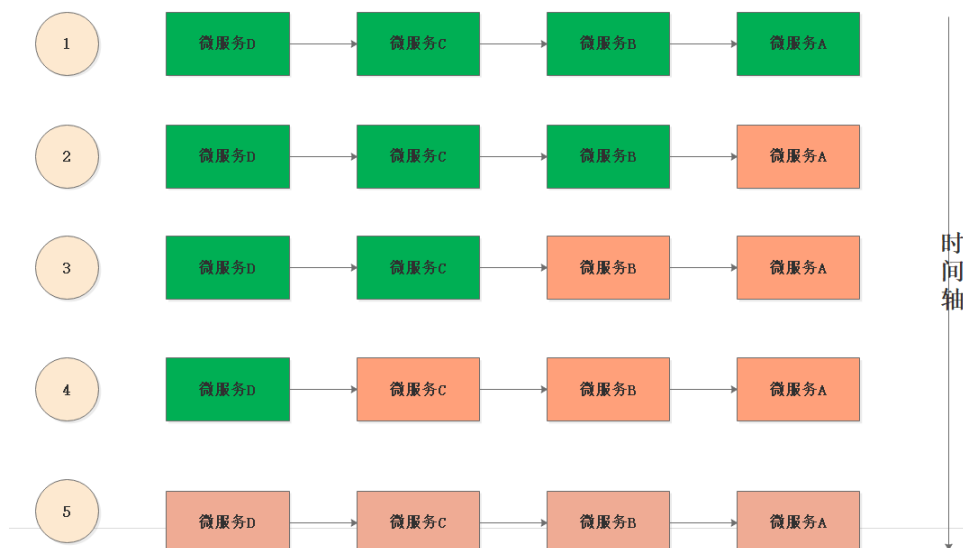


一:什么是雪崩效应?



业务场景,高并发调用

①: 正常情况下, 微服务A B C D 都是正常的

②: 随着时间推移, 在某一个时间点 微服务A突然挂了, 此时的微服务B 还在疯狂的调用微服务A,由于A已经挂了, **所以B调用A必须等待服务调用超时**。而我们知道每次B->A的时候B都会去创建线程 (而线程由是计算机的资源 比如cpu 内存等) 。由于是高并发场景 **B就会阻塞大量的线程**。那么B所在的机器就会去创建线程, 但是计算机资源是有限的, 最后B的服务器就会宕机。**(说白了微服务B 活生生的被猪队友微服务A给拖死的)**

③: 由于**微服务A这个猪队友**活生生的把微服务B给拖死, 导致微服务B也宕机了, 然后也会导致微服务C D 出现类似的情况, 最终我们的猪队友A成功的把微服务 B C D 都拖死了。这种情况也叫做**服务雪崩**。也有一个**专业术语(cascading failures) 级联故障**。

二:容错三板斧

2.1)超时:超时机制你懂的, 配置一下超时时间, 例如1秒——每次请求在1秒内必须返回, 否则到点就把线程掐死, 释放资源!

思路: 一旦超时, 就释放资源。由于释放资源速度较快, 应用就不会那么容易被拖死

代码演示:(针对调用方处理)

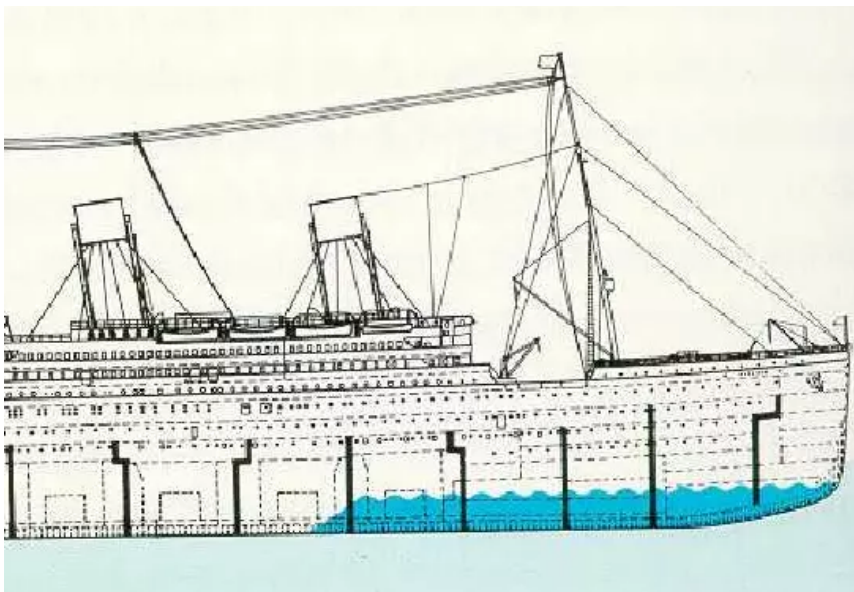
```
1
2 /**
3  * 第一步:设置RestTemplate的超时时间
4  */
5 @Configuration
6 public class WebConfig {
7
8     @Bean
9     public RestTemplate restTemplate() {
10         //设置restTemplate的超时时间
11         SimpleClientHttpRequestFactory requestFactory = new SimpleClientHttpRequestFactory();
12         requestFactory.setReadTimeout(1000);
13         requestFactory.setConnectTimeout(1000);
14         RestTemplate restTemplate = new RestTemplate(requestFactory);
```

```

15  return restTemplate;
16  }
17  }
18  /**
19  *第二步:进行超时异常处理
20  */
21  try{
22  ResponseEntity<ProductInfo> responseEntity= restTemplate.getForEntity(uri+orderInfo.getProduct
No(), ProductInfo.class);
23  productInfo = responseEntity.getBody();
24  }catch (Exception e) {
25  throw new RuntimeException("调用超时");
26  }
27
28  /**
29  * 设置全局异常处理
30  */
31  @ControllerAdvice
32  public class TulingExceptionHandler {
33
34  @ExceptionHandler(value = {RuntimeException.class})
35  @ResponseBody
36  public Object dealBizException() {
37  OrderVo orderVo = new OrderVo();
38  orderVo.setOrderNo("-1");
39  orderVo.setUserName("容错用户");
40  return orderVo;
41  }
42  }
43

```

2.2)舱壁隔离模式

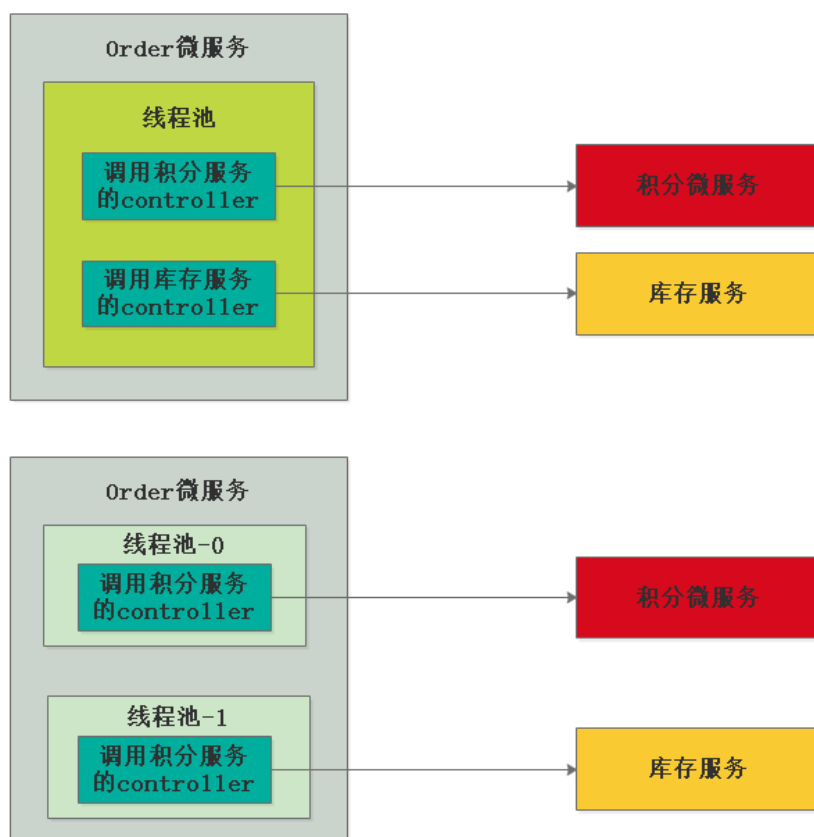


有兴趣的可以先了解一下船舱构造——一般来说，现代的轮船都会分很多舱室，舱室之间用钢板焊死，彼此隔离。这样即使有某个/某些船舱进水，也不会影响其他舱室，浮力够，船不会沉。

2.2.1)代码中的舱壁隔离(线程池隔离模式)

M类使用线程池1，N类使用线程池2，彼此的线程池不同，并且为每个类分配的线程池大小，例如coreSize=10。举个例子：M类调用B服务，N类调用C服务，如果M类和N类使用相同的线程池，那么如果B服务挂了，M类调用B服务的接口并发又很高，你没有任何保护措施，你的服务就很可能被M类拖死。而如果M类有自己的线程池，N类也有自己的线程池，如果B服务挂了，M类顶多是将自己的线程池占满，不会影响N类的线程池——于是N类依然能正常工作，

思路：不把鸡蛋放在一个篮子里。你有你的线程池，我有我的线程池，你的线程池满了和我没关系，你挂了也和我没关系。

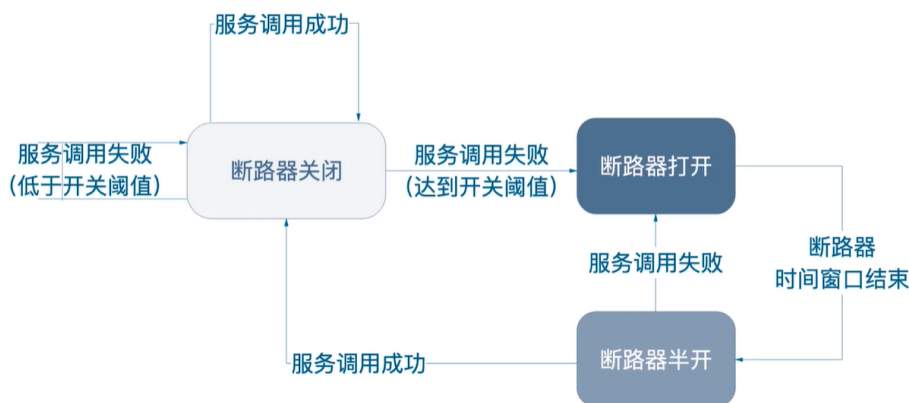


2.3) 断路器模式

现实世界的断路器大家肯定都很了解，每个人家里都会有断路器。断路器实时监控电路的情况，如果发现电路电流异常，就会跳闸，从而防止电路被烧毁。

软件世界的断路器可以这样理解：实时监测应用，如果发现在**一定时间内失败次数/失败率达到一定阈值**，就“跳闸”，断路器打开——此时，请求直接返回，而不去调用原本调用的逻辑。

跳闸一段时间后（例如15秒），断路器会进入半开状态，这是一个**瞬间态**，此时允许一次请求调用该调的逻辑，如果成功，则断路器关闭，应用正常调用；如果调用依然不成功，断路器继续回到打开状态，过段时间再进入半开状态尝试——通过“跳闸”，应用可以保护自己，而且避免浪费资源；而通过半开的设计，可实现应用的“自我修复”。



三: Sentinel 流量控制, 容错, 降级 (Hystrix在Sentinel面前就是弟弟)

3.1) 什么是Sentinel?

A lightweight powerful flow control component enabling reliability and monitoring for microservices. (轻量级的流量控制、熔断降级 Java 库)

github官网地址:<https://github.com/alibaba/Sentinel>

wiki:<https://github.com/alibaba/Sentinel/wiki/>

3.1.1) Sentinel的初体验 tulingvip04-ms-alibaba-sentinel-helloworld

V1版本

①: 添加依赖包

```

1 <dependency>
2 <groupId>com.alibaba.csp</groupId>
3 <artifactId>sentinel-core</artifactId>
4 <version>1.7.1</version>
5 </dependency>

```

②: 编写测试 controller

```

1 @Slf4j
2 public class SentinelHelloController {
3
4     @Autowired
5     private BusServiceImpl busiService;
6     /**
7      * 初始化流控规则
8      */
9     @PostConstruct
10    public void init() {
11
12        List<FlowRule> flowRules = new ArrayList<>();
13
14        //创建流控规则对象
15        FlowRule flowRule = new FlowRule();
16        //设置流控规则 QPS
17        flowRule.setGrade(RuleConstant.FLOW_GRADE_QPS);
18        //设置受保护的资源
19        flowRule.setResource("HelloSentinel");
20        //设置受保护的资源的阈值
21        flowRule.setCount(1);

```

```

22
23 flowRules.add(flowRule);
24
25 //加载配置好的规则
26 FlowRuleManager.loadRules(flowRules);
27 }
28
29 /**
30 * 频繁请求接口 http://localhost:8080/helloSentinelV1
31 * 这种做法的缺点:
32 * 1)业务侵入性很大,需要在你的controoler中写入 非业务代码..
33 * 2)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
34 * @return
35 */
36 @RequestMapping("/helloSentinelV1")
37 public String testHelloSentinelV1() {
38
39     Entry entity =null;
40     //关联受保护的资源
41     try {
42         entity = SphU.entry("helloSentinelV1");
43         //开始执行 自己的业务方法
44         busiService.doBusi();
45         //结束执行自己的业务方法
46     } catch (BlockException e) {
47         log.info("testHelloSentinelV1方法被流控了");
48         return "testHelloSentinelV1方法被流控了";
49     }finally {
50         if(entity!=null) {
51             entity.exit();
52         }
53     }
54     return "OK";
55 }
56
57 }

```

测试效果: <http://localhost:8080/helloSentinelV1>

2020-02-13 18:34:49.942	INFO	12880	com.tuling.service.BusiServiceImpl	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:50.066	INFO	12880	[nio-8080-exec-1] com.tuling.service.BusiServiceImpl	: 执行业务方法:执行时机:Thu Feb 13 18:34:50 CST 2020
2020-02-13 18:34:50.203	INFO	12880	[nio-8080-exec-3] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:50.341	INFO	12880	[nio-8080-exec-10] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:50.475	INFO	12880	[nio-8080-exec-7] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:50.747	INFO	12880	[nio-8080-exec-9] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:50.896	INFO	12880	[nio-8080-exec-1] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:51.032	INFO	12880	[nio-8080-exec-3] com.tuling.service.BusiServiceImpl	: 执行业务方法:执行时机:Thu Feb 13 18:34:51 CST 2020
2020-02-13 18:34:51.178	INFO	12880	[nio-8080-exec-10] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:51.330	INFO	12880	[nio-8080-exec-7] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:51.456	INFO	12880	[nio-8080-exec-9] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:51.601	INFO	12880	[nio-8080-exec-1] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:51.724	INFO	12880	[nio-8080-exec-3] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:51.872	INFO	12880	[nio-8080-exec-10] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:51.988	INFO	12880	[nio-8080-exec-7] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:52.111	INFO	12880	[nio-8080-exec-9] com.tuling.service.BusiServiceImpl	: 执行业务方法:执行时机:Thu Feb 13 18:34:52 CST 2020
2020-02-13 18:34:52.280	INFO	12880	[nio-8080-exec-1] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:52.366	INFO	12880	[nio-8080-exec-3] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:52.501	INFO	12880	[nio-8080-exec-10] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:52.657	INFO	12880	[nio-8080-exec-7] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:52.784	INFO	12880	[nio-8080-exec-9] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:52.933	INFO	12880	[nio-8080-exec-1] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了
2020-02-13 18:34:53.061	INFO	12880	com.tuling.service.BusiServiceImpl	: 执行业务方法:执行时机:Thu Feb 13 18:34:53 CST 2020
2020-02-13 18:34:53.185	INFO	12880	[nio-8080-exec-10] c.t.c.HelloWorldSentinelController	: testHelloSentinelV1方法被流控了

v1版本的缺陷:如下

业务侵入性很大,需要在你的controoler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

V2版本: 基于v1版本 再添加一个依赖

```
1 <dependency>
2   <groupId>com.alibaba.csp</groupId>
3   <artifactId>sentinel-annotation-aspectj</artifactId>
4   <version>1.7.1</version>
5 </dependency>
```

编写controller

```
1 @PostConstruct
2 public void init() {
3
4     List<FlowRule> flowRules = new ArrayList<>();
5
6     /**
7      * 定义 helloSentinelV2 受保护的资源的规则
8      */
9     //创建流控规则对象
10    FlowRule flowRule2 = new FlowRule();
11    //设置流控规则 QPS
12    flowRule2.setGrade(RuleConstant.FLOW_GRADE_QPS);
13    //设置受保护的资源
14    flowRule2.setResource("helloSentinelV2");
15    //设置受保护的资源的阈值
16    flowRule2.setCount(1);
17
18    flowRules.add(flowRule2);
19
20
21    //加载配置好的规则
22    FlowRuleManager.loadRules(flowRules);
23 }
24
25 @RequestMapping("/helloSentinelV2")
26 @SentinelResource(value = "helloSentinelV2",blockHandler = "testHelloSentinelV2BlockMethod")
27 public String testHelloSentinelV2() {
28     busiService.doBusi();
29     return "OK";
30 }
31
32 public String testHelloSentinelV2BlockMethod(BlockException e) {
33     return "testHelloSentinelV2方法被流控了..." + e;
34 }
```

测试效果:http://localhost:8080/helloSentinelV2

优缺点:

```

/**
 * 频繁请求接口 http://localhost:8080/helloSentinelV2
 * 优点: 需要配置aspectj的切面SentinelResourceAspect ,添加注解@SentinelResource
 * 解决了v1版本中 sentinel的业务侵入代码问题,通过blockHandler指定被流控后调用的方法.
 * 缺点: 若我们的controller中的方法逐步变多,那么受保护的方法也越来越多, 会导致一个问题
 * blockHandler的方法也会越来越多 引起方法急剧膨胀 怎么解决
 *
 * 注意点:
 * blockHandler 对应处理 BlockException 的函数名称,
 * 可选项。blockHandler 函数访问范围需要是 public, 返回类型需要与原方法相匹配,
 * 参数类型需要和原方法相匹配并且最后加一个额外的参数,
 * 类型为 BlockException。blockHandler 函数默认需要和原方法在同一个类中
 * @return
 */

```

V3:版本 基于V2缺点改进

```

1  @PostConstruct
2  public void init() {
3
4      List<FlowRule> flowRules = new ArrayList<>();
5      /**
6       * 定义 helloSentinelV3 受保护的资源的规则
7       */
8      //创建流控规则对象
9      FlowRule flowRule3 = new FlowRule();
10     //设置流控规则 QPS
11     flowRule3.setGrade(RuleConstant.FLOW_GRADE_QPS);
12     //设置受保护的资源
13     flowRule3.setResource("helloSentinelV3");
14     //设置受保护的资源的阈值
15     flowRule3.setCount(1);
16     flowRules.add(flowRule3);
17
18     //加载配置好的规则
19     FlowRuleManager.loadRules(flowRules);
20 }
21
22 /**
23  * 我们看到了v2中的缺点,我们通过blockHandlerClass 来指定处理被流控的类
24  * 通过testHelloSentinelV3BlockMethod 来指定blockHandlerClass 中的方法名称
25  * ***这种方式 处理异常流控的方法必须要是static的
26  * 频繁请求接口 http://localhost:8080/helloSentinelV3
27  * @return
28  */
29 @RequestMapping("/helloSentinelV3")
30 @SentinelResource(value = "helloSentinelV3",blockHandler = "testHelloSentinelV3BlockMethod",blockHandlerClass = BlockUtils.class)
31 public String testHelloSentinelV3() {
32     busiService.doBusi();

```



```

33     return "OK";
34 }
35
36 异常处理类:
37 @Slf4j
38 public class BlockUtils {
39
40     public static String testHelloSentinelV3BlockMethod(BlockException e){
41         log.info("testHelloSentinelV3方法被流控了");
42         return "testHelloSentinelV3方法被流控了";
43     }
44 }

```

测试: <http://localhost:8080/helloSentinelV3>

```

: 执行业务方法:执行时机:Thu Feb 13 18:46:06 CST 2020
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: 执行业务方法:执行时机:Thu Feb 13 18:46:07 CST 2020
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: 执行业务方法:执行时机:Thu Feb 13 18:46:08 CST 2020
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: testHelloSentinelV3方法被流控了
: 执行业务方法:执行时机:Thu Feb 13 18:46:09 CST 2020
: testHelloSentinelV3方法被流控了

```

缺点: 不能动态的添加规则。 如何解决问题 请看3.2。

3.2) 如何在工程中快速整合Sentinel

```

1 <!--加入sentinel-->
2 <dependency>
3     <groupId>com.alibaba.cloud</groupId>
4     <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
5 </dependency>
6
7
8 <!--加入actuator-->

```



```

9 <dependency>
10 <groupId>org.springframework.boot</groupId>
11 <artifactId>spring-boot-starter-actuator</artifactId>
12 </dependency>

```

添加Sentinel后，会暴露/actuator/sentinel端点<http://localhost:8080/actuator/sentinel>

而Springboot默认是没有暴露该端点的，所以我们需要自己设置

```

1 server:
2   port: 8080
3 management:
4   endpoints:
5     web:
6     exposure:
7     include: '*'

```

← → ↺ localhost:8080/actuator/sentinel




```

{
  "blockPage": null,
  "appName": "order-center",
  "consoleServer": null,
  "coldFactor": "3",
  "rules": {
    "systemRules": [],
    "authorityRule": [],
    "paramFlowRule": [],
    "flowRules": [],
    "degradeRules": []
  },
  "metricsFileCharset": "UTF-8",
  "filter": {
    "order": -2147483648,
    "urlPatterns": [
      "/*"
    ],
    "enabled": true
  },
  "totalMetricsFileCount": 6,
  "datasource": {},
  "clientIp": "192.168.159.1",
  "clientPort": "8719",
  "logUsePid": false,
  "metricsFileSize": 52428800,
  "logDir": "C:\\Users\\zhuwei\\logs\\csp\\",
  "heartbeatIntervalMs": 10000
}

```

3.3)我们需要整合Sentinel-dashboard(哨兵流量卫兵)

下载地址:<https://github.com/alibaba/Sentinel/releases> **老师下载的是1.6.3版本的**

Assets	
 sentinel-dashboard-1.7.0.jar	20.2 MB
 sentinel-envoy-rls-token-server-1.7.0.jar	12.1 MB
 Source code (zip)	
 Source code (tar.gz)	

第一步:执行 java -jar sentinel-dashboard-1.7.0.jar 启动 (就是一个Springboot工程)

第二步: 访问我们的sentinel控制台(1.6版本加入登陆页面) <http://192.168.159.8:8080/>

默认账号密码 sentinel/sentinel



第三步:我们的微服务tulingvip04-ms-alibaba-sentinel-order 也整合好了sentinel, 我们也搭建好了Sentinel控制台了, 临门一脚 为微服务添加sentinel的控制台地址

```
1 spring:
2   cloud:
3     sentinel:
4       transport:
5         dashboard: 192.168.159.8:8080
```

四: sentinel监控性能指标详解

4.1)实时监控面板

在这个面板中我们监控我们接口的通过的QPS和拒绝的QPS

****在没有设置流控规则** 我们是看不到拒绝的QPS**



设置流控规则为2的时候的监控图



4.2) 簇点链路 用来显示微服务的所监控的API



4.3) 流控设置

簇点链路 选择具体的访问的API 然后点击流控按钮

新增流控规则

资源名

/selectOrderInfoById/1

针对来源

default

阈值类型

QPS

线程数

单机阈值

2

是否集群

☐

流控模式

直接

关联

链路

流控效果

快速失败

Warm Up

排队等待

关闭高级选项

新增并继续添加

新增

取消

含义：

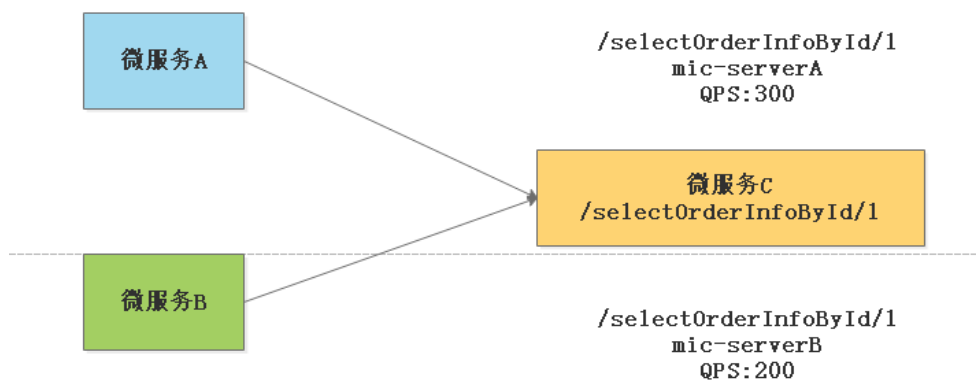
资源名称：为我们接口的API /selectOrderInfoById/1

针对来源：这里是默认的default (标示不针对来源)，还有一种情况就是假设微服务A需要调用这个资源，微服务B也需要调用这个资源，那么我们就可以单独的为微服务A和微服务B进行设置阈值。

阈值类型：分为QPS和线程数 假设阈值为2

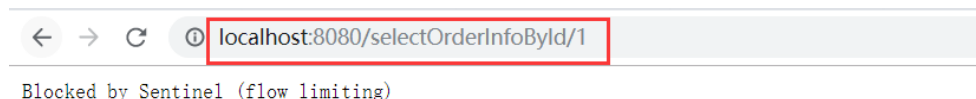
QPS类型：只得是每秒钟访问接口的次数>2就进行限流

线程数：为接受请求该资源 分配的线程数>2就进行限流。



流控模式:

①:直接:这种很好理解, 就是达到设置的阈值后直接被流控抛出异常 疯狂的这个路径



②:关联

业务场景 我们现在有二个api, 第一个是保存订单, 一个是查询订单,假设我们希望优先操作是"保存订单"

编辑流控规则

资源名: /findById/1

针对来源: default

阈值类型: ☒ QPS ☐ 线程数

单机阈值: 1

是否集群: ☐

流控模式: ☐ 直接 ☒ 关联 ☐ 链路

关联资源: /saveOrder

流控效果: ☒ 快速失败 ☐ Warm Up ☐ 排队等待

关闭高级选项

保存 取消

测试:写二个读写测试接口

```

2  * 方法实现说明:模仿 流控模式【关联】 读接口
3  * @author:smlz
4  * @param orderNo
5  * @return:
6  * @exception:
7  * @date:2019/11/24 22:06
8  */
9  @RequestMapping("/findById/{orderNo}")
10 public Object findById(@PathVariable("orderNo") String orderNo) {
11     log.info("orderNo:{},执行查询操作");
12     return orderInfoMapper.selectOrderInfoById(orderNo);
13 }
14
15 /**
16 * 方法实现说明:模仿流控模式【关联】 写接口(优先)
17 * @author:smlz
18 * @return:
19 * @exception:
20 * @date:2019/11/24 22:07
21 */
22 @RequestMapping("/saveOrder")
23 public String saveOrder() {
24     log.info("执行保存操作,模仿返回订单ID");
25     return UUID.randomUUID().toString();
26 }

```

测试代码: 写一个for循环一直调用我们的写接口, 让写接口QPS达到阈值

```

1 public class TestSentinelRule {
2
3     public static void main(String[] args) throws InterruptedException {
4         RestTemplate restTemplate = new RestTemplate();
5
6         for(int i=0;i<1000;i++) {
7             restTemplate.postForObject("http://localhost:8080/saveOrder",null,String.class);
8             Thread.sleep(200);
9         }
10    }
11 }

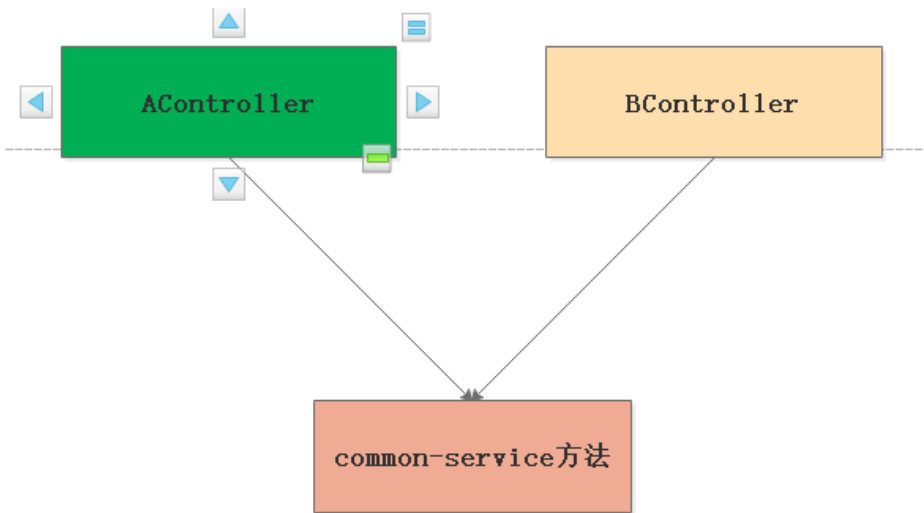
```

此时访问我们的读接口: 此时被限制流了.

← → ↺ ⓘ localhost:8080/findById/1

Blocked by Sentinel (flow limiting)

③:链路 (用法说明,本地实验没成功 用alibaba 未毕业版本0.9.0可以测试出效果 API级别的限制流量)



编辑流控规则

资源名

common

针对来源

default

阈值类型

☒ QPS

☐ 线程数

单机阈值

2

是否集群

☐

流控模式

☐ 直接

☐ 关联

☒ 链路

入口资源

/findAll

流控效果

☒ 快速失败

☐ Warm Up

☐ 排队等待

关闭高级选项

保存

取消

代码:

```
1 @GetMapping("/findAll")
2 public String findAll() {
3     this.orderServiceImpl.common();
4     return "findAll";
5 }
6
7 @GetMapping("/findAllByCondition")
8 public String findAllByCondition() {
9     this.orderServiceImpl.common();
10    return "findAllByCondition";
11 }
12
13 @Service
```

```

14 @Slf4j
15 public class OrderServiceImpl {
16
17     @SentinelResource("common")
18     public String common() {
19         log.info("common.....");
20         return "common";
21     }
22 }

```

根据流控规则来说: 只会限制/findAll的请求, 不会限制/findAllByCondition规则

流控效果:

①:快速失败(直接抛出异常) 每秒的QPS 操过1 就直接抛出异常

源码:com.alibaba.csp.sentinel.slots.block.flow.controller.DefaultController

编辑流控规则

资源名	common		
针对来源	default		
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数	单机阈值	1
是否集群	<input type="checkbox"/>		
流控模式	<input checked="" type="radio"/> 直接 <input type="radio"/> 关联 <input type="radio"/> 链路		
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待		

关闭高级选项

← → ↺ localhost:8080/findAll

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Nov 26 15:27:46 CST 2019

There was an unexpected error (type=Internal Server Error, status=500).

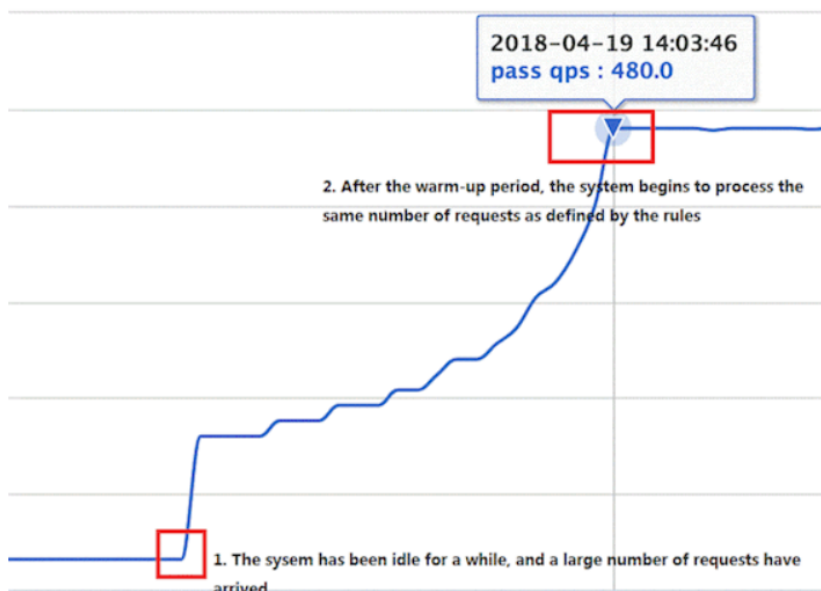
No message available

②:预热(warmUp)

当流量突然增大的时候, 我们常常会希望系统从空闲状态到繁忙状态的切换的时间长一些。即如果系统在此之前长期处于空闲的状态, 我们希望处理请求的数量是缓步的增多, 经过预期的时间以后, 到达系统处理请求个数的最大值。Warm Up (冷启动, 预热) 模式就是为了实现这个目的的。

冷加载因子: codeFactor 默认是3

默认 coldFactor 为 3, 即请求 QPS 从 threshold / 3 开始, 经预热时长逐渐升至设定的 QPS 阈值。



编辑流控规则

资源名

common

针对来源

default

阈值类型

☒ QPS ☐ 线程数

单机阈值

100

是否集群

☐

流控模式

☒ 直接 ☐ 关联 ☐ 链路

流控效果

☐ 快速失败 ☒ Warm Up ☐ 排队等待

预热时长

10

关闭高级选项

上图设置: 就是QPS从 $100/3=33$ 开始算 经过10秒钟, 到达一百的QPS 才进行限制流量

详情文档:<https://github.com/alibaba/Sentinel/wiki/%E9%99%90%E6%B5%81---%E5%86%B7%E5%90%AF%E5%8A%A8>

源码:com.alibaba.csp.sentinel.slots.block.flow.controller.WarmUpController

③:排队等待

这种方式适合用于请求以突刺状来到, 这个时候我们不希望一下子把所有的请求都通过, 这样可能会把系统压垮; 同时我们也期待系统以稳定的速度, 逐步处理这些请求, 以起到“削峰填谷”的效果, 而不是拒绝所有请求。

选择排队等待的阈值类型必须是QPS

编辑流控规则

资源名

/findAll

针对来源

default

阈值类型

QPS

线程数

单机阈值

10

是否集群

流控模式

直接

关联

链路

流控效果

快速失败

Warm Up

排队等待

超时时间

20000

关闭高级选项

保存

取消

单机阈值:10表示 每秒通过的请求个数是10,那么每隔100ms通过一次请求.

每次请求的最大等待时间为20000=20s, 超过20S就丢弃请求。

具体文档:

<https://github.com/alibaba/Sentinel/wiki/%E6%B5%81%E9%87%8F%E6%8E%A7%E5%88%B6-%E5%8C%80%E9%80%9F%E6%8E%92%E9%98%9F%E6%A8%A1%E5%BC%8F>

具体源码:com.alibaba.csp.sentinel.slots.block.flow.controller.RateLimiterController

4.4)降级规则

①:rt（平局响应时间）

编辑降级规则

资源名

/testRt

降级策略

RT

异常比例

异常数

RT

1

时间窗口

5

保存

取消

平均响应时间 (DEGRADE_GRADE_RT): 当 1s 内持续进入 5 个请求, 对应时刻的平均响应时间 (秒级) 均超过阈值 (count, 以 ms 为单位) , 那么在接下的时间窗口 (DegradeRule 中的 timeWindow, 以 s 为单位) 之内, 对这个方法的调用都会自动地熔断 (抛出 DegradeException) 。注意 Sentinel 默认统计的 RT 上限是 4900 ms, 超出此阈值的都会算作 4900 ms, 若需要变更此上限可以通过启动配置项 -Dcsp.sentinel.statistic.max.rt=xxx 来配置

②:异常比例 (DEGRADE_GRADE_EXCEPTION_RATIO):

当资源的每秒请求量 ≥ 5 ，并且每秒异常总数占通过量的比值超过阈值（DegradeRule 中的 count）之后，资源进入降级状态，即在接下的时间窗口（DegradeRule 中的 timeWindow，以 s 为单位）之内，对这个方法的调用都会自动地返回。异常比率的阈值范围是 [0.0, 1.0]，代表 0% - 100%。

③： **异常数 (DEGRADE_GRADE_EXCEPTION_COUNT)**：当资源近 1 分钟的异常数目超过阈值之后会进行熔断。注意由于统计时间窗口是分钟级别的，若 timeWindow 小于 60s，则结束熔断状态后仍可能再进入熔断状态。

编辑降级规则

资源名: /testExCount

降级策略: ☒ RT ☐ 异常比例 ☒ 异常数

异常数: 4 时间窗口: 10

min 分钟级别统计

保存 取消

4.5) 热点参数:

业务场景：秒杀业务，比如商城做促销秒杀，针对苹果11（商品id=5）进行9.9秒杀活动，那么这个时候，我们去请求订单接口(商品id=5)的请求流量十分大，我们就可以通过热点参数规则来控制商品id=5的请求的并发量。而其他正常商品的请求不会收到限制。那么这种热点参数规则很使用。

order-skill

限流模式: QPS 模式

参数索引: 1

单机阈值: 1 统计窗口时长: 5 秒

是否集群: ☐

参数例外项

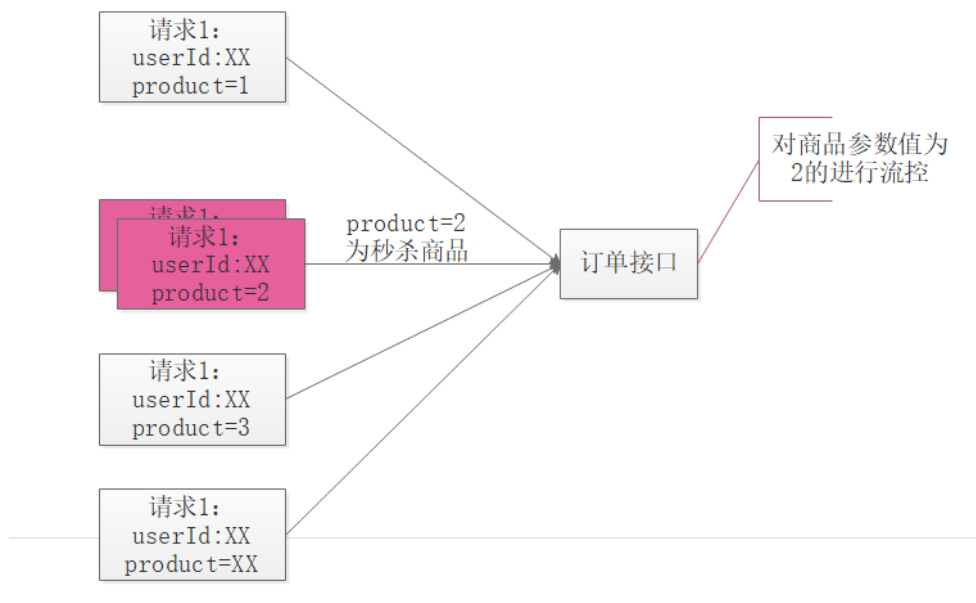
参数类型: 例外项参数值 限流阈值: 限流阈值 + 添加

参数值	参数类型	限流阈值	操作
1	java.lang.String	1000	删除

关闭高级选项

若第二个参数的 值为1 可以单独设置qps阈值流量

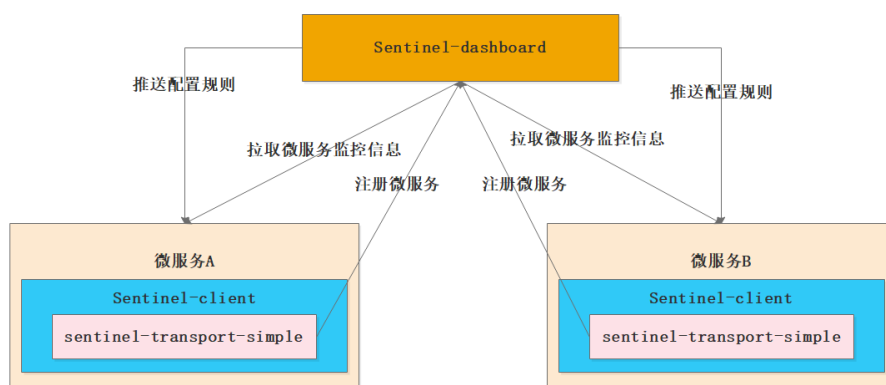
保存 取消



五:Sentinel-dashboard 控制台和 我们的微服务通信原理.

5.1)控制台如何获取到微服务的监控信息?

5.2)在控制台配置规则，如何把规则推送给微服务的?

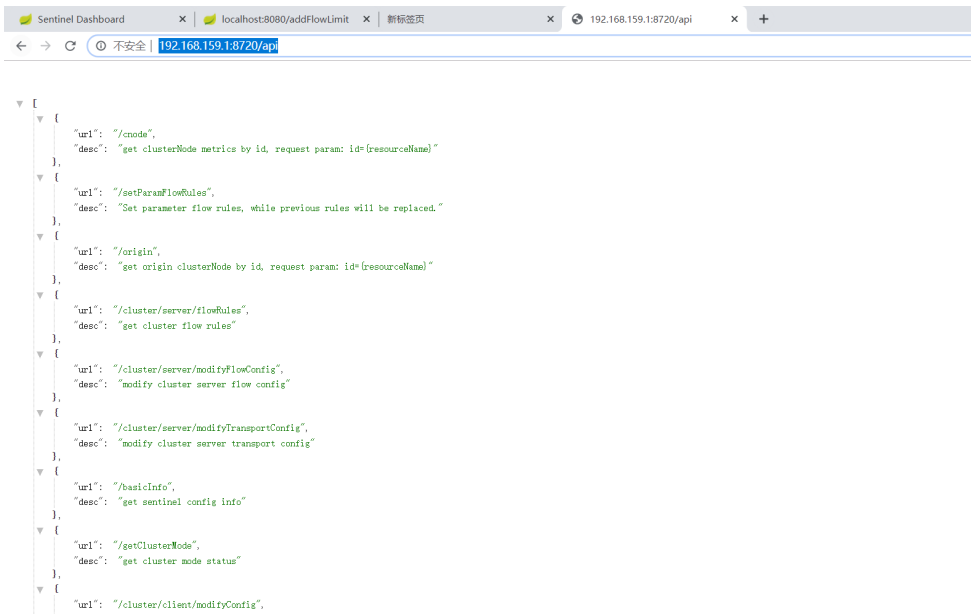


我们通过观察到sentinel-dashboard的机器列表上观察注册服务微服务信息。我们的控制台就可以通过这些微服务的注册信息跟我们的具体的微服务进行通信。

order-center

机器列表 实例总数 1, 健康 1, 失联 0.						
关键字						
机器名	IP 地址	端口号	Sentinel 客户端版本	健康状态	心跳时间	操作
DESKTOP-NS33J0G	192.168.159.1	8720	1.6.3	健康	2019/11/27 20:23:43	
共 1 条记录, 每页 10 条记录, 第 1 / 1 页						

5.3)微服务整合sentinel时候的提供的一些接口API地址: <http://localhost:8720/api>



5.4) 我们可以通过代码设置规则(我们这里用流控规则为例)

```
1 @RestController
2 public class AddFlowLimitController {
3
4     //添加流控规则
5     @RequestMapping("/addFlowLimit")
6     public String addFlowLimit() {
7         List<FlowRule> flowRuleList = new ArrayList<>();
8         //为哪个流控规则设置流控规则
9         FlowRule flowRule = new FlowRule("/testAddFlowLimitRule");
10
11         //设置QPS阈值
12         flowRule.setCount(1);
13
14         //设置流控模型为QPS模型
15         flowRule.setGrade(RuleConstant.FLOW_GRADE_QPS);
16
17         flowRuleList.add(flowRule);
18
19         FlowRuleManager.loadRules(flowRuleList);
20
21         return "success";
22     }
23
24     @RequestMapping("/testAddFlowLimitRule")
25     public String testAddFlowLimitRule() {
26         return "testAddFlowLimitRule";
27     }
28
29 }
```

添加效果截图: 执行:<http://localhost:8080/addFlowLimit>

编辑流控规则

资源名

/testAddFlowLimitRule

针对来源

default

阈值类型

☒ QPS
 ☐ 线程数

单机阈值

1

是否集群

☐

高级选项

保存

取消

Sentinel具体配置项

<https://github.com/alibaba/Sentinel/wiki/%E5%90%AF%E5%8A%A8%E9%85%8D%E7%BD%AE%E9%A1%B9>

5.5)对SpringMvc端点保护关闭(一般应用场景是做压测需要关闭)

```

1 spring:
2
3   cloud:
4     nacos:
5     discovery:
6     server-addr: localhost:8848
7     sentinel:
8     transport:
9     dashboard: localhost:9999
10    filter:
11    enabled: false #关闭Spring mvc的端点保护

```

那么我们的这种类型的接口 不会被sentinel保护

```

@RequestMapping("/findAll")
public String findAll() throws InterruptedException {
    orderServiceImpl.common();
    return "findAll";
}

```

只有加了@SentinelResource的注解的资源才会被保护

```

@SentinelResource("order-skill")
@RequestMapping("/orderSave")
public String saveOrder(@RequestParam(value = "userId",required = false)String
                        @RequestParam(value = "productId",required = false)
                        log.info("userId:{}",userId);
                        log.info("productId:{}",productId);
                        return userId;
}

```

