ch04-服务容错

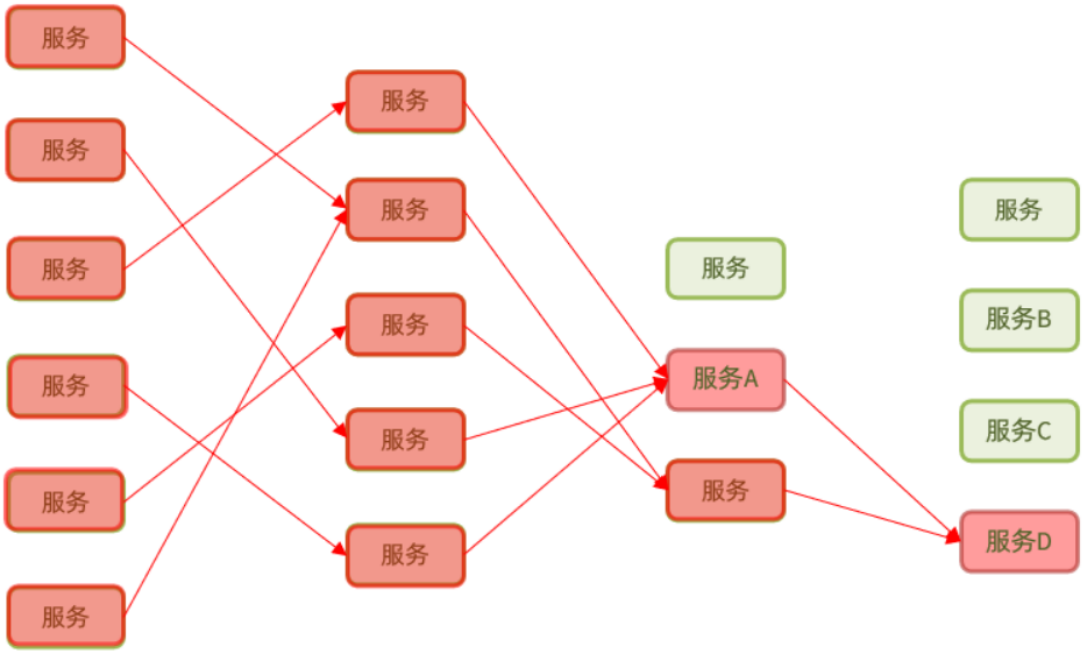
公众号：锋哥聊编程

主讲：小锋

1、什么是服务雪崩?

服务异常可能导致服务雪崩

在分布式服务的系统内，很多的用户请求在系统内部都是存在级联式远程调用的。如下图所示：一次请求先后经过Service A、B、C、D，如果此时服务D发生异常，长时间无法响应或者根本不响应，将导致Service C服务调用无法正常响应，进而导致Service B和Service A的响应也出现问题。这种因为服务调用链中某一个服务不可达或超时等异常情况，导致其上游的服务也出现响应异常或者崩溃。当这种情况在高并发环境下就会导致整个系统响应超时、资源等待耗尽，这种现象就是“**服务雪崩**”。



服务重试可能导致服务雪崩

- 服务请求重试机制在很大程度上解决了由于网络瞬时不可达的问题，导致服务请求失败的问题。但是在很多的情况下：造成“服务雪崩”的元凶正是“服务重试”机制。
- 某个服务本来就已经出现问题了，造成资源占用无法释放、请求延时等问题。这时在请求失败之后又不断的发送重试请求，在原本就无法释放的资源基础上继续膨胀式占用，导致整个系统资源耗尽。导致服务雪崩。
- 那么是不是我们就应该将“服务重试”配置关闭掉呢？当然也不是，你不能因为马路上发生了车祸，就不让所有人开车。

2、解决服务雪崩的办法

解决服务雪崩之一：服务熔断

什么是熔断

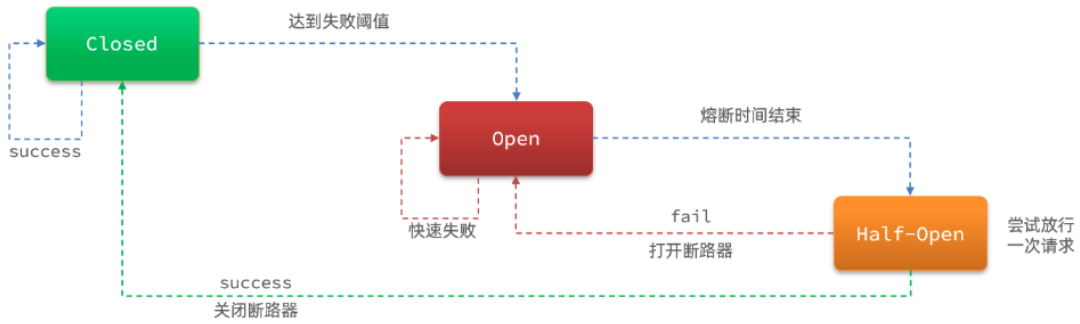
理解“熔断”这个词的由来，可以帮助我们跟好的理解“熔断”在微服务体系应用的意义。

1. 熔断机制的英文是circuit breaker mechanism，其中circuit breaker在电工学里就是断路器的意思。当电路中出现短路时，断路器会立即断开电路，保护电路负载的安全。
2. 后来熔断机制被引入股票交易。最早起源于美国，1987年10月19日，纽约股票市场爆发了史上最大的一次崩盘事件，道琼斯工业指数一天之内重挫508.32点，跌幅达22.6%，由于没有熔断机制和涨跌幅限制，许多百万富翁一夜之间沦为贫民，这一天也被美国金融界称为“黑色星期一”。2020年（今年）由于新冠疫情的影响，美国股市多次触发熔断机制，在一段时间内暂停交易，进而对整个市场起到一定的保护作用。

什么是服务熔断

服务熔断：指的是在服务提供者的错误率达到一定的比例之后，断路器就会熔断一段时间，不再去请求服务提供者，从而避免上游服务被拖垮，进而达到保护整体系统可用性的目的。

熔断恢复：熔断时间过了以后再去尝试请求服务提供者，一旦服务提供者的服务能力恢复，请求将继续可以调用服务提供者，此过程完全不需认为参与。



上图是“断路器”的状态转换图

- 断路器默认处于“关闭”状态，当服务提供者的错误率到达阈值，就会触发断路器“开启”。
- 断路器开启后进入熔断时间，到达熔断时间终点后重置熔断时间，进入“半开启”状态
- 在半开启状态下，如果服务提供者的服务能力恢复，则断路器关闭熔断状态。进而进入正常的服务状态。

- 在半开启状态下，如果服务提供者的服务能力未能恢复，则断路器再次触发服务熔断，进入熔断时间。

解决服务雪崩之二：服务降级

服务熔断后由一个问题是：原来的服务已经不能调用了，那该去调用谁呢？总不能不响应吧！这时候就需要做“服务降级”处理了！

所谓服务降级，就是一种兜底服务策略。白话说就是“实在不行就怎样”。例如：感冒了想去医院看病，但挂不上号，实在不行就先回家吃点药睡觉呗。“实在不行”的处理方法，我们成为“Fallback”方法。

在服务消费方进行服务降级

当服务提供者故障触发调用者服务的熔断机制，服务调用者就不再调用远程服务方法，而是调用本地的Fallback方法。此时你需要预先提供一个处理方法，作为服务降级之后的执行方法，Fallback返回值一般是设置的默认值或者来自缓存。

在服务提供方进行服务降级

除了可以在服务调用端实现服务降级，还可以在服务提供端实现服务降级。实际上在大型的微服务系统中，服务提供者和服务消费者并没有严格的区分，很多的服务既是提供者，也是消费者。

当然，除了服务熔断会触发服务降级和程序运行时异常，还有其他几种异常也可以触发服务降级

- 响应超时
- 达到服务限流标准
- Hystrix线程池或信号量爆满

解决服务雪崩之三：服务限流

服务限流：通过对并发访问/请求进行限速或者一个时间窗口内的请求数量进行限制来保护系统，一旦达到限制速率则可以拒绝服务。拒绝服务之后，可以有如下的处理方式：

- 定向到错误页或告知资源没有了
- 排队或等待（比如秒杀、评论、下单）
- 降级（返回默认数据或缓存数据）

3、Hystrix和Sentinel的区别

- 在SpringCloud当中支持多种服务保护技术：
  - Netfix Hystrix
  - Sentinel
  - Resilience4J

早期比较流行的是Hystrix框架，但目前国内实用最广泛的还是阿里巴巴的Sentinel框架，这里我们做下对比：

	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	基于慢调用比例或异常比例	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WarmUp模式、匀速排队模式	不支持
系统自适应保护	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

4、服务容错（一）：Hystrix

Hystrix介绍

Hystrix是一个用于微服务系统的延迟和容错库，旨在远程系统、服务和第三方库出现故障的时候，隔离服务之间的接口调用，防止级联故障导致服务雪崩。

- Hystrix github官网：https://github.com/Netflix/Hystrix
- Hystrix项目目前已经进入到维护阶段，不再开发新版本。即便如此，Hystrix的很多概念和设计思想都非常有价值，仍然值得学习
- Hystrix进入维护阶段之后，Netflix的建议是使用resilience4j,但是目前国内使用者比较少。更多的还是使用了 Spring Cloud Alibaba的sentinel。(后面章节为大家介绍)
- 无论是Setinel还是Hystrix都借鉴了Hystrix的设计，所以学习Hystrix还是非常有必要的。

注意： Spring Cloud微服务系统使用Sentinel代替Hystrix。除非是老项目会用Hystrix，新项目一定要用Sentinel。

微服务集成Hystrix（提供方）

在微服务提供方可以集成Hystrix做服务熔断，下面步骤是在mafeng-user整合Hystrix

导入Hystrix依赖

```
<dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
        </dependency>
```

注解开启Hystrix

- 在服务入口启动类上面加上`@EnableCircuitBreaker`注解

```
@EnableCircuitBreaker
```

方法上添加熔断

通过在方法上加上HystrixCommand注解和HystrixProperty注解来实现某个方法的服务熔断配置。

```
@HystrixCommand(
            commandProperties = {
                    @HystrixProperty(name =
"metrics.rollingStats.timeInMilliseconds", value = "10000"), //统计窗口时间
                    @HystrixProperty(name = "circuitBreaker.enabled", value =
"true"),  //启用熔断功能
                    @HystrixProperty(name =
"circuitBreaker.requestVolumeThreshold", value = "10"),  //20个请求失败触发熔断
                    @HystrixProperty(name =
"circuitBreaker.errorThresholdPercentage", value = "60"),  //请求错误率超过60%触发
熔断
                    @HystrixProperty(name =
"circuitBreaker.sleepWindowInMilliseconds", value = "30000"),//熔断后开始尝试恢复的
时间
            }
    )
```

- 熔断开关：enabled=ture，打开断路器状态转换的功能。
- 熔断阈值配置
  - requestVolumeThreshold=10，表示在Hystrix默认的时间窗口10秒钟之内有10个请求失败（没能正常返回结果），则触发熔断。断路器由“关闭状态”进入“开启状态”。
  - errorThresholdPercentage=60，表示在Hystrix默认的时间窗口10秒钟之内有60%以上的请求失败（没能正常返回结果），则触发熔断。断路器由“关闭状态”进入“开启状态”。
- 熔断恢复时间：sleepWindowInMilliseconds=30000，表示断路器开启之后30秒钟进入半开启状态。(为了后面测试方便，我们把熔断恢复时间设置为30秒)

测试熔断效果

为了方便测试，我在UserController的findById方法上添加以下代码

```
UserController.java
                @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "3
            }
        )
    public User findById(@PathVariable("id")Long id, HttpServletRequest request,String name){
        /*System.out.println("授权信息："+request.getHeader("Authorization"));
        System.out.println("name："+request.getParameter("name"));
        System.out.println("name："+name);*/

        //System.out.println("password="+password);

        if(id==1){
            throw new RuntimeException("模拟错误");
        }

        return userService.getById(id);
    }
```

启动mafeng-user。

正常查询请求：http://localhost:9201/user/2

异常查询请求：http://localhost:9201/user/1

当频繁访问异常查询请求后（10秒内超过6次失败），发现原来正常查询订单请求会变得不可用，此时Hystrix其实已经生效了！在mafeng-user后台会看到这行错误提示：

```
l-06 15:49:03.014 ERROR 20460 --- [nio-9401-exec-3] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.s

ang.RuntimeException Create breakpoint : Hystrix circuit short-circuited and is OPEN
com.netflix.hystrix.AbstractCommand.handleShortCircuitViaFallback(AbstractCommand.java:979) ~[hystrix-c
com.netflix.hystrix.AbstractCommand.applyHystrixSemantics(AbstractCommand.java:557) ~[hystrix-core-1.5.
com.netflix.hystrix.AbstractCommand.access$200(AbstractCommand.java:60) ~[hystrix-core-1.5.18.jar:1.5.1
com.netflix.hystrix.AbstractCommand$4.call(AbstractCommand.java:419) ~[hystrix-core-1.5.18.jar:1.5.18]
```

代表熔断器已经打开

Hystrix服务降级Fallback处理

刚才我可以看到当mafeng-user触发了熔断条件后确实发生了熔断，但对用户而言，并没有友好的提示信息！这时我们可以给熔断机制添加一个Fallback处理方法，这个做法也称为“服务降级”。

添加 fallbackMethod属性

```
/**
 * 根据id查询用户
 */
@GetMapping("{id}")
@HystrixCommand(
        fallbackMethod = "findByIdFallback",
        commandProperties = {
                @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds", value = "10000"), //统计窗口时间
                @HystrixProperty(name = "circuitBreaker.enabled", value = "true"),  //启用熔断功能
                @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"),  //20个请求失败触发熔断
                @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60"),  //请求错误率超过60%触发熔断
                @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "30000"),//熔断后开始尝试恢复的时间
        }
)
public User findById(@PathVariable("id")Long id, HttpServletRequest request,String name){
```

添加降级Fallback方法

```
 public User findByIdFallback(@PathVariable("id")Long id, HttpServletRequest
request,String name){
        User user = new User();
        user.setUsername("熔断降级中...");
        return user;
    }
```

查看效果

Hystrix服务降级全局配置

上面的示例我们是在方法上添加服务熔断， 虽然可以实现功能，但是无疑增大了我们的代码量，而且非常冗余。为了解决这个问题，我们可以使用全局配置来实现：

```
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 10000 # 设置hystrix的超时时间为6000ms
      circuitBreaker:
        #在当10秒的时间内，最近20次调用请求，请求错误率超过60%，则触发熔断5秒，期间快速失败，
以下都是默认值
        requestVolumeThreshold: 10
        errorThresholdPercentage: 50
        sleepWindowInMilliseconds: 30000
      #设置统计的时间窗口值的毫秒值，circuit break 的打开会根据1个rolling window的统计来
计算。
      #若rolling window被设为10000毫秒，则rolling window会被分成n个buckets，
```

```
      #每个bucket包含success，failure，timeout，rejection的次数的统计信息。默认10000。
      metrics:
        rollingStats:
          timeInMilliseconds: 10000
```

全局配置完成之后，想让哪一个方法实现断路器功能，就在哪一个方法上加上注解：

```
@HystrixCommand
```

对于服务降级方法，我们也可以定义全局的降级方法

在Controller类上面添加以下注解：

```
@DefaultProperties(defaultFallback = "globalFallbackMethod")  //全局服务降级处理方法
```

为了代码解耦，我们设计一个BaseController，让UserController继承

```
public class BaseController {

    /**
     * 全局服务降级处理方法
     * @return
     */
    public User globalFallbackMethod(){
        User user = new User();
        user.setUsername("全局服务熔断中...");
        return user;
    }
}
```

```
public class UserController extends BaseController
```

项目中方法返回值并不是User对象，而是统一返回结果对象，如R，Result，AjaxResponse等。

注意：方法服务熔断的优先级比全局配置高！所以 针对系统内的绝大部分接口调用采用全局配置的方式，针对个别个性化重点业务接口使用方法注解配置即可！

Hystrix结合Feign服务降级（消费方）

下面为大家介绍服务降级的另一类方法：在FeignClient上实现服务降级。这种方式是在服务调用方（或叫消费方）实现服务熔断降级。

开启Feign的服务熔断

```
feign:
  hystrix:
    enabled: true
```

声明Fallback处理类

在FeignClient注解增加fallback处理实现类

```
@FeignClient(name = "mafeng-user",path = "user",configuration =
DefaultFeignConfiguration.class,fallback = UserClientFallback.class)
public interface UserClient {
}
```

编写Fallback处理类

```
@Component
public class UserClientFallback implements UserClient{
    @Override
    public User findById(Long id) {
        User user = new User();
        user.setUsername("Feign接口熔断中...");
        return user;
    }
}
```

Hystrix DashBoard服务监控

搭建Hystrix DashBoard

1）导入依赖

```
<dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-hystrix-
dashboard</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
    </dependencies>
```

2）启动类

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardApplication {
    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardApplication.class,args);
    }
}
```

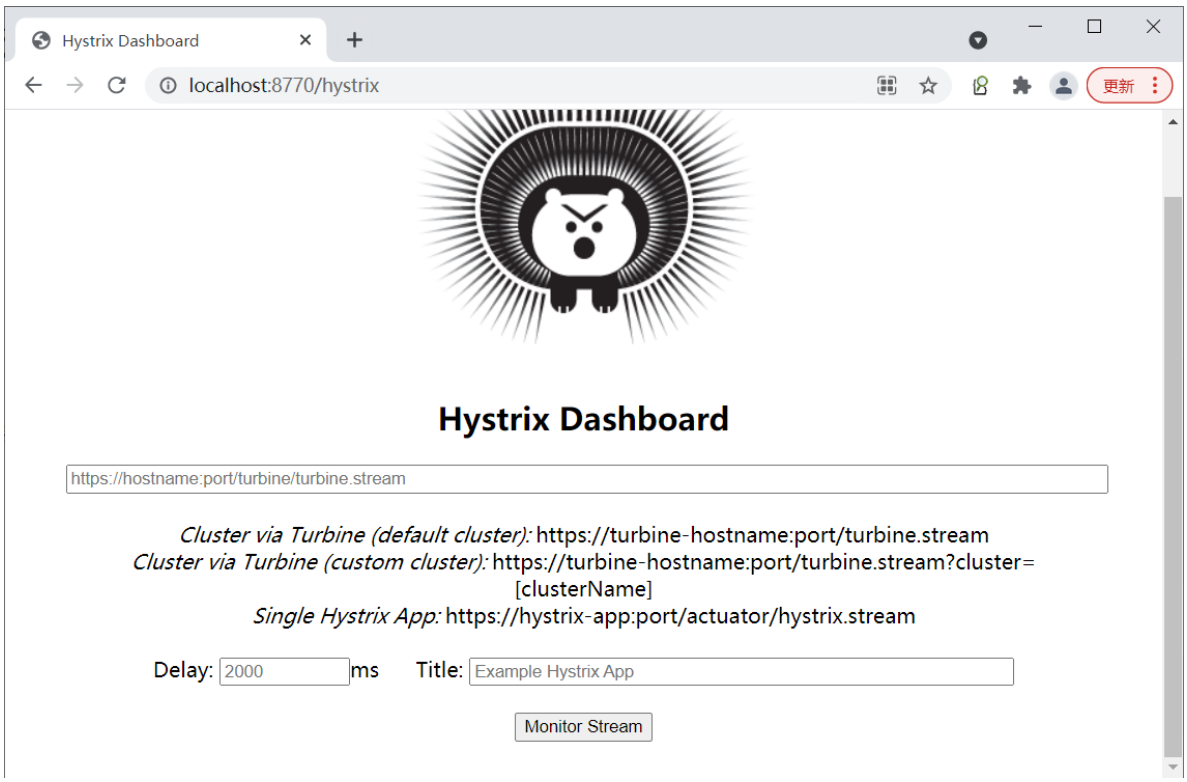
3）application.yml

```
server:
  port: 8770

hystrix:
  dashboard:
    proxy-stream-allow-list: "*"
```

4）启动并访问

http://localhost:8770/hystrix



微服务整合Hystrix DashBoard

1）导入依赖

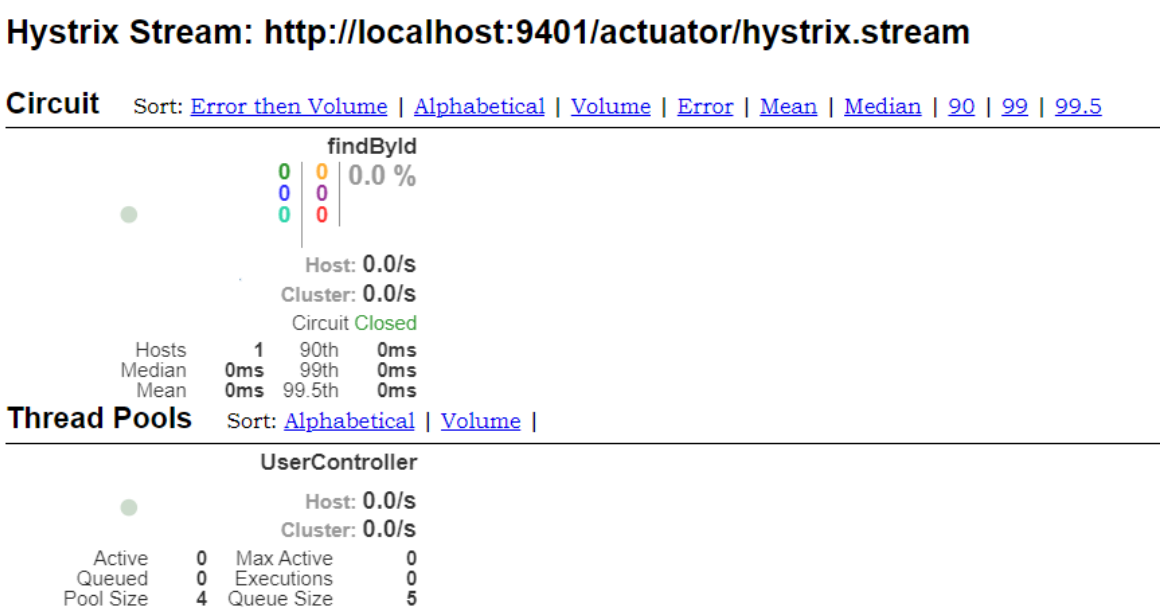
```
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
```

2）application.yml

```
management:
  endpoints:
    web:
      exposure:
        include: refresh,health,hystrix.stream
```

3）输入监听地址

http://localhost:9401/actuator/hystrix.stream



5、服务容错（二）：Sentinel

Sentinel介绍

Sentinel是阿里巴巴开源的一款微服务流量控制组件。官网地址：https://sentinelguard.io/zh-cn/index.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、集群流量控制、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 扩展接口。您可以通过实现扩展接口来快速地定制逻辑。例如定制规则管理、适配动态数据源等。

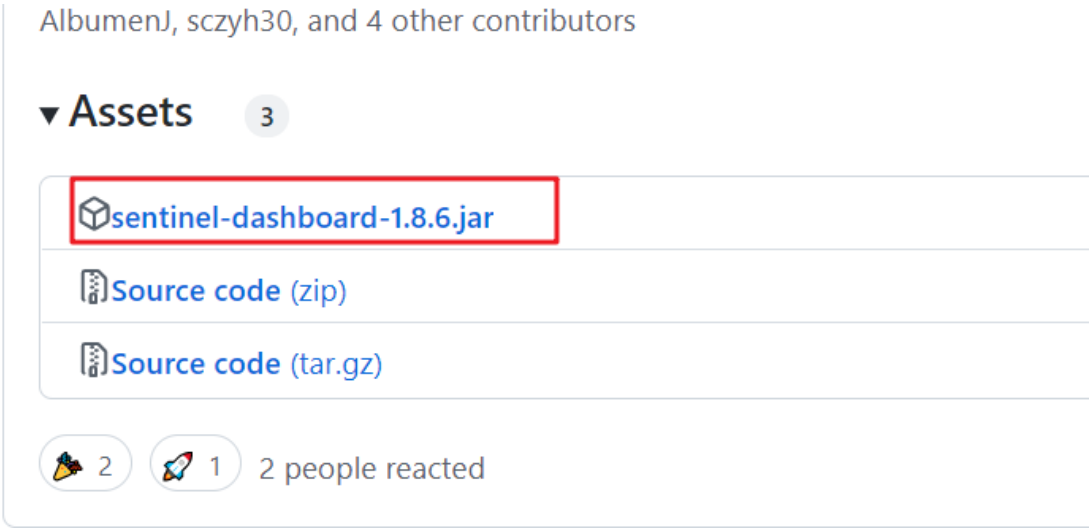
Sentinel控制台启动

sentinel官方提供了UI控制台，方便我们对系统做限流设置。大家可以在GitHub下载。

下载

我们需要下载并安装的是DashBoard控制台，下载地址：https://github.com/alibaba/Sentinel/releases。下载如下图所示的

sentinel-dashboard-x.y.z.jar



启动

注意：启动 Sentinel 控制台需要 JDK 版本为 1.8 及以上版本。

使用如下命令启动控制台：

```
nohup java -Dserver.port=8090 \
-Dcsp.sentinel.heartbeat.client.ip=192.168.66.133 \
-Dproject.name=sentinel-dashboard -jar \
sentinel-dashboard-1.8.6.jar &
```

- `-Dserver.port=8080` 用于指定 Sentinel 控制台端口为 `8090`。默认是8080。我们给它改成不常用的端口。
- `-Dcsp.sentinel.heartbeat.client.ip=192.168.66.133` 控制台部署的地址，指定控制台后客户端会自动向该地址发送心跳包。 （多网卡环境下如果不做这个配置，会报出连接超时的异常）
- `-Dproject.name=sentinel-dashboard` 指定Sentinel控制台程序的名称

从 Sentinel 1.6.0 起，Sentinel 控制台引入基本的**登录**功能，默认用户名和密码都是 `sentinel`。当然也可以通过JVM参数的方式进行修改。

- `-Dsentinel.dashboard.auth.username=sentinel` 用于指定控制台的登录用户名为 `sentinel`；
- `-Dsentinel.dashboard.auth.password=123456` 用于指定控制台的登录密码为 `123456`；如果省略这两个参数，默认用户和密码均为 `sentinel`；
- `-Dserver.servlet.session.timeout=7200` 用于指定 Spring Boot 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；

登录

访问：http://192.168.66.133:8090

微服务集成Sentinel客户端

通过maven坐标在微服务模块mafeng-user、mafeng-order中加入sentinel客户端

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

在项目的配置中加上sentinel配置，因为我是用了nacos，所以去nacos修改配置文件。如果你的服务没有使用配置中心，在application.yml里面配置就可以了。

```
spring:
  cloud:
    sentinel:
      transport:
        dashboard: 192.168.66.133:8090
```

只需要完成上述的配置，代码不需要有任何的调整，我们就可以通过实时监控查看服务内的流量QPS以及平均响应时长等信息。

需要注意的是只有服务接口被访问的情况下，在sentinel里面才可以看到监控信息。

流控规则：QPS限流

如何添加流控规则

在菜单左侧的“簇点链路”和流控规则都可以针对“服务接口”添加流控规则

- 当我们的服务接口资源被访问的时候，就会出现在“簇点链路”列表中，我们可以针对该服务接口资源配置流程控制规则。在“簇点链路”还可以配置降级规则、热点以及授权

在流控规则页面也有“新增流控规则”按钮，添加完成之后的流控规则，出现在流控规则页面列表中。

QPS限流

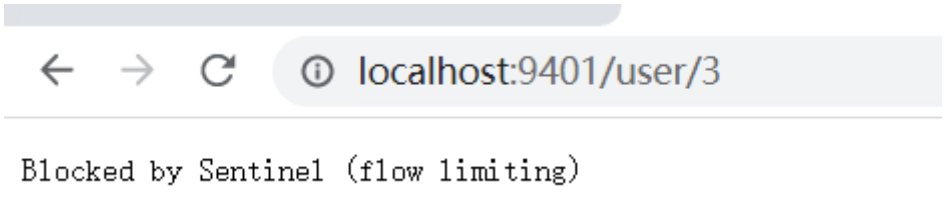
点击“新增流控规则”按钮之后，弹出如下的配置页面：

- 资源名称：表示我们针对哪个接口资源进行流控规则配置，如："/user/{id}"
- 针对来源：表示针对哪一个服务访问当前接口资源的时候进行限流，default表示不区分访问来源。如填写服务名称：mafeng-order，表示mafeng-order访问前接口资源的时候进行限流，其他服务访问该接口资源的时候不限流。
- 阈值类型/单机阈值：QPS，每秒钟请求数量。上图配置表示每秒钟超过1次请求的时候进行限流。
- 流控模式：直接，当达到限流标准时就直接限流
- 流控效果：快速失败。很简单的说就是达到限流标准后，请求就被拦截，直接失败。（HTTP状态码：429 too many request)
- 是否集群：默认情况下我们的限流策略都是针对单个服务的，Sentinel提供了集群限流的功能。

上面的限流规则用一句话说：**对于任何来源的请求，当超过每秒1次的标准之后就直接限流，访问失败抛出异常（BlockException）！**

测试QPS限流

只要稍微快点刷新http://localhost:9401/user/3这个接口就会出现下面提示，证明被限流了！



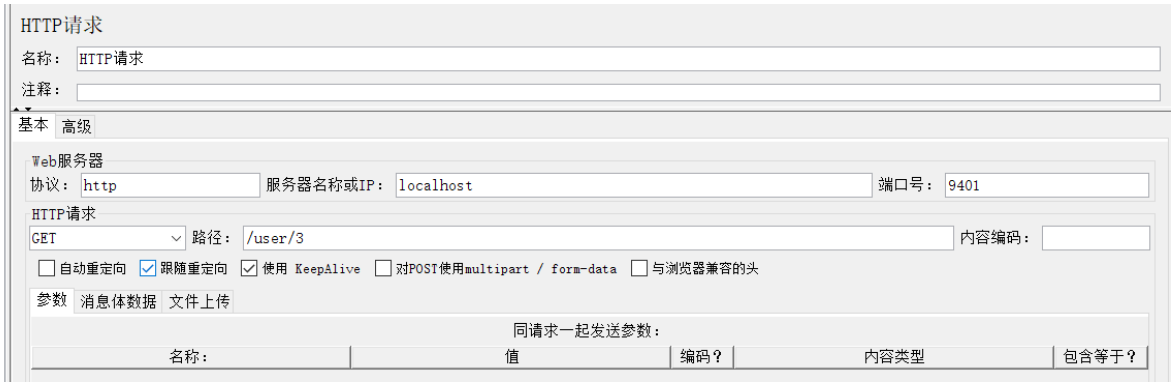
流控规则：线程数限流

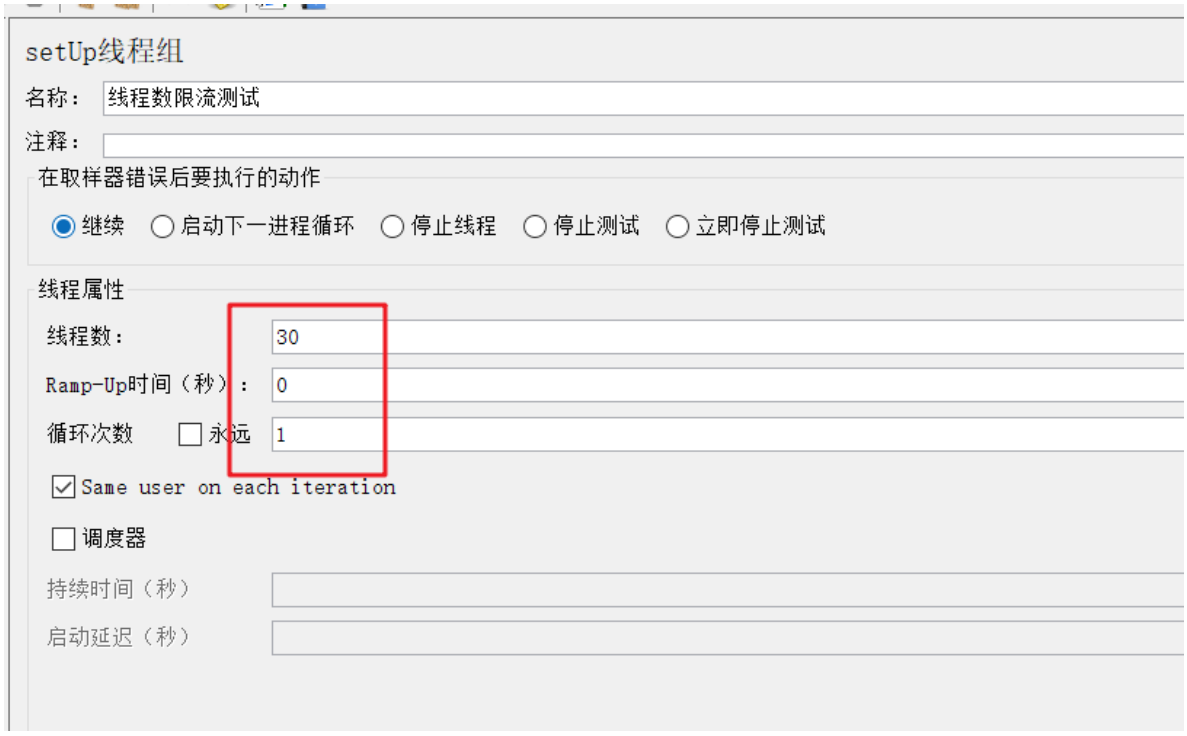
线程数限流设置

- 资源名称：表示我们针对哪个接口资源进行流控规则配置，如：“/user/{id}”
- 针对来源：表示针对哪一个服务访问当前接口资源的时候进行限流，default表示不区分访问来源。如填写服务名称：mafeng-order，表示mafeng-order访问前接口资源的时候进行限流，其他服务访问该接口资源的时候不限流。
- 阈值类型/单机阈值：线程数。**表示开启n个线程处理资源请求。**
- 流控模式：直接，当所有线程都被占用时，新进来的请求就直接限流
- 流控效果：快速失败。很简单的说就是达到限流标准后，请求就被拦截，直接失败。（HTTP状态码：429 too many request)

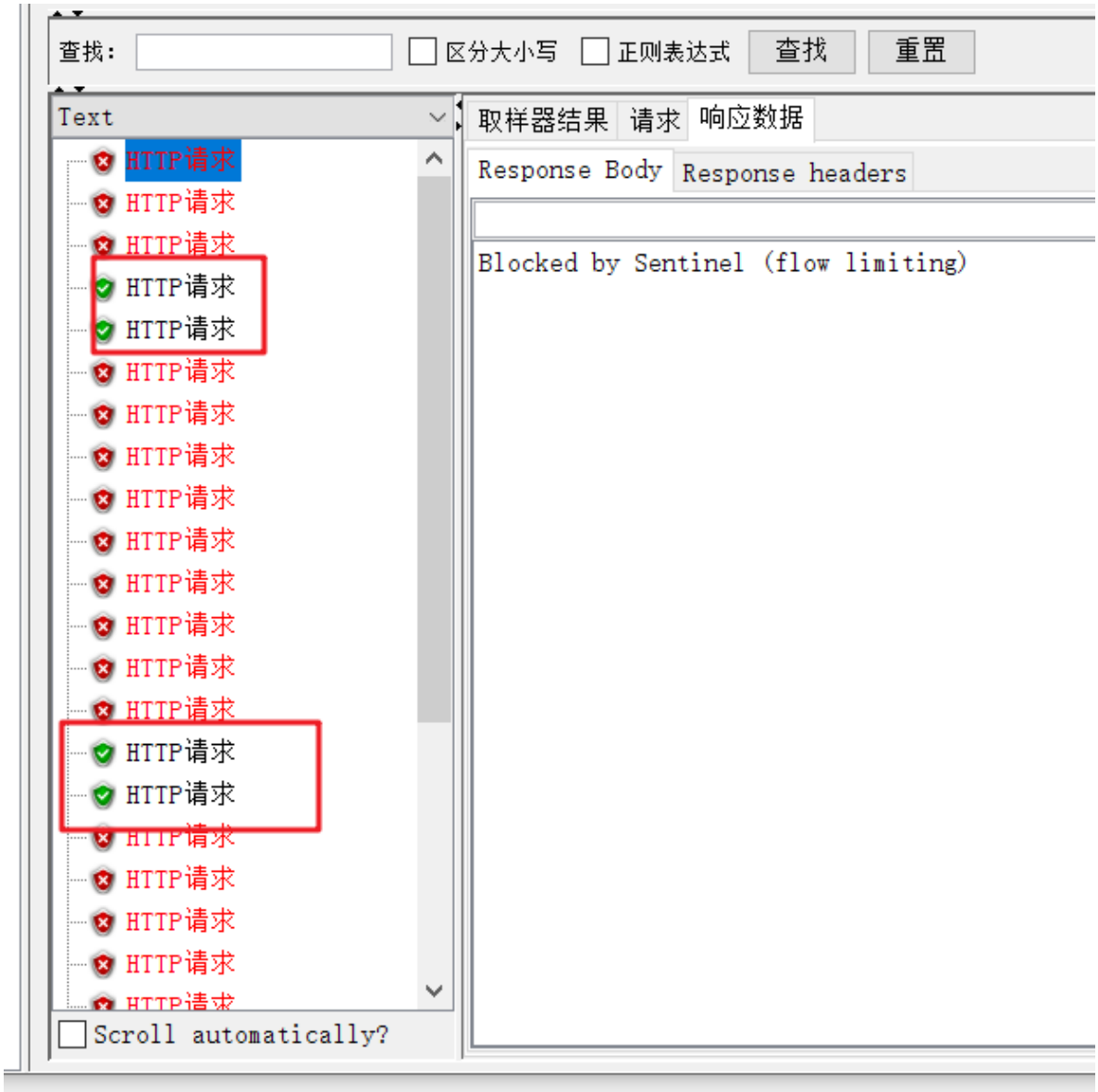
上面的限流规则用一句话说：**对于任何来源的请求，mafeng-user服务端“/user/{id}”资源接口的2个线程都被占用的时候，其他访问失败！**

JMeter测试效果




setUp线程组
名称：
线程数限流测试
注释：
在取样器错误后要执行的动作
继续
启动下一进程循环
停止线程
停止测试
立即停止测试
线程属性
线程数：
30
Ramp-Up时间（秒）：
0
循环次数
永远
1
Same user on each iteration
调度器
持续时间（秒）
启动延迟（秒）

测试结果：


查找：
区分大小写
正则表达式
查找
重置
Text
HTTP请求
取样器结果
请求
响应数据
Response Body
Response headers
Blocked by Sentinel (flow limiting)
Scroll automatically?

流控规则：关联限流

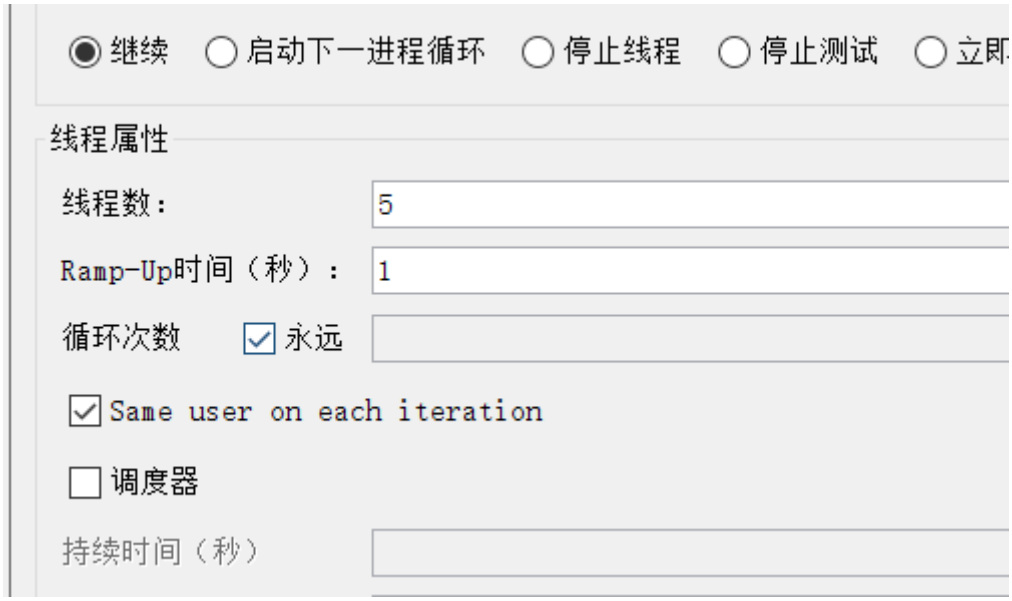
关联限流设置

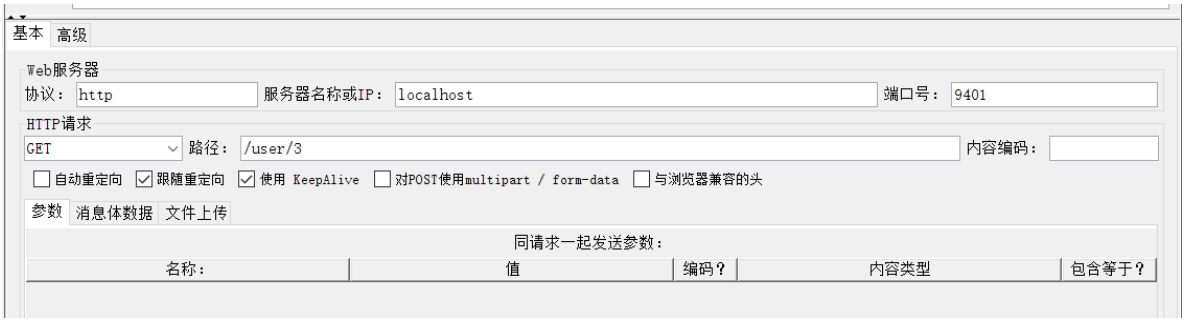
图的配置表示：

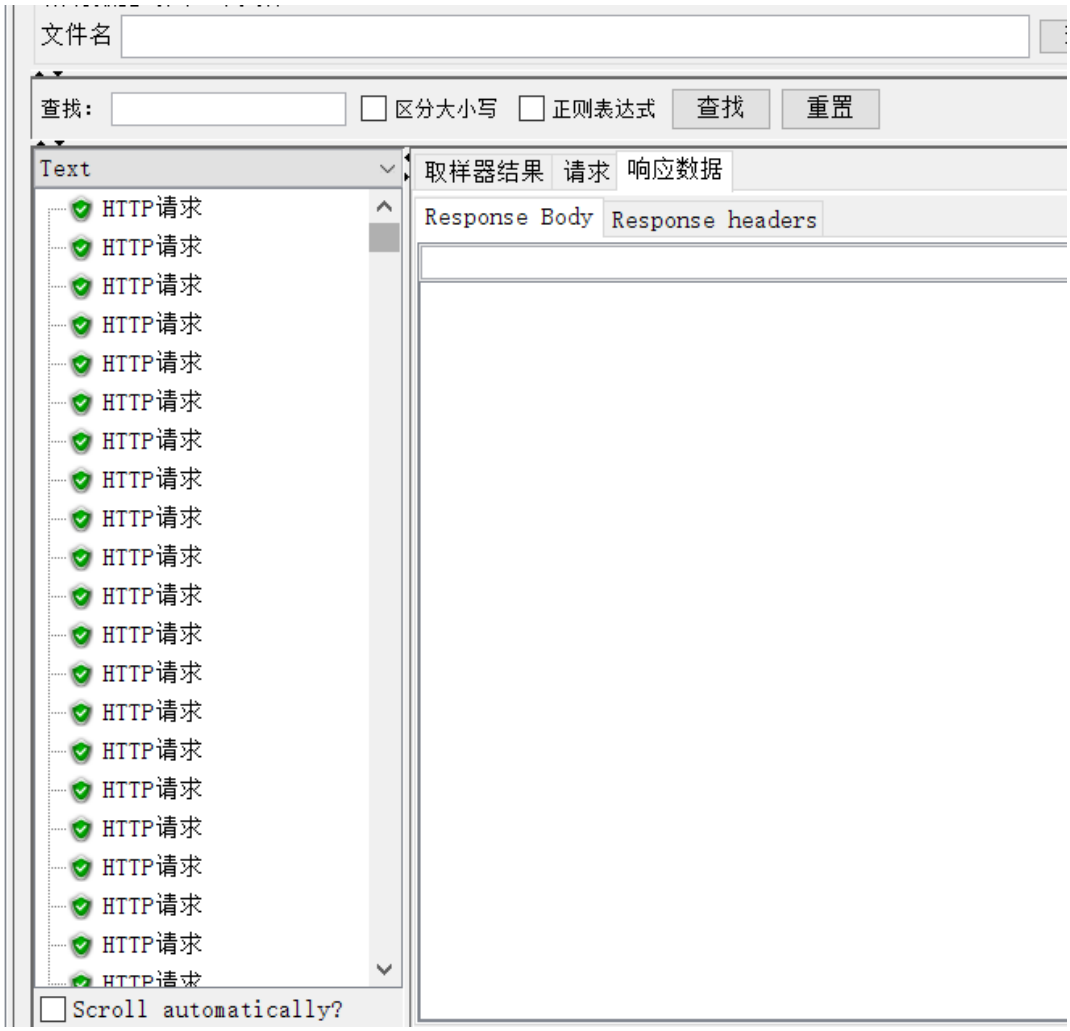
- 资源名：指需要被真正限流的接口资源。
- 关联资源：被限流接口所关联的接口。当大量的并发请求达到“/user/{id}”关联资源接口的限流标准的时候，“/role/list”资源将被限流。流控效果是快速失败。

大家注意不要把限流关系弄反了！限流规则是为了限制“资源”，而不是“关联资源”！

JMeter测试效果

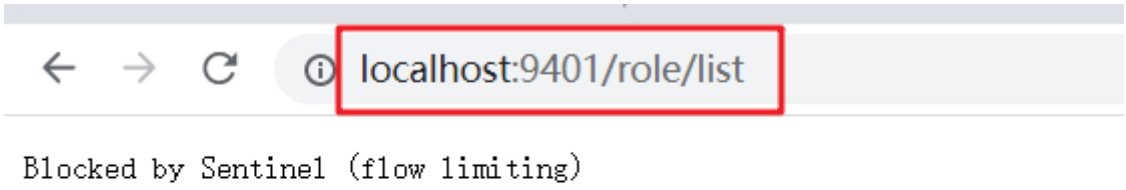






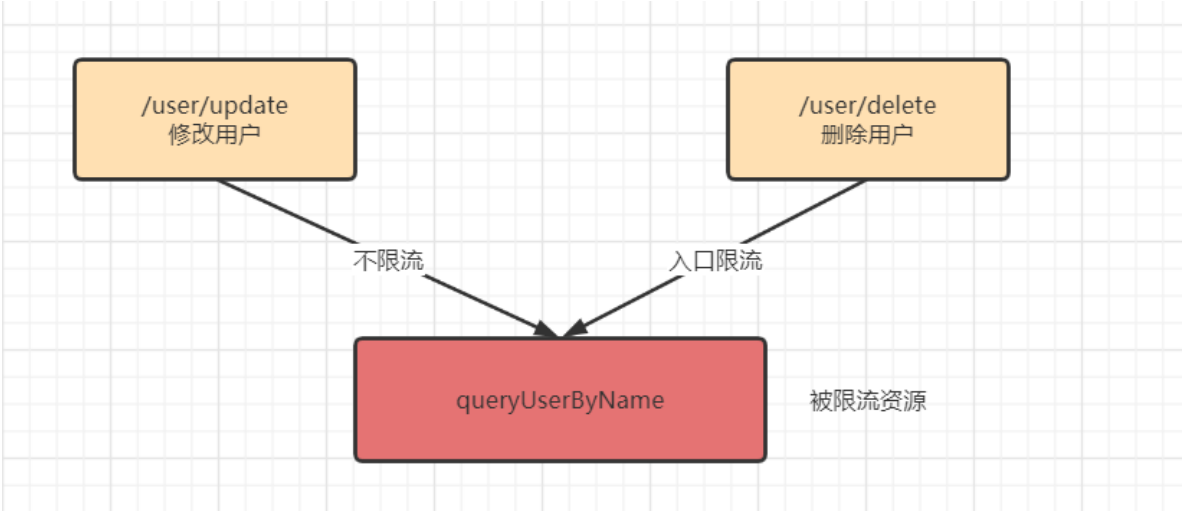
针对“/user/{id}”接口1秒5次请求，虽然为该接口配置限流规则（1秒1次），但是访问并未失败

反而，访问/role/list接口时发现被限流了！



流控规则：链路限流

需求说明



- 我们针对queryUserByName资源进行流控规则配置，入口为:"/user/delete”。
- 期望实现的效果是从"/user/delete”访问queryUserByName资源被限流，从“/user/update”入口访问queryUserByName资源不被限流。

添加模拟代码

UserController

```
    /**
     * 模拟修改用户接口
     */
    @GetMapping("/update")
    public String update(){
        userService.queryUserByName();
        return "修改用户";
    }

    /**
     * 模拟删除用户接口
     */
    @GetMapping("/delete")
    public String delete(){
        userService.queryUserByName();
        return "删除用户";
    }
```

UserService

```
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    @Override
    public void queryUserByName() {
        System.out.println("根据用户名查询用户");
    }
}
```

在业务层添加资源标记

在queryUserByName方法上添加 @SentinelResource 注解

```
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements Us
    @Override
    @SentinelResource("queryUserByName")
    public void queryUserByName() {
        System.out.println("根据用户名查询用户");
    }
}
```

添加Sentinel限流标记才能成为流控资源

配置Sentinel

Sentinel默认会让相同资源路径聚合成一个，这样无法在多个资源访问同一个资源实现限流。这时，需要关闭路径聚合功能

```
spring:
  cloud:
    sentinel:
      transport:
        dashboard: 192.168.66.133:8090
      web-context-unify: false  # 关闭路径聚合功能，不关闭无法实现链路限流
```

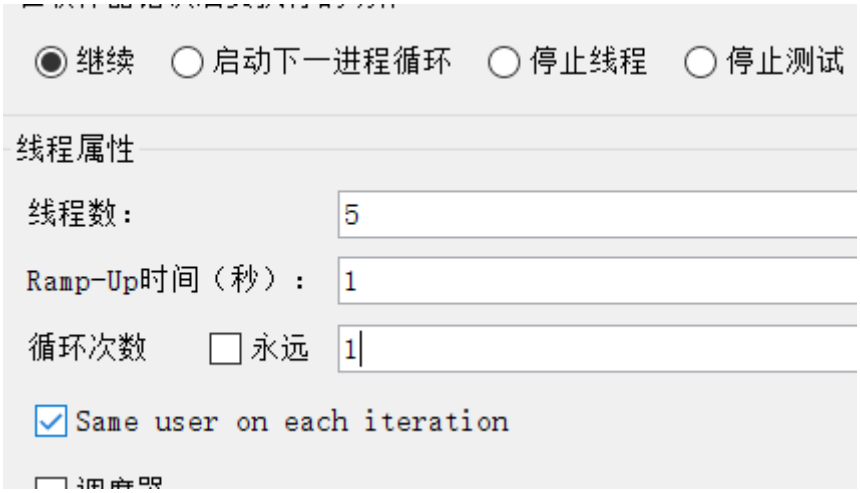
链路限流配置

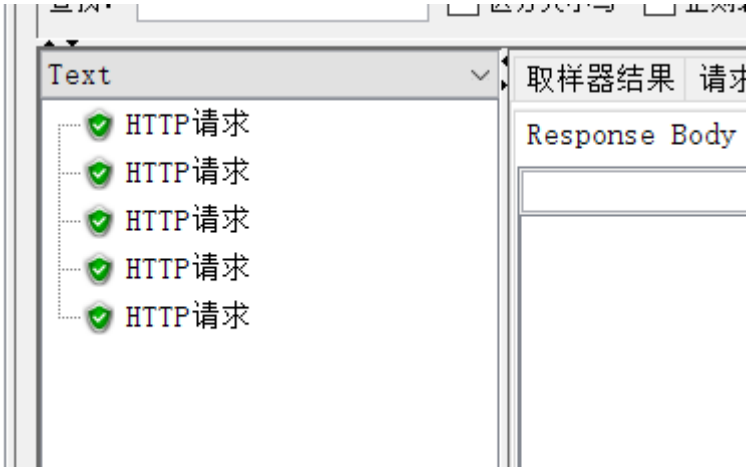
上面的规则配置的含义是：

- 从"/user/delete"访问queryUserByName资源被限流，从其他入口访问queryUserByName资源不被限流。
- 限流规则是：1秒钟只允许访问一次，超出之后访问失败。

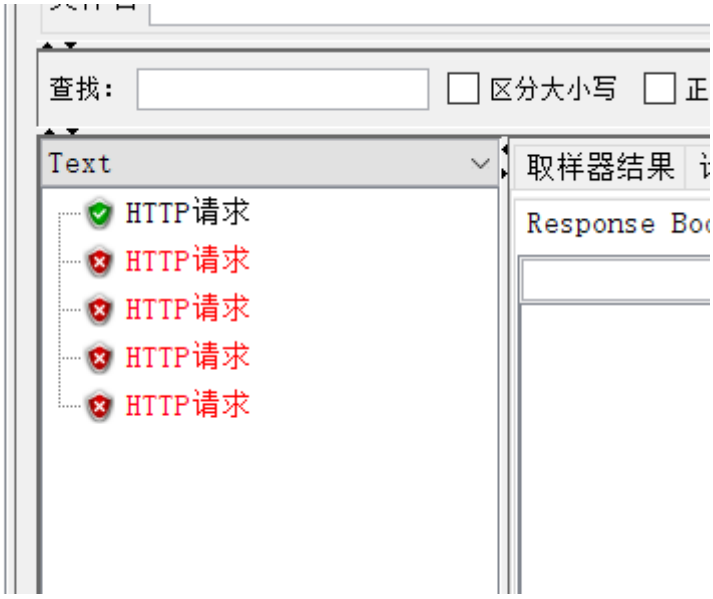
JMeter测试效果

访问/user/update接口





访问/user/delete接口：



流控效果：WarmUp

什么是WarmUp

Warm Up（`RuleConstant.CONTROL_BEHAVIOR_WARM_UP`）方式，即预热/冷启动方式。当系统流量长期处于低水位的情况下，当流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮。通过"冷启动"，让通过的流量缓慢增加，在一定时间内逐渐增加到阈值上限，给冷系统一个预热的时间，避免冷系统被压垮。

为什么冷系统容易被压垮?
一般在我们系统内部会有线程池，比如：数据库连接线程池。在系统较为空闲的时候，数据库连接线城池内只有少量的连接。假设突然大量的请求并发而至，数据库连接池会去创建新的连接，用来支撑高并发请求。但是这个连接创建的过程需要时间，有可能这边连接池内新连接还没创建完成，这些少量的连接支撑不住就会被压垮。

所以在类似这种场景下，Warm Up让流量缓慢爬升，从而给数据库连接池创建连接一个缓冲的时间，就显得非常有必要了!

WarmUp配置

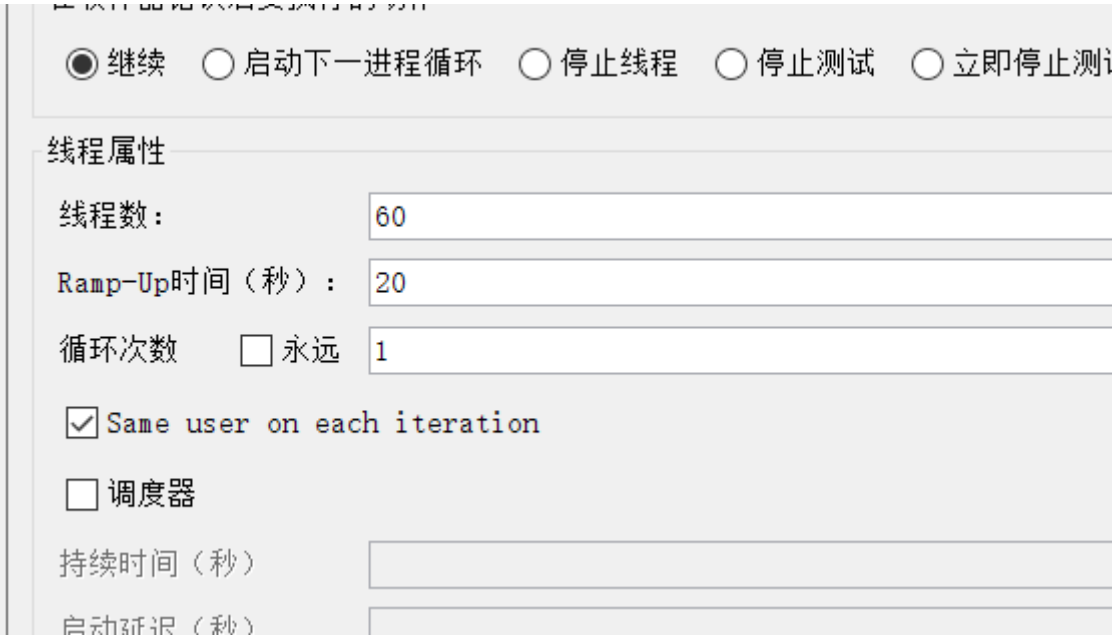
Warm Up配置有三个要素：

- 冷启动因子coldFactoer，默认等于3
- 预热时长（配置项）
- QPS单机阈值（配置项）

举例：当预热时长=8，QPS单机阈值=3

- 当并发请求到达的时候，**实际的单机阈值**是：QPS单机阈值配置/coldFactoer=3/3=1,也就是每秒钟只能一个请求访问成功。
- 预热时长为8秒，**实际的单机阈值**在8秒钟内逐步由1 -> 2 -> 3，最终等于QPS单机阈值配置。

JMeter测试效果

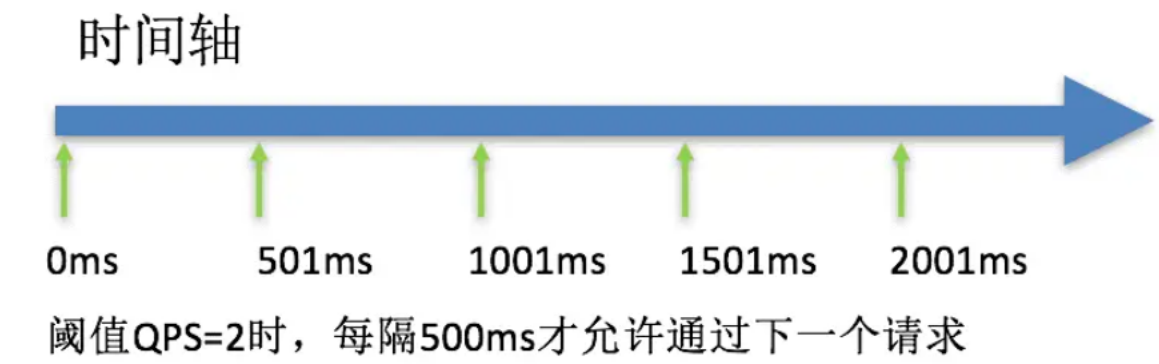


流控效果：匀速排队

匀速排队配置

匀速排队：就是让请求以均匀的速度通过，阈值类型必须是QPS。

上图的配置表示的是：“/user/{id}”资源服务接口，每秒钟匀速通过2个请求。当每秒请求大于2的时候，多余的请求排队等待，等待的时间是5000ms。如果5000ms以内请求得不到处理，就被限流访问失败！



JMeter测试效果

请求刚开始发送的时候，我们配置的匀速通过阈值是2，所以每秒处理2个请求。先发送的请求先进入排队队列，在5秒之内发送的请求几乎都被成功处理了，后来队列里面的请求积压的越来越多，导致后面不断有请求超时（超过5秒）。

热点参数限流

什么是热点参数限流

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。

Sentinel 利用 LRU 策略统计最近最常访问的热点参数，结合令牌桶算法来进行参数级别的流控。

热点参数限流配置

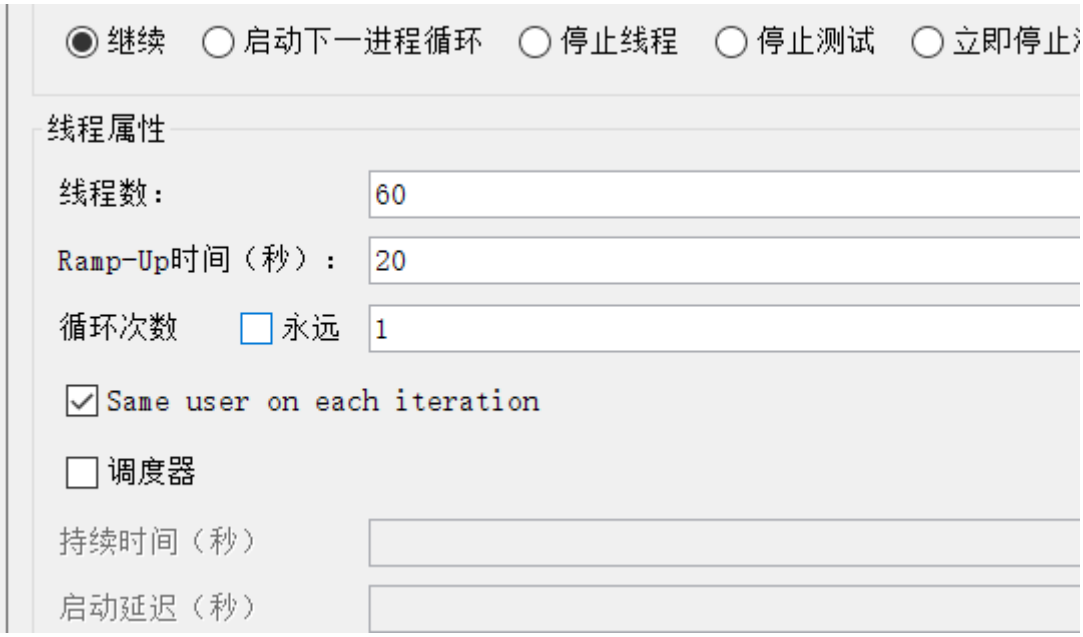
1）给热点资源接口添加流控标记

```
//@HystrixCommand
@SentinelResource("hotUser")
public User findById(@PathVariable("id")Long id, HttpServletRequest request, String name){
    /*System.out.println("授权信息："+request.getHeader("Authorization"));
    System.out.println("name："+request.getParameter("name"));
    System.out.println("name："+name);*/
```

2）配置热点参数

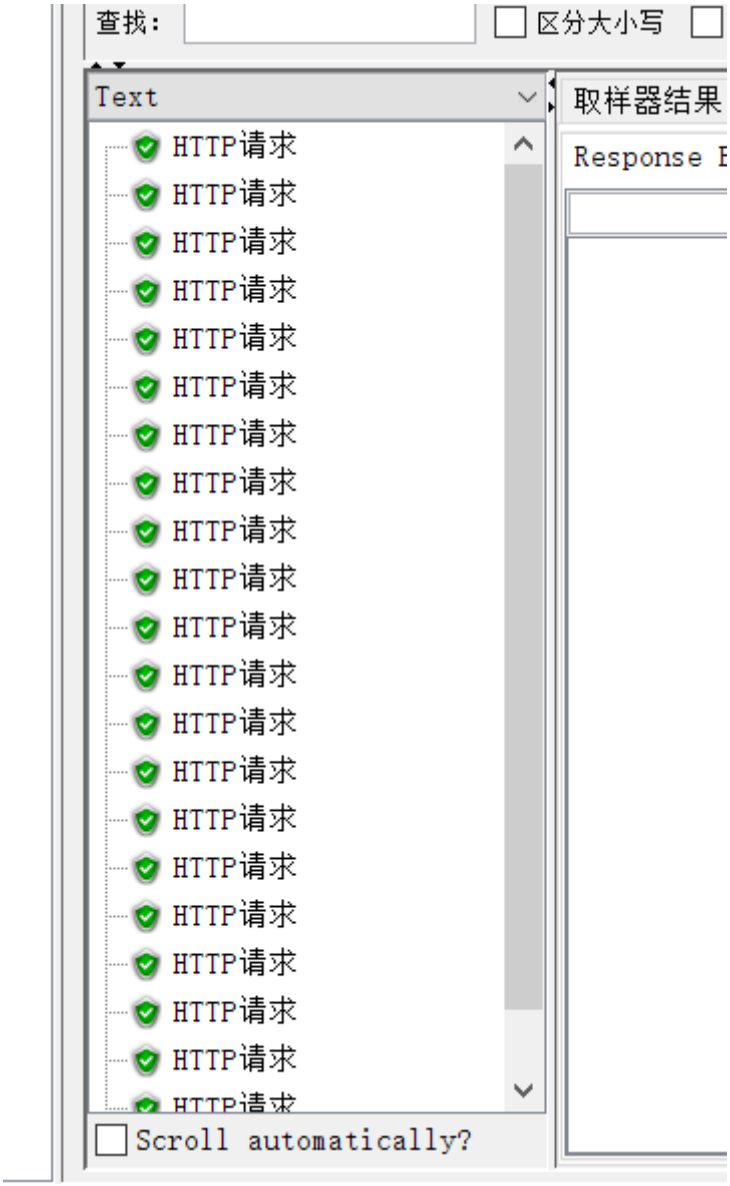
/user/4的请求QPS限流为5，其他参数QPS限流为1。

JMeter测试效果



测试/user/3

测试/user/4



熔断降级：慢调用比例

Sentinel**熔断降级**会在调用链路中某个资源出现不稳定状态时（例如调用超时或异常比例升高），对这个资源的调用进行限制，让请求快速失败，避免影响到其它的资源而导致级联错误。当资源被降级后，在接下来的降级时间窗口之内，对该资源的调用都自动熔断（默认行为是抛出`DegradeException`）。

一共有三种熔断降级策略：

- 慢调用比例
- 异常比例
- 异常数量

慢调用比例配置

熔断降级：异常数和比例

异常数配置

异常比例配置

Sentinel异常处理

如果要自定义异常时的返回结果，需要实现BlockExceptionHandler接口：

```
public interface BlockExceptionHandler {
    /**
     * 处理请求被限流、降级、授权拦截时抛出的异常：BlockException
     */
    void handle(HttpServletRequest request, HttpServletResponse response,
BlockException e) throws Exception;
}
```

这个方法有三个参数：

- HttpServletRequest request：request对象
- HttpServletResponse response：response对象
- BlockException e：被sentinel拦截时抛出的异常

这里的BlockException包含多个不同的子类：

异常	说明
FlowException	限流异常
ParamFlowException	热点参数限流的异常
DegradeException	降级异常
AuthorityException	授权规则异常
SystemBlockException	系统规则异常

自定义Sentinel处理类

```
@Component
```

```
public class SentinelExceptionHandler implements BlockExceptionHandler {
    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
BlockException e) throws Exception {
        String msg = "未知异常";
        int status = 429;

        if (e instanceof FlowException) {
            msg = "请求被限流了";
        } else if (e instanceof ParamFlowException) {
            msg = "请求被热点参数限流";
        } else if (e instanceof DegradeException) {
            msg = "请求被降级了";
        }

        response.setContentType("application/json;charset=utf-8");
        response.setStatus(status);
        response.getWriter().println("{\"msg\": " + msg + ", \"status\": " +
status + "}");
    }
}
```