第9章 微服务架构基础

广东财经大学信息学院
罗 东 俊 博士
ZSUJONE@126.COM

（内部资料，请勿外传）

目的和要求

- 了解微服务的基本概念和常见的解决方案。
- 了解**Spring Cloud**和**Spring Cloud Alibaba**的基本组件及其功能。
- 掌握微服务注册、配置和调用原理和实践方法。
- 掌握微服务容错原理和实践方法。
- 掌握微服务网关原理和实践方法。
- 掌握微服务分布式事务原理和实践方法。
- 掌握微服务链路追踪原理和实践方法。

主要内容

- 9.1 微服务概述
- 9.2 微服务注册、配置和调用
- 9.3 微服务容错
- 9.4 微服务网关
- 9.5 微服务分布式事务
- 9.6 微服务链路追踪

9.3 微服务容错

- 9.3.1 概述
- 9.3.2 使用Sentinel中间件

9.3.1 概述

- 1.微服务容错简介
- 2.Sentinel简介

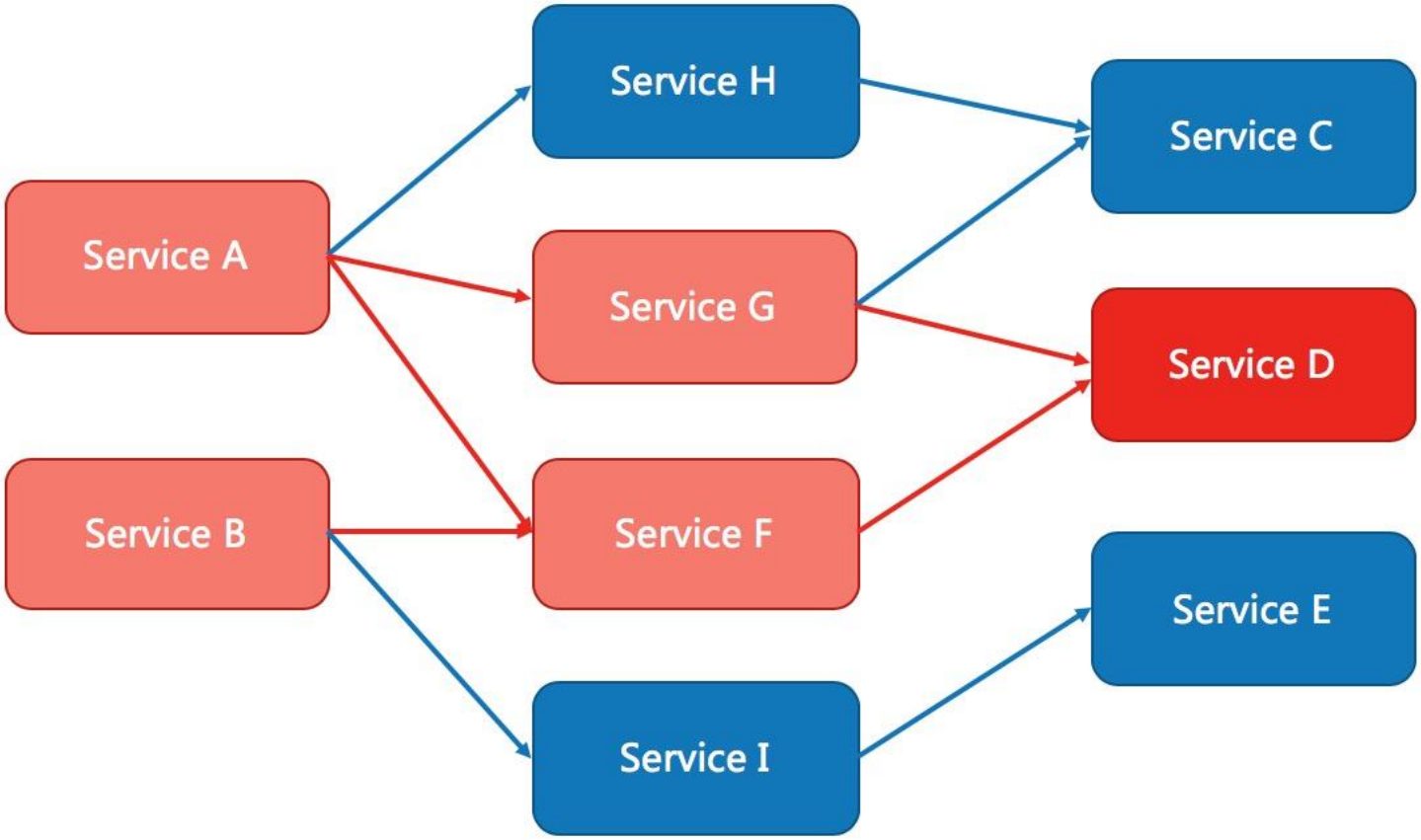
1.微服务容错简介

- 1）雪崩效应
- 2）常见容错思路
- 3）常见容错产品

1）雪崩效应

- 在分布式系统中，由于网络原因或自身的原因，服务一般无法保证100%可用。
  - ✧如果一个服务出现了问题，调用这个服务的线程就会出现阻塞情况，此时若有大量的请求涌入，就会出现多条线程阻塞等待，进而导致服务瘫痪。
- 由于服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩效应”。

服务调用关系示例


Service H
Service C
Service A
Service G
Service D
Service B
Service F
Service E
Service I

2）常见容错思路

- ①限流
- ②隔离
- ③熔断
- ④降级
- ⑤超时

①限流

- 限流就是限制系统的输入和输出流量。
- 常见的限流算法：
  - ✧漏桶算法
  - ✧令牌桶算法
  - ✧滑动时间窗口算法

漏桶算法

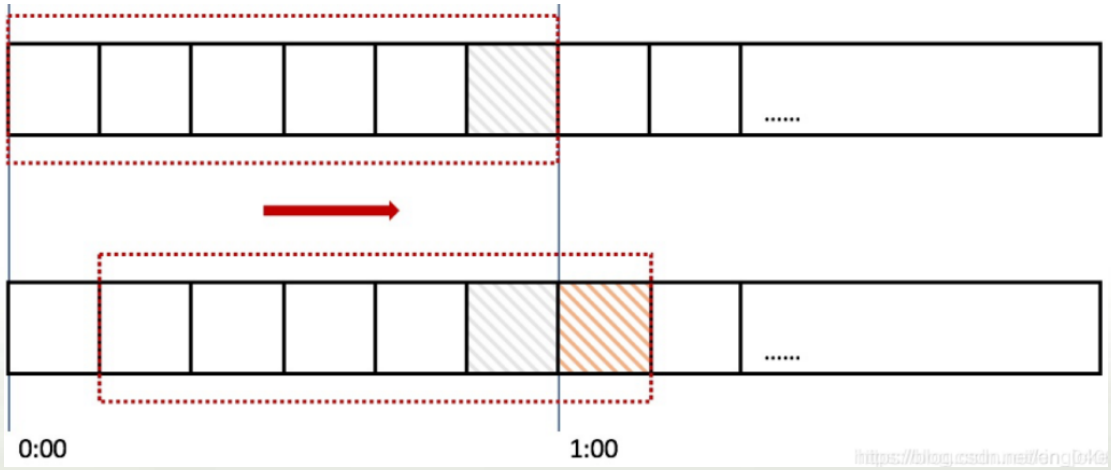
- 漏桶算法的思路：一个固定容量漏桶中的水总按照固定速率流出。
  - ✧如果桶是空的则不需要流出水滴；
  - ✧流入漏桶的水可以以任意速率流入，一旦超出桶的容量，则溢出（被丢弃）。

令牌桶算法

- 基本思路：
  - ✧令牌桶用来存放固定数量的令牌，并以一定速率往桶中放令牌，如果桶中令牌数达到上限，就丢弃令牌；
  - ✧每次请求调用需先获取令牌，只有拿到令牌，才有机会继续执行，否则选择等待可用令牌、或者直接被拒绝。
- 令牌桶算法是对漏桶算法的一种改进，能够在限制调用的平均速率同时还允许一定程度的突发调用（如：在令牌上限时）。

滑动时间窗口算法

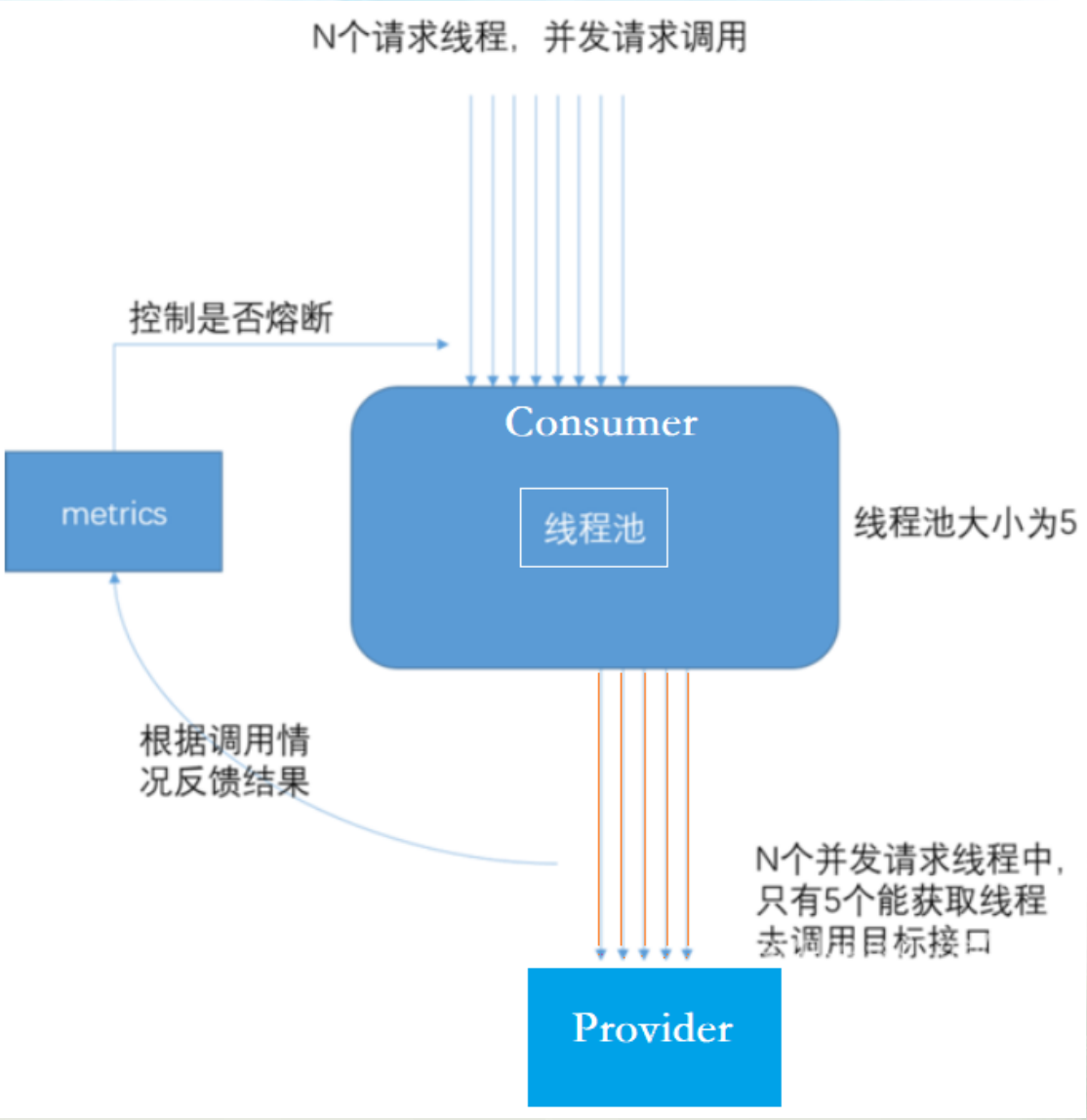
- 该算法是在一个固定长度滑动窗口时间内进行统计限流。
- 窗口移动方式为：
  - ✧窗口开始时间片为上一窗口的第二时间片，结束时间片为上一窗口的下一时间片。



②隔离

- 隔离是指当某个模块有故障发生时，能将问题和影响隔离在该模块内部，而不波及其它模块，不影响整体的系统服务。
- 常见的隔离方式有：
  - ✧线程池隔离
  - ✧信号量隔离

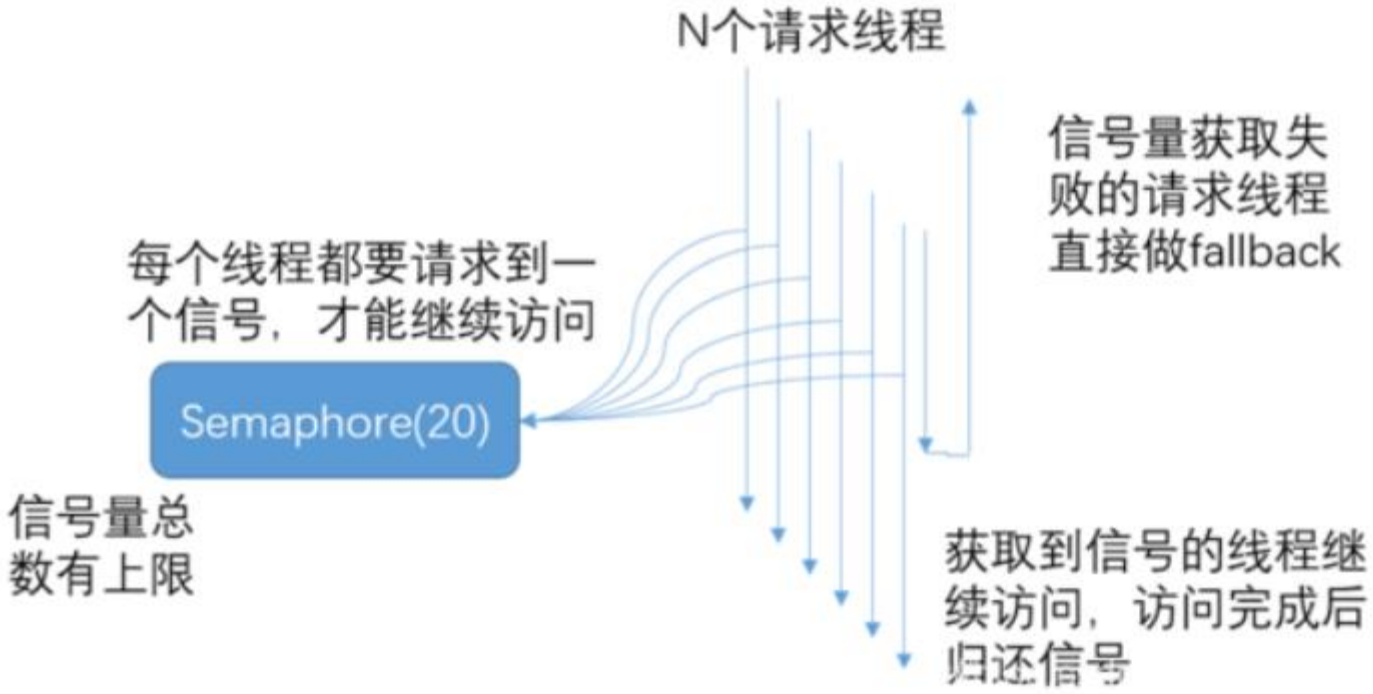
线程池隔离


N个请求线程，并发请求调用
控制是否熔断
metrics
Consumer
线程池
线程池大小为5
根据调用情
况反馈结果
N个并发请求线程中，
只有5个能获取线程
去调用目标接口
Provider

线程池隔离

- 线程池隔离基本思想是：
  - 每个Consumer都设计有一个线程池，线程池有固定大小n；
  - 大量用户请求线程到达时，只有n个请求线程才能拿到Consumer线程池中的线程去调用目标接口；
  - 而后Consumer根据调用情况反馈结果，控制是否熔断。
  - 这样，即使Provider发生故障也不会导致大量用户请求堆积在Consumer而使整个系统崩溃，实现了把故障只隔离在Provider的效果。

信号量隔离


N个请求线程
信号量获取失败的请求线程直接做fallback
每个线程都要请求到一个信号，才能继续访问
Semaphore(20)
信号量总数有上限
获取到信号的线程继续访问，访问完成后归还信号

信号量隔离

- 信号量隔离基本思想是：
  - ✧每个Consumer都设计有一个信号量，信号量的信号总数有固定上限n；
  - ✧大量用户请求线程到达时，每个线程都要请求得到一个信号才能继续访问，访问完成后归还信号；信号获取失败的请求线程直接做fallback。
  - ✧这样，即使Provider发生故障也不会导致大量用户请求堆积在Consumer而使整个系统崩溃，实现了把故障只隔离在Provider的效果。

两种隔离方式的区别

线程池和信号量的区别？		
	线程池隔离	信号量隔离
线程	请求线程和调用provider线程不是同一条线程	请求线程和调用provider线程是同一条线程
开销	排队、调度、上下文开销等	无线程切换，开销低
异步	支持	不支持
并发支持	支持（最大线程池大小）	支持（最大信号量上限）
传递Header	无法传递http Header	可以传递http Header
支持超时	能支持超时	不支持超时

1.什么情况下，用线程池隔离？

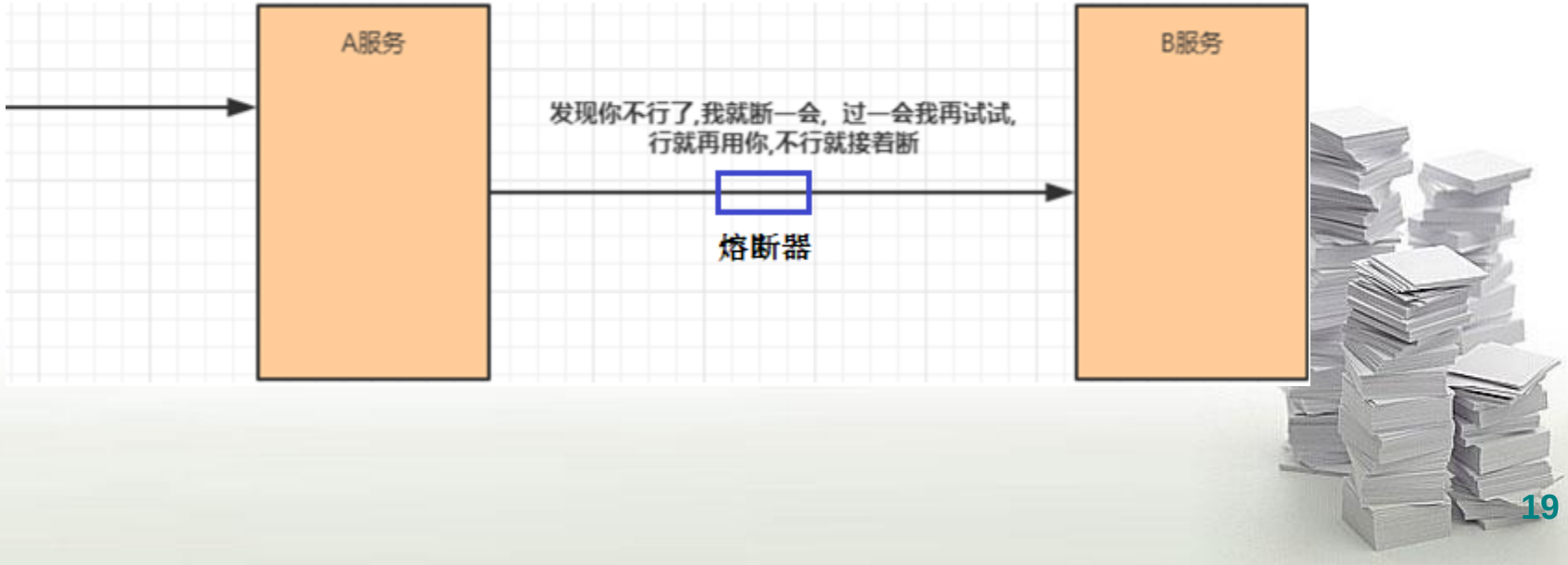
请求并发量大，并且**耗时长**（请求耗时长一般是计算量大，或读数据库）：采用线程隔离策略，这样的话，可以保证大量的容器(tomcat)线程可用，不会由于服务原因，一直处于阻塞或等待状态，快速失败返回。

2.什么情况下，用信号量隔离？

请求并发量大，并且**耗时短**（请求耗时短可能是计算量小，或读缓存）：采用信号量隔离策略，因为这类服务的返回通常会非常的快，不会占用容器线程太长时间，而且也减少了线程切换的一些开销，提高了缓存服务的效率。

③熔断

- 熔断是指当下游服务因访问压力过大而响应变慢或失败，上游服务为了保护系统整体的可用性，可以暂时切断对下游服务的调用。
- 特点：牺牲局部，保全整体。

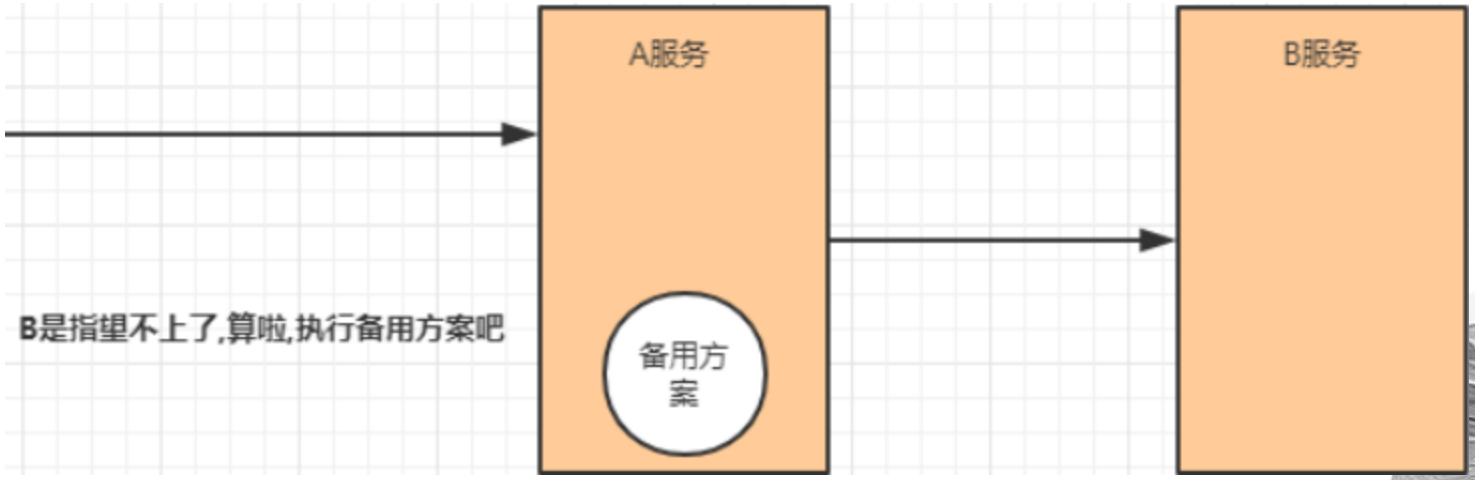


熔断器的三种状态

- 服务熔断一般有三种状态：
  - ✧熔断关闭状态（Closed）
    - 服务没有故障时，熔断器所处的状态，对调用方的调用不做任何限制。
  - ✧熔断开启状态（Open）
    - 后续对该服务接口的调用不再经过网络，直接执行本地的fallback方法。
  - ✧半熔断状态（Half-Open）
    - 尝试恢复服务调用，允许有限的流量调用该服务，并监控调用成功率。
      - ✓如果成功率达到预期，则说明服务已恢复，进入熔断关闭状态；
      - ✓如果成功率仍旧很低，则重新进入熔断关闭状态。

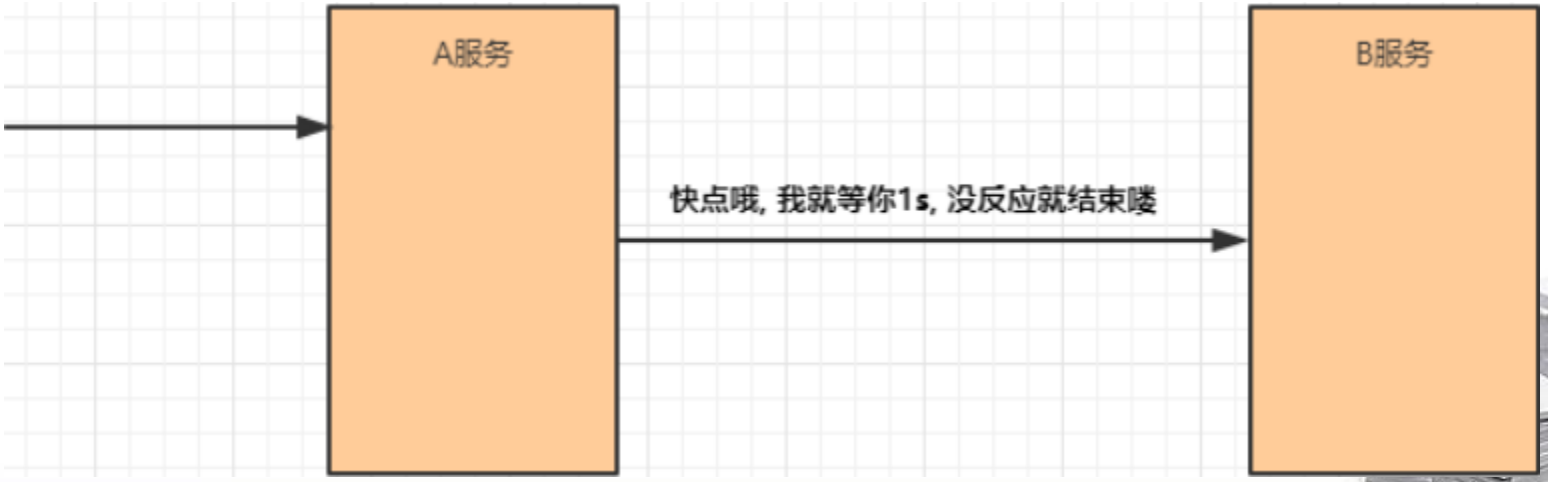
④降级

- 降级其实就是为服务提供一个托底方案，一旦服务无法正常调用，就使用托底方案。



⑤超时

- 在上游服务调用下游服务的时候，设置一个最大响应时间，如果超过这个时间，下游未作出反应，就断开请求，释放掉线程。



3）常见容错产品

- **Hystrix**
  - ✧ **Hystrix**是由**Netflix**开源的一个延迟和容错库，用于隔离访问远程系统、服务或者第三方库，防止级联失败，从而提升系统的可用性与容错性。
- **Resilience4J**
  - ✧ **Resilicence4J**一款非常轻量、简单，并且文档非常清晰、丰富的熔断工具，这也是**Hystrix**官方推荐的替代产品。
  - ✧ 不仅如此，**Resilicence4j**还原生支持**Spring Boot 1.x/2.x**，而且监控也支持和**prometheus**等多款主流产品进行整合。
- **Sentinel**
  - ✧ **Sentinel**是阿里巴巴开源的一款断路器实现，本身在阿里内部已经被大规模采用，非常稳定。

特性比较

	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
流量整形	支持预热模式、匀速器模式、预热排队模式	不支持	简单的 Rate Limiter 模式
系统自适应保护	支持	不支持	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	不提供控制台，可对接其它监控系统

2. Sentinel简介

- **Sentinel是阿里开源的面向分布式服务架构的一套用于服务容错的综合性解决方案。**
  - ✧它以流量为切入点，从限流、流量整形、熔断降级、系统负载保护、热点防护等多个维度来帮助开发者保障微服务的稳定性。
  - ✧它通过一个轻量级的控制台，提供机器发现、单机资源实时监控以及规则管理等功能。
    - 机器发现：收集 **Sentinel** 客户端发送的心跳包，用于判断机器是否在线。
    - 监控：通过 **Sentinel** 客户端暴露的监控 **API**，定期拉取并且聚合应用监控信息，最终可以实现秒级的实时监控。
    - 规则管理：统一管理和推送规则。

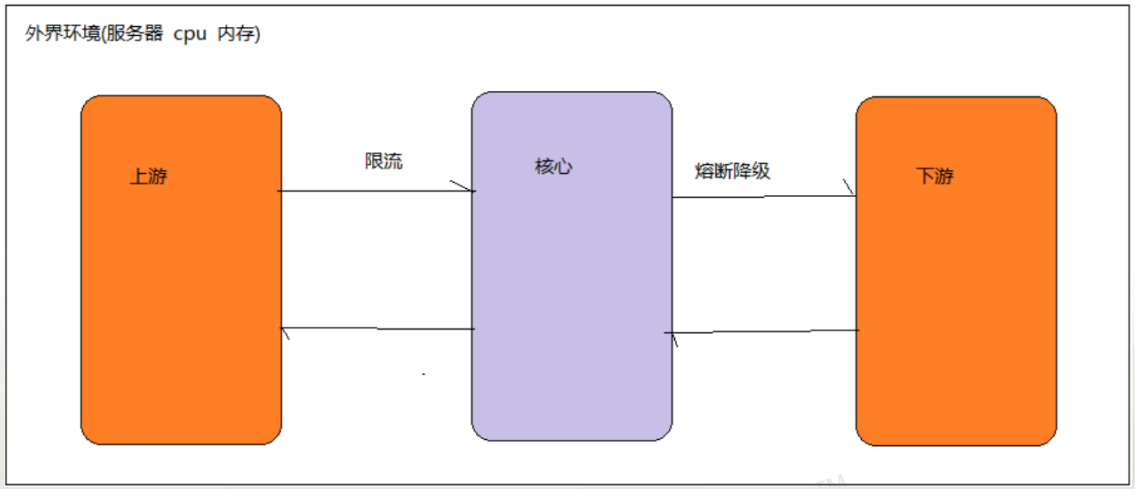
Sentinel体系结构

1）Sentinel中的核心概念

- 资源：Sentinel要保护的内容。
  - ✧是Sentinel的关键概念。
  - ✧它可以是Java应用程序中的任何内容，可以是一个服务，也可以是一个方法，甚至可以是一段代码。
- 规则：定义如何保护资源。
  - ✧它作用在资源上，主要包括流控规则、熔断降级规则、热点规则、授权规则以及系统规则。

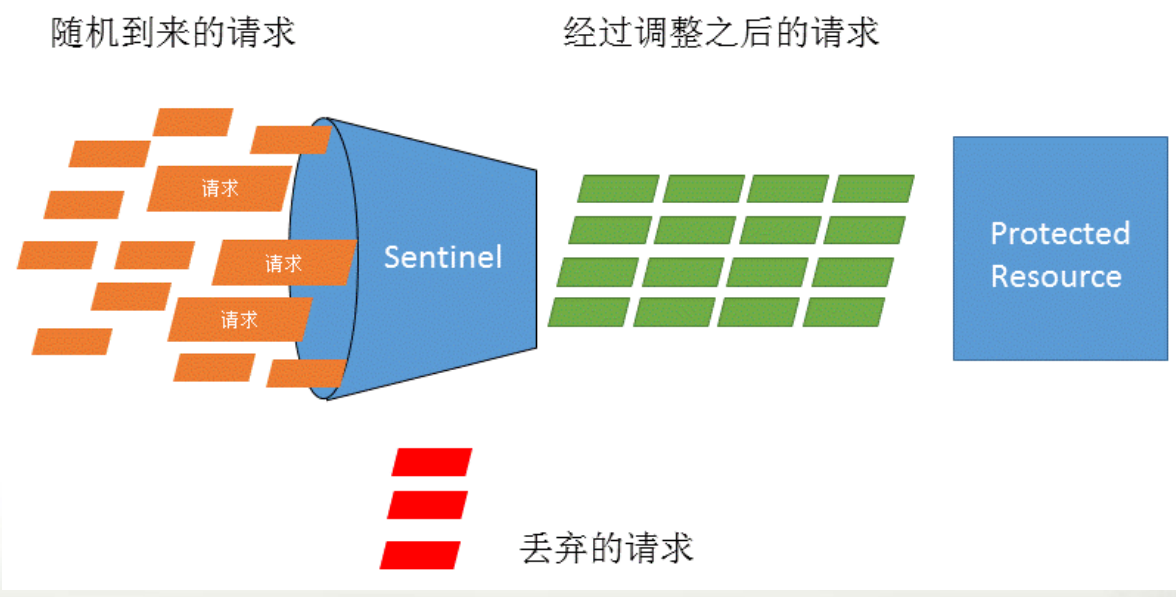
2）Sentinel主要功能

- Sentinel作为保护微服务的中间件产品，具有以下主要功能：
  - ✧①流量控制
  - ✧②熔断降级
  - ✧③系统负载保护



①流量控制

- Sentinel作为一个调配器，可以根据需要把随机的请求调整成合适的形状。

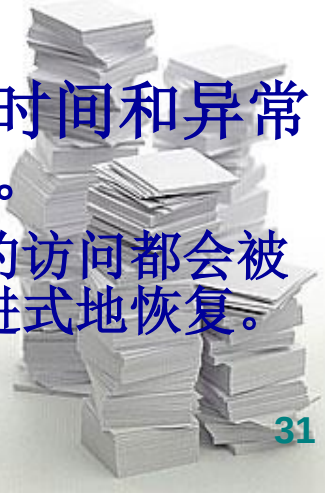


流量控制设计理念

- 流量控制有以下几个角度:
  - ✧资源的调用关系，例如资源的调用链路，资源和资源之间的关系；
  - ✧运行指标，例如 QPS（每秒请求数量）、线程池、系统负载等；
  - ✧控制的效果，例如直接限流、冷启动、排队等。
- Sentinel的设计理念是让您自由选择控制的角度，并进行灵活组合，从而达到想要的效果。

②熔断降级

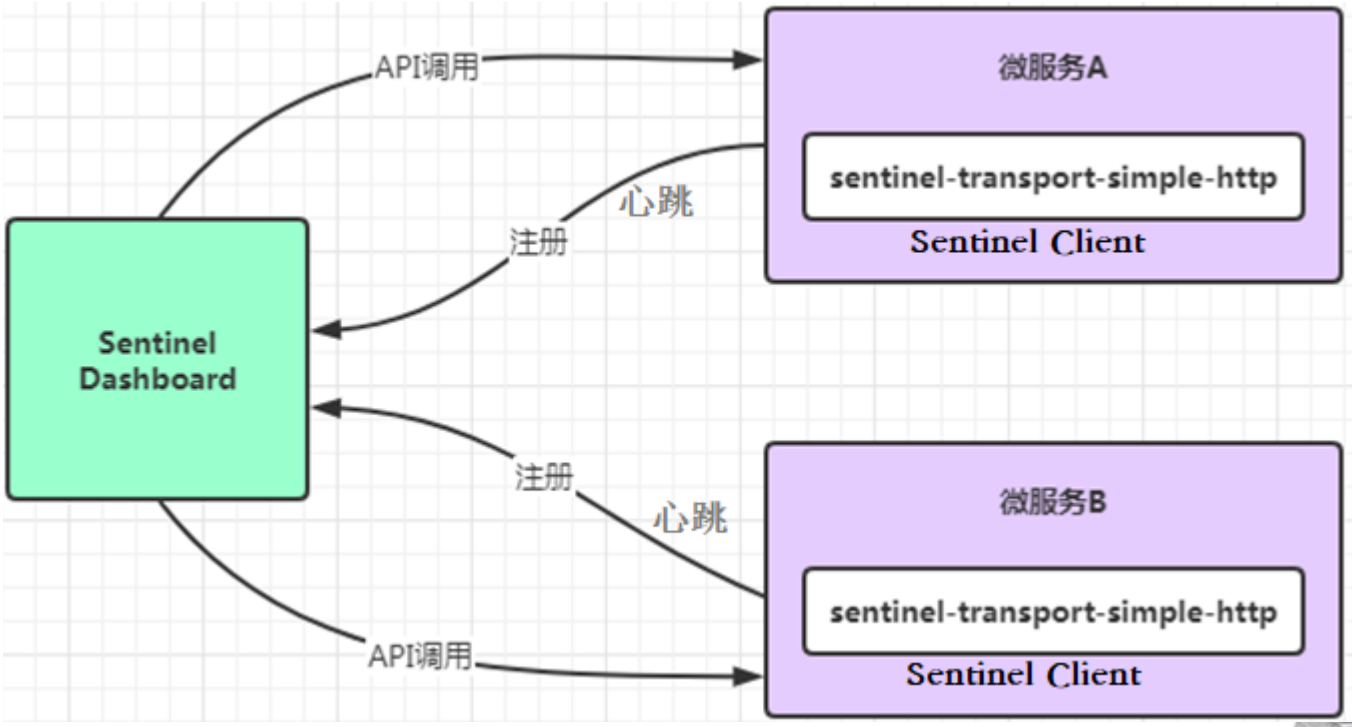
- 除了流量控制以外，及时对调用链路中的不稳定因素进行熔断也是 **Sentinel** 的使命之一。
- **Sentinel**采取了两种手段:
  - ✧通过并发线程数进行限制：通过限制访问资源并发线程的数量，来减少不稳定资源对其它资源的影响。
    - 当某个资源出现不稳定的情况下，例如响应时间变长，对资源的直接影响就是会造成线程数的逐步堆积。
    - 当线程数在特定资源上堆积到一定的数量之后，对该资源的新请求就会被拒绝。
  - ✧针对慢调用和异常对资源进行降级：根据响应时间和异常等不稳定因素来快速对不稳定的调用进行熔断。
    - 当依赖的资源出现响应时间过长后，所有对该资源的访问都会被直接拒绝，直到过了指定的时间窗口之后才重新渐进式地恢复。



③系统负载保护

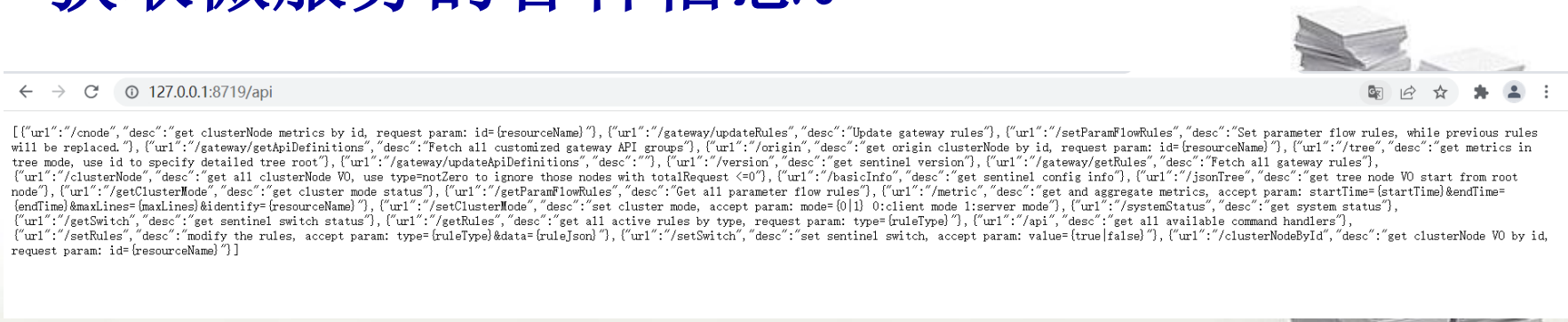
- Sentinel同时提供系统维度的自适应保护能力，能让系统的入口流量和系统的负载达到一个平衡，保证系统在能力范围之内处理最多的请求。

3）Sentinel通信原理



通信原理

- **Sentinel客户端（一个Simple HTTP Server）在启动的时候会开一个8719（+1）端口，把自己的服务信息注册到Sentinel控制台上，并且在这个端口下会暴漏一些如下图所示的接口，Sentinel控制台通过调用这些接口获取微服务的各种信息。**

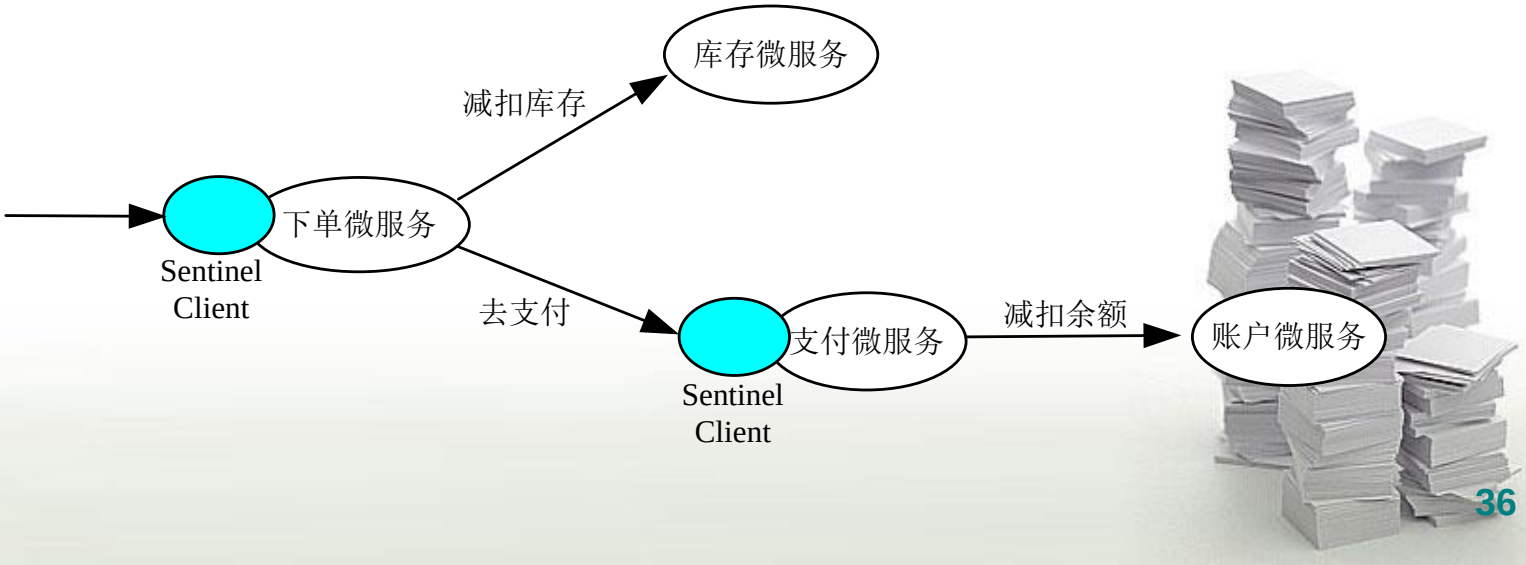


通信原理

- 当在控制台中配置了某个规则如流控规制时，Sentinel控制台会调用Sentinel客户端暴漏的接口把规则推给Sentinel客户端，保存在Sentinel客户端内存。
  - ✧如果Sentinel客户端配置了文件持久化，流控规也会在文件中存储一份。
  - ✧当请求到来的时候，Sentinel客户端通过拦截器（AbstractSentinelInterceptor）把流控规制从内存中取出来判断是否达到限流要求，达到了就进行限流，未达到就放行。

9.3.2 使用Sentinel中间件

- 本小节以Sentinel为服务容错解决方案，通过规则配置实现对上一节搭建的电商项目微服务架构应用进行容错保护。
- 受容错保护的电商项目微服务架构：



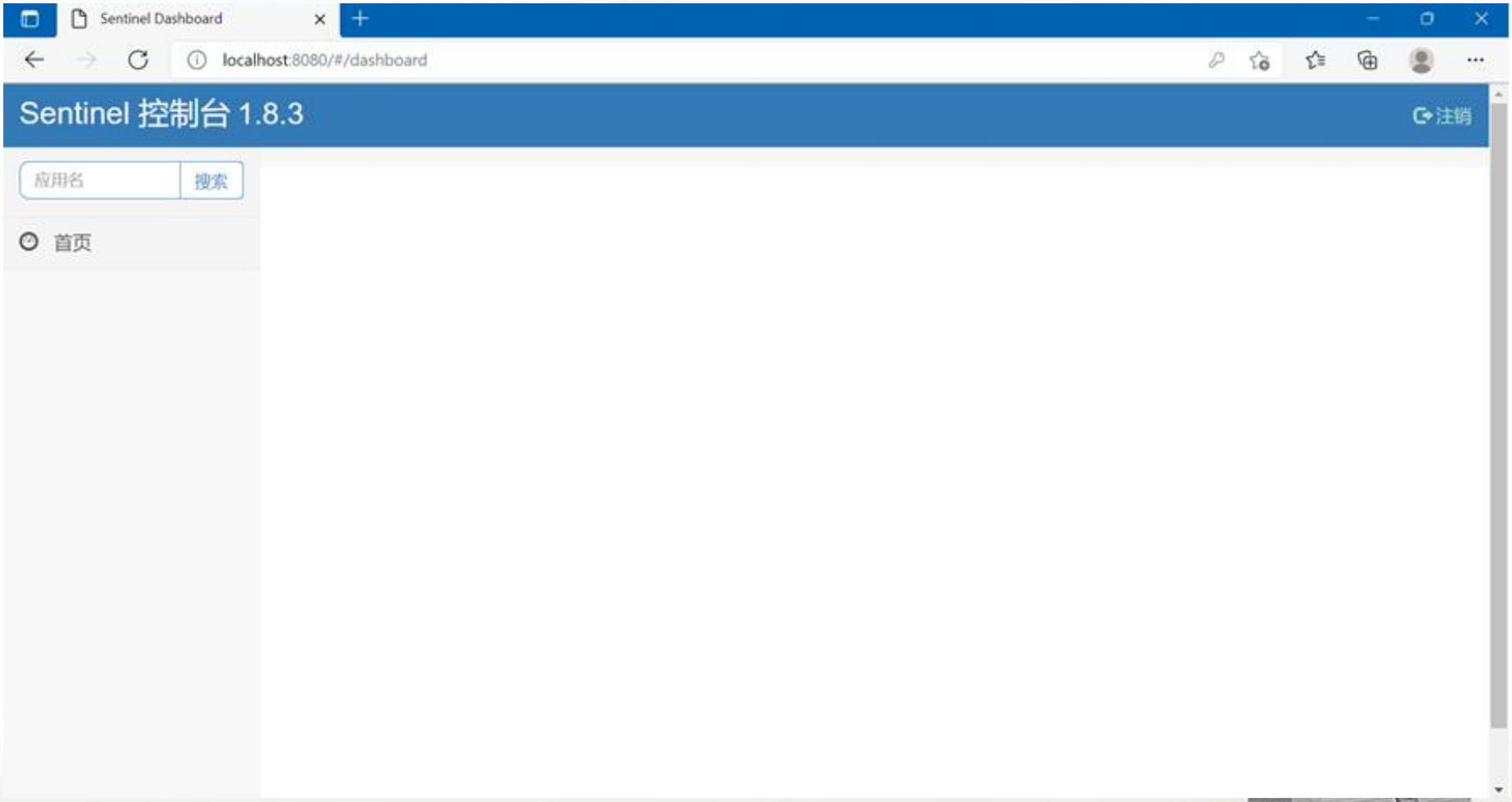
搭建步骤

- 1.安装Sentinel DashBoard
- 2.添加Sentinel依赖和配置
- 3.添加规则
- 4.规则持久化
- 5.开启Feign支持
- 6.自定义微服务容错处理
- 7.效果测试

1. 安装Sentinel DashBoard

- 下载地址: **https://github.com/alibaba/Sentinel/releases/download/1.8.3/sentinel-dashboard-1.8.3.jar**
- 执行以下命令，启动**Sentinel DashBoard**：
  - **java -jar sentinel-dashboard-1.8.3.jar**
- 启动后访问**http://localhost:8080**，输入默认的登录用户名和密码：**sentinel**，即可看到**Sentinel DashBoard**界面。

Sentinel DashBoard界面



关于Sentinel DashBoard启动

- 使用命令java -jar sentinel-dashboard-1.8.3.jar启动Sentinel DashBoard时还可以带以下参数：
  - -Dserver.port=8080：指定服务端端口号
  - -Dsentinel.dashboard.auth.username=sentinel：指定登录用户名
  - -Dsentinel.dashboard.auth.password=123456：指定登录密码
  - -Dserver.servlet.session.timeout=7200：指定服务端session的过期时间，如7200表示7200秒；60m表示60分钟，默认为30分钟。

2. 添加Sentinel依赖和配置

- 1）引入Sentinel依赖
- 2）添加Sentinel配置
- 3）监控微服务

1）引入Sentinel依赖

- 在chapter09工程的payment-service模块pom.xml中引入Sentinel客户端依赖：

```
<!--sentinel客户端依赖-->
 <dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
 </dependency>
```

2）添加Sentinel配置

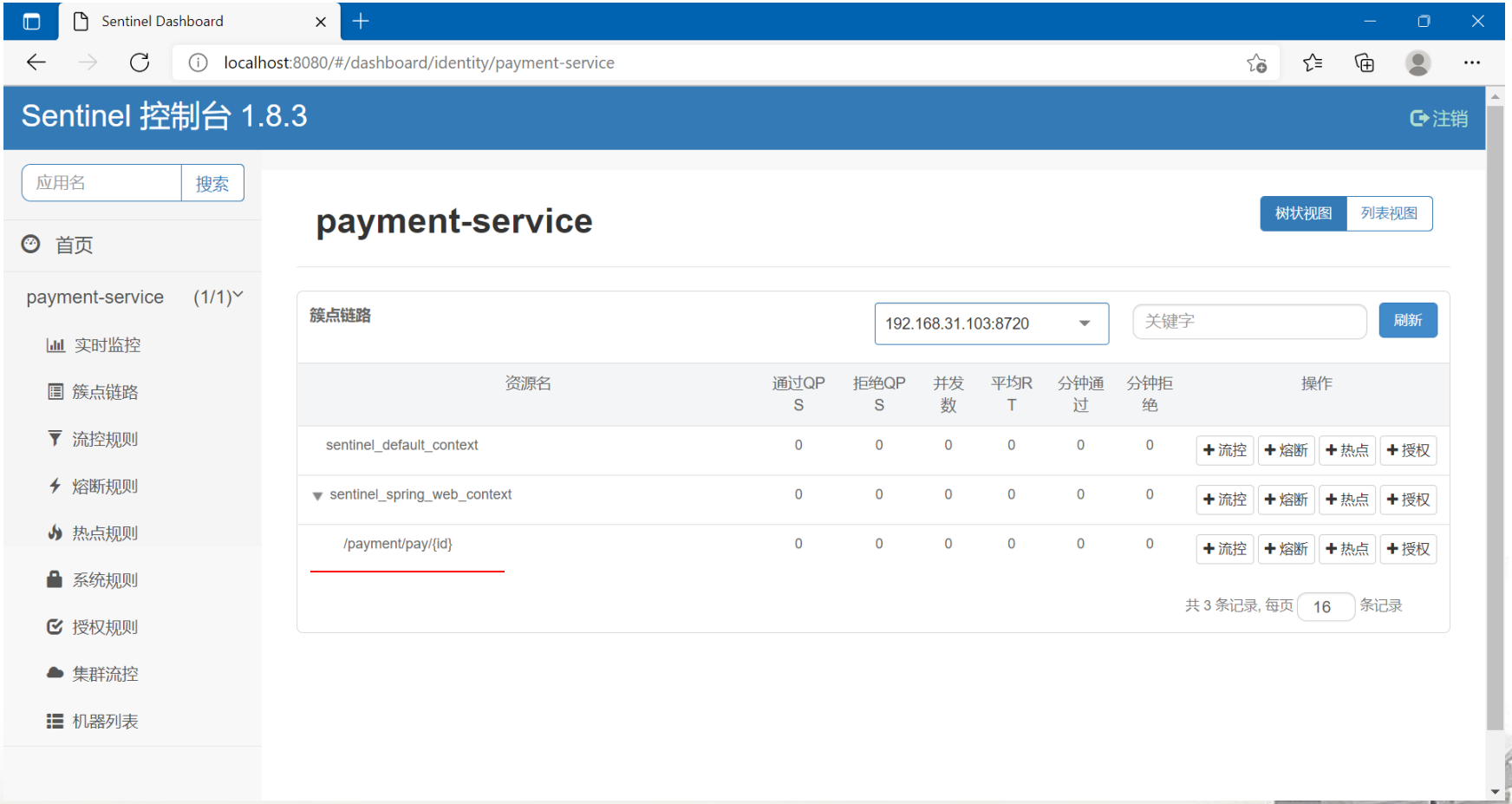
- 在payment-service模块的application.properties中添加以下配置：

```
#配置Sentinel DashBoard地址
spring.cloud.sentinel.transport.dashboard=localhost:8080
#取消Sentinel DashBoard懒加载(sentinel客户端项目一旦启动，sentinel客户端的信息就出现在Sentinel DashBoard上)
spring.cloud.sentinel.eager=true
```

3）监控微服务

- 启动payment-service服务和chapter09工程其他服务，payment-service微服务被列入到了Sentinel DashBoard监控列表中。
- 然后，访问：http://localhost:9010/ordering/purchase/p002/a001，URI资源“/payment/pay/{id}”被列入到了Sentinel DashBoard簇点链路列表中。

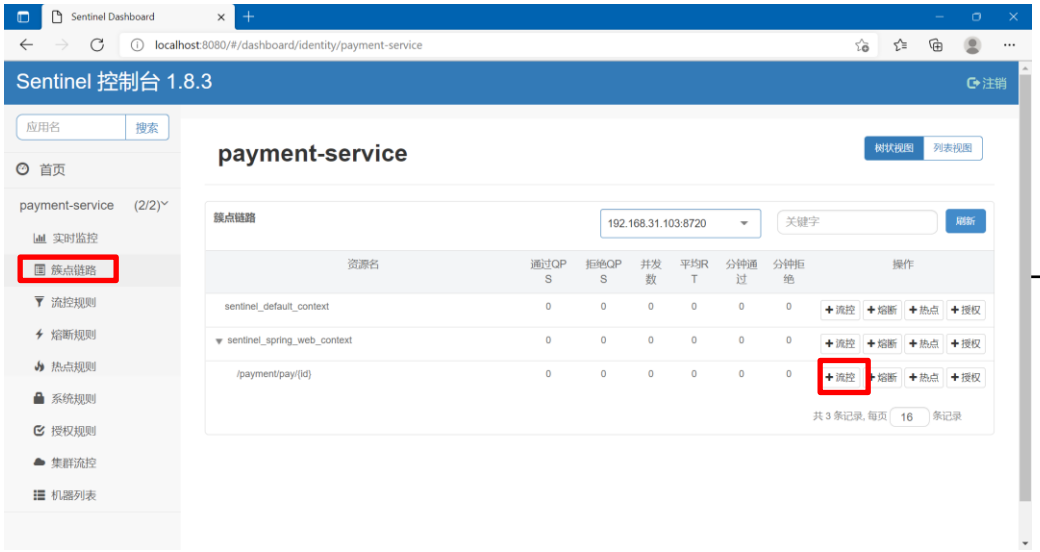
被监控资源列表


Sentinel Dashboard
localhost:8080/#/dashboard/identity/payment-service
Sentinel 控制台 1.8.3
注销
应用名
搜索
首页
payment-service (1/1)
实时监控
簇点链路
流控规则
熔断规则
热点规则
系统规则
授权规则
集群流控
机器列表
payment-service
树状视图
列表视图
簇点链路
192.168.31.103:8720
关键字
刷新
资源名
通过QPS
拒绝QPS
并发数
平均RT
分钟通过
分钟拒绝
操作
sentinel_default_context
sentinel_spring_web_context
/payment/pay/{id}
+流控
+熔断
+热点
+授权
共 3 条记录, 每页 16 条记录

3.添加规则

- 1）流控规则
- 2）熔断规则
- 3）热点规则
- 4）授权规则
- 5）系统规则

1）流控规则


Sentinel Dashboard
localhost:8080/#/dashboard/identity/payment-service
Sentinel 控制台 1.8.3
注销
应用名
搜索
首页
payment-service (2/2)
实时监控
簇点链路
流控规则
熔断规则
热点规则
系统规则
授权规则
集群流控
机器列表
payment-service
树状视图
列表视图
簇点链路
192.168.31.103:8720
关键字
刷新
资源名
通过QPS
拒绝QPS
并发数
平均RT
分钟通过
分钟拒绝
操作
sentinel_default_context
sentinel_spring_web_context
/payment/pay/{id}
+流控 +熔断 +热点 +授权
共 3 条记录, 每页 16 条记录


新增流控规则
资源名
/payment/pay/{id}
针对来源
default
阈值类型
QPS
并发线程数
单机阈值
单机阈值
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
新增并继续添加
新增
取消

规则参数组合

资源名 /payment/pay/{id}
针对来源 default
阈值类型 ◉ QPS ○ 并发线程数 单机阈值 单机阈值
是否集群 ☐
流控模式 ○ 直接 ◉ 关联 ○ 链路
关联资源 关联资源
流控效果 ○ 快速失败 ◉ Warm Up ○ 排队等待
预热时长 秒
关闭高级选项

资源名 /payment/pay/{id}
针对来源 default
阈值类型 ○ QPS ◉ 并发线程数 单机阈值 单机阈值
是否集群 ☐
流控模式 ○ 直接 ○ 关联 ◉ 链路
入口资源 入口资源
关闭高级选项

资源名 /payment/pay/{id}
针对来源 default
阈值类型 ◉ QPS ○ 并发线程数 单机阈值 单机阈值
是否集群 ☐
流控模式 ○ 直接 ○ 关联 ◉ 链路
入口资源 入口资源
流控效果 ○ 快速失败 ○ Warm Up ◉ 排队等待
超时时间 毫秒
关闭高级选项

资源名 /payment/pay/{id}
针对来源 default
阈值类型 ◉ QPS ○ 并发线程数 均摊阈值 单机均摊阈值
是否集群 ☑ 集群阈值模式 ◉ 单机均摊 ○ 总体阈值
失败退化 ☐ ⓘ 如果 Token Server 不可用是否退化到单机限流
新增并继续添加 新增 取消

规则配置

- 一条流控规则主要配置以下几个因素：
  - ✧①资源名
  - ✧②针对来源
  - ✧③阈值类型和阈值
  - ✧④是否集群
  - ✧⑤流控模式
  - ✧⑥流控效果

①资源名

- 指受保护资源的名称，默认是uri，如：
  - /payment/pay/{id}
- 也可使用注解@SentinelResource(“资源名”)标注在受保护资源（方法）上来声明资源点，然后使用注解中的“资源名”。

声明资源点

```
@GetMapping("/pay/{id}")
@SentinelResource("pay")
public String pay(@PathVariable("id") String accountId) {
    String payResult=accountFeignClient.reduce(accountId);
    boolean flag = paymentService.pay(accountId);
    String serverPort=environment.getProperty("server.port");
    return payResult+" 支付服务端口："+serverPort;
}
```

②针对来源

- 表示对调用的来源进行限流。
- 该字段的值有以下三种选项，分别对应不同的场景：
  - ✧ **default**：表示不区分调用者。
    - 来自任何调用者的请求都将进行限流统计。
    - 如果这个资源名的调用总和超过了这条规则定义的阈值，则触发限流。
  - ✧ **{some_origin_name}**：表示针对特定的调用者。
    - 只有来自这个调用者的请求才会进行流量控制。
  - ✧ **other**：表示针对除**{some_origin_name}**以外的其余调用者的流量进行流量控制。
    - 例如，资源**A**配置了一条针对调用者**caller1**的限流规则，同时又配置了一条调用者为**other**的规则，那么任意来自非**caller1**对的调用，都不能超过**other**这条规则定义的阈值。
- **{some_origin_name}**和**other**对某个来源控制进行限流需要在该资源的服务模块中额外建一个解析来源的类。
  - ✧ 只要**Sentinel**保护的资源被访问，**Sentinel**就会调用该来源解析类去解析访问来源。

来源解析类

```
package com.scst.parser;
import com.alibaba.csp.sentinel.adapter.spring.webmvc.callback.RequestOriginParser;
import org.springframework.stereotype.Component;
import javax.servlet.http.HttpServletRequest;
@Component
public class MyRequestOriginParser implements RequestOriginParser {
    @Override
    public String parseOrigin(HttpServletRequest request) {
        //如果是{some_origin_name}这种针对来源，表示请求的参数origin的值必须和针对来源一样才能限流
        String origin = request.getParameter("origin");
        return origin;
    }
}
```

③阈值类型和阈值

- 阈值类型：
  - ✧QPS（每秒请求数量）：表示一秒内有多少个请求；
  - ✧线程数：表示同时有多少个线程请求。
- 单机阈值：表示对应的阈值类型请求数达到指定的值时就会限流。

④是否集群

- 在集群流控下，阈值模式可设置为单机均摊阈值或总阈值。
- 集群流控中共有两种身份：
  - ✧**Token Client**：集群流控客户端，用于向所属**Token Server**通信请求**token**。集群限流服务端会返回给客户端结果，决定是否限流。
  - ✧**Token Server**：集群流控服务端，处理来自 **Token Client**的请求，根据配置的集群规则判断是否应该发放**token**（告知客户端是否允许流量通过）。
- 集群流控可以精确地控制整个集群的调用总量，结合单机限流兜底，可以更好地发挥流量控制的效果。

流控集群配置

3个payment-service微服务实例组建一个流控集群。

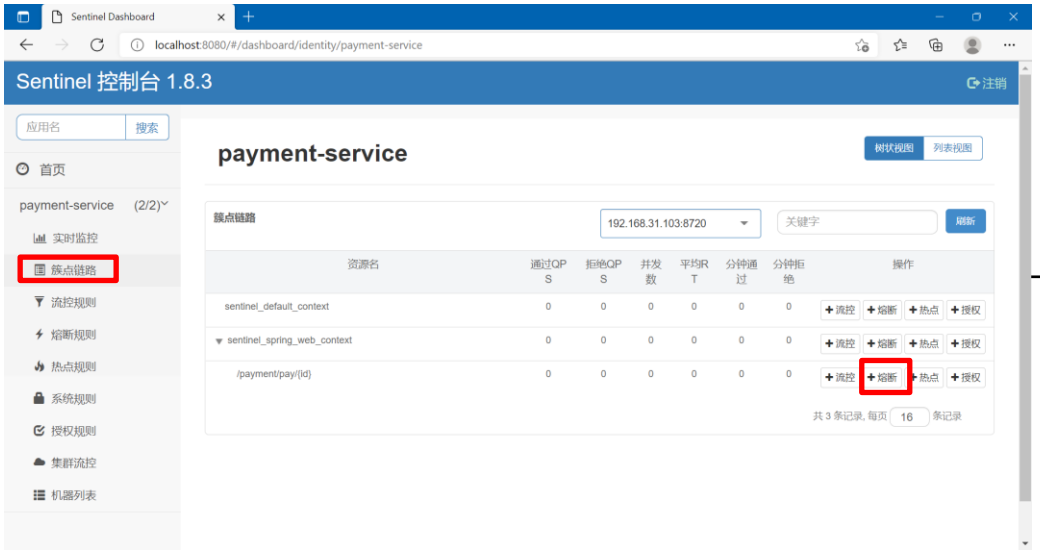
⑤流控模式

- sentinel有三种流控模式：
  - ✧直接：表示直接对当前受保护的资源名资源进行限流；
  - ✧关联：表示对关联资源的访问请求达到阈值时，限制对当前受保护的资源名资源的访问；
  - ✧链路：表示对当前受保护的资源名资源的入口资源的访问请求达到阈值时，限制对当前受保护的资源名资源的访问。
    - 如：一个链路上/test2/{app}是/services资源的入口资源，如果对/test2/{app}的访问请求达到阈值1，则限制对当前资源/services的访问

⑥流控效果

- 流控效果分三种：
  - ✧快速失败：表示当达到阈值时就直接失败，返回限流错误提示；
  - ✧**warm up**：表示阈值可变，可在预热时长内从开始阈值增长到最大阈值，其中，最大阈值即设置的**QPS**阈值，开始阈值为最大阈值**/3**；
    - 适用于将突然增大的流量转换为缓步增长的场景。
  - ✧排队等待：又称匀速排队，表示超过阈值的请求先放到队列，如果在设置的超时时间内还未能被处理，则返回限流错误信息。（漏桶算法）

2）熔断规则


Sentinel Dashboard
localhost:8080/#/dashboard/identity/payment-service
Sentinel 控制台 1.8.3
注销
应用名
搜索
首页
payment-service (2/2)
实时监控
簇点链路
流控规则
熔断规则
热点规则
系统规则
授权规则
集群流控
机器列表
payment-service
树状视图
列表视图
簇点链路
192.168.31.103:8720
关键字
刷新
资源名
通过QPS
拒绝QPS
并发数
平均RT
分钟通过
分钟拒绝
操作
sentinel_default_context
sentinel_spring_web_context
/payment/pay/{id}
+流控 +熔断 +热点 +授权
共 3 条记录, 每页 16 条记录


新增熔断规则
资源名
/payment/pay/{id}
熔断策略
慢调用比例
异常比例
异常数
最大 RT
RT (毫秒)
比例阈值
取值 [0.0, 1.0]
熔断时长
熔断时长(s)
s
最小请求数
5
统计时长
1000
ms
新增并继续添加
新增
取消

规则参数组合

资源名 /payment/pay/{id}

熔断策略 ○ 慢调用比例 ◉ 异常比例 ○ 异常数

比例阈值 取值范围 [0.0,1.0]

熔断时长 熔断时长(s) s 最小请求数 5

统计时长 1000 ms

新增并继续添加 新增 取消

资源名 /payment/pay/{id}

熔断策略 ○ 慢调用比例 ○ 异常比例 ◉ 异常数

异常数 异常数

熔断时长 熔断时长(s) s 最小请求数 5

统计时长 1000 ms

新增并继续添加 新增 取消

规则配置

- 熔断规则就是设置当满足什么条件的时候，对资源提供服务进行降级，抛出异常“**Blocked by Sentinel (flow limiting)**”。
- 熔断规则的熔断策略可有3种配置：
  - ①慢调用比例
  - ②异常比例
  - ③异常数

①慢调用比例

- 慢调用是指请求的响应时间大于“最大RT”设置值（最大响应时间）的调用。
- 该策略是：
  - ✧当单位“统计时长”内请求数大于“最小请求数”，并且慢调用的比例大于“比例阈值”时，则在接下来的“熔断时长”内的请求会自动被熔断。
  - ✧经过“熔断时长”后，熔断器会进入探测恢复状态（Half-Open状态）。
    - 若接下来的一个请求的响应时间小于“最大RT”，则结束熔断；
    - 否则，再次被熔断。

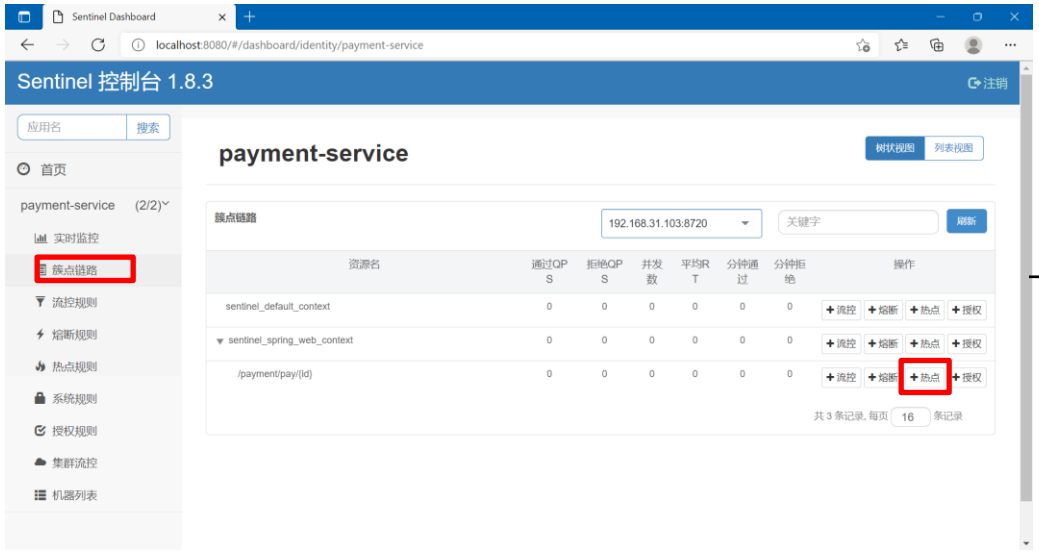
②异常比例

- 该策略是：
  - ✧当单位“统计时长”内请求数大于“最小请求数”，并且异常的比例大于“比例阈值”时，则在接下来的“熔断时长”内的请求会自动被熔断。
  - ✧经过“熔断时长”后，熔断器会进入探测恢复状态（**Half-Open**状态）。
    - 若接下来的一个请求成功完成（没有异常），则结束熔断；
    - 否则，再次被熔断。

③异常数

- 该策略是：
  - ✧当单位“统计时长”内请求数大于“最小请求数”，并且异常数大于“异常阈值”时，则在接下来的“熔断时长”内的请求会自动被熔断。
  - ✧经过“熔断时长”后，熔断器会进入探测恢复状态（**Half-Open**状态）。
    - 若接下来的一个请求成功完成（没有异常），则结束熔断；
    - 否则，再次被熔断。

3）热点规则


Sentinel Dashboard
localhost:8080/#/dashboard/identity/payment-service
Sentinel 控制台 1.8.3
注销
应用名
搜索
首页
payment-service (2/2)
实时监控
簇点链路
流控规则
熔断规则
热点规则
系统规则
授权规则
集群流控
机器列表
payment-service
树状视图
列表视图
簇点链路
192.168.31.103:8720
关键字
刷新
资源名
通过QPS
拒绝QPS
并发数
平均RT
分钟通过
分钟拒绝
操作
sentinel_default_context
sentinel_spring_web_context
/payment/pay/{id}
流控
熔断
热点
授权
共 3 条记录, 每页 16 条记录


新增热点规则
资源名
pay
限流模式
QPS 模式
参数索引
请填入传入的热点参数的索引（从 0 开始）
单机阈值
0
统计窗口时长
1
秒
是否集群
新增并继续添加
新增
取消

规则参数组合

资源名	pay
限流模式	QPS 模式
参数索引	请填入传入的热点参数的索引（从 0 开始）
均摊阈值	0
是否集群	☑ 集群阈值模式：◉ 单机均摊 ○ 总体阈值
失败退化	☑ ⓘ 若选择，则 Token Server 不可用时将退化到单机限流

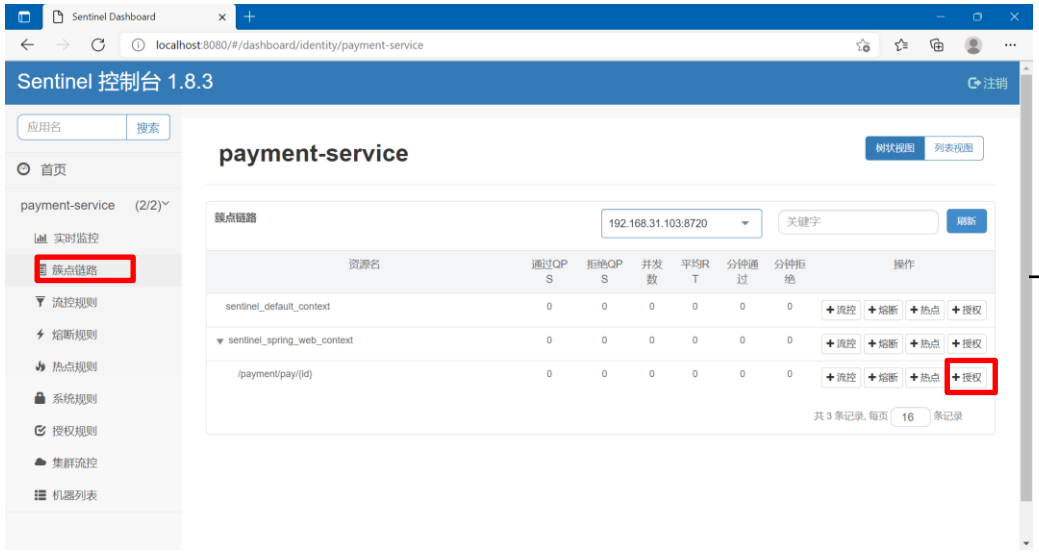
规则配置

- 热点即经常访问的参数，热点规则就是通过参数索引、单机阈值、统计窗口时长等设置如何对包含热点参数的资源调用进行限流，表示在单位“统计窗口时长”内，对指定索引的参数，如果资源调用QPS大于阈值将被限流。
  - ✧它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（Least Recently Used）置换策略统计最近最常访问的热点参数，结合令牌桶算法进行参数级别的流控。
  - ✧必须使用@SentinelResource(“资源名”)定义热点规则的资源名，而不能直接使用uri作为资源名。

规则编辑

- 一条热点规则被设置后，如果想增加其参数例外项，可编辑此规则：对该参数指定的“参数值”另设“限流阈值”。

4）授权规则


Sentinel Dashboard
localhost:8080/#/dashboard/identity/payment-service
Sentinel 控制台 1.8.3
注销
应用名
搜索
首页
payment-service (2/2)
实时监控
簇点链路
流控规则
熔断规则
热点规则
系统规则
授权规则
集群流控
机器列表
payment-service
树状视图
列表视图
簇点链路
192.168.31.103:8720
关键字
刷新
资源名
通过QPS
拒绝QPS
并发数
平均RT
分钟通过
分钟拒绝
操作
sentinel_default_context
sentinel_spring_web_context
/payment/pay/{id}
+流控 +熔断 +热点 +授权
共 3 条记录, 每页 16 条记录


新增授权规则
资源名
/payment/pay/{id}
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
新增并继续添加
新增
取消

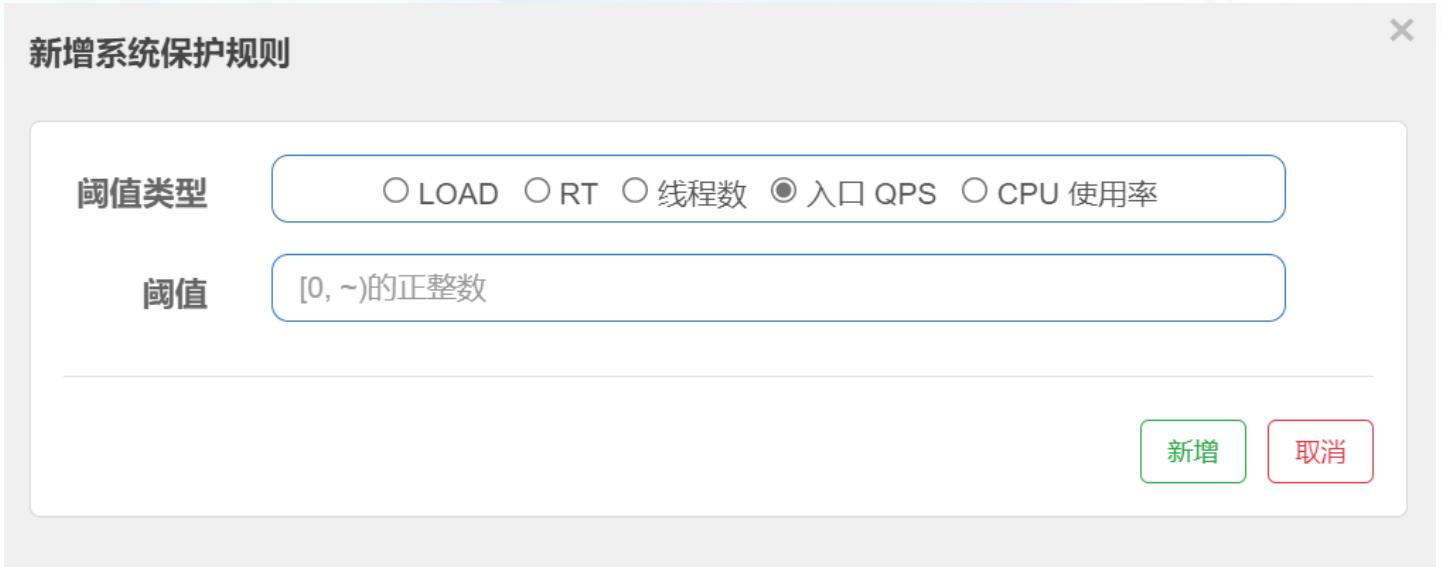
规则配置

- 授权规则配置表示针对请求来源配置黑白名单。
  - ✧若配置白名单则只有请求来源位于白名单内时才可通过；
  - ✧若配置黑名单则请求来源位于黑名单时不通过，其余的请求通过。
- 授权规则对某个来源控制进行限流需要在该资源的服务模块中额外建一个解析来源的类。
  - ✧**Sentinel**提供了**RequestOriginParser**接口来解析访问来源，只要**Sentinel**保护的资源被访问，**Sentinel**就会调用该接口的实现类去解析访问来源。
  - ✧来源可以放到路径参数里，也可以放到请求头里，也可直接用**URI**。
    - 比如把**token**放到请求头里，通过**token**解析出用户**id**

来源解析类

```
package com.scst.parser;
import com.alibaba.csp.sentinel.adapter.spring.webmvc.callback.RequestOriginParser;
import org.springframework.stereotype.Component;
import javax.servlet.http.HttpServletRequest;
@Component
public class MyRequestOriginParser implements RequestOriginParser {
    @Override
    public String parseOrigin(HttpServletRequest request) {
        //来源origin必须在sentinel授权配置的白名单或黑名单里才可对其进行访问控制
        String origin = request.getRequestURI();
        return origin;
    }
}
```

5）系统规则


新增系统保护规则
阈值类型
LOAD RT 线程数 入口 QPS CPU 使用率
阈值
[0, ~)的正整数
新增
取消


新增系统保护规则
阈值类型
LOAD RT 线程数 入口 QPS CPU 使用率
阈值
[0, 1]的小数，代表百分比
新增
取消

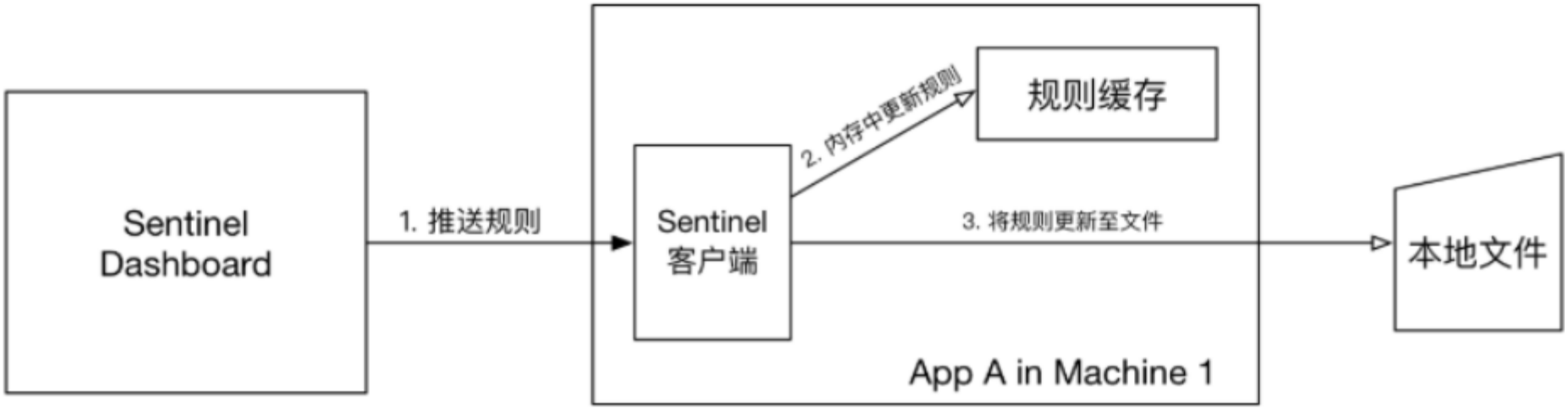
规则配置

- 系统规则是从整体应用维度（不是资源维度）对入口流量进行控制，从单台机器的以下5个方面监控应用总体数据，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。
  - ✧ **Load**（仅对 **Linux/Unix-like** 机器生效）：当系统**load1**超过阈值，且系统当前的并发线程数超过系统容量时才会触发系统保护。
    - 系统容量由系统的 **maxQps * minRt** 计算得出。设定参考值一般是 **CPU cores * 2.5**。
  - ✧ **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。
  - ✧ 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。
  - ✧ **CPU**使用率：当单台机器上所有入口流量的**CPU**使用率达到阈值即触发系统保护。

4.规则持久化

- 1）持久化到文件
- 2）持久化到Nacos

1）持久化到文件


Sentinel
Dashboard
1. 推送规则
Sentinel
客户端
2. 内存中更新规则
规则缓存
3. 将规则更新至文件
本地文件
App A in Machine 1

搭建步骤

- ①创建工具类
- ②添加配置
- ③查看持久化结果

①创建工具类

- 在**payment-service**模块中创建工具类**com.scst.utils.FilePersistence**，在该类中实现流控规则、熔断规则、热点规则、授权规则、系统规则的持久化和加载。
  - ✧该类会在模块启动时被加载进行规则初始化。

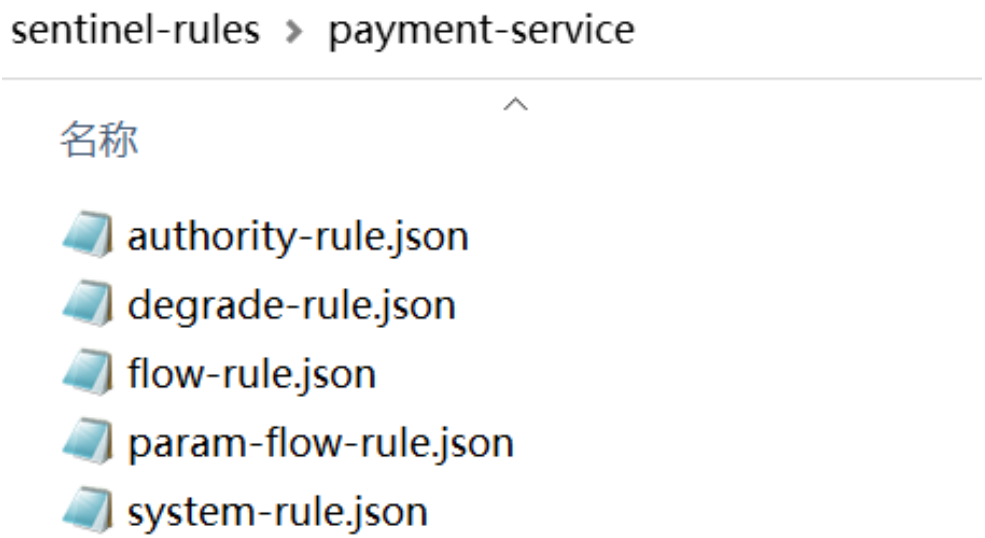
FilePersistence类

```
package com.scst.utils;
import com.alibaba.csp.sentinel.init.InitFunc;
……

//规则持久化
public class FilePersistence implements InitFunc {

    @Override
    public void init() throws Exception {
        init("payment-service");
    }
    ……
}
```

②添加配置

- 在payment-service模块的resources下创建配置目录META-INF/services，然后新建文件com.alibaba.csp.sentinel.init.InitFunc，在该文件中添加工具类的全路径：
  - ✧com.scst.utils.FilePersistence

③查看持久化结果



规则持久化后即使**Sentinel DashBorad**离线也将起作用。

2）持久化到Nacos

- 当用Nacos做为规则持久化时，在Sentinel客户端项目启动的时候会根据配置的Nacos数据源把规则从Nacos中读取出来显示在Sentinel控制台中，并加载到Sentinel客户端内存。
  - ✧注意：当前Sentinel版本在Sentinel控制台中修改规则后不会同步推送到Nacos中，只能在Nacos中修改规则后在Sentinel控制台显示。

搭建步骤

- ①添加pom依赖
- ②添加数据源配置
- ③在nacos中配置流控规则
- ④查看流控规则

①添加pom依赖

- 在chapter09工程的ordering-service模块pom.xml中引入Sentinel客户端依赖和Nacos数据源依赖：

```
<!--Sentinel客户端依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
<!--Nacos数据源依赖-->
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

②添加数据源配置

- 在ordering-service模块的application.properties中添加以下配置：

```
#配置Sentinel DashBoard地址
spring.cloud.sentinel.transport.dashboard=localhost:8080
#配置Sentinel规则持久化(存储到Nacos的JSON格式配置文件中)
spring.cloud.sentinel.datasource.ds1.nacos.server-addr=localhost:8848
spring.cloud.sentinel.datasource.ds1.nacos.username=nacos
spring.cloud.sentinel.datasource.ds1.nacos.password=nacos
spring.cloud.sentinel.datasource.ds1.nacos.data-id=${spring.application.name}.json
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
```

③在nacos中配置流控规则

新建配置

* Data ID: ordering-service.json

* 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:

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

ordering-service.json

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

字段含义

- **resource**：资源名，即限流规则的作用对象
- **limitApp**：针对来源，若为**default** 则不区分调用来源
- **grade**：阈值类型（**0:**并发线程数、 **1:QPS**）
- **count**：限流阈值
- **strategy**：流控模式（**0:**直接、**1:**关联、**2:**链路）
- **controlBehavior**：流控效果（**0:**快速失败、**1:Warm Up**、**2:**排队等待）
- **clusterMode**：是否为集群模式

④查看流控规则

ordering-service

+ 新增流控规则

流控规则

192.168.31.103:8722

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
/ordering/configs	default	直接	QPS	1	单机	快速失败	编辑 删除

共 1 条记录, 每页 10 条记录

5.开启Feign支持

- Feign作为微服务客户端同样受Sentinel的限流控制。
- 开启Feign客户端的支持只需在Feign客户端项目application.properties中如下配置即可，此时如果微服务请求通过Feign客户端调用，则调用链会在Sentinel控制台显示出来。
  - feign.sentinel.enabled=true

Feign客户端调用的容错处理

- 在注解@FeignClient中提供了以下2个属性（二选一）用于当调用发生异常时指定容错类，而该容错类需实现FeignClient接口。
  - ①fallback：指定的容错类仅仅实现FeignClient接口
  - ②fallbackFactory：指定的容错类还可提供具体的异常信息。

增强后的PaymentFeignClient接口

```
package com.scst.feignclient;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
//声明Feign客户端,其中value用于指定被调用的微服务,fallback或fallbackFactory用于指定容错类
//@FeignClient(value = "payment-service",fallback = PaymentFallback.class)
@FeignClient(value = "payment-service",fallbackFactory = PaymentFallbackFactory.class)
public interface PaymentFeignClient {
    //调用pay方法:http://[payment-service address]/payment/pay/{id}
    @GetMapping("/payment/pay/{id}")
    public String pay(@PathVariable("id") String accountId);
}
```

PaymentFallback类

```
package com.scst.feignclient;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
//容错类，要求必须实现被容错的接口,并为每个方法实现容错方案
@Component
public class PaymentFallback implements PaymentFeignClient {
    @Override
    public String pay(String accountId){
        return "FeignClient调用payment-service微服务失败："+ HttpStatus.INTERNAL_SERVER_ERROR;
    }
}
```

PaymentFallbackFactory类

```
package com.scst.feignclient;
import feign.hystrix.FallbackFactory;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
//可拿到具体错误的容错类，也要求必须实现被容错的接口,并为每个方法实现容错方案
@Component
public class PaymentFallbackFactory implements FallbackFactory<PaymentFeignClient> {
    @Override
    public PaymentFeignClient create(Throwable cause){
        return new PaymentFeignClient() {
            @Override
            public String pay(String accountId) {
                cause.printStackTrace();
                return "FeignClient调用payment-service微服务失败："+ HttpStatus.INTERNAL_SERVER_ERROR;
            }
        };
    }
}
```

6.自定义微服务容错处理

- 1）自定义异常返回信息
- 2）自定义异常处理策略

1）自定义异常返回信息

- 当发生限流降级产生BlockException异常时，Sentinel都是向请求方返回[Blocked by Sentinel (flow limiting)]错误信息。
- 事实上，BlockException包含以下5个Sentinel子异常，并且Sentinel提供了一个BlockExceptionHandler接口，让开发人员可以根据子异常类型返回不同错误提示语。
  - ✧ FlowException：限流异常
  - ✧ DegradeException：熔断降级异常
  - ✧ ParamFlowException：参数限流异常
  - ✧ AuthorityException：授权异常
  - ✧ SystemBlockException：系统负载异常
- 为此，在payment-service模块中增加一个BlockExceptionHandler的实现类MyBlockExceptionInfoHandler，当发生BlockException时，Sentinel会自动调用该实现类返回错误提示语。

MyBlockExceptionInfoHandler类

```
package com.scst.handler;
import com.alibaba.csp.sentinel.adapter.spring.webmvc.callback.BlockExceptionHandler;
import com.alibaba.csp.sentinel.slots.block.BlockException;
import com.alibaba.csp.sentinel.slots.block.authority.AuthorityException;
import com.alibaba.csp.sentinel.slots.block.degrade.DegradeException;
import com.alibaba.csp.sentinel.slots.block.flow.FlowException;
import com.alibaba.csp.sentinel.slots.block.flow.param.ParamFlowException;
import com.alibaba.csp.sentinel.slots.system.SystemBlockException;
import com.alibaba.fastjson.JSON;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.stereotype.Component;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

MyBlockExceptionInfoHandler类

```
@Component
public class MyBlockExceptionInfoHandler implements BlockExceptionHandler {
    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response, BlockException e) throws IOException {
        response.setContentType("application/json;charset=utf-8");
        ResponseData data = null;
        if (e instanceof FlowException) {
            data = new ResponseData(-1, "流控规则不通过");
        } else if (e instanceof DegradeException) {
            data = new ResponseData(-2, "熔断规则不通过");
        } else if (e instanceof ParamFlowException) {
            data = new ResponseData(-3, "热点规则不通过");
        } else if (e instanceof AuthorityException) {
            data = new ResponseData(-4, "授权规则不通过");
        }else if (e instanceof SystemBlockException) {
            data = new ResponseData(-5, "系统规则不通过");
        }
        response.getWriter().write(JSON.toJSONString(data));
    }
}
```

MyBlockExceptionInfoHandler类

```
@Data
@AllArgsConstructor//全参构造
@NoArgsConstructor//无参构造
class ResponseData {
    private int code;
    private String message;
}
```

2）自定义异常处理策略

- Sentinel主要提供了2种应用保护方式：
  - ✧①直接拦截对controller的uri请求
  - ✧②通过@SentinelResource注解

①直接拦截对controller的uri请求

- 当我们为资源配置了规则后，Sentinel底层会通过一个拦截器对请求controller的uri进行拦截：
  - ✧com.alibaba.csp.sentinel.adapter.spring.webmvc.SentinelWebInterceptor
- 可以通过如下配置关闭对uri的保护：
  - ✧spring.cloud.sentinel.filter.enabled=false

②通过@SentinelResource注解

- @SentinelResource不但可以声明一个资源点，还可通过@SentinelResource的blockHandler、fallback等属性来指定该资源点出现异常时的处理策略。

@SentinelResource的属性和作用

属性	作用	是否必须
value	资源名称	是
entryType	entry类型，标记流量的方向，取值IN/OUT，默认是OUT	否
blockHandler	处理BlockException的方法名称。方要求: 1. 必须是 public 2.返回类型与原资源方法一致 3. 参数类型需与原资源方法相匹配，并在最后加BlockException 类型的参数。 4. 默认需和原资源方法在同一个类中。若希望使用其他类的方法，可配置blockHandlerClass，并指定使用blockHandlerClass里面的哪个方法来处理。	否
blockHandlerClass	blockHandler外置类名。对应的处理方法必须static修饰，否则无法解析，其他要求：同blockHandler。	否
fallback	处理任何类型异常（除了 exceptionsToIgnore 里面排除掉的异常类型）的方法名称。方法要求: 1. 必须是 public 2.返回类型与原资源方法一致 3. 参数类型需与原资源方法相匹配，并在最后加Throwable 类型的参数。 4. 默认需和原资源方法在同一个类中。若希望使用其他类的方法，可配置fallbackClass，并指定使用fallbackClass里面的哪个方法来处理。	否
fallbackClass	fallback外置类名。对应的处理方法必须static修饰，否则无法解析，其他要求：同fallback。	否
defaultFallback	用于通用的 fallback 逻辑。若同时配置了fallback和defaultFallback，以fallback为准方法参数列表为空，或者有一个 Throwable 类型的参数，其他要求同fallback。	否
exceptionsToIgnore	指定排除掉哪些异常。排除的异常不会计入异常统计，也不会进入fallback逻辑，而是原样抛出。	否
exceptionsToTrace	需要trace的异常	Throwable

增强后的资源方法

```
@GetMapping("/pay/{id}")
@SentinelResource(value = "pay",blockHandlerClass = SentinelResourceExceptionHandler.class,
        blockHandler = "blockHandler",
        fallbackClass = SentinelResourceExceptionHandler.class,
        fallback = "fallback")
public String pay(@PathVariable("id") String accountId) {
    String payResult=accountFeignClient.reduce(accountId);
    boolean flag = paymentService.pay(accountId);
    String serverPort=environment.getProperty("server.port");
    return payResult+" 支付服务端口： "+serverPort;
}
```

异常外置处理类

```
package com.scst.handler;
import com.alibaba.csp.sentinel.slots.block.BlockException;
import lombok.extern.slf4j.Slf4j;
@Slf4j
public class SentinelResourceExceptionHandler {
    //BlockException时进入的方法
    public static String blockHandler(String accountId,BlockException ex) {
        log.error("{}", ex);
        return "账号："+accountId+"，[此为被限流降级处理结果]";
    }
    //Throwable时进入的方法
    public static String fallback(String accountId,Throwable throwable) {
        log.error("{}", throwable);
        return "账号："+accountId+"，[此为被容错处理结果]";
    }
}
```

7.效果测试

- 分别启动chapter09工程中的4个微服务，其中payment-service服务启动3个实例。
- 分别配置以下3个规则，然后访问受控资源，观察测试结果。
  - ✧流控规则
  - ✧授权规则
  - ✧系统规则

流控规则

新增流控规则

资源名 pay

针对来源 default

阈值类型 ◉ QPS ○ 并发线程数　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

新增并继续添加　新增　取消

授权规则

新增授权规则

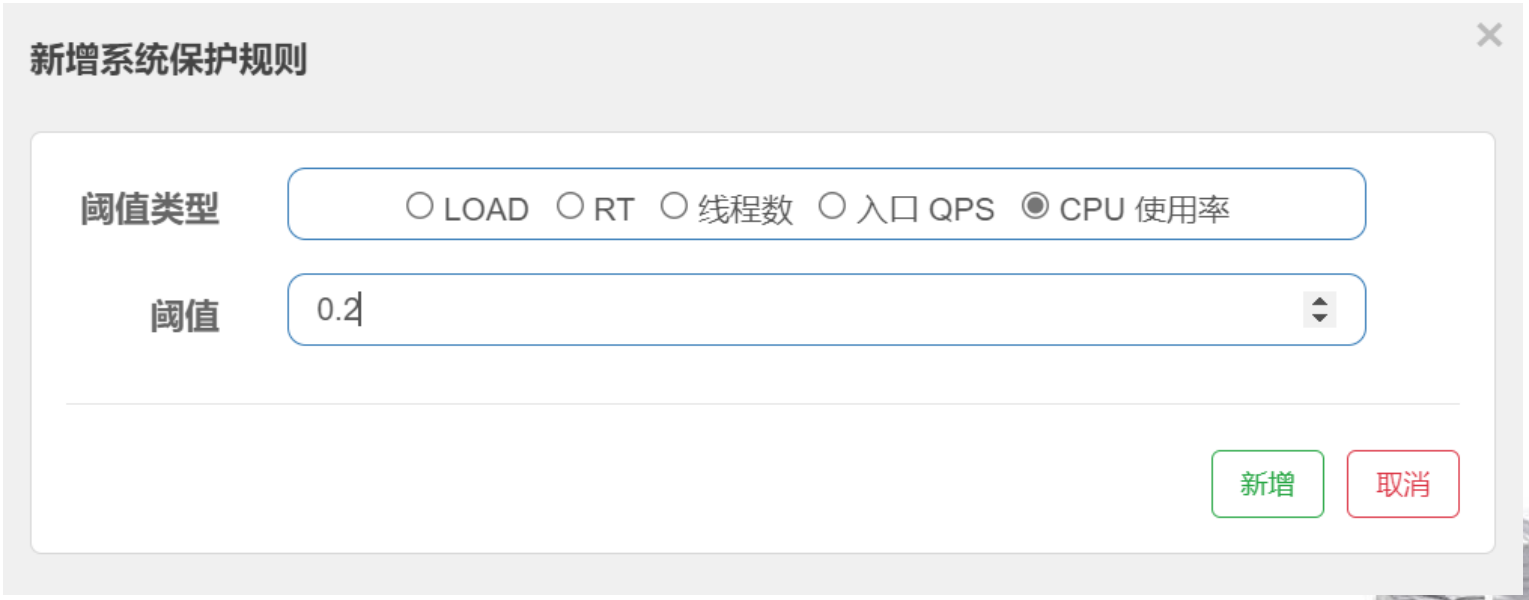
资源名 /payment/pay/{id}

流控应用 /payment/pay/a002,/payment/pay/a001

授权类型 ○ 白名单 ◉ 黑名单

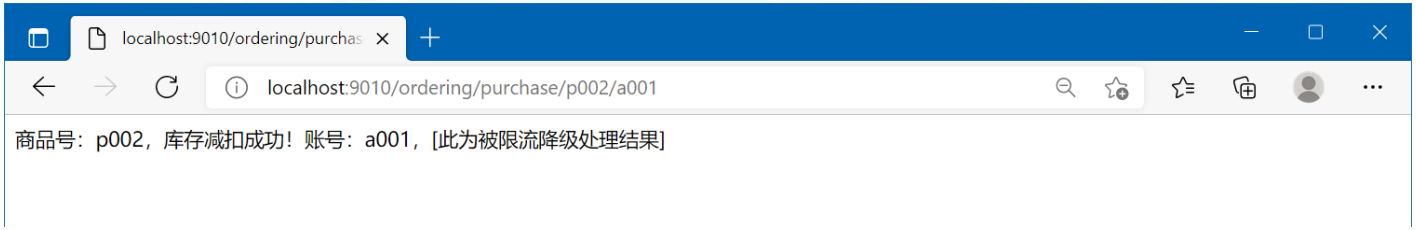
新增并继续添加 新增 取消

系统规则

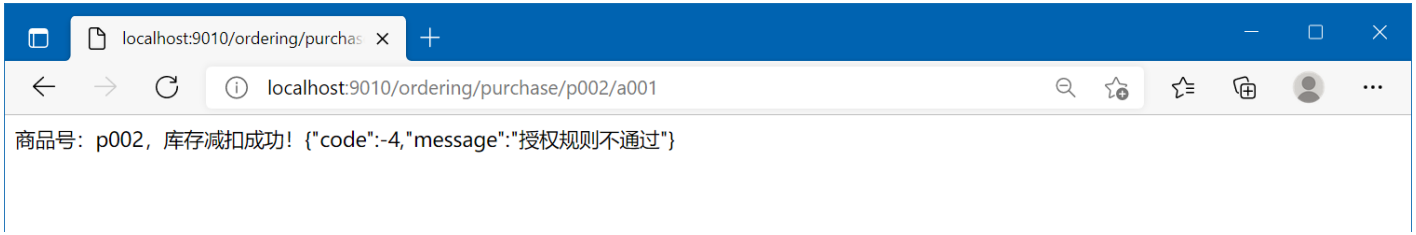


测试结果

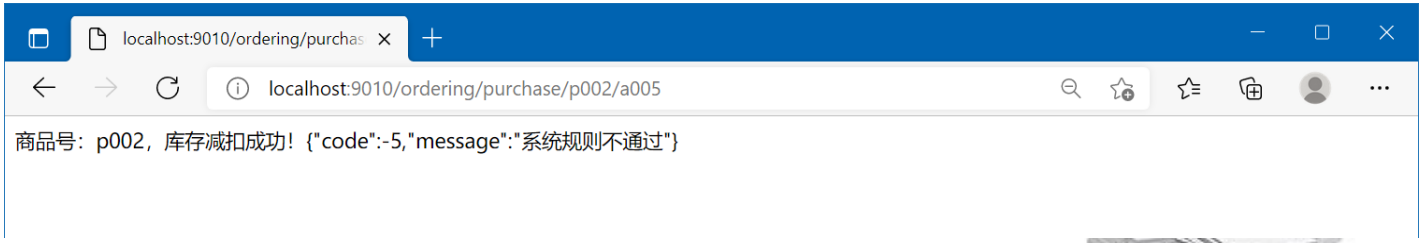
流控规则结果

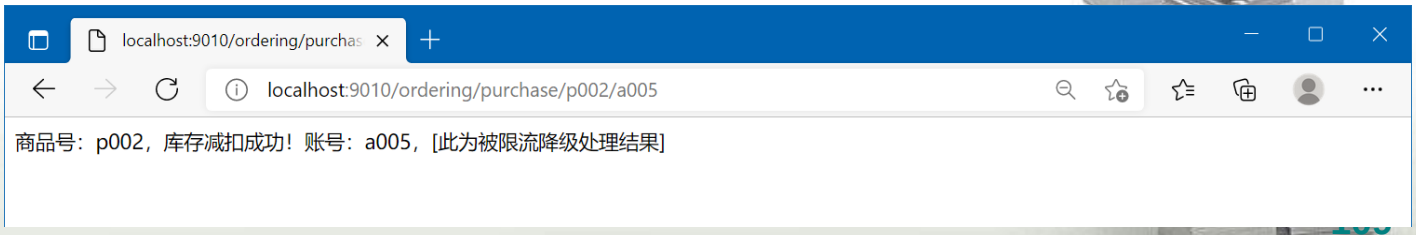


授权规则结果



系统规则结果






聪明等于勤奋
劳力就是幸福