链滴

sentinel

作者：wenyl

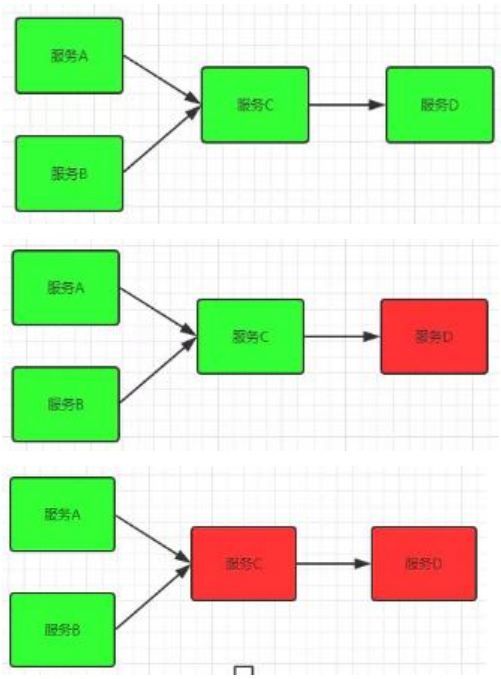
原文链接：https://ld246.com/article/1636016765028

来源网站：链滴

1、基本概念

1.1、服务雪崩

微服务架构中，一个系统被拆分成多个子服务，这些服务之间存在相互调用关系，某个服务不可用之，导致调用它的服务不可用，最终导致整个服务链崩溃，就出现服务雪崩。



预防服务雪崩有以下几个方案：

- 服务熔断：某个服务崩溃后，暂停对该服务的调用；

● 服务降级：对不是很关键的服务来说，当服务负荷过高或响应过慢，则关闭对该服务的访问，返回个定义好的兜底数据，等服务负荷降低在开放对该服务的访问；

● 服务限流：不是核心的服务可以使用服务降级来处理，但是核心的服务一定要保证能够正常访问，此我们对服务进行限流处理，限制他的并发和请求量，保证服务能够正常运行。

2、Sentinel简介

sentinel是一个轻量级的流量控制、熔断降级Java 组件，是分布式系统的流量防卫兵；

Sentinel 以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。

更多信息可以参考sentinel的github地址

3、下载安装

在sentinel版本列表上下载自己需要的版本的jar包，然后java -jar启动成功就可以访问了，默认端口是080，用户名密码都是sentinel

4、客户端

我使用了spring cloud组件，在pom中引入

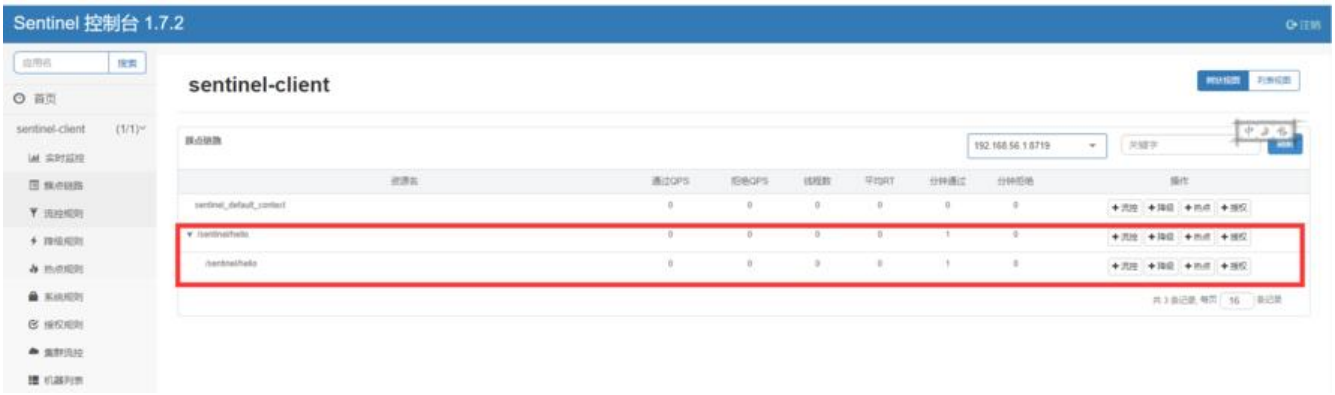
```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
        <version>0.9.0.RELEASE</version>
    </dependency>
```

然后在配置文件中配置对应选项（nacos选项是因为我引入了nacos作为注册中心），重点关注sentine部分，dashboard指定了sentinel安装地址，eager表示spring容器加载就启动心跳检测去链接sentine，默认是false，需要我们请求服务后才会对sentinel发起心跳检测，简单来说，当我们配置eager为fal e，项目启动后去查看sentinel控制台，是看不到服务的，需要对服务进行访问后才能看到，而如果配为true那么项目启动后去访问sentinel就直接可以看到这个服务。

```
spring:
  cloud:
    nacos:
      discovery:
```

```
      server-addr: 10.116.8.56:8848
    sentinel:
      transport:
        dashboard: 10.116.8.56:8080
      eager: true
  application:
    name: sentinel-client
server:
  port: 8079
```

启动后我们访问一个一个系统接口，再去sentinel控制台就可以看到对该接口的监控



通过客户端http://客户端IP:8719/api可以查看服务暴露给sentinel的路径列表

5、配置

5.1、流控规则

流控规则针对资源名进行配置

- 阈值类型
  - QPS（每秒请求数）
  - 线程数
- 单机阈值
  - 单机环境箱超过阈值则对服务限流
- 集群类型
  - 勾选集群类型后界面如下，可以选择均摊和总体阈值两种策略

- 流控规则
  - 链路：只限制链路上指定入口的流量

例如我现在有个service层接口，我将它注册为一个资源，名为sayHello，在controller有两个接口调

了这个资源，分别为/sentinel/hello1和/sentinel/hello2，现在我对sayHello进行链路流量控制，设流控模式为线路，指定入口资源为/sentinal/hello1，则只有hello1这个几口会被限制访问sayHello源，sayHello2则不会被限制

- 直接：表示超过指定阈值直接限制访问

- 关联：表示关联资源超过指定阈值，则限定指定的资源访问，例如，我配置两个资源，当关联源（/sentinel/hello2）访问阈值超过1则限制资源（/sentinel/hello1）的访问

- 流控效果

原文链接：sentinel

- 快速失败

达到阈值直接限制访问，需要将阈值类型设置为QPS

- warm up（预热）

如下，设置阈值为30，预热时间为5，则QPS从10（sentinel有个冷加载因子，默认为3，QPS从设置阈值/冷加载因子开始递增）开始经过5秒逐步达到阈值，需要将阈值类型设置为QPS

- 排队等待

让请求匀速通过，需要将阈值类型设置为QPS

5.2、降级规则

降级规则共有三个配置项，分别为：

- RT：平均响应时间

一秒内进入的请求响应时间超过平均响应阈值时间则会在指定时间窗口内进行服务降级

- 异常比例

一秒内进入的请求，错误的请求数量占比超过指定比例，就会在指定时间窗口内进行服务降级，如下请求异常比例超过0.6，就在5s内限制访问

- 异常数

一分钟内错误请求数超过阈值怎进行降级，如下，一分钟异常请求数超过300则在5s内进行服务降级

5.3、热点规则

对指定资源的参数进行限制，在指定时间内超过规定的访问次数，则限制访问

我们建立一个接口，并注册为sentinelResource

```
@GetMapping("/paramsBlock")
@SentinelResource(value = "paramsBlock",blockHandler = "hello2Block")
public String hello2(@RequestParam("id") String id, @RequestParam("name") String name)

    return helloService.sayHello()+id+"-"+name;
}

public String hello2Block(String id,String name,BlockException blo){
    return "参数限流"+id+"-"+name;
}
```

这个接口有两个参数，我们对第一个进行限制，超过阈值就会触发流量控制

上面规定了某个参数限流，在高级选项中，可以配置例参数的例外，即对某个参数的值进行限制，例，童颜是参数0，我们指定了一秒内最大访问量为10，但是，我们在下面给了他一个例外，就是当这参数值等于1的时候，他的限流阈值不是10而是100

5.4、系统规则

系统规则有五个可选配置项

- **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。

5.5、授权规则

来源访问控制规则，白名单可以访问，黑名单不能访问主要有以下配置项：

- 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appA,appB。
- strategy：限制模式，AUTHORITY_WHITE 为白名单模式，AUTHORITY_BLACK 为黑名单模式，认为白名单模式。

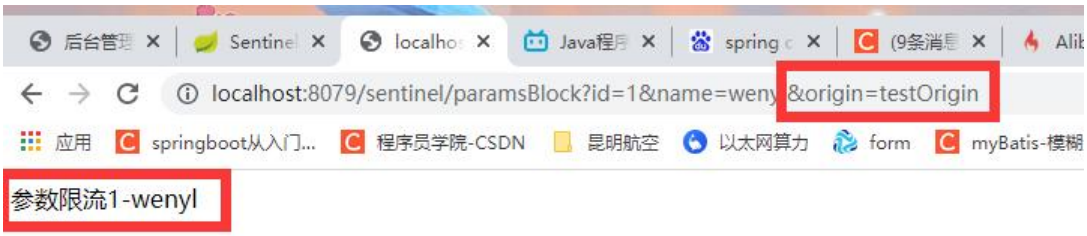
sentinel的启动配置中，默认加载了过滤器，我们需要实现RequestOriginParser定义解析的规则，如：我定义一个解析器，获取request字段中的origin属性，这个属性的值就是白名单黑名单列表

```
/**
 * @author Mr.Wen
 * @version 1.0
 * @date 2021-11-04 16:24
```

```
 */
@Component
public class RequestOriginParserDefinition implements RequestOriginParser {
    @Override
    public String parseOrigin(HttpServletRequest request) {
        return request.getParameter("origin");
    }
}
```

建立好了之后，去sentinel建立规则，只允许testOrigin访问paramsBlock资源

然后访问路径就会被限流了



6、配置持久化

sentinel配置在服务重启后就丢失了，因此肯定是需要持久化的，不然每次都重新配置得累死

这个官方给出来的三种配置模式：

推送模式	说明	优点	缺点
原始模式	API 将规则推送至客户端并直接更新到内存中，扩展写数据源（`WritableDataSource`）	简单，无任何依赖	不保证一致性；规则保存在内存中，重启即消失。严重不建议用于生产环境
Pull 模式	扩展写数据源（`WritableDataSource`），客户端主动向某个规则管理中心定期轮询拉取规则，这个规则中心可以是 RDBMS、文件 等	简单，无任何依赖；规则持久化	不保证一致性；实时性不保证，拉取过于频繁也可能会有性能问题。
Push 模式	扩展读数据源（`ReadableDataSource`），规则中心统一推送，客户端通过注册监听器的方式时刻监听变化，比如使用 Nacos、Zookeeper 等配置中心。这种方式有更好的实时性和一致性保证。**生产环境下一般采用 push 模式的数据源。**	规则持久化；一致性；快速	引入第三方依赖

这里我们以ncaos为例，进行配置：

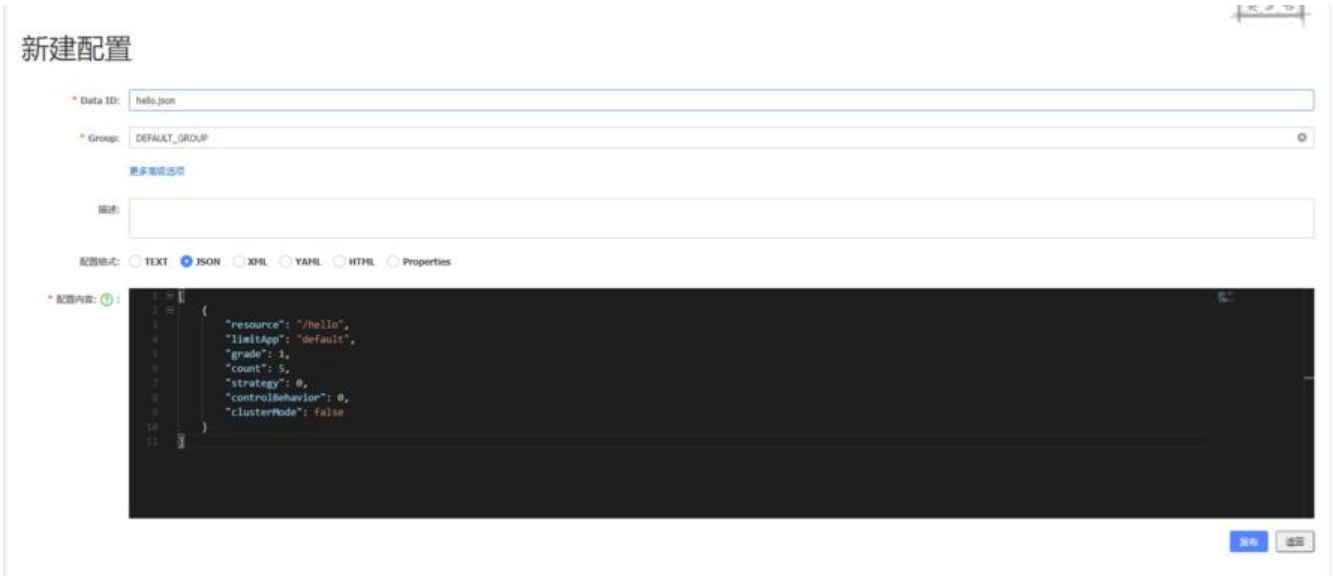
下载对应版本的maven依赖

```
<dependency>
        <groupId>com.alibaba.csp</groupId>
        <artifactId>sentinel-datasource-nacos</artifactId>
        <version>1.7.2</version>
    </dependency>
```

配置文件中dashboard是sentinel服务端IP端口，client-ip和port是客户端地址（也就是服务地址）

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: 10.116.8.56:8848
    sentinel:
      transport:
        dashboard: 10.116.8.56:8080
        client-ip: 10.116.10.99
        port: 8719
      eager: true
      datasource:
        ds1:
          nacos:
            server-addr: 10.116.8.56:8848
            data-id: sentinelHello
            group-id: DEFAULT_GROUP
            data-type: json
            rule-type: flow
```

nacos新增配置



启动项目可以看到配置出现在sentinel中

原文链接：sentinel