主讲老师： Fox

有道笔记地址：https://note.youdao.com/s/TEpaBciM

1. 需求场景

对于很多电商系统而言，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。针对电商系统，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

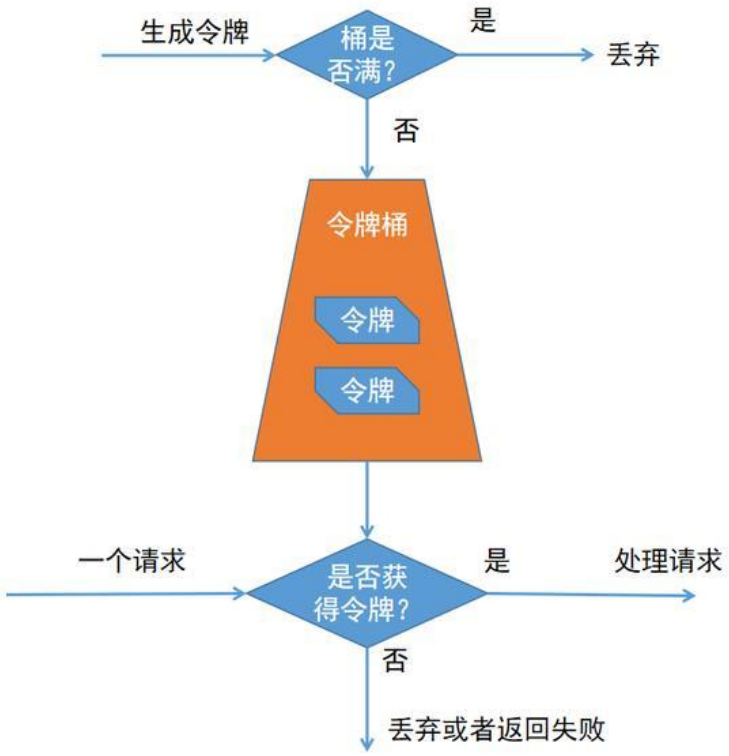
2.1 网关限流

业务场景

商品详情页入口流量防护：黑白名单，限制同一个 ip 访问频率，限制查询商品接口调用频率

方案一： 基于 redis+lua 脚本限流

gateway 官方提供了 RequestRateLimiter 过滤器工厂，基于 redis+lua 脚本方式采用令牌桶算法实现了限流。

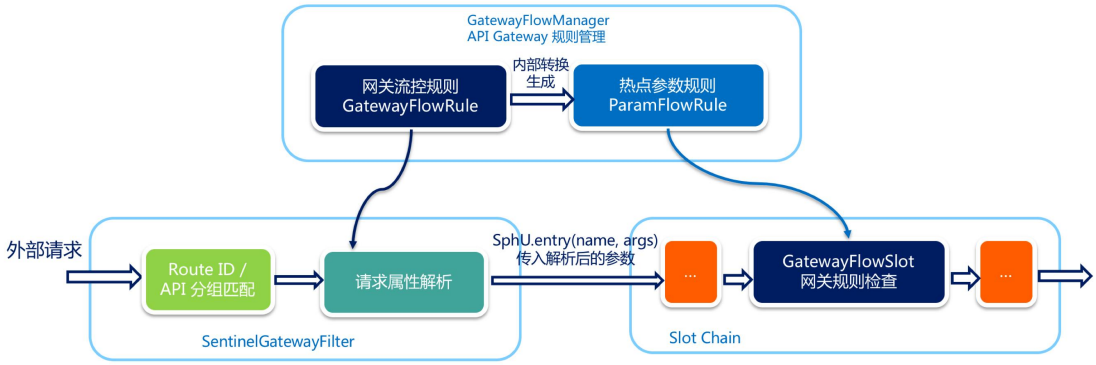


具体使用可以参考微服务专题 Spring Cloud Gateway 实战课上讲解

方案二：整合 sentinel 限流

利用 Sentinel 的网关流控特性，在网关入口处进行流量防护，或限制 API 的调用频率。

Spring Cloud Gateway 接入 Sentinel 实现限流的原理：



2.2 网关接入 sentinel 实战

1) 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>
```

2) 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: 192.168.65.103:8000
```

3) 启动 sentinel 控制台

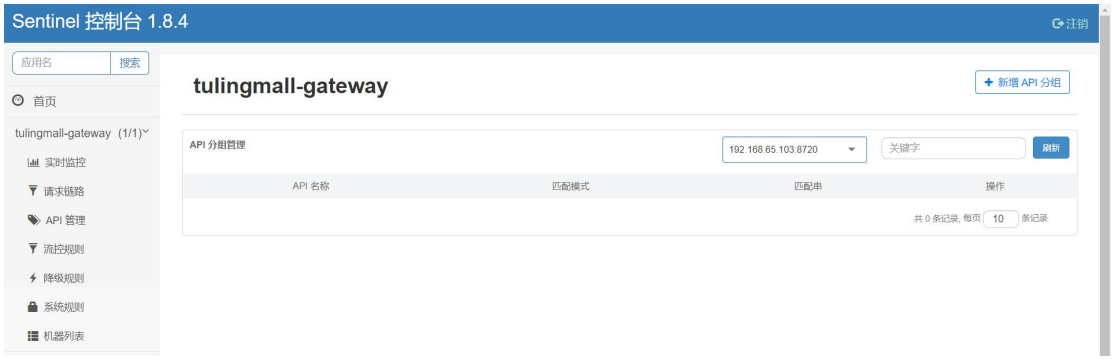
下载地址： https://github.com/alibaba/Sentinel/releases/download/1.8.4/sentinel-dashboard-1.8.4.jar

```
java    -Dserver.port=8000    -jar sentinel-dashboard-1.8.4.jar
```

访问 http://192.168.65.103:8000/#/login ,默认用户名密码： sentinel/sentinel

4）测试

启动网关服务，商品服务，访问商品详情接口 http://localhost:8888/pms/productInfo/27



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：

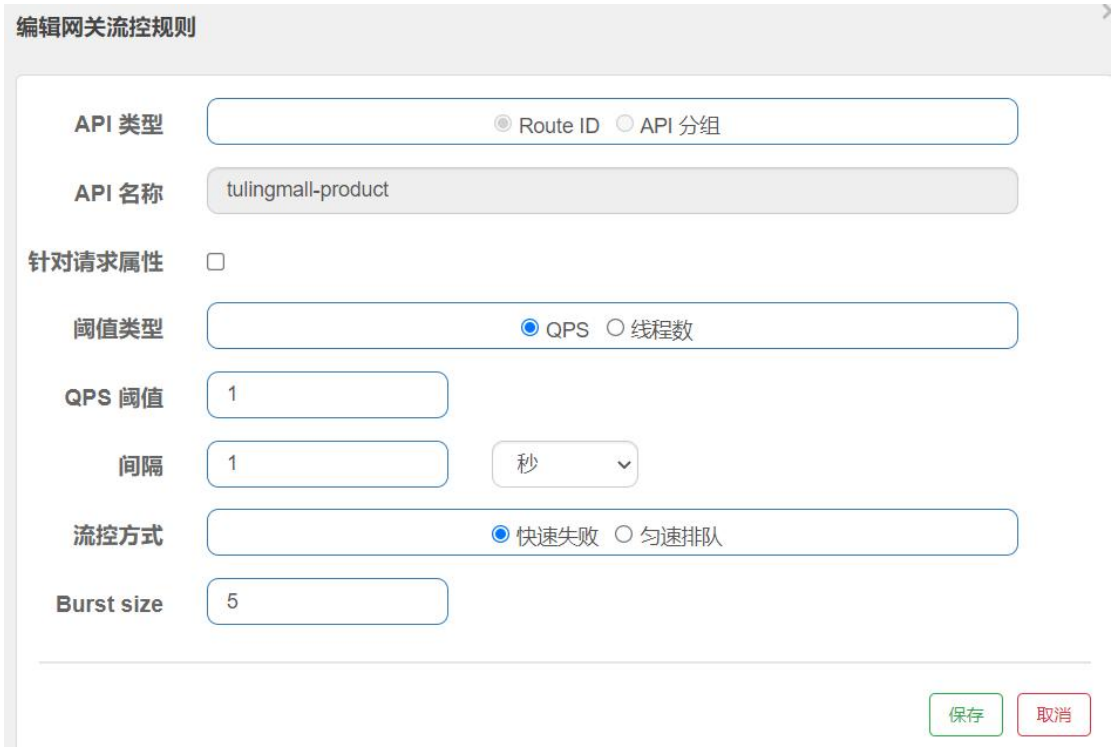
route 维度：即在 Spring 配置文件中配置的路由条目，资源名为对应的 routeId

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

需求：对商品详情接口进行流控

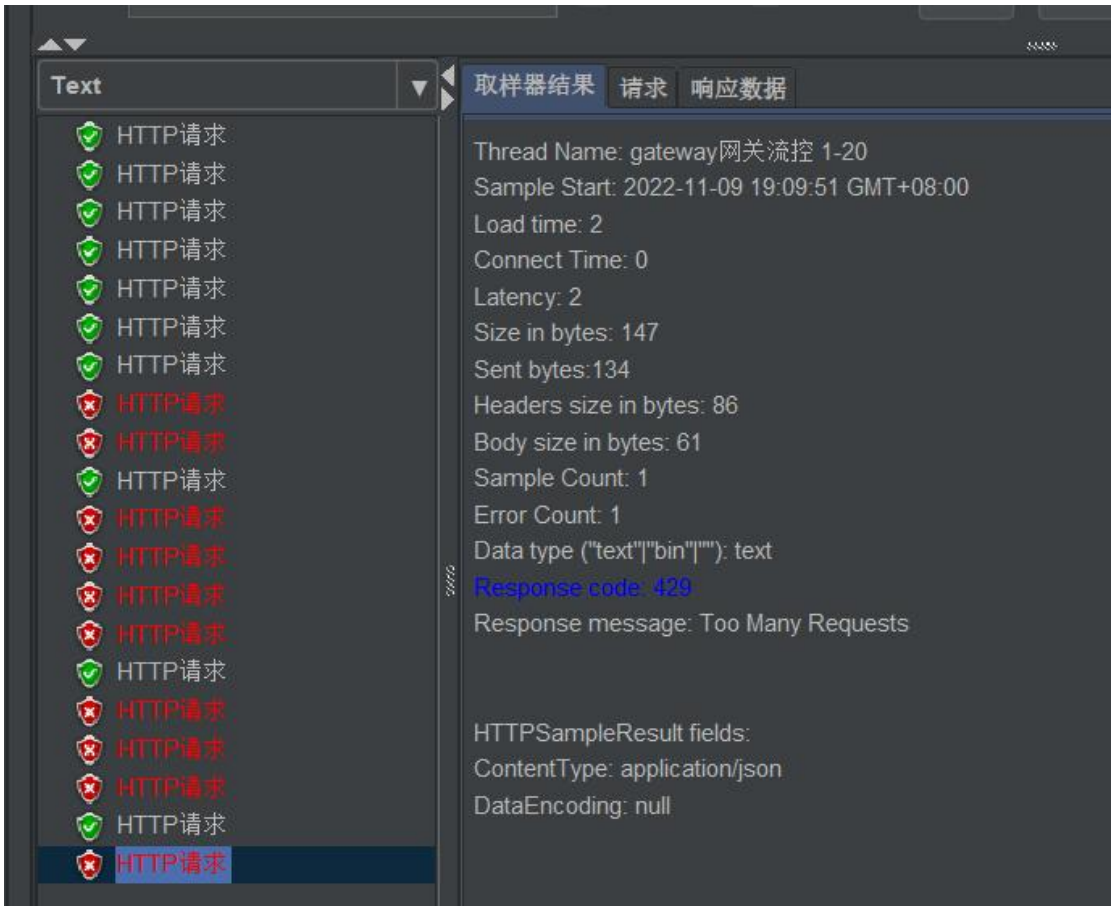
1）配置流控规则



2）jemeter 压测配置

压测接口：http://localhost:8888/pms/productInfo/27

3) 查看 jemeter 压测结果

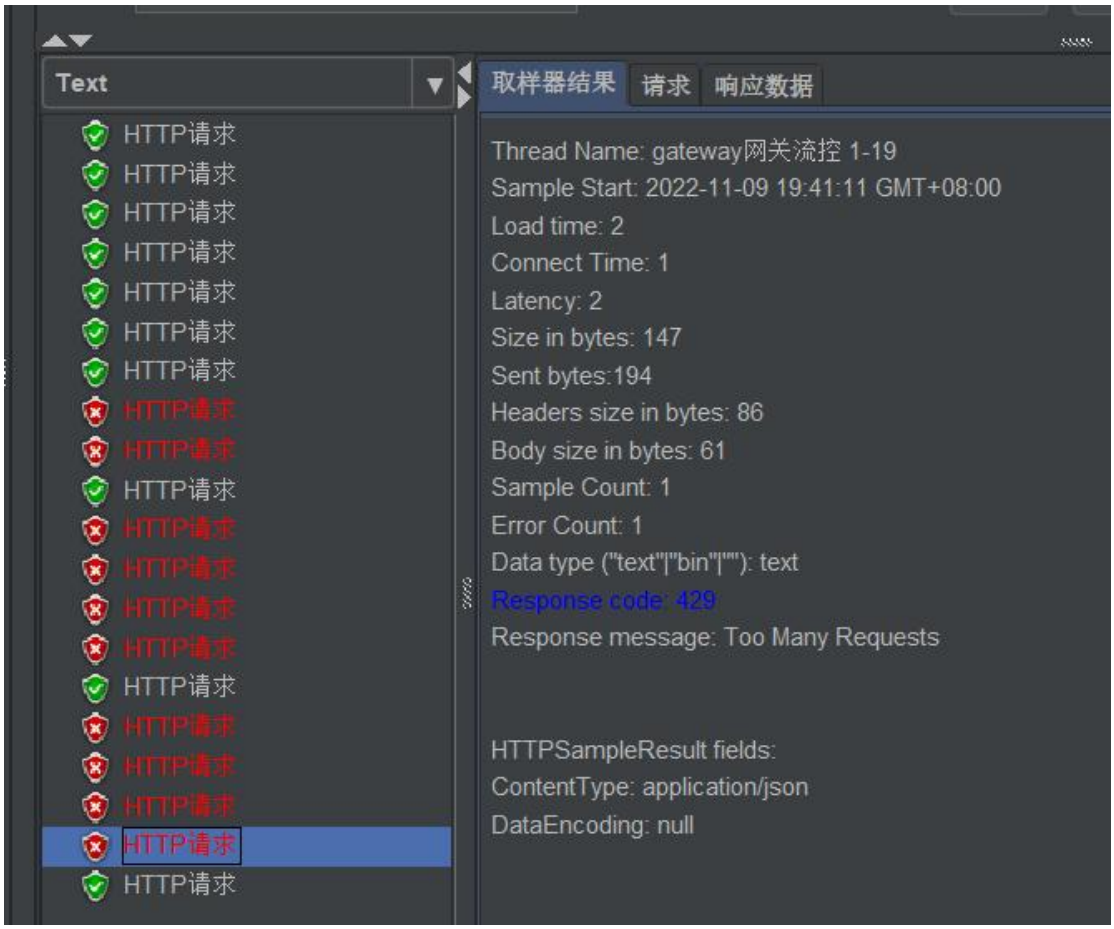


4）查看 sentinel 控制台实时监控效果

拓展：针对请求属性进行流控

jemeter 测试接口增加请求头

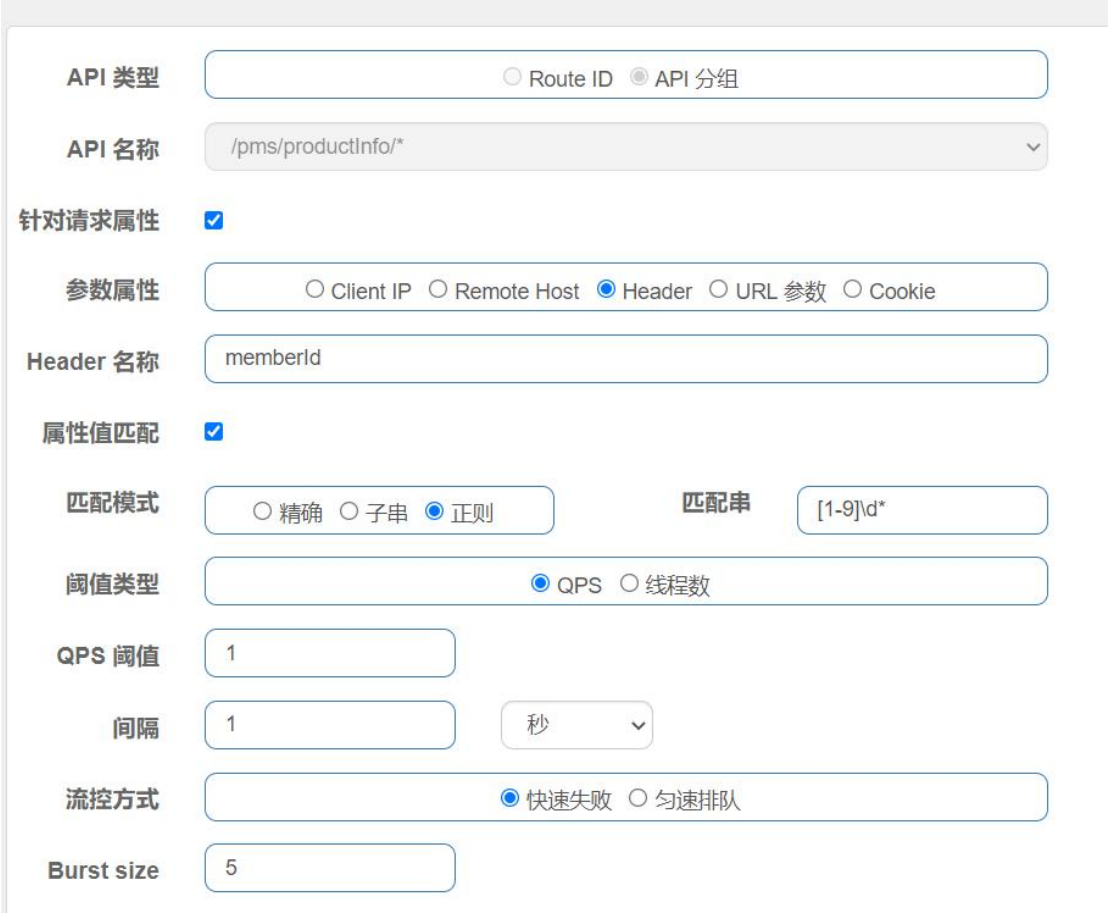
jemeter 测试效果



API 维度限流

1）新增商品详情接口的 API 分组

2） 配置规则



3）测试

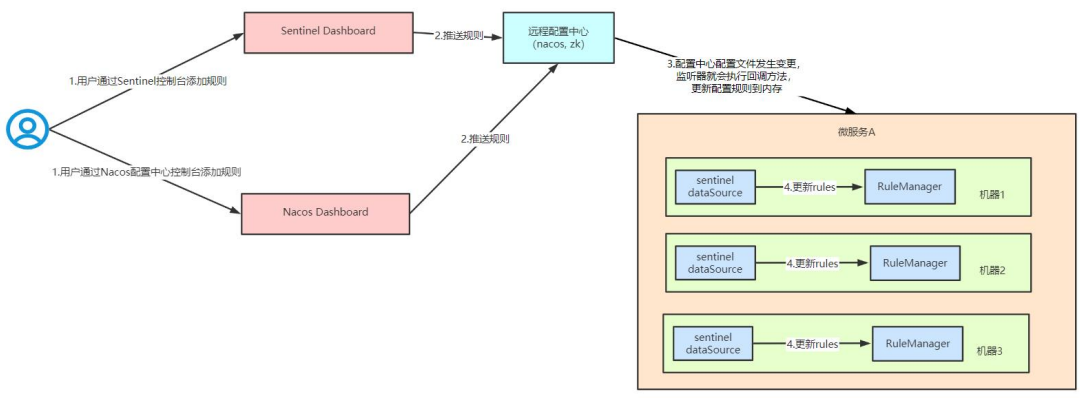
商品详情接口：http://localhost:8888/pms/productInfo/27　　测试结果：被流控

购物车商品详情接口：http://localhost:8888/pms/cartProduct/27　　测试结果：没有被流控

思考：针对商品详情接口的流控需求，Sentinel 网关流控两种模式如何选择？

拓展：生产环境接入 Sentinel 规则持久化配置

规则持久化改造思路



网关服务配置

1）引入依赖

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

2) application.yml 添加 datasource 配置

```
spring:
  application:
    name: tulingmall-gateway
  main:
    allow-bean-definition-overriding: true
  cloud:
    sentinel:
      transport:
        dashboard: 192.168.65.103:8000

      datasource:
        gateway-flow-rules:
          nacos:
            server-addr: 192.168.65.103:8848
            dataId: ${spring.application.name}-gateway-flow-rules
            groupId: SENTINEL_GROUP
            data-type: json
            rule-type: gw-flow
        gateway-api-rules:
          nacos:
            server-addr: 192.168.65.103:8848
            dataId: ${spring.application.name}-gateway-api-rules
            groupId: SENTINEL_GROUP
            data-type: json
            rule-type: gw-api-group
```

3) 启动持久化改造后的 Sentinel 控制台

指定端口和 nacos 配置中心地址

```
java -Dserver.port=8000 -Dsentinel.nacos.config.serverAddr=192.168.65.103:8848 -jar tulingmall-sentinel-dashboard-1.8.4.jar
```

改造后的 Sentinel 控制台：

4）测试规则是否持久化

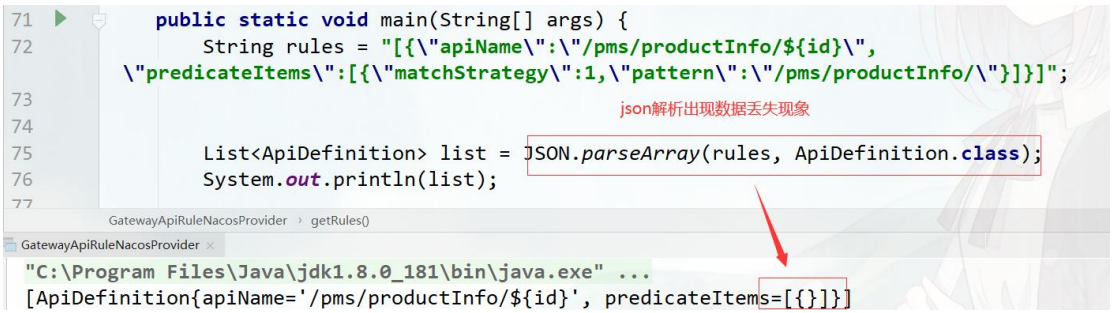
注意：网关规则改造的坑

1. 网关规则实体转换

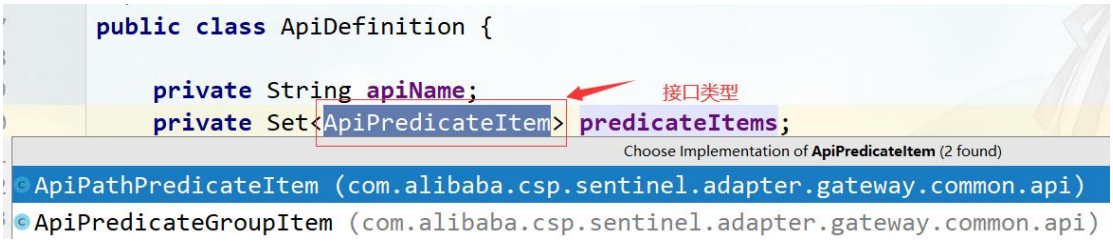
RuleEntity---》Rule　利用 RuleEntity#toRule
#网关规则实体
ApiDefinitionEntity---》ApiDefinition　利用 ApiDefinitionEntity#toApiDefinition
GatewayFlowRuleEntity----->GatewayFlowRule　利　用 GatewayFlowRuleEntity#toGatewayFlowRule

2. json 解析丢失数据

json 解析 ApiDefinition 类型出现数据丢失的现象　天坑



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;
        }

}
```

2.3 应用层限流

微服务接入 sentinel

1) 引入依赖

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

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

2) 接入 sentinel 控制台，修改 application.yml 配置

```
spring:
  cloud:
    sentinel:
      transport:
        dashboard: 192.168.65.103:8000
      datasource:
        flow-rules:
          nacos:
            server-addr: 192.168.65.103:8848
            dataId: ${spring.application.name}-flow-rules
            groupId: SENTINEL_GROUP      # 注意 groupId 对应 Sentinel Dashboard 中的定
义
            data-type: json
            rule-type: flow
        degrade-rules:
          nacos:
            server-addr: 192.168.65.103:8848
            dataId: ${spring.application.name}-degrade-rules
```

```
            groupId: SENTINEL_GROUP
            data-type: json
            rule-type: degrade
        param-flow-rules:
          nacos:
            server-addr: 192.168.65.103:8848
            dataId: ${spring.application.name}-param-flow-rules
            groupId: SENTINEL_GROUP
            data-type: json
            rule-type: param-flow
        authority-rules:
          nacos:
            server-addr: 192.168.65.103:8848
            dataId: ${spring.application.name}-authority-rules
            groupId: SENTINEL_GROUP
            data-type: json
            rule-type: authority
        system-rules:
          nacos:
            server-addr: 192.168.65.103:8848
            dataId: ${spring.application.name}-system-rules
            groupId: SENTINEL_GROUP
            data-type: json
            rule-type: system
```

3) 启动持久化改造后的 Sentinel 控制台

指定端口和 nacos 配置中心地址

```
java -Dserver.port=8000 -Dsentinel.nacos.config.serverAddr=192.168.65.103:8848 -jar tulingmall-sentinel-dashboard-1.8.4.jar
```

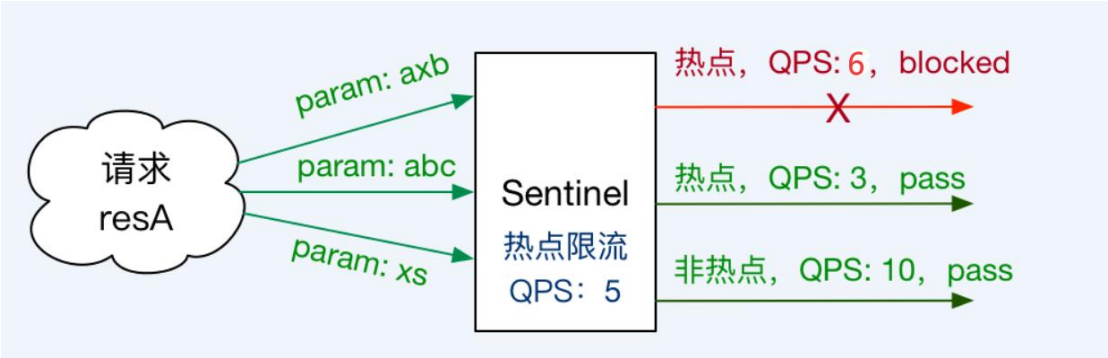
场景：商品详情接口热点参数限流

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：

热点规则需要使用@SentinelResource("resourceName")注解，否则不生效

参数必须是 7 种基本数据类型才会生效

3. 降级实战

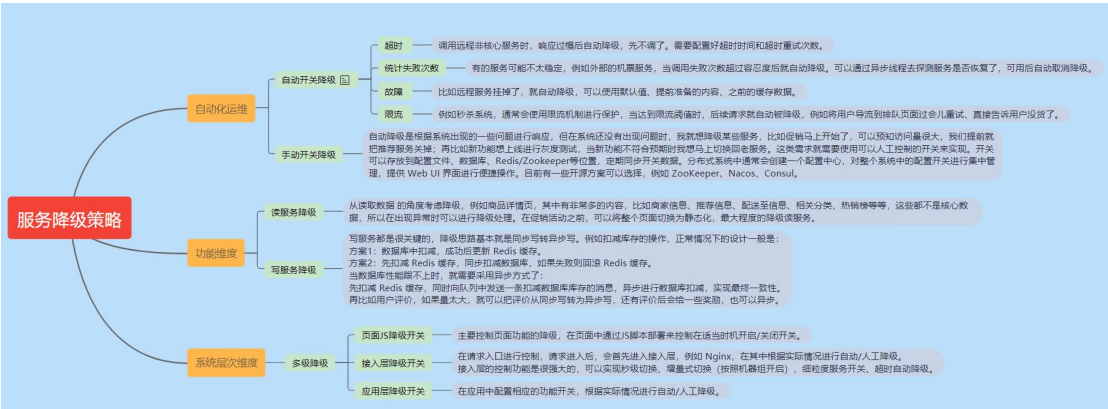
降级就是当系统的容量达到一定程度时，限制或者关闭系统的某些非核心功能，从而把有限的资源保留给更核心的业务。

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。

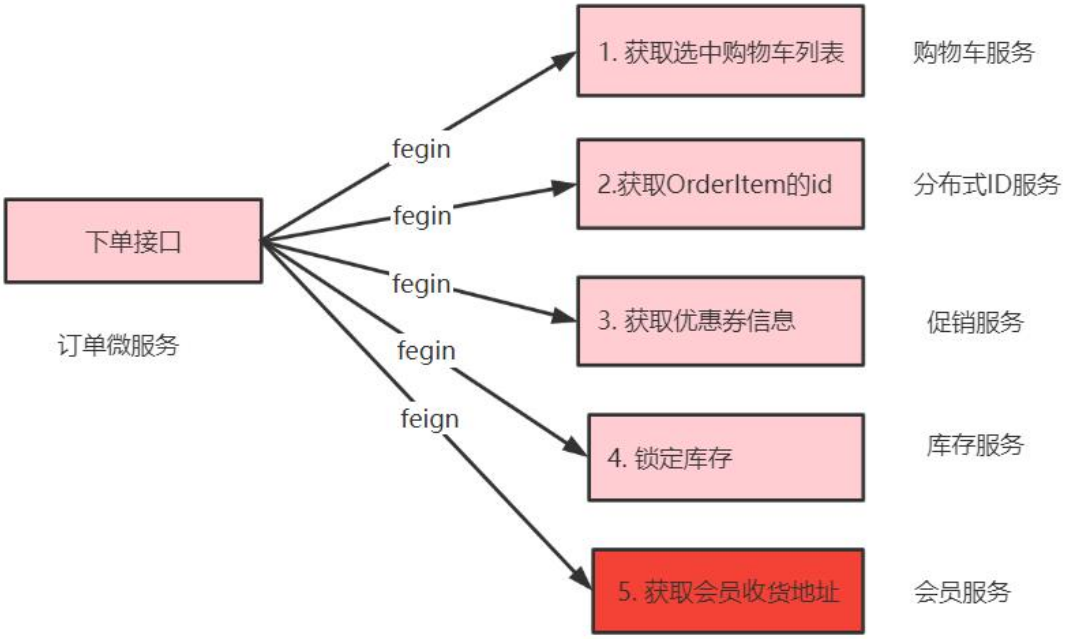
3.1 服务降级的策略

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



3.2 应用层降级实战

场景：用户下单接口



如果会员服务出现问题，会影响整个下单链路。

Sentinel 熔断降级

OpenFeign 整合 Sentinel

1）开启 Sentinel 对 Feign 的支持：

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

2) feign 接口配置 fallbackFactory

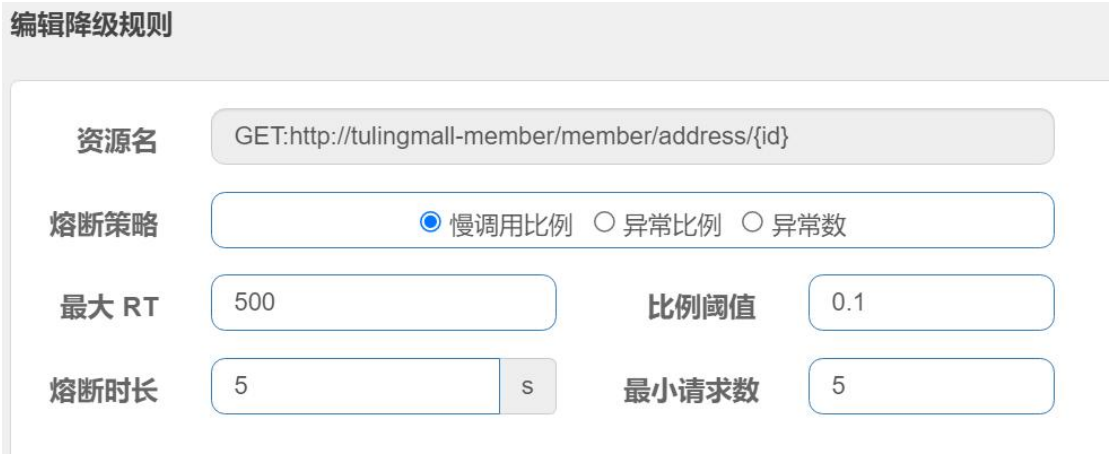
```
@FeignClient(name = "tulingmall-member",path = "/member",
        fallbackFactory = UmsMemberFeginFallbackFactory.class)
public interface UmsMemberFeignApi {
```

3) UmsMemberFeginFallbackFactory 中编写降级逻辑

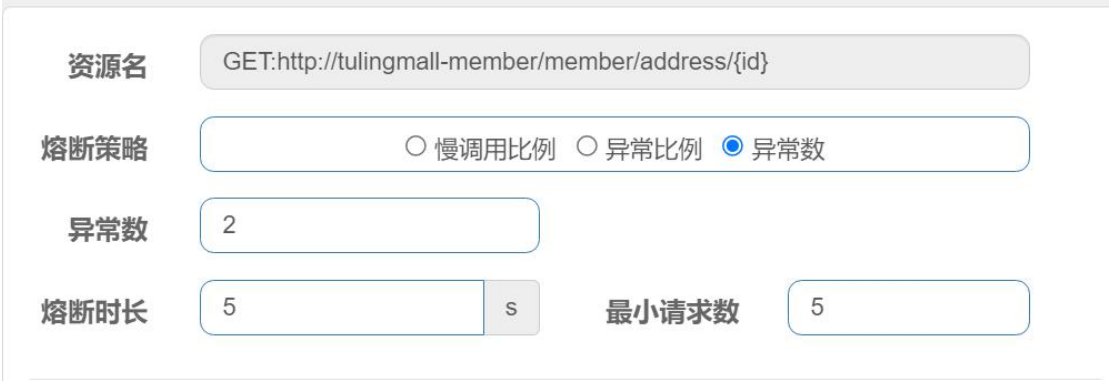
```
@Component
public class UmsMemberFeginFallbackFactory implements
FallbackFactory<UmsMemberFeignApi> {

    @Override
    public UmsMemberFeignApi create(Throwable throwable) {

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

```
UmsMemberReceiveAddress();
                    defaultAddress.setName("默认地址");
                    defaultAddress.setId(-1L);
                    defaultAddress.setDefaultStatus(0);
                    defaultAddress.setPostCode("-1");
                    defaultAddress.setProvince("默认省份");
                    defaultAddress.setCity("默认 city");
                    defaultAddress.setRegion("默认 region");
                    defaultAddress.setDetailAddress("默认详情地址");
                    defaultAddress.setMemberId(-1L);
                    defaultAddress.setPhoneNumber("199xxxxxx");
                    return CommonResult.success(defaultAddress);
                }
            };
        }


}
```

配置基于响应时间的降级规则



配置基于异常数的降级规则



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

```
system-rules:
  nacos:
    server-addr: 192.168.65.103:8848
    dataId: ${spring.application.name}-system-rules
    groupId: SENTINEL_GROUP
    data-type: json
    rule-type: system
```


阈值类型
LOAD
RT
线程数
入口 QPS
CPU 使用率
阈值
1