主讲老师：Fox

> 有道笔记：https://note.youdao.com/s/Rr6CtNIp

# 1. 服务雪崩及其解决方案

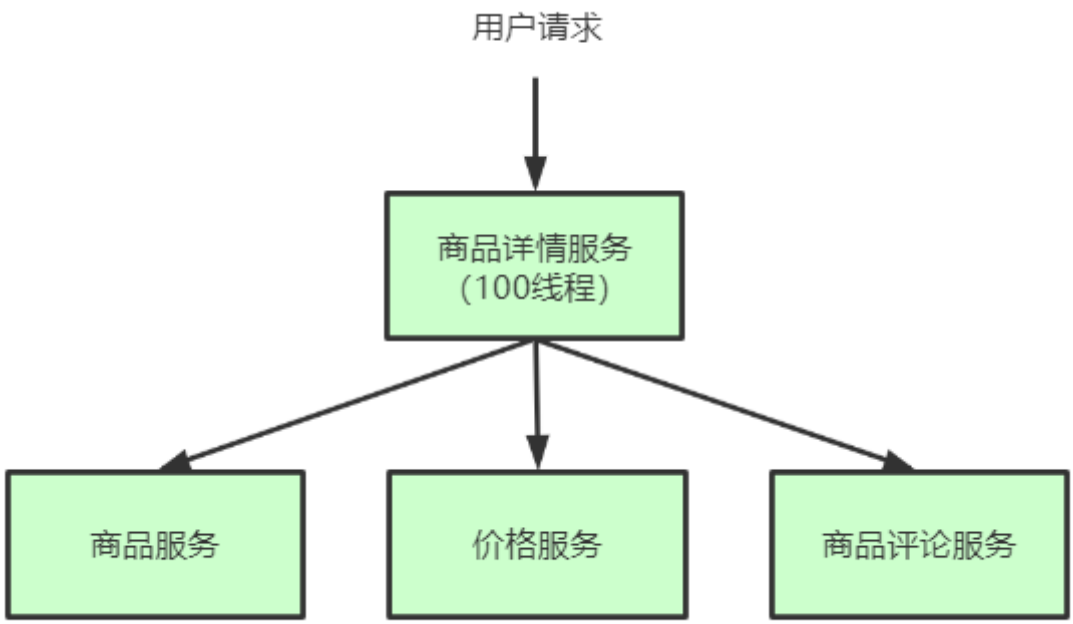
1、当服务访问量达到一定程度，流量扛不住的时候，该如何处理?

2、服务之间相互依赖，当服务A出现响应时间过长，影响到服务B的响应，进而产生连锁反应，直至影响整个依赖链上的所有服务，该如何处理?

如何保证微服务运行期间的稳定性，这是分布式、微服务开发不可避免的问题。

## 1.1 什么是服务雪崩

在一个高度服务化的系统中,我们实现的一个业务逻辑通常会依赖多个服务,比如:商品详情展示服务会依赖商品服务, 价格服务, 商品评论服务. 如图所示:




调用三个依赖服务会共享商品详情服务的线程池. 如果其中的商品评论服务不可用, 就会出现线程池里所有线程都因等待响应而被阻塞, 从而造成服务雪崩. 如图所示:

在微服务调用链路中，因服务提供者的不可用导致服务调用者的不可用,并将不可用逐渐放大的过程，就叫服务雪崩效应。

**导致服务不可用的原因:**

- **程序有Bug**：代码循环调用的逻辑问题，资源未释放引起的内存泄漏等问题；
- **大流量请求**：在秒杀和大促开始前,如果准备不充分,瞬间大量请求会造成服务提供者的不可用；
- **硬件故障**：可能为硬件损坏造成的服务器主机宕机, 网络硬件故障造成的服务提供者的不可访问；

- **缓存击穿**：一般发生在缓存应用重启, 缓存失效时高并发，所有缓存被清空时,以及短时间内大量缓存失效时。大量的缓存不命中, 使请求直击后端,造成服务提供者超负荷运行,引起服务不可用。

在服务提供者不可用的时候，会出现大量重试的情况：用户重试、代码逻辑重试，这些重试最终导致：进一步加大请求流量。所以归根结底导致雪崩效应的最根本原因是：大量请求线程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。

## 1.2 解决方案

解决雪崩问题的常见方式有四种：

### 超时机制

设定超时时间，请求超过一定时间没有响应就返回错误信息，不会无休止地等待。

在不做任何处理的情况下，服务提供者不可用会导致消费者请求线程强制等待，而造成系统资源耗尽。加入超时机制，一旦超时，就释放资源。由于释放资源速度较快，一定程度上可以抑制资源耗尽的问题。但是这种解决方案只是缓解了雪崩问题，并不能解决雪崩问题。

### 服务限流

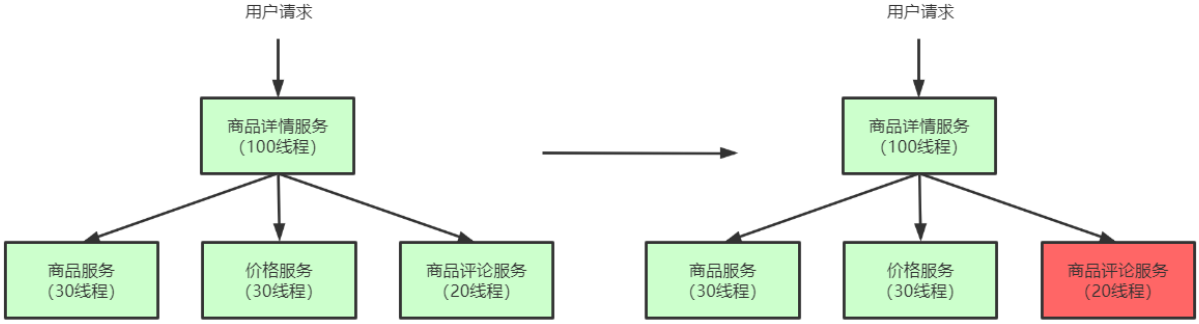
限制业务访问的QPS，避免服务因为流量的突增而故障。这种是从预防层面来解决雪崩问题。

### 舱壁隔离(资源隔离)

资源隔离分为进程隔离，线程隔离和信号量隔离。隔离机制的本质就是将服务调用的粒度划分的更小，以此来减少服务生产崩溃而对服务调用带来的影响，避免服务雪崩现象产生。

比如限定每个业务能使用的线程数，避免耗尽整个线程池的资源。它是通过划分线程池的线程，让每个业务最多使用n个线程，进而提高容灾能力。但是这种模式会导致一定的资源浪费，比如，服务C已经宕机，但每次还是会尝试让服务A去向服务C发送请求。

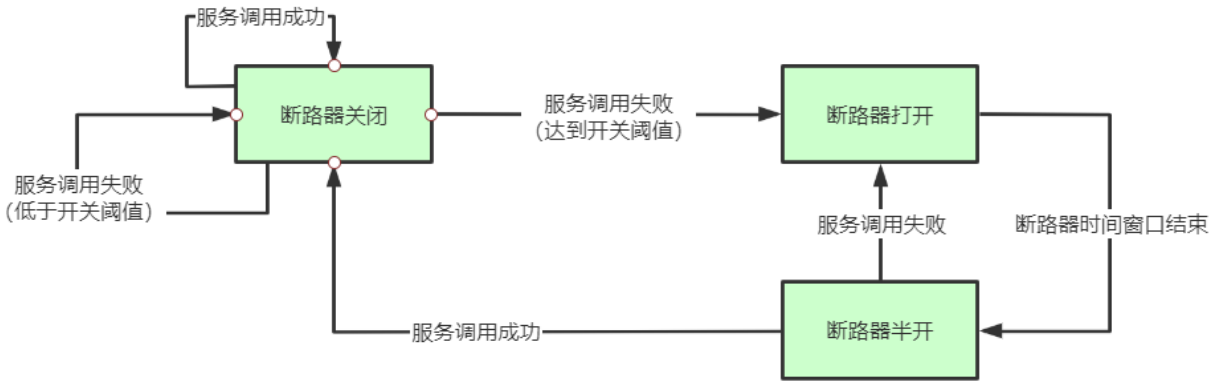
如下图所示, 当商品评论服务不可用时, 即使商品服务独立分配的20个线程全部处于同步等待状态,也不会影响其他依赖服务的调用。



**服务熔断降级**

这种模式主要是参考电路熔断，如果一条线路电压过高，保险丝会熔断，防止火灾。放到我们的系统中，如果某个目标服务调用慢或者有大量超时，此时，熔断该服务的调用，对于后续调用请求，不在继续调用目标服务，直接返回，快速释放资源。如果目标服务情况好转则恢复调用。

当依赖的服务有大量超时时，在让新的请求去访问根本没有意义，只会无畏的消耗现有资源。比如我们设置了超时时间为3s,如果短时间内有大量请求在3s内都得不到响应，就意味着这个服务出现了异常，此时就没有必要再让其他的请求去访问这个依赖了，这个时候就应该使用断路器避免资源浪费。




有服务熔断，必然要有服务降级。

所谓降级，就是当某个服务熔断之后，服务将不再被调用，此时客户端可以自己准备一个本地的fallback（回退）回调，返回一个缺省值。 例如：(备用接口/缓存/mock数据) 。这样做，虽然服务水平下降，但好歹可用，比直接挂掉要强，当然这也要看适合的业务场景。

## 1.3 技术选型: Sentinel or Hystrix

Hystrix 的关注点在于以 隔离 和 熔断 为主的容错机制，超时或被熔断的调用将会快速失败，并可以提供 fallback 机制。

而 Sentinel 的侧重4点在于：

- 多样化的流量控制
- 熔断降级
- 系统负载保护
- 实时监控和控制台

| | Sentinel | Hystrix |
| --- | --- | --- |
| 隔离策略 | 信号量隔离 | 线程池隔离/信号量隔离 |
| 熔断降级策略 | 基于响应时间或失败比率 | 基于失败比率 |
| 实时指标实现 | 滑动窗口 | 滑动窗口（基于 RxJava） |
| 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 | 支持慢启动、匀速器模式 | 不支持 |
| 系统负载保护 | 支持 | 不支持 |
| 控制台 | 开箱即用，可配置规则、查看秒级监控、机器发现等 | 不完善 |
| 常见框架的适配 | Servlet、Spring Cloud、Dubbo、gRPC 等 | Servlet、Spring Cloud Netflix |

# 2. 流量治理组件Sentinel实战

## 2.1 Sentinel 是什么

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。Sentinel 是面向分布式、多语言异构化服务架构的流量治理组件，主要以流量为切入点，从流量路由、流量控制、流量整形、熔断降级、系统自适应过载保护、热点流量防护等多个维度来帮助开发者保障微服务的稳定性。

> 官方文档： https://sentinelguard.io/zh-cn/docs/introduction.html

Sentinel具有以下特征:

- **丰富的应用场景**： Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息削峰填谷、实时熔断下游不可用应用等。
- **完备的实时监控**： Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器秒级数据，甚至 500 台以下规模的集群的汇总运行情况。
- **广泛的开源生态**： Sentinel 提供开箱即用的与其它开源框架/库的整合模块，例如与 Spring Cloud、Dubbo、gRPC 的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入 Sentinel。
- **完善的 SPI 扩展点**： Sentinel 提供简单易用、完善的 SPI 扩展点。您可以通过实现扩展点，快速的定制逻辑。例如定制规则管理、适配数据源等。

## 2.2 Sentinel核心概念

**资源（Resource）**

资源是 Sentinel 的关键概念。它可以是 Java 应用程序中的任何内容，例如，由应用程序提供的服务，或由应用程序调用的其它应用提供的服务，甚至可以是一段代码。在接下来的文档中，我们都会用资源来描述代码块。

**只要通过 Sentinel API 定义的代码，就是资源，能够被 Sentinel 保护起来。**大部分情况下，可以使用方法签名，URL，甚至服务名称作为资源名来标示资源。

**规则(Rule)**

围绕资源的实时状态设定的规则，可以包括流量控制规则、熔断降级规则以及系统保护规则。所有规则可以动态实时调整。

## 2.3 Sentinel快速开始

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 8 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（见 主流框架适配）
- 控制台（Dashboard）：Dashboard 主要负责管理推送规则、监控、管理机器信息等。

### 基于API实现资源保护

在官方文档中，定义的Sentinel进行资源保护的几个步骤：

- 定义资源
- 定义规则
- 检验规则是否生效

#### 1）引入依赖

```
<dependency>
      <groupId>com.alibaba.csp</groupId>
      <artifactId>sentinel-core</artifactId>
      <version>1.8.6</version>
</dependency>
```

#### 2）定义受保护的资源和流控规则

```
@RestController
@Slf4j
public class HelloController {

```

```
    private static final String RESOURCE_NAME = "HelloWorld";

    @RequestMapping(value = "/hello")
    public String hello() {
        try (Entry entry = SphU.entry(RESOURCE_NAME)) {
            // 被保护的逻辑
            log.info("hello world");
            return "hello world";
        } catch (BlockException ex) {
            // 处理被流控的逻辑
            log.info("blocked!");
            return "被流控了";
        }

    }
    /**
     * 定义流控规则
     */
    @PostConstruct
    private static void initFlowRules(){
        List<FlowRule> rules = new ArrayList<>();
        FlowRule rule = new FlowRule();
        //设置受保护的资源
        rule.setResource(RESOURCE_NAME);
        // 设置流控规则 QPS
        rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
        // 设置受保护的资源阈值
        // Set limit QPS to 20.
        rule.setCount(1);
        rules.add(rule);
        // 加载配置好的规则
        FlowRuleManager.loadRules(rules);
    }
}
```

**3）测试**

测试效果：http://localhost:8800/hello

**缺点：**

- 业务侵入性很强，需要在controller中写入非业务代码.
- 配置不灵活 若需要添加新的受保护资源 需要手动添加 init方法来添加流控规则

## 基于@SentinelResource注解埋点实现资源保护

Sentinel 提供了 @SentinelResource 注解用于定义资源，并提供了 AspectJ 的扩展用于自动定义资源、处理 BlockException 等。

> 注意：注解方式埋点不支持 private 方法。

@SentinelResource 注解包含以下属性：

- value： 资源名称，必需项（不能为空）
- blockHandler: 定义当资源内部发生了BlockException应该进入的方法（捕获的是Sentinel定义的BlockEcxeption异常）。如果同时配置了blockHandler和fallback，出现BlockException时将进入BlockHandler方法中处理
- fallback: 定义的是资源内部发生了Throwable应该进入的方法。默认处理所有的异常，如果我们不配置blockHandler，其抛出BlockEcxeption也将会进入fallback方法中
- exceptionsToIgnore：配置fallback可以忽略的异常

源码入口：com.alibaba.csp.sentinel.annotation.aspectj.SentinelResourceAspect

### 1）引入依赖

```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-annotation-aspectj</artifactId>
    <version>1.8.6</version>
</dependency>
```

### 2）配置切面支持

```
@Configuration
public class SentinelAspectConfiguration {

    @Bean
    public SentinelResourceAspect sentinelResourceAspect() {
        return new SentinelResourceAspect();
    }
}
```

3）HelloController中编写测试逻辑，添加@SentinelResource，并配置blockHandler和fallback

```
@SentinelResource(value = "hello world",blockHandler = "handleException",
    fallback = "fallbackException")
@RequestMapping("/hello2")
public String hello2() {
    int i = 1 / 0;
    return "helloworld";
}

// Block 异常处理函数，参数最后多一个 BlockException，其余与原方法hello2一致.
public String handleException(BlockException ex){
    return "被流控了";
}

// Fallback 异常处理函数，参数与原方法hello2一致或加一个 Throwable 类型的参数.
public String fallbackException(Throwable t){
    return "被异常降级了";
}

```

**使用注意事项：**

- 方法必须和被保护资源处于同一个类
- 方法参数列表和受保护资源一致（blockHandler最后增加一个BlockException，fallback可选增加Throwable）
- 方法返回值必须和受保护资源相同

单独新建一个类用于处理受保护资源的BlockException和fallback，更加的解耦且灵活。可以通过配置blockHandlerClass和fallbackClass实现

```
@SentinelResource(value = RESOURCE_NAME,
        blockHandler = "handleException",blockHandlerClass = ExceptionUtil.class,
        fallback = "fallbackException",fallbackClass = ExceptionUtil.class)
@RequestMapping("/hello2")
public String hello2() {
```

```
6
7       int i = 1 / 0;
8
9       return "helloworld ";
10  }
11
12  // 编写ExceptionUtil，注意如果指定了class，方法必须是static方法
13  public class ExceptionUtil {
14
15      public static String fallbackException(Throwable t){
16          return "===被异常降级啦===";
17      }
18
19      public static String handleException(BlockException ex){
20          return "===被限流啦===";
21      }
22  }
23
24
```

使用注意事项：

- 指定处理方法必须是public static的
- 方法参数列表和受保护资源一致（blockHandler最后增加一个BlockException，fallback可选增加Throwable）
- 方法返回值必须和受保护资源相同

#### 4) 测试

测试效果：http://localhost:8800/hello2

## 2.4 Java应用接入Sentinel控制台配置流控规则

Sentinel 提供一个轻量级的开源控制台，它提供机器发现以及健康情况管理、监控（单机和集群），规则管理和推送的功能。

> 注意：Sentinel 控制台目前仅支持单机部署。

### 启动Sentinel控制台

#### 1）下载sentinel控制台jar包

```
https://github.com/alibaba/Sentinel/releases/download/1.8.6/sentinel-dashboard-1.8.6.jar
```

### 2）启动sentinel控制台

使用如下命令启动控制台：

```

java -Dserver.port=8080 -Dcsp.sentinel.dashboard.server=localhost:8080 -Dproject.name=sentinel-dashboard -jar sentinel-dashboard-1.8.6.jar
```

如若8080端口冲突，可使用 -Dserver.port=新端口 进行设置。


3）访问http://localhost:8080/#/login ,默认用户名密码： sentinel/sentinel

## Java应用接入Sentinel控制台

### 1）引入依赖

Java应用需要引入 Transport 模块来与 Sentinel 控制台进行通信。通过 pom.xml 引入 JAR 包:

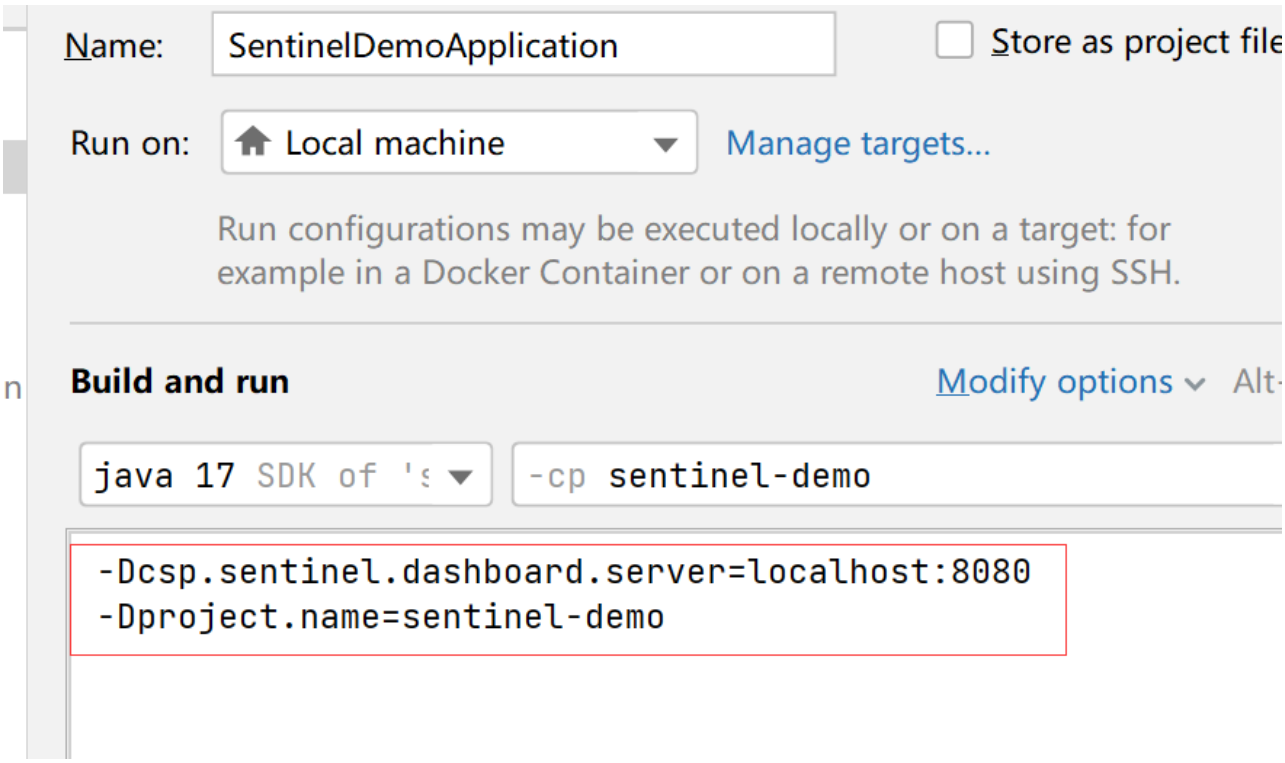
```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-transport-simple-http</artifactId>
    <version>1.8.6</version>
</dependency>
```

### 2）Java应用添加相应的 JVM 参数

- 应用名称：-Dproject.name=xxxx

- 控制台地址：-Dcsp.sentinel.dashboard.server=ip:port
- 应用启动 HTTP API Server 的端口号：-Dcsp.sentinel.api.port=xxxx （默认是 8719），端口冲突会依次递增

> 更多的参数参见： 启动参数文档




3）运行Java应用，访问测试接口：http://localhost:8800/hello

当首次访问对应的资源后 等待一段时间即可在控制台上看到对应的应用以及相应的监控信息。可以通过 curl http://ip:port/tree 命令查看调用链，正常情况下会显示出已访问资源的调用链。

注意: Sentinel 会在客户端首次调用时候进行初始化，开始向控制台发送心跳包。因此需要确保客户端有访问量，才能在控制台上看到监控数据。

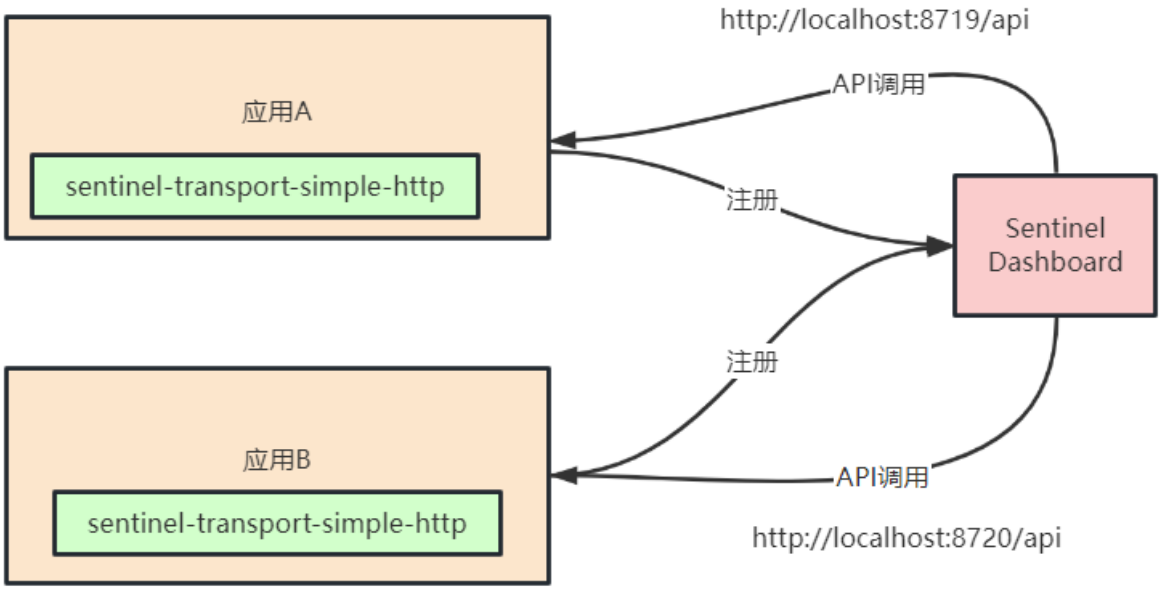
**sentinel日志查看路径：**

- 控制台推送规则的日志默认位于 ${user.home}/logs/csp/sentinel-dashboard.log
- 客户端接收规则日志默认位于 ${user.home}/logs/csp/sentinel-record.log.xxx

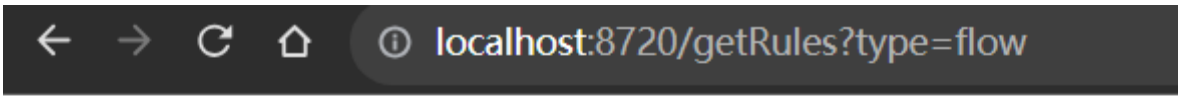
**4）测试：通过sentinel控制台配置流控规则是否生效**

## Java应用和Sentinel Dashboard通信原理

Sentinel控制台与Java应用之间，实现了一套服务发现机制，集成了Sentinel的应用都会将元数据传递给Sentinel控制台，架构图如下所示：




测试：http://localhost:8720/getRules?type=flow 获取流控规则信息





```
[
  - {
        clusterMode: false,
        controlBehavior: 0,
        count: 1,
        grade: 1,
        limitApp: "default",
        maxQueueingTimeMs: 500,
        resource: "HelloWorld",
        strategy: 0,
        warmUpPeriodSec: 10,
    }
]
```


## 2.5 微服务整合Sentinel实战

### 快速开始

### 1）引入依赖

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

### 2）业务代码中配置需要保护的资源

当 SpringBoot 应用接入 Sentinel starter 后，可以针对某个 URL 进行流控。所有的 URL 就自动成为 Sentinel 中的埋点资源，可以针对某个 URL 进行流控。或者使用@SentinelResource 注解用来标识资源是否被限流、降级。

```
// mvc接口方法自动埋点
@RequestMapping("/info/{id}")
public R info(@PathVariable("id") Integer id){
    UserEntity user = userService.getById(id);
    return R.ok().put("user", user);
}

// 使用@SentinelResource 注解用来标识资源是否被限流、降级
@SentinelResource(value = "getUser",blockHandler = "handleException")
public UserEntity getById(Integer id) {
    return userDao.getById(id);
}

public UserEntity handleException(Integer id, BlockException ex) {
    UserEntity userEntity = new UserEntity();
    userEntity.setUsername("===被限流降级啦===");
    return userEntity;
}
```

### 3）添加yml配置，为微服务设置sentinel控制台地址

```
server:
  port: 8800

spring:
```

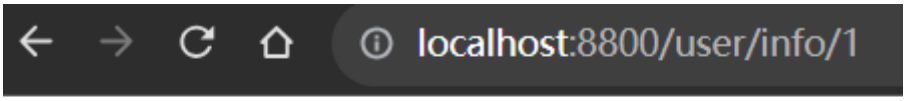
```yaml
  application:
    name: mall-user-sentinel-demo
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848

    sentinel:
      transport:
        # 添加sentinel的控制台地址
        dashboard: 127.0.0.1:8080
        # 指定应用与Sentinel控制台交互的端口，应用会起一个HttpServer占用该端口
        # port: 8719

```

- spring.cloud.sentinel.transport.port 端口配置会在应用对应的机器上启动一个 Http Server，该 Server 会与 Sentinel 控制台做交互。比如 Sentinel 控制台添加了一个限流规则，会把规则数据 push 给这个 Http Server 接收，Http Server 再将规则注册到 Sentinel 中。

### 4）启动sentinel控制台，在sentinel控制台中设置流控规则

- **资源名**: 接口的API
- **针对来源**: 默认是default，当多个微服务都调用这个资源时，可以配置微服务名来对指定的微服务设置阈值
- **阈值类型**: 分为QPS和线程数 假设阈值为10
- **QPS类型**: 只得是每秒访问接口的次数>10就进行限流
- **线程数**: 为接受请求该资源分配的线程数>10就进行限流


测试： http://localhost:8800/user/info/1 因为QPS是1，所以1秒内多次访问会出现如下情形：




Blocked by Sentinel (flow limiting)

## 通过Sentinel 对外暴露的 Endpoint查看相关的配置

Sentinel Endpoint 里暴露的信息包括当前应用的所有规则信息、日志目录、当前实例的 IP，Sentinel Dashboard 地址应用与 Sentinel Dashboard 的心跳频率等等信息。

**1）引入依赖**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

**2）暴露监控端点**

添加Sentinel后，需要暴露/actuator/sentinel端点

```
#暴露actuator端点
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

3）访问http://localhost:8800/actuator/sentinel， 可以查看流控规则信息

```
- rules: {
    systemRules: [ ],
    authorityRule: [ ],
    paramFlowRule: [ ],
  - flowRules: [
    - {
        id: null,
        resource: "/user/info/{id}",
        limitApp: "default",
        grade: 1,
        count: 1,                    流控规则信息
        strategy: 0,
        refResource: null,
        controlBehavior: 0,
        warmUpPeriodSec: 10,
        maxQueueingTimeMs: 500,
        clusterMode: false,
      - clusterConfig: {
            flowId: null,
            thresholdType: 0,
            fallbackToLocalWhenFail: true,
            strategy: 0,
            sampleCount: 10,
            windowIntervalMs: 1000,
            resourceTimeout: 2000,
            resourceTimeoutStrategy: 0,
            acquireRefuseStrategy: 0,
            clientOfflineTime: 2000,
        },
      }
    ],
    degradeRules: [ ],
```

### RestTemplate整合Sentinel

Spring Cloud Alibaba Sentinel 支持对 RestTemplate 的服务调用使用 Sentinel 进行保护，在构造 RestTemplate bean的时候需要加上 @SentinelRestTemplate 注解。

### 1）RestTemplate添加@SentinelRestTemplate注解

```
@Bean
@LoadBalanced
@SentinelRestTemplate(
        blockHandler = "handleBlockException",blockHandlerClass = ExceptionUtil.class,
        fallback = "handleFallback",fallbackClass = ExceptionUtil.class
)
```

```
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

@SentinelRestTemplate 注解的属性支持限流(blockHandler, blockHandlerClass)和降级(fallback, fallbackClass BlockException中的DegradeException)的处理。

其中 blockHandler 或 fallback 属性对应的方法必须是对应 blockHandlerClass 或 fallbackClass 属性中的静态方法。

该方法的参数跟返回值

跟 org.springframework.http.client.ClientHttpRequestInterceptor#interceptor 方法一致，其中参数多出了一个 BlockException 参数用于获取 Sentinel 捕获的异常。

> 源码跟踪:
> com.alibaba.cloud.sentinel.custom.SentinelBeanPostProcessor
> com.alibaba.cloud.sentinel.custom.SentinelProtectInterceptor#intercept

```
// UserController.java
@RequestMapping(value = "/findOrderByUserId2/{id}")
public R  findOrderByUserId2(@PathVariable("id") Integer id) {

    //利用@LoadBalanced，restTemplate需要添加@LoadBalanced注解
    String url = "http://mall-order/order/findOrderByUserId/"+id;
    R result = restTemplate.getForObject(url,R.class);
    return result;
}


public class ExceptionUtil {
    /**
     * 注意：  static修饰，参数类型不能出错
     * @param request  org.springframework.http.HttpRequest
     * @param body
     * @param execution
     * @param ex
     * @return
     */
    public static SentinelClientHttpResponse handleBlockException(HttpRequest request,
                                                                  byte[] body,
ClientHttpRequestExecution execution, BlockException ex) {
```

```
        R r = R.error(-1, "===被限流啦===");
        try {
            return new SentinelClientHttpResponse(new
ObjectMapper().writeValueAsString(r));
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
        return null;
    }

    public static SentinelClientHttpResponse handleFallback(HttpRequest request,
                                                             byte[] body,
ClientHttpRequestExecution execution, BlockException ex) {
        R r = R.error(-2, "===被异常降级啦===");
        try {
            return new SentinelClientHttpResponse(new
ObjectMapper().writeValueAsString(r));
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

### 2）添加yml配置，开启sentinel对resttemplate的支持，默认开启，可忽略

```

#开启sentinel对resttemplate的支持，false则关闭 ，默认true
resttemplate:
  sentinel:
    enabled: true
```

### 3） 测试

**Sentinel RestTemplate 限流的资源规则提供两种粒度：**

- httpmethod:schema://host:port：协议、主机和端口
- httpmethod:schema://host:port/path：协议、主机、端口和路径


**测试限流：**

在sentinel控制台中对被保护的restTemplate资源进行限流配置

访问 http://localhost:8800/user/findOrderByUserId2/1，测试限流是否生效

**测试降级**

修改服务提供者mall-order，模拟业务异常

```
@RequestMapping("/findOrderByUserId/{userId}")
public R findOrderByUserId(@PathVariable("userId") Integer userId) {

    //模拟异常
    if(userId==5){
        throw new IllegalArgumentException("非法参数异常");
    }

    log.info("根据userId:"+userId+"查询订单信息");
    List<OrderEntity> orderEntities = orderService.listByUserId(userId);
    return R.ok().put("orders", orderEntities);
}
```

在sentinel控制台中对被保护的restTemplate资源进行熔断降级规则配置

访问http://localhost:8800/user/findOrderByUserId2/5，测试降级是否生效

**OpenFeign整合Sentinel**

Sentinel 适配了 Feign 组件。

> 源码跟踪：com.alibaba.cloud.sentinel.feign.SentinelInvocationHandler#invoke

**1）yml配置文件中开启 Sentinel 对 Feign 的支持**

```yaml
feign:
  sentinel:
    enabled: true  #开启Sentinel 对 Feign 的支持
```

### 2）在Feign的声明式接口上添加fallback或者fallbackFactory属性

```
//
@FeignClient(value = "mall-order",path = "/order",fallback = FallbackOrderFeignService.class)
public interface OrderFeignService {

    @RequestMapping("/findOrderByUserId/{userId}")
    public R findOrderByUserId(@PathVariable("userId") Integer userId);

}

@Component   //必须交给spring 管理
public class FallbackOrderFeignService implements OrderFeignService {
    @Override
    public R findOrderByUserId(Integer userId) {
        return R.error(-1,"=======服务降级了========");
    }
}

@FeignClient(value = "mall-order",path = "/order",fallbackFactory = FallbackOrderFeignServiceFactory.class)
public interface OrderFeignService {

    @RequestMapping("/findOrderByUserId/{userId}")
    public R findOrderByUserId(@PathVariable("userId") Integer userId);
}

@Component
public class FallbackOrderFeignServiceFactory implements FallbackFactory<OrderFeignService> {
    @Override
    public OrderFeignService create(Throwable throwable) {

```

```
30          return new OrderFeignService() {
31              @Override
32              public R findOrderByUserId(Integer userId) {
33                  return R.error(-1,"=======服务降级了========");
34              }
35          };
36      }
37  }
```

**3）测试**

访问：http://localhost:8800/user/findOrderByUserId/1，生成如下sentinel openFeign的资源链路

**测试限流**

在sentinel控制台配置流控规则

访问：http://localhost:8800/user/findOrderByUserId/1，测试流控规则是否生效

**测试降级**

访问http://localhost:8800/user/findOrderByUserId/5 (userId为5是mall-order服务会抛出异常)，测试降级是否生效

关闭mall-order服务，访问http://localhost:8800/user/findOrderByUserId/1，测试降级是否生效