不区分来源

grade: 限流阈值类型(0-根据并发数量来限流 1-根据QPS来进行流量控制)

count: 单机阈值 (限流阈值)

strategy: 调用关系限流策略, 调用关系限流策略, 选项为资源本身 (0), 关联资源 (1), 链路路口 (2)

controlBehavior: 流量控制效果(直接拒绝、WarmUP、匀速排队)

clusterMode: 是否集群

流控

Field	说明	默认值
resource	资源名，资源名是限流规则的作用对象	
count	限流阈值	
grade	限流阈值类型，是按照 QPS 还是线程数	QPS 模式
limitApp	是否根据调用者来限流	否
strategy	判断的根据是资源自身，还是根据其它资源 (<code>refResource</code>)，还是根据链路入口 (<code>refResource</code>)	根据资源本身
controlBehavior	发生拦截后的流量整形和控制策略 (直接拒绝 / 排队等待 / 慢启动模式)	直接拒绝

降级

Field	说明	默认值
resource	资源名，资源名是限流规则的作用对象	
count	限流阈值	
grade	降级模式，根据 RT 降级还是根据异常比例降级	RT
timeWindow	降级的时间	

热点

属性	说明	默认值
resource	资源名，必填	
count	限流阈值，必填	
grade	限流模式	QPS 模式
durationInSec	统计窗口时间长度 (单位为秒)，1.6.0 版本开始支持	1s
controlBehavior	流控效果 (支持快速失败和匀速排队模式)，1.6.0 版本开始支持	快速失败
maxQueueingTimeMs	最大排队等待时长 (仅在匀速排队模式生效)，1.6.0 版本开始支持	0ms
paramIdx	热点参数的索引，必填，对应 <code>SphU.entry(xxx, args)</code> 中的参数索引位置	
paramFlowItemList	参数例外项，可以针对指定的参数值单独设置限流阈值，不受前面 <code>count</code> 阈值的限制。仅支持基本类型	
clusterMode	是否是集群参数流控规则	false
clusterConfig	集群流控相关配置	

系统

Field	说明	默认值
resource	资源名，即限流规则的作用对象	-
limitApp	对应的黑名单/白名单，不同 origin 用 <code>,</code> 分隔，如 <code>appA, appB</code>	default，代表不区分调用来源
strategy	限制模式， <code>AUTHORITY_WHITE</code> 为白名单模式， <code>AUTHORITY_BLACK</code> 为黑名单模式，默认为白名单模式	AUTHORITY_WHITE

授权

Field	说明	默认值
resource	资源名，即限流规则的作用对象	-
limitApp	对应的黑名单/白名单，不同 origin 用 , 分隔，如 <code>appA,appB</code>	default，代表不区分调用来源
strategy	限制模式， <code>AUTHORITY_WHITE</code> 为白名单模式， <code>AUTHORITY_BLACK</code> 为黑名单模式，默认为白名单模式	<code>AUTHORITY_WHITE</code>

4、重启后访问Sentinel

`http://localhost:8888/order`

访问几次接口之后，就可以在Sentinel Dashboard 中看到在nacos中配置的规则信息了，并且项目服务重启依然存在

Sentinel 控制台 1.7.2注销

应用名 搜索

首页

itheima-apollo (1/1)

实时监控

簇点链路

流控规则

降级规则

热点规则

系统规则

授权规则

集群流控

机器列表

itheima-apollo 新增流控规则

流控规则

172.16.43.175:8719 关键字 刷新

资源名	来源应用	流控模式	阈值类型	阈值	阈值模式	流控效果	操作
/order	default	直接	QPS	5	单机	快速失败	编辑 删除

共 1 条记录, 每页 10 条记录

重要：

在nacos控制台修改配置，刷新sentinel控制台后是更新后的信息，同时会自动推送给应用程序，使配置生效。

而且是实时推送

带来的问题

对于接口的限流规则可以通过两个地方修改：Sentinel控制台、Nacos控制台

- Sentinel控制台中修改规则：仅存在于服务的内存中，不会修改Nacos中的配置值，重启后恢复原来的值。
- Nacos控制台中修改规则：服务的内存中规则会更新，Nacos中持久化规则也会更新，重启后依然保持

3 微服务下声明式服务调用

3.1 Feign声明式调用概述

Feign简介

Feign 是一个http请求调用的轻量级框架，可以以Java接口注解的方式调用Http请求

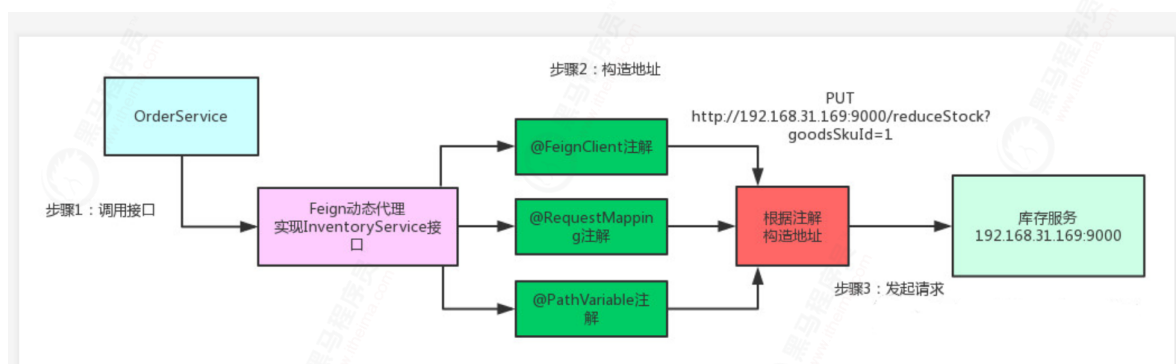
Feign通过处理注解，将请求模板化，当实际调用的时候，传入参数，根据参数再应用到请求上，进而转化成真正的请求，这种请求相对而言比较直观

Feign解决了哪些问题

开发者只需要关注Feign注解模板的开发，而不用关注Http 请求本身，简化了Http 请求的过程，使得Http请求变得简单和容易理解

3.2 Feign工作原理及流程

Feign调用流程



Feign工作原理

- 首先通过@EnableFeignCleints注解开启FeignCleint
- 根据Feign的规则实现接口，并加@FeignCleint注解
- 程序启动后，会进行包扫描，扫描所有的@ FeignCleint的注解的类，并将这些信息注入到ioc容器中。
- 当接口的方法被调用，通过jdk的代理，来生成具体的RequesTemplate

- RequestTemplate在生成Request
- Request交给Client去处理，其中Client可以是URLConnection、HttpClient也可以是Okhttp
- 最后Client被封装到LoadBalanceClient类，这个类结合类Ribbon做到了负载均衡。

3.3 声明式调用与熔断

本章节场景：

我们将建立3个Module，父项目是itheima-service-openfeign，里面没有任何代码，它用来管理我们的聚合工程。

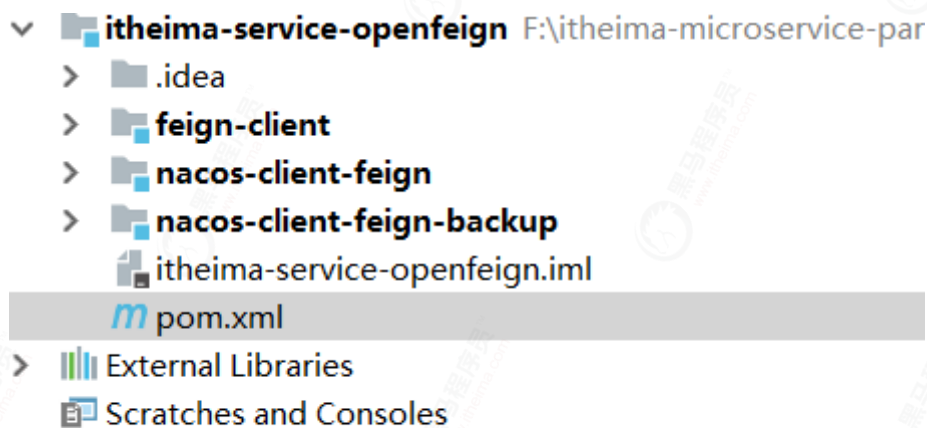
```
<modules>
  <module>nacos-client-feign</module>
  <module>nacos-client-feign-backup</module>
  <module>feign-client</module>
</modules>
```

nacos-client-feign、nacos-client-feign-backup是服务注册中心客户端

feign-client是服务调用的具体实现

具体场景：

将nacos-client-feign、nacos-client-feign-backup注册到服务端，通过feign-client实现声明式服务调用



1. 创建服务调用模块

创建feign-client模块，使用Feign来远程调用其他微服务

2. 引入Feign起步依赖

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

3. 修改application.yml

```
spring:
  application:
    name: feign-client
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.23.139:8848
  server:
    port: 8765
  feign:
    hystrix:
      enabled: true
#nacos-client-feign:
# ribbon:
#   NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule

#ribbon:
# #建立连接之后，读取响应资源超时时间
# ReadTimeout: 5000
# #建立连接超时时间
# ConnectTimeout: 500
#hystrix:
# command:
#   default:
#     execution:
#       timeout:
#         enabled:
# isolation:
#   thread:
#     timeoutInMilliseconds: 5000
```

4. 开启Feign服务调用

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class NacosFeignClientApplication {

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

}
```

5. 创建Feign调用接口

```

@FeignClient(value = "nacos-client-feign", fallback = NacosClientHystrix.class)
public interface NacosClientFeign {

    @GetMapping(value = "/stock")
    String getStock(@RequestParam(value = "name") String name);

}

```

熔断类

```

@Component
public class NacosClientHystrix implements NacosClientFeign {

    @Override
    public String getStock(String name) {
        return "进入Hystrix熔断方法";
    }

}

```

7. 创建Feign调用类

```

@Service
public class FeignService {

    @Autowired
    NacosClientFeign nacosClientFeign;

    public String feign(String name) {
        return nacosClientFeign.getStock(name);
    }

}

```

8. 访问入口

```

@RestController
public class FeignController {

    @Autowired
    FeignService feignService;

    @GetMapping("/feign")
    public String feign(@RequestParam(defaultValue = "itheima", required = false)
String name) {
        return feignService.feign(name);
    }

}

```

9. 启动

启动nacos-client-feign、nacos-client-feign-backup、feign-client

NACOS

首页 文档 博客 社区 En nacos

NACOS 1.3.1

配置管理 服务管理 服务列表 订阅者列表 权限控制 命名空间 集群管理

public

服务列表 | public

服务名称 请输入服务名称 分组名称 请输入分组名称 隐藏空服务: 查询 创建服务

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-client-feign	DEFAULT_GROUP	1	2	2	false	详情 示例代码 删除
feign-client	DEFAULT_GROUP	1	1	1	false	详情 示例代码 删除

3.3.1 负载均衡验证

http://localhost:8765/feign

hi itheima,i am from port:8763

通过Feign进行服务与服务之间调用会轮流（负载均衡：默认轮询）显示以下信息

hi itheima,i am from port:8763
hi itheima,i am from port:8769

在yaml文件中修改下负载均衡策略为随机

```
#nacos-client-feign:  
# ribbon:  
#   NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

发现变成了随机调用

hi itheima,i am from port:8769

3.3.2 断服与超时熔断验证

1、停止nacos-client-feign所有实例

← → ↺ ④ localhost:8765/hi

进入Hystrix熔断方法

2、超时设置

调用方设置

```
spring:
  application:
    name: feign-client
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.23.139:8848
  server:
    port: 8765
  feign:
    hystrix:
      enabled: true
#nacos-client-feign:
# ribbon:
#   NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule

ribbon:
  #建立连接之后，读取响应资源超时时间
  ReadTimeout: 50000
  #建立连接超时时间
  ConnectTimeout: 500
hystrix:
  command:
    default:
      execution:
        timeout:
          enabled: true
      isolation:
        thread:
          timeoutInMilliseconds: 50000
```

3.4 熔断高频出错问题

问题

在进行feign调用的是一直发生熔断

feign和hystrix的超时时间都调整了，就是一直不起作用

1、熔断产生的原因

openFeign默认支持Ribbon（软负载）和Hystrix（熔断）

1)关于Ribbon（软负载）

而Ribbon最重要的两个超时包含：

连接超时ConnectTimeout

读超时ReadTimeout

默认情况下，也就是没有任何配置下，Feign的超时时间会被Ribbon覆盖，两个超时时间都是1秒

如果调用的逻辑复杂，用时超过1秒，肯定会发生熔断现象（读超时ReadTimeout引起）

2)关于Hystrix（熔断）

如果Hystrix也开启（可以通过配置文件自定义）了超时，也需要将它的超时时间设置一下，

否则会 and 上面的Ribbon一样，会覆盖掉Feign

2、熔断解决方案一

在调用方的yaml配置文件中添加配置

ribbon:

#建立连接之后，读取响应资源超时时间

ReadTimeout: 5000

#建立连接超时时间

ConnectTimeout: 500

hystrix:

command:

default:

execution:

timeout:

enabled: true

isolation:

thread:

timeoutInMilliseconds: 5000

注意：

上面ReadTimeout: 5000和timeoutInMilliseconds: 5000

一定要大于业务逻辑处理时间，比如你业务处理为10秒，此处就应该大于10秒，否则还会出现熔断

如果上面的 hystrix.command.execution.timeout.enabled设置为false，那么

timeoutInMilliseconds的值就可以不用设置了

3、熔断解决方案二

在调用方的yaml配置文件中添加配置

hystrix:

command:

default:

execution:

timeout:

enabled: true

isolation:

thread:

timeoutInMilliseconds: 5000

feign:

hystrix:

enabled: true

client:

config:

#这里也可以填写自己的业务服务名称也可以填default，表示所有服务生效

default:


```
#只关注readTimeout  
connectTimeout: 500  
readTimeout: 5000
```

注意：**feign**的配置会覆盖**ribbon**，也就是说**Feign**配置会优先于**Ribbon**配置。配置完之后也是同样的效果

- 1、默认情况下，也就是没有任何配置下，Ribbon会覆盖Feign，超时时间都是1秒
- 2、如果Hystrix+ribbon开启它也会覆盖Feign
- 3、Hystrix+feign的配置会覆盖ribbon，