主讲老师： Fox

学习本课程基础：

- **熟悉gateway网关使用。**
- **熟悉nginx使用。**
- **熟悉sentinel的应用，会涉及网关规则持久化改造。**

这节课实战部分跟不上的同学可以补一下之前的微服务gateway网关和Sentinel的课程

1. 秒杀场景介绍

1.1 秒杀场景的特点

1）**秒杀具有瞬时高并发的特点**，秒杀请求在时间上高度集中于某一特定的时间点（秒杀开始那一秒），这样一来，就会导致一个特别高的流量峰值，它对资源的消耗是瞬时的。

2）但是对秒杀这个场景来说，最终能够抢到商品的人数是固定的，也就是说 100 人和 10000 人发起请求的结果都是一样的，**并发度越高，无效请求也越多**。

3）但是**从业务上来说，秒杀活动是希望更多的人来参与的**，也就是开始之前希望有更多的人来刷页面，但是真正开始下单时，秒杀请求并不是越多越好。

1.2 流量消峰

服务器的处理资源是恒定的，你用或者不用它的处理能力都是一样的，所以出现峰值的话，很容易导致忙到处理不过来，闲的时候却又没有什么要处理。流量削峰，一是可以让服务端处理变得更加平稳，二是可以节省服务器的资源成本。针对秒杀这一场景，削峰从本质上来说就是更多地延缓用户请求的发出，以便减少和过滤掉一些无效请求，它遵从“请求数要尽量少”的原则。流量削峰的比较常见的思路：排队、答题、分层过滤。

1.3 兜底方案

对于很多秒杀系统而言，在诸如双十一这样的大流量的迅猛冲击下，都曾经或多或少发生过宕机的情况。当一个系统面临持续的大流量时，它其实很难单靠自身调整来恢复状态，你必须等待流量自然下降或者人为地把流量切走才行，这无疑会严重影响用户的购物体验。

我们可以在系统达到不可用状态之前就做好流量限制，防止最坏情况的发生。针对秒杀系统，在遇到大流量时，更多考虑的是运行阶段如何保障系统的稳定运行，常用的手段：限流，降级，拒绝服务。

2. 限流实战

限流相对降级是一种更极端的保存措施，限流就是当系统容量达到瓶颈时，我们需要通过限制一部分流量来保护系统，并做到既可以人工执行开关，也支持自动化保护的措施。

限流既可以是在客户端限流，也可以是在服务端限流。限流的实现方式既要支持 URL 以及方法级别的限流，也要支持基于 QPS 和线程的限流。

客户端限流

好处：可以限制请求的发出，通过减少发出无用请求从而减少对系统的消耗。

缺点：当客户端比较分散时，没法设置合理的限流阈值：如果阈值设的太小，会导致服务端没有达到瓶颈时客户端已经被限制；而如果设的太大，则起不到限制的作用。

服务端限流

好处：可以根据服务端的性能设置合理的阈值

缺点：被限制的请求都是无效的请求，处理这些无效的请求本身也会消耗服务器资源。

在限流的实现手段上来讲，基于 QPS 和线程数的限流应用最多，最大 QPS 很容易通过压测提前获取，例如我们的系统最高支持 1w QPS 时，可以设置 8000 来进行限流保护。线程数限流在客户端比较有效，例如在远程调用时我们设置连接池的线程数，超出这个并发线程请求，就将线程进行排队或者直接超时丢弃。

限流必然会导致一部分用户请求失败，因此在系统处理这种异常时一定要设置超时时间，防止因被限流的请求不能 fast fail（快速失败）而拖垮系统。

限流的方案

- 前端限流

- 接入层nginx限流
- 网关限流
- 应用层限流

2.1 nginx限流

https://nginx.org/en/docs/

- ngx_http_keyval_module
- ngx_http_limit_conn_module
- ngx_http_limit_req_module
- ngx_http_log_module

```
# window下nginx强制关闭命令
taskkill /fi "imagename eq nginx.EXE" /f
# 启动nginx
start nginx.exe
# 重新加载配置
nginx.exe -s reload
```

limit_conn_zone&limit_conn

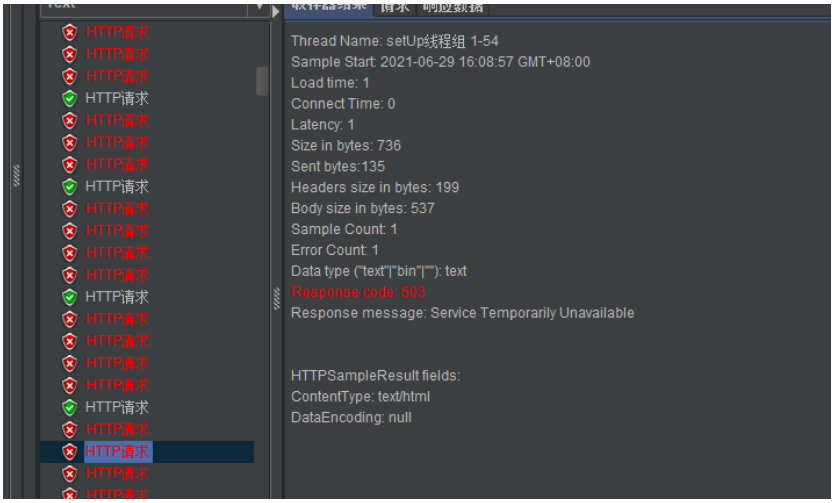
ngx_http_limit_conn_module 可以对于一些服务器流量异常、负载过大，甚至是大流量的恶意攻击访问等，进行并发数的限制；该模块可以根据定义的键来限制每个键值的连接数，只有那些正在被处理的请求（这些请求的头信息已被完全读入）所在的连接才会被计数。

```

# 限制连接数
limit_conn_zone $binary_remote_addr zone=addr:10m;

server {
 location /download/ {
 # 指定每个给定键值的最大同时连接数，同一IP同一时间只允许有1个连接
 limit_conn addr 1;
 }
```

> 客户端的IP地址作为键。
> binary_remote_addr变量的长度是固定的4字节，存储状态在32位平台中占用32字节或64字节，在64位平台中占用64字节。
> 1M共享空间可以保存3.2万个32位的状态，1.6万个64位的状态。
> 如果共享内存空间被耗尽，服务器将会对后续所有的请求返回 503 (Service Temporarily Unavailable) 错误。

缺陷： 前端做LVS或反向代理，会出现大量的503错误，需要设置白名单（对某些ip不做限制）

测试： http://localhost/pms/productInfo/29



limit_req_zone&limit_req

通过ngx_http_limit_req_module 模块可以通过定义的键值来限制请求处理的频率。特别的，可以限制来自单个IP地址的请求处理频率。 限制的方法如同漏斗，每秒固定处理请求数，推迟过多请求。

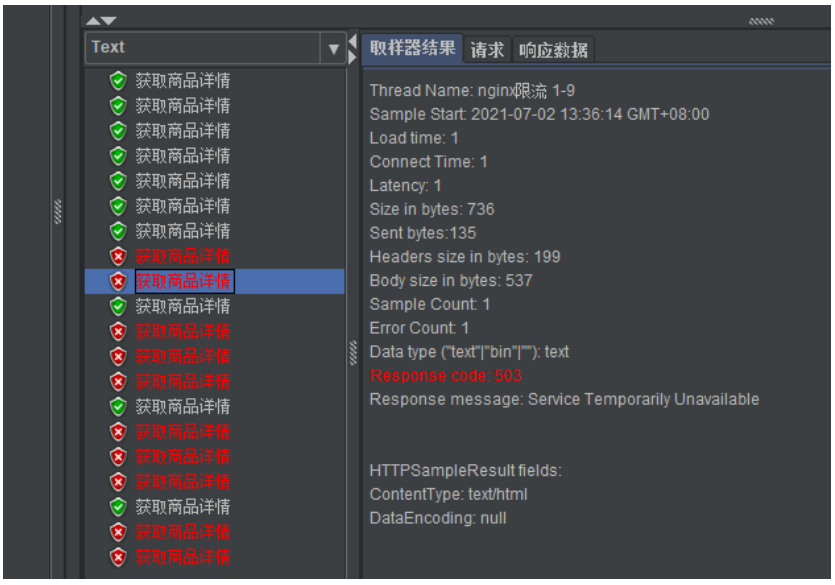
```
http {
```

```
# 限制请求数，大小为10m，平均处理的频率不能超过每秒1次
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;

...

server {

...

 location /search/ {
 # 允许超出频率限制的请求数为5，默认会被延迟处理，如果不希望延迟处理，可以使用nodelay参数
 limit_req zone=one burst=5 nodelay;
 }
```

区域名称为one，大小为10m，平均处理的请求频率不能超过每秒一次。键值是客户端IP。

> 使用$binary_remote_addr变量，可以将每条状态记录的大小减少到64个字节，这样1M的内存可以保存大约1万6千个64字节的记录
> 如果限制域的存储空间耗尽了，对于后续所有请求，服务器都会返回503（Service Temporarily Unavailable）错误
> 速度可以设置为每秒处理请求数和每分钟处理请求数，其值必须是整数，所以如果你需要每秒处理少于1个的请求，2秒处理一个请求，可以使用30r/m

测试：

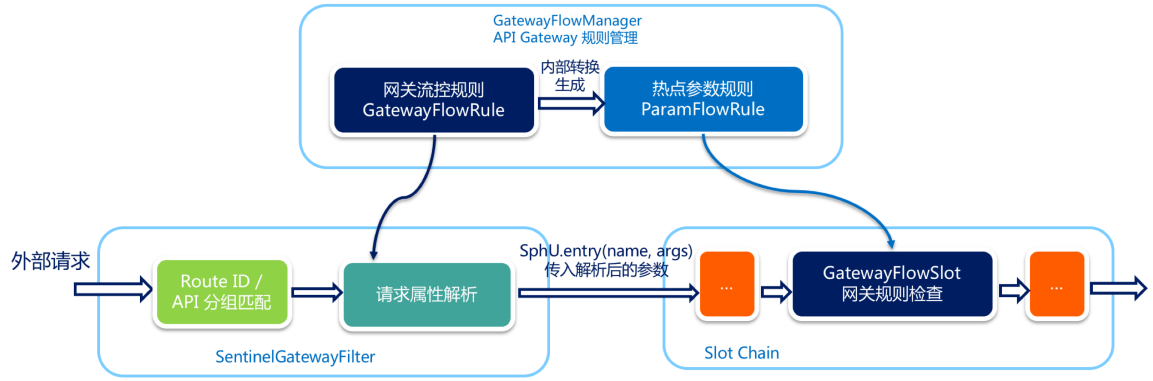


利用Lua限流

https://github.com/openresty/lua-resty-limit-traffic

2.2 网关限流

spring cloud gateway接入sentinel实现限流的原理：



2.2.1 网关接入sentinel控制台

建议sentinel 控制台和微服务sentinel版本一一对应，否则可能出现兼容性问题导致规则配置失效

引入依赖

```
<!--添加Sentinel的依赖-->
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>

<!-- gateway接入sentinel -->
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-alibaba-sentinel-gateway</artifactId>
</dependency>
```

接入sentinel控制台，修改application.yml配置

```
spring:
 application:
 name: tulingmall-gateway
 main:
 allow-bean-definition-overriding: true
 cloud:
 sentinel:
 transport:
 dashboard: 127.0.0.1:8000
```

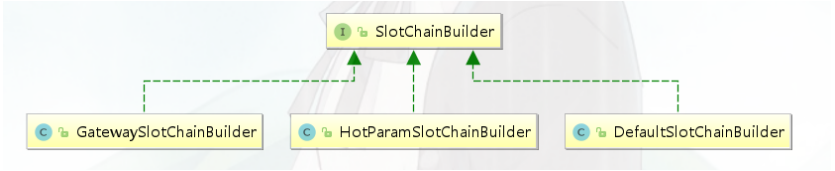
启动sentinel控制台

```
java -Dserver.port=8000 -Dsentinel.nacos.config.serverAddr=tl.nacos.com:8848 -jar sentinel-dashboard-1.7.1.jar
```

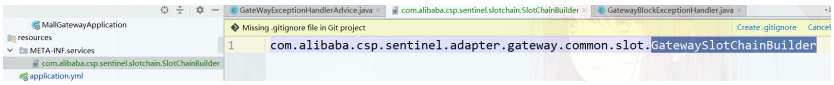
网关接入控制台后界面展示：

Sentinel1.7.1版本，gateway网关规则不生效的问题

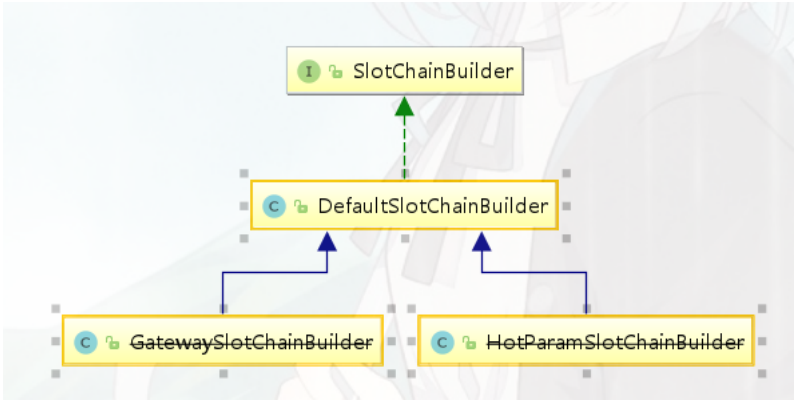
根本原因： SlotChain中没有添加GatewayFlowSlot ，默认生效的是HotParamSlotChainBuilder



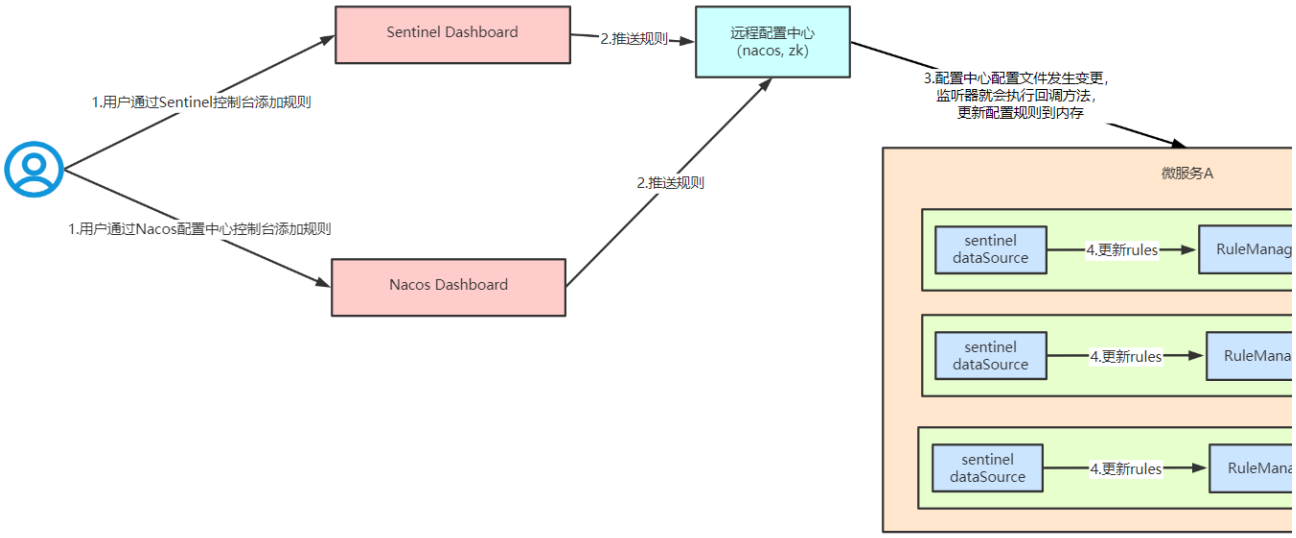
解决思路： 使用GatewaySlotChainBuilder，将GatewayFlowSlot加入到SlotChain
可以利用SPI实现：在当前微服务添加GatewaySlotChainBuilder的spi文件



Sentinel1.8.0 中SlotChain处理策略，统一在DefaultSlotChainBuilder中处理了



2.2.2 Sentinel规则持久化配置



引入依赖

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

application.yml添加datasource配置

```
spring:
  cloud:
  sentinel:
  transport:
  # 添加sentinel的控制台地址
  dashboard: 127.0.0.1:8000
  datasource:
  gateway-flow-rules:
  nacos:
   server-addr: 127.0.0.1:8848
   dataId: ${spring.application.name}-gateway-flow-rules
   groupId: SENTINEL_GROUP
   data-type: json
   rule-type: gw-flow
```

```
gateway-api-rules:
nacos:
server-addr: 127.0.0.1:8848
dataId: ${spring.application.name}-gateway-api-rules
groupId: SENTINEL_GROUP
data-type: json
rule-type: gw-api-group
```

启动持久化改造后的sentinel dashboard

指定端口和nacos配置中心地址

```
java -Dserver.port=8000 -Dsentinel.nacos.config.serverAddr=tl.nacos.com:8848 -jar tuling-sentinel-dashboard.jar
```

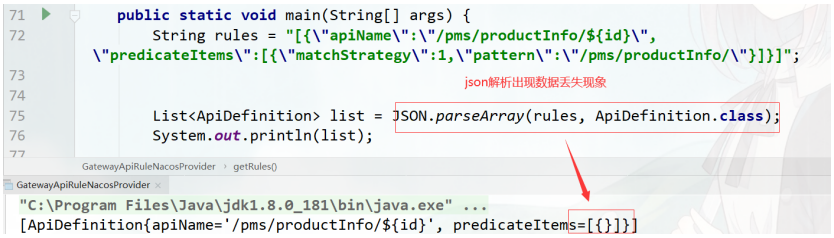
注意：网关规则改造的坑

1. 网关规则实体转换

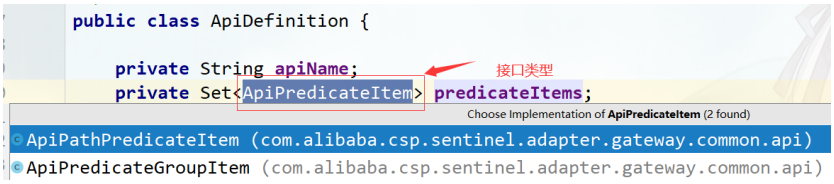
```
RuleEntity--->》Rule 利用RuleEntity#toRule
#网关规则实体
ApiDefinitionEntity--->》ApiDefinition 利用ApiDefinitionEntity#toApiDefinition
GatewayFlowRuleEntity----->GatewayFlowRule 利用GatewayFlowRuleEntity#toGatewayFlowRule
```

2. json解析丢失数据

json解析ApiDefinition类型出现数据丢失的现象



排查原因： *ApiDefinition的属性Set<ApiPredicateItem> predicateItems中元素 是接口类型，JSON解析丢失数据*



解决方案：重写实体类ApiDefinition2,再转换为ApiDefinition

```
//GatewayApiRuleNacosProvider.java

@Override
public List<ApiDefinitionEntity> getRules(String appName,String ip,Integer port) throws Exception {
 String rules = configService.getConfig(appName + NacosConfigUtil.GATEWAY_API_DATA_ID_POSTFIX,
 NacosConfigUtil.GROUP_ID, NacosConfigUtil.READ_TIMEOUT);
 if (StringUtil.isEmpty(rules)) {
 return new ArrayList<>();
 }

 // 注意 ApiDefinition的属性Set<ApiPredicateItem> predicateItems中元素 是接口类型，JSON解析丢失数据
 // 重写实体类ApiDefinition2,再转换为ApiDefinition
 List<ApiDefinition2> list = JSON.parseArray(rules, ApiDefinition2.class);

 return list.stream().map(rule ->
 ApiDefinitionEntity.fromApiDefinition(appName, ip, port, rule.toApiDefinition()))
 .collect(Collectors.toList());
}


public class ApiDefinition2 {
 private String apiName;
 private Set<ApiPathPredicateItem> predicateItems;

 public ApiDefinition2() {
```

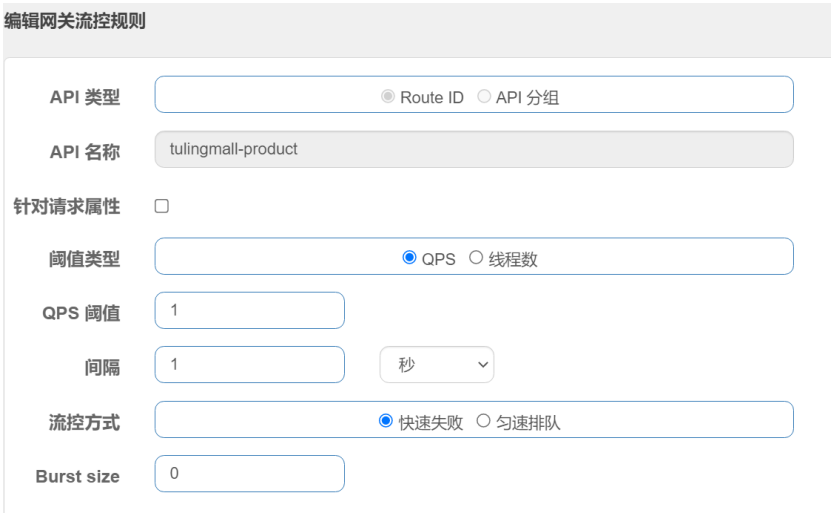
```
}

public String getApiName() {
return apiName;
}

public void setApiName(String apiName) {
this.apiName = apiName;
}

public Set<ApiPathPredicateItem> getPredicateItems() {
return predicateItems;
}

public void setPredicateItems(Set<ApiPathPredicateItem> predicateItems) {
this.predicateItems = predicateItems;
}

@Override
public String toString() {
return "ApiDefinition2{" + "apiName='" + apiName + '\'' + ", predicateItems=" + predicateItems + '}';
}


public ApiDefinition toApiDefinition() {
ApiDefinition apiDefinition = new ApiDefinition();
apiDefinition.setApiName(apiName);

Set<ApiPredicateItem> apiPredicateItems = new LinkedHashSet<>();
apiDefinition.setPredicateItems(apiPredicateItems);

if (predicateItems != null) {
for (ApiPathPredicateItem predicateItem : predicateItems) {
apiPredicateItems.add(predicateItem);
}
}

return apiDefinition;
}

}
```

从 1.6.0 版本开始，Sentinel 提供了 Spring Cloud Gateway 的适配模块，可以提供两种资源维度的限流：

- route 维度：即在 Spring 配置文件中配置的路由条目，资源名为对应的 routeId
- 自定义 API 维度：用户可以利用 Sentinel 提供的 API 来自定义一些 API 分组

route维度限流

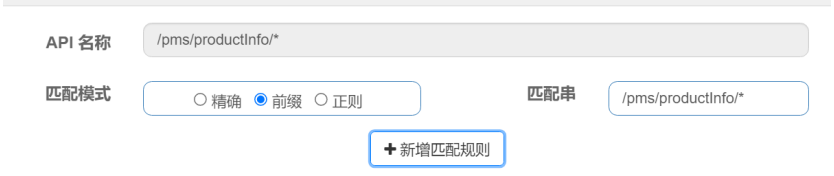
配置流控规则



测试： http://localhost:8888/pms/productInfo/29

配置流控规则

API维度限流



2.3 应用层限流

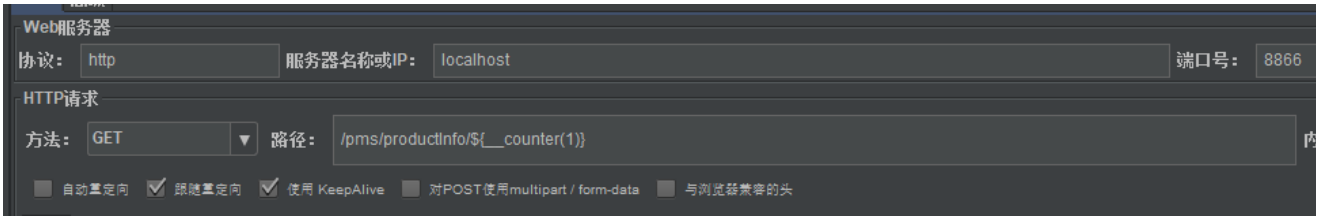
场景： 商品详情接口

系统第一次上线启动，或者系统在 redis 故障的情况下重新启动，这时在高并发的场景下就会出现所有的流量 都会打到 mysql（原始数据库） 上去，导致 mysql 崩溃。因此需要通过缓存预热的方案，提前给 redis 灌入部分数据后再提供服

务。

jemeter测试： 模拟2秒内查询商品id为1-5000的商品信息

/pms/productInfo/${__counter(,)}



压测直接访问DB的接口： 吞吐量： 20-60

```
public PmsProductParam getProductInfo1(Long id){
 PmsProductParam productInfo = portalProductDao.getProductInfo(id);
 if(null == productInfo){
 return null;
 }
 checkFlash(id, productInfo);
 return productInfo;
}
```

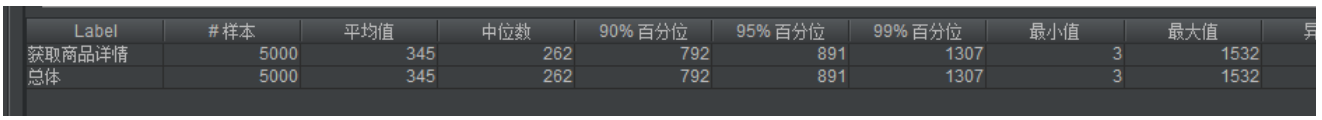
Label	# 样本	平均值	中位数	90% 百分位	95% 百分位	99% 百分位	最小值	最大值	异常 %	吞吐量	接收 KB/sec
获取商品详情	100	219	216	274	288	314	84	334	0.00%	46.8/sec	95.69
总体	100	219	216	274	288	314	84	334	0.00%	46.8/sec	95.69

压测访问缓存的接口：

```
public PmsProductParam getProductInfo2(Long id) {
 PmsProductParam productInfo = null;
 // 查询本地缓存
 productInfo = cache.get(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE + id);
 if (null != productInfo) {
 return productInfo;
 }
 // 查询redis缓存
 productInfo = redisOpsUtil.get(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE + id, PmsProductParam.class);
 if (productInfo != null) {
 //设置本地缓存
 cache.setLocalCache(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE + id, productInfo);
 return productInfo;
 }
 // 查询DB
 productInfo = portalProductDao.getProductInfo(id);
 if (null == productInfo) {
 return null;
 }
 checkFlash(id, productInfo);
 // 设置redis缓存
 redisOpsUtil.set(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE + id, productInfo, 3600, TimeUnit.SECONDS);
 // 设置本地缓存
 cache.setLocalCache(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE + id, productInfo);
 return productInfo;
}
```

第一次访问缓存击穿的吞吐量： 20-60

之后吞吐量： 1000-2800



Label	# 样本	平均值	中位数	90% 百分位	95% 百分位	99% 百分位	最小值	最大值
获取商品详情	5000	345	262	792	891	1307	3	1532
总体	5000	345	262	792	891	1307	3	1532

思考： 在没有事先进行缓存预热的情况下，如何避免更多的请求直接访问到数据库？

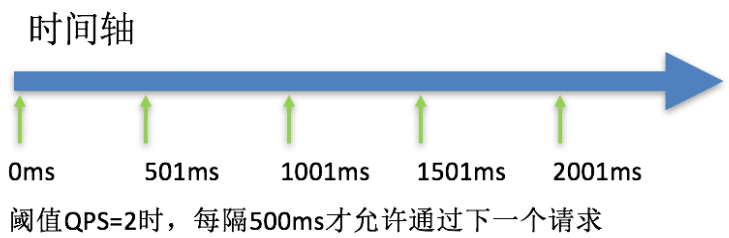
当对数据库访问达到阈值，可以对商品详情请求限流

配置流控规则

思考：排队等待可以应用于什么场景？

匀速排队（`RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER`）方式会严格控制请求通过的间隔时间，也即是让请求以均匀的速度通过，对应的是漏桶算法。

该方式的作用如下图所示：



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。

> 注意：匀速排队模式暂时不支持 QPS > 1000 的场景。

场景： 对秒杀接口进行流控

时间	通过 QPS	拒绝QPS	响应时间（ms）
21:27:47	199.0	0.0	1,003.0
21:27:46	201.0	0.0	947.0
21:27:45	201.0	0.0	835.0
21:27:44	201.0	0.0	1,383.0
21:27:43	197.0	0.0	1,034.0
21:27:42	201.0	0.0	964.0

热点参数限流

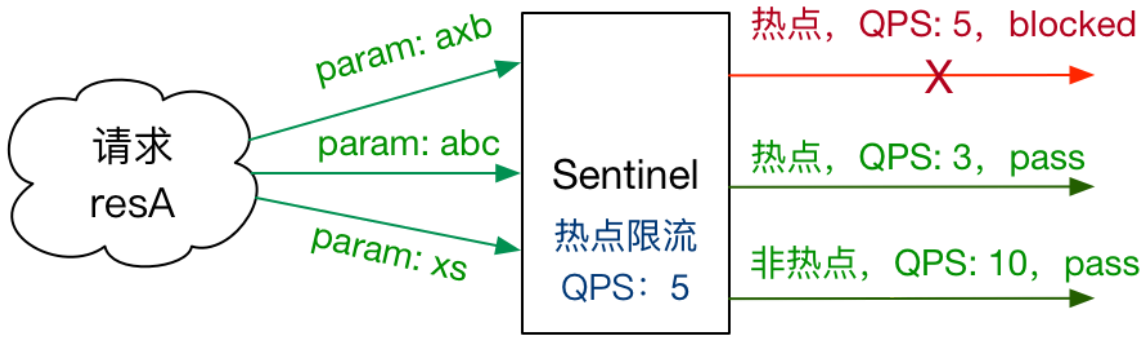
何为热点？热点即经常访问的数据。商家不定期做一些“商品秒杀”、“商品推广”活动，导致“营销活动”、“商品详情”、“交易下单”等链路应用出现 **缓存热点访问** 的情况：

- 活动时间、活动类型、活动商品之类的信息不可预期，导致 **缓存热点访问** 情况不可提前预知；
- **缓存热点访问** 出现期间，应用层少数 **热点访问 key** 产生大量缓存访问请求，冲击分布式缓存系统，大量占据内网带宽，最终影响应用层系统稳定性；

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。



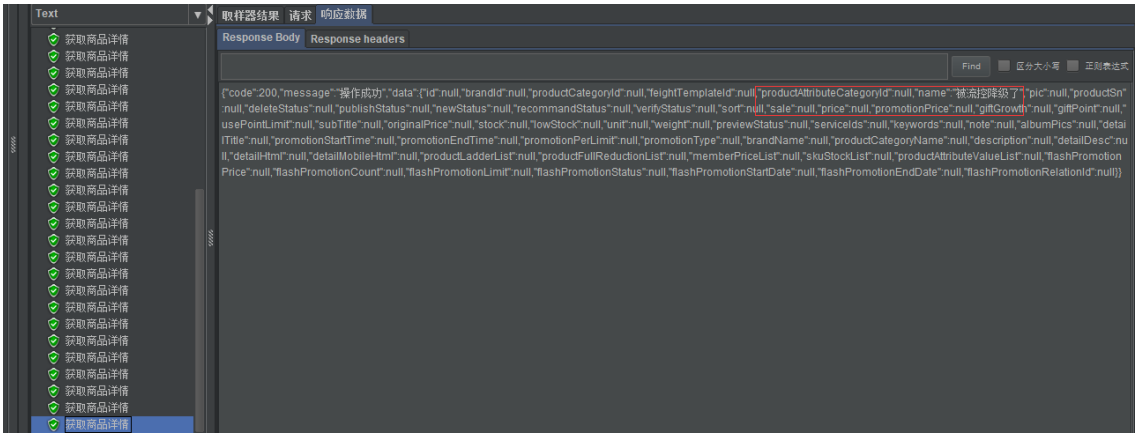
注意：

1. 热点规则需要使用@SentinelResource("resourceName")注解，否则不生效
2. 参数必须是7种基本数据类型才会生效

```
@RequestMapping(value = "/productInfo/{id}", method = RequestMethod.GET)
@SentinelResource(value = "getProductInfo")
public CommonResult getProductInfo(@PathVariable Long id) {
    PmsProductParam pmsProductParam=pmsProductService.getProductInfo(id);
    return CommonResult.success(pmsProductParam);
}
```

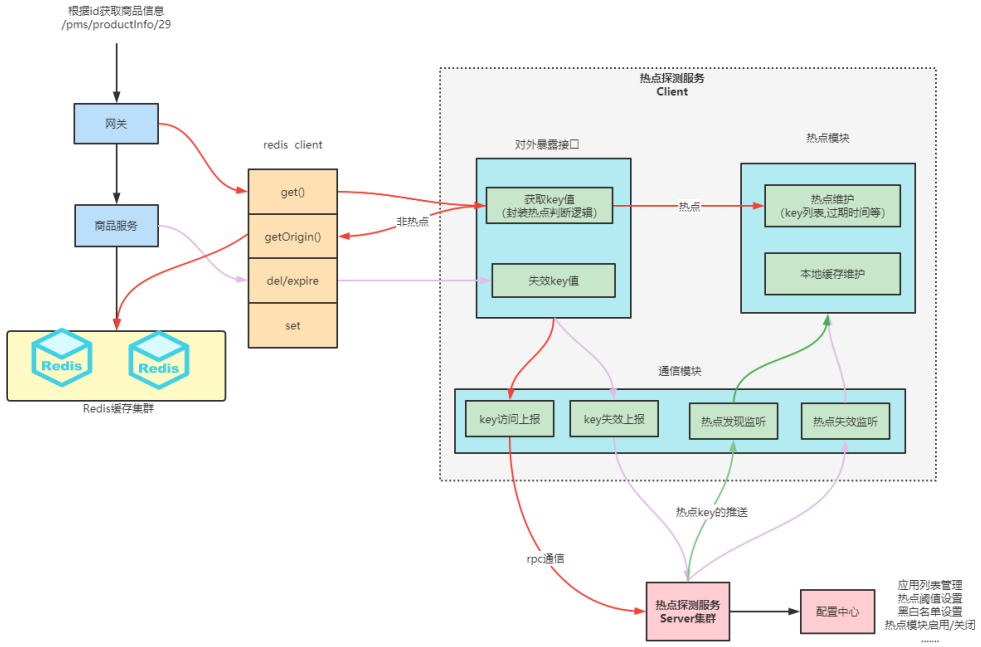
配置热点参数限流规则

测试： http://localhost:8866/pms/productInfo/26



思考： 如何快速且准确的发现热点访问key？

热点探测功能设计思路



3. 降级实战

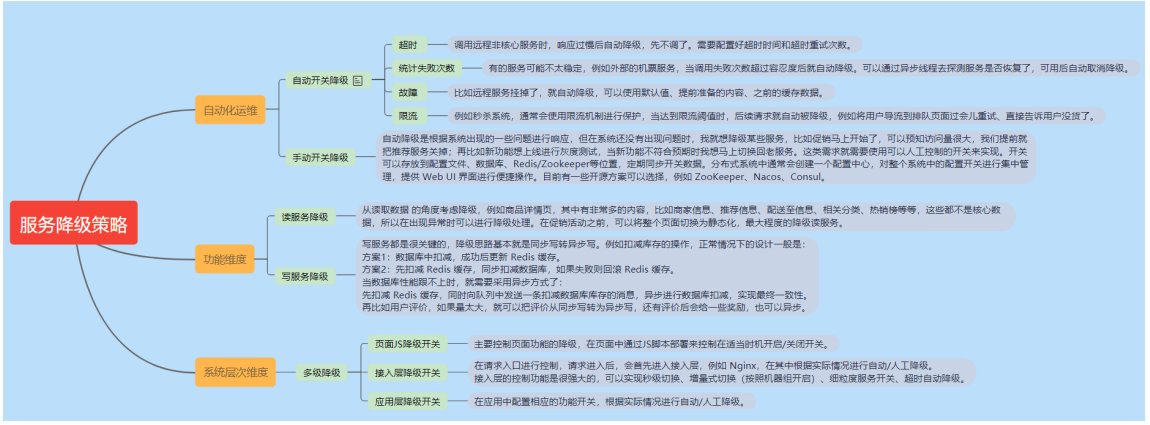
降级就是当系统的容量达到一定程度时，限制或者关闭系统的某些非核心功能，从而把有限的资源保留给更核心的业务。

比如降级方案可以这样设计：当秒杀流量达到 5w/s 时，把成交记录的获取从展示 20 条降级到只展示 5 条。“从 20 改到 5”这个操作由一个开关来实现，也就是设置一个能够从开关系统动态获取的系统参数。

降级的核心目标是牺牲次要的功能和用户体验来保证核心业务流程的稳定，是一个不得已而为之的举措。例如在双 11 零点时，如果优惠券系统扛不住，可能会临时降级商品详情的优惠信息展示，把有限的系统资源用在保障交易系统正确展示优惠信息上，即保障用户真正下单时的价格是正确的。

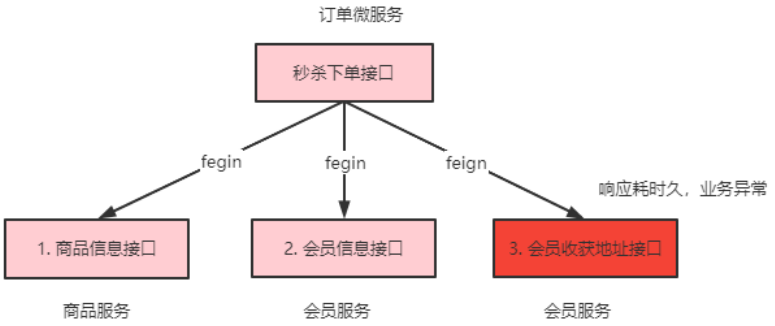
3.1 服务降级的策略

https://www.processon.com/view/link/60dc6e485653bb2a8d08850d



3.2 应用层降级实战

场景： 秒杀下单 /order/miaosha/generateOrder

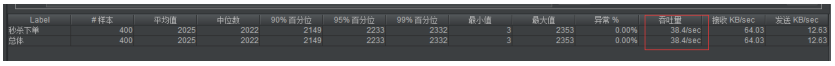


如果会员服务出现问题，会影响整个下单链路。

模拟查询会员地址信息出现网络问题和业务异常

```
@ApiOperation("显示收货地址详情")
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
@ResponseBody
public CommonResult<UmsMemberReceiveAddress> getItem(@PathVariable Long id,@RequestHeader("memberId") long memberId)
{
 if(memberId==3){
 try {
 //模拟网络问题
 Thread.sleep(2000);
 } catch (InterruptedException e) {
  e.printStackTrace();
 }
 }
 if(memberId==4){
 //模拟业务异常
 throw new IllegalArgumentException("非法参数异常");
 }
 UmsMemberReceiveAddress address = memberReceiveAddressService.getItem(id,memberId);
 return CommonResult.success(address);
}
```

测试： memberId为3的用户压测



Label	# 样本	平均值	中位数	90% 百分位	95% 百分位	99% 百分位	最小值	最大值	异常 %	吞吐量	接收 KB/sec	发送 KB/sec
秒杀下单	400	2025	2022	2149	2233	2332	3	2353	0.00%	38.4/sec	64.03	12.63
总体	400	2025	2022	2149	2233	2332	3	2353	0.00%	38.4/sec	64.03	12.63

Sentinel熔断降级

OpenFeign整合Sentinel

配置文件打开 Sentinel 对 Feign 的支持：feign.sentinel.enabled=true

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

feign接口配置fallbackFactory

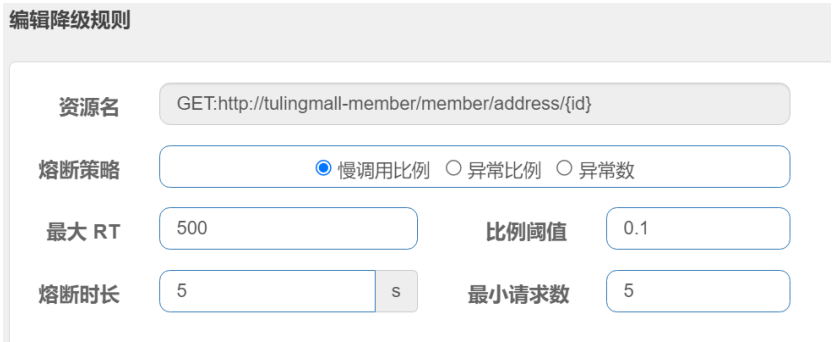
```
@FeignClient(name = "tulingmall-member",path = "/member",
 fallbackFactory = UmsMemberFeginFallbackFactory.class)
public interface UmsMemberFeignApi {
```

UmsMemberFeginFallbackFactory中编写降级逻辑

```
@Component
public class UmsMemberFeginFallbackFactory implements FallbackFactory<UmsMemberFeignApi> {

 @Override
 public UmsMemberFeignApi create(Throwable throwable) {

 return new UmsMemberFeignApi() {
 @Override
 public CommonResult<UmsMemberReceiveAddress> getItem(Long id) {
 //TODO 业务降级
```

```
11  UmsMemberReceiveAddress defaultAddress = new UmsMemberReceiveAddress();
12  defaultAddress.setName("默认地址");
13  defaultAddress.setId(-1L);
14  defaultAddress.setDefaultStatus(0);
15  defaultAddress.setPostCode("-1");
16  defaultAddress.setProvince("默认省份");
17  defaultAddress.setCity("默认city");
18  defaultAddress.setRegion("默认region");
19  defaultAddress.setDetailAddress("默认详情地址");
20  defaultAddress.setMemberId(-1L);
21  defaultAddress.setPhoneNumber("199xxxxxx");
22  return CommonResult.success(defaultAddress);
23  }
24
25  @Override
26  public CommonResult<String> updateUmsMember(UmsMember umsMember) {
27  return null;
28  }
29
30  @Override
31  public CommonResult<PortalMemberInfo> getMemberById() {
32  return null;
33  }
34
35  @Override
36  public CommonResult<List<UmsMemberReceiveAddress>> list() {
37  return null;
38  }
39  };
40  }
41
42
43  }
```

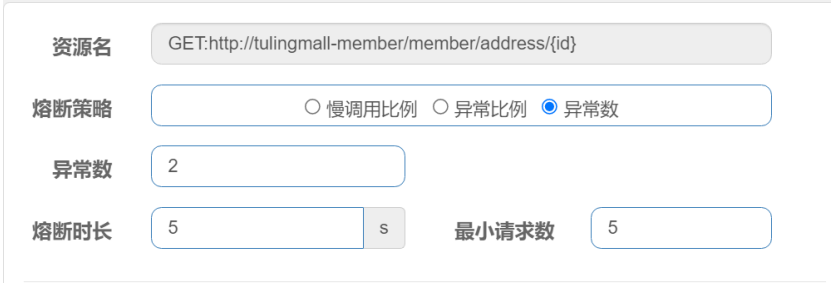
会员收货地址接口配置基于响应时间的降级规则



测试：

Label	# 样本	平均值	中位数	90% 百分位	95% 百分位	99% 百分位	最小值	最大值	异常 %	吞吐量	接收 KB/sec	发送 KB/sec
秒杀下单	2000	1019	834	2216	2531	2732	5	4764	0.00%	178.9/sec	231.09	58.89
总体	2000	1019	834	2216	2531	2732	5	4764	0.00%	178.9/sec	231.09	58.89

会员收货地址接口配置基于异常数的降级规则



测试

Label	# 样本	平均值	中位数	90% 百分位	95% 百分位	99% 百分位	最小值	最大值	异常 %	吞吐量	接收 KB/sec	发送 KB/sec
秒杀下单	2000	556	44	2055	2371	2743	3	3006	0.00%	199.6/sec	258.90	65.69
总体	2000	556	44	2055	2371	2743	3	3006	0.00%	199.6/sec	258.90	65.69

4. 拒绝服务

拒绝服务可以说是一种不得已的兜底方案，用以防止最坏情况发生，防止因把服务器压跨而长时间彻底无法提供服务。当系统负载达到一定阈值时，例如 CPU 使用率达到 90% 或者系统 load 值达到 2*CPU 核数时，系统直接拒绝所有请求，这种方式是最暴力但也最有效的系统保护方式。

例如秒杀系统，我们可以在以下环节设计过载保护：

- 在最前端的 Nginx 上设置过载保护，当机器负载达到某个值时直接拒绝 HTTP 请求并返回 503 错误码。

  阿里针对nginx开发的过载保护扩展插件sysguard：_https://github.com/alibaba/nginx-http-sysguard

- 在 Java 层同样也可以设计过载保护。 比如Sentinel提供了系统规则限流

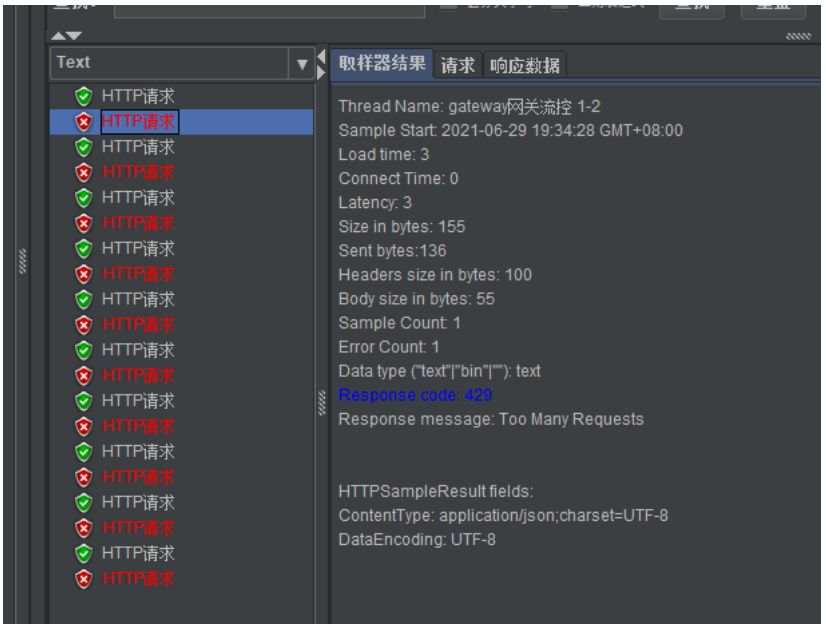
Sentinel系统规则限流

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

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

系统规则持久化yml配置

```yaml
system-rules:
  nacos:
  server-addr: tl.nacos.com:8848
  dataId: ${spring.application.name}-system-rules
  groupId: SENTINEL_GROUP
  data-type: json
  rule-type: system
```



文档：12 秒杀场景兜底方案之限流&降级实战.n...

链接：http://note.youdao.com/noteshare?id=825387831d6e5d22dab51a2aa9cad35e&sub=8D3AE906B7664246B1A37CE632E61306