

## 1. Sentinel介绍

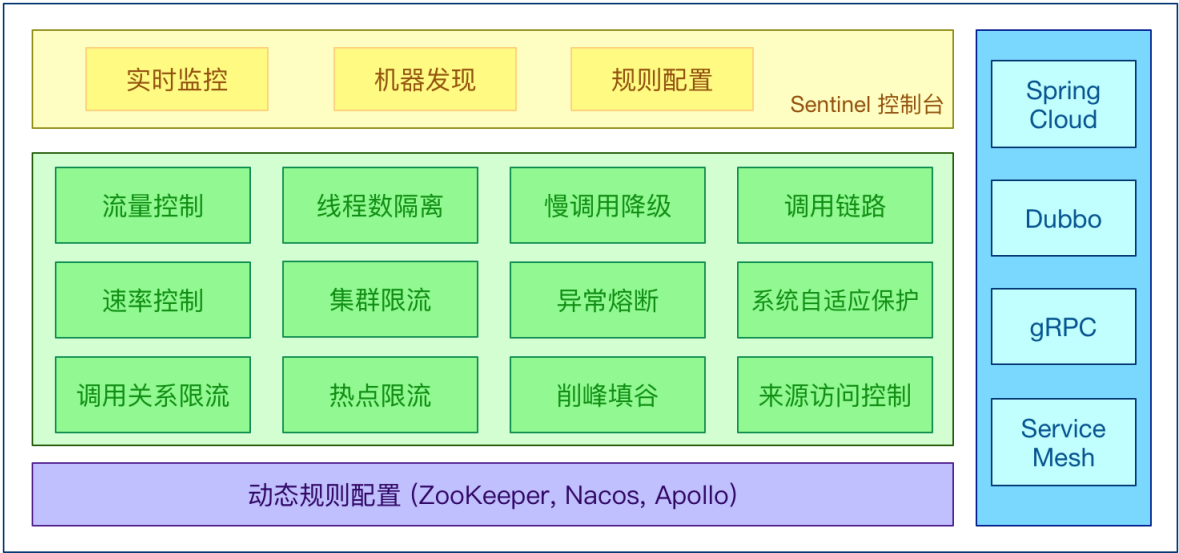
Sentinel是面向分布式、多语言异构化服务架构的流量治理组件，主要以流量为切入点，从流量控制、流量路由、熔断降级、系统自适应保护等多个维度来帮助用户保障微服务的稳定性。

- 1. 官网地址: <https://sentinelguard.io/zh-cn/>
- 2. 文档地址: <https://sentinelguard.io/zh-cn/docs/introduction.html>
- 3. 源码地址: <https://github.com/alibaba/Sentinel>

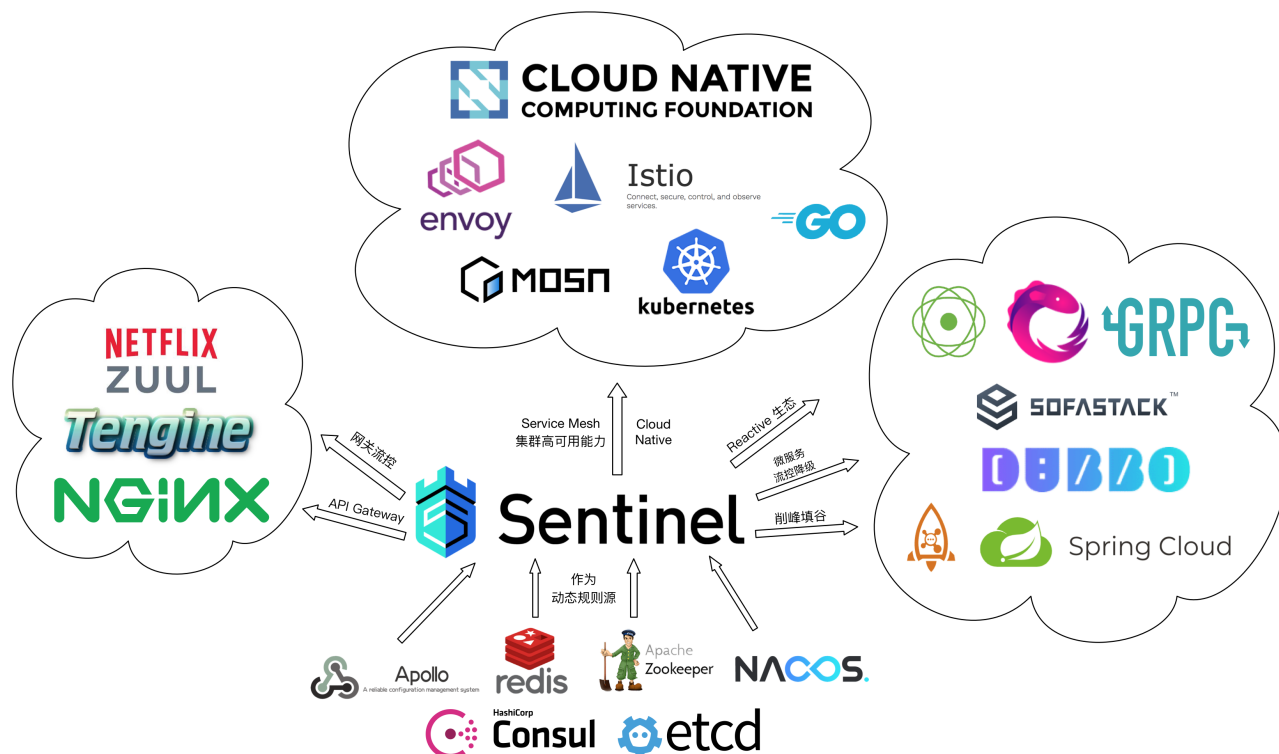
## 2. Sentinel特性

- **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 o/C++ 等多语言的原生实现。
- **完善的 SPI 扩展机制**：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。

### 2.1 Sentinel主要特性



## 3. Sentinel的开源生态



## 4. Sentinel的组成

- 核心库（Java 客户端）不依赖任何框架/库，能够运行于所有 Java 运行时环境，同时对 Dubbo / Spring Cloud 等框架也有较好的支持。
- 控制台（Dashboard）基于 Spring Boot 开发，打包后可以直接运行，不需要额外的 Tomcat 等应用容器。

## 5. Sentinel 基本概念

### • 5.1 资源

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。

只要通过 Sentinel API 定义的代码，就是资源，就能够被 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。

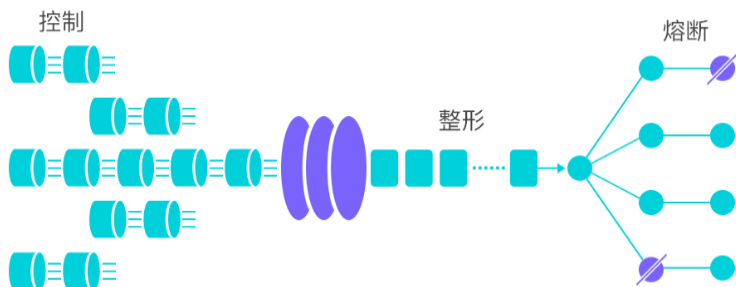
### • 5.2 规则

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。

## 6. Sentinel功能和设计理念

## • 6.1 流量控制

流量控制在网络传输中是一个常用的概念，它用于调整网络包的发送数据。然而，从系统稳定性角度考虑，在处理请求的速度上，也有非常多的讲究。任意时间到来的请求往往是随机不可控的，而系统的处理能力是有限的。我们需要根据系统的处理能力对流量进行控制。Sentinel 作为一个调配器，可以根据需要把随机的请求调整成合适的形状，如下图所示：



## • 6.2 熔断降级

除了流量控制以外，降低调用链路中的不稳定资源也是 Sentinel 的使命之一。由于调用关系的复杂性，如果调用链路中的某个资源出现了不稳定，最终会导致请求发生堆积。

当调用链路中某个资源出现不稳定，例如，表现为 timeout，异常比例升高的时候，则对这个资源的调用进行限制，并让请求快速失败，避免影响到其它的资源，最终产生雪崩的效果。

### 1. 通过并发线程数进行限制

和资源池隔离的方法不同，Sentinel 通过限制资源并发线程的数量，来减少不稳定资源对其它资源的影响。这样不但没有线程切换的损耗，也不需要您预先分配线程池的大小。当某个资源出现不稳定的情况下，例如响应时间变长，对资源的直接影响就是会造成线程数的逐步堆积。当线程数在特定资源上堆积到一定的数量之后，对该资源的新请求就会被拒绝。堆积的线程完成任务后才开始继续接收请求。

### 2. 通过响应时间对资源进行降级

除了对并发线程数进行控制以外，Sentinel 还可以通过响应时间来快速降级不稳定的资源。当依赖的资源出现响应时间过长后，所有对该资源的访问都会被直接拒绝，直到过了指定的时间窗口之后才重新恢复。

## • 6.3 系统负载保护

Sentinel 同时提供 系统维度的自适应保护能力。防止雪崩，是系统防护中重要的一环。当系统负载较高的时候，如果还持续让请求进入，可能会导致系统崩溃，无法响应。在集群环境下，网络负载均衡会把本应这台机器承载的流量转发到其它的机器上去。如果这个时候其它的机器也处在一个边缘状态的时候，这个增加的流量就会导致这台机器也崩溃，最后导致整个集群不可用。

针对这个情况，Sentinel 提供了对应的保护机制，让系统的入口流量和系统的负载达到一个平衡，保证系统在能力范围之内处理最多的请求。

## 7. Sentinel快速开始

### 1. 创建项目

## 2. 导入Maven依赖

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

## 3. 编写application.yml

```
server:
  port: xxxx
```

## 4. 编写启动类

```
package com.etoak;

import com.alibaba.csp.sentinel.slots.block.RuleConstant;
import com.alibaba.csp.sentinel.slots.block.flow.FlowRule;
import com.alibaba.csp.sentinel.slots.block.flow.FlowRuleManager;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.ArrayList;
import java.util.List;

@SpringBootApplication
public class SentinelApp {

    public static void main(String[] args) {
        // 初始化限流规则
        initFlowRules();
        SpringApplication.run(SentinelApp.class, args);
    }

    /**
     * 初始化限流规则
     */
    public static void initFlowRules() {
        List<FlowRule> rules = new ArrayList<>();
        // 限流规则
        FlowRule rule = new FlowRule();
        // 设置限流资源: "hello"
        rule.setResource("hello");
        // 设置阈值类型: QPS
        rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
        // 设置单机阈值: 1, 表示 1秒1个请求
        rule.setCount(1);
        rules.add(rule);
    }
}
```

```
        FlowRuleManager.loadRules(rules);
    }
}
```

## 5. 编写HelloController进行测试

```
@RestController
public class HelloController {

    @RequestMapping("hello")
    public String hello() {
        Entry entry = null;
        try {
            // 资源
            entry = SphU.entry("hello");
            return "success: " + LocalDateTime.now();
        } catch (Exception e) {
            return "限流: " + LocalDateTime.now();
        } finally {
            if (entry != null) {
                entry.exit();
            }
        }
    }
}
```

## 8. Sentinel控制台

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。

Sentinel 控制台是流量控制、熔断降级规则统一配置和管理的入口，它为用户提供了机器自发现、簇点链路自发现、监控、规则配置等功能。在 Sentinel 控制台上，我们可以配置规则并实时查看流量控制效果。

### • 8.1 功能

- **查看机器列表以及健康情况**：收集 Sentinel 客户端发送的心跳包，用于判断机器是否在线。
- **监控 (单机和集群聚合)**：通过 Sentinel 客户端暴露的监控 API，定期拉取并且聚合应用监控信息，最终可以实现秒级的实时监控。
- **规则管理和推送**：统一管理推送规则。
- **鉴权**：生产环境中鉴权非常重要。这里每个开发者需要根据自己的实际情况进行定制。

**注意**：Sentinel 控制台目前仅支持单机部署。Sentinel 控制台项目提供 Sentinel 功能全集示例，不作为开箱即用的生产环境控制台，不提供安全可靠保障。若希望在生产环境使用请根据 [文档](#) 自行进行定制和改造。

## • 8.2 启动控制台

1. **获取控制台**：<https://github.com/alibaba/Sentinel/releases>

2. 下载源码，自行编译

控制台源码地址：<https://github.com/alibaba/Sentinel/tree/master/sentinel-dashboard>

下载后编译：`mvn clean package`

3. 运行控制台

```
java -jar sentinel-dashboard-x.x.x.jar
```

4. 访问控制台

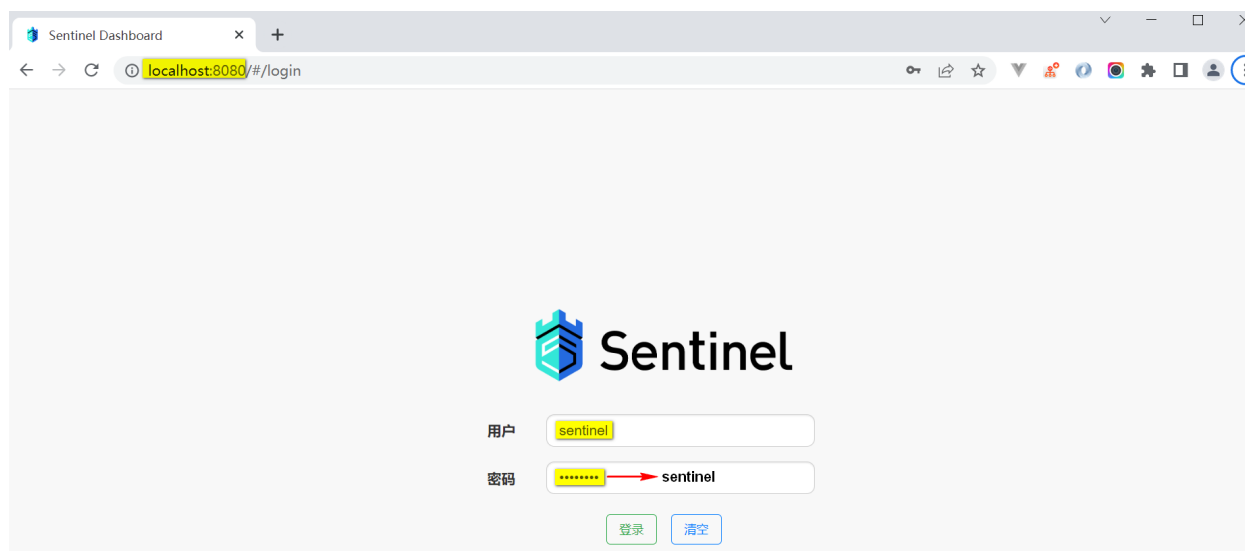
**默认端口号**：8080

可以通过 `java -jar -Dserver.port=9999 sentinel-dashboard-x.x.x.jar` 修改端口号

浏览器输入 <http://localhost:9999>

用户名：`sentinel`

密码：`sentinel`



## 9. 流量控制（限流）

### • 9.1 限流阈值类型（grade） - 流量控制的两种统计类型

1. 基于QPS/并发数限流

QPS（Queries Per Second）：每秒钟请求数

由 `FlowRule.grade` 字段来定义。其中，0 代表根据并发数量来限流，1 代表根据 QPS 来进行流量控制。

当请求该资源的QPS达到设定阈值时，进行限流

2. 基于线程数限流

当请求该资源的线程数达到阈值时，进行限流  
允许请求进入，但是如果没有空闲线程，则失败

● 9.2 流量控制概述

一条限流规则主要由下面几个因素组成，我们可以组合这些元素来实现不同的限流效果：

- **resource**：资源名，即限流规则的作用对象
- **grade**：限流阈值类型，QPS 或线程数
- **count**：限流阈值
- **strategy**：根据调用关系选择策略

● 9.3 限流策略（strategy） - 流控模式

1. 直接

新增流控规则

资源名

资源名

针对来源

default

阈值类型

☒ QPS ☐ 并发线程数

单机阈值

单机阈值

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

取消

2. 关联

当两个资源之间具有资源争抢或者依赖关系的时候，这两个资源便具有了关联。

比如对数据库同一个字段的读操作和写操作存在争抢，读的速度过高会影响写得速度，写的速度过高会影响读的速度。如果放任读写操作争抢资源，则争抢本身带来的开销会降低整体的吞吐量。

可使用关联限流来避免具有关联关系的资源之间过度的争抢，举例来说，**read\_db** 和 **write\_db** 这两个资源分别代表数据库读写，我们可以给 **read\_db** 设置限流规则来达到写优先的目的，这样当写库操作过于频繁时，读数据的请求会被限流。

**简单来说：**就是当与 **A资源 (read\_db)** 关联的 **B资源(write\_db)** 的请求达到阈值时，对 **A资源 (read\_db)** 进行流控；

新增流控规则

资源名

/flow/read

针对来源

default

对/flow/read进行限流

阈值类型

☒ QPS ☐ 并发线程数

单机阈值

5

是否集群

☐

流控模式

☐ 直接 ☒ 关联 ☐ 链路

关联资源

/flow/write

当/flow/write的QPS达到5次/秒时,

流控效果

☒ 快速失败 ☐ Warm Up ☐ 排队等待

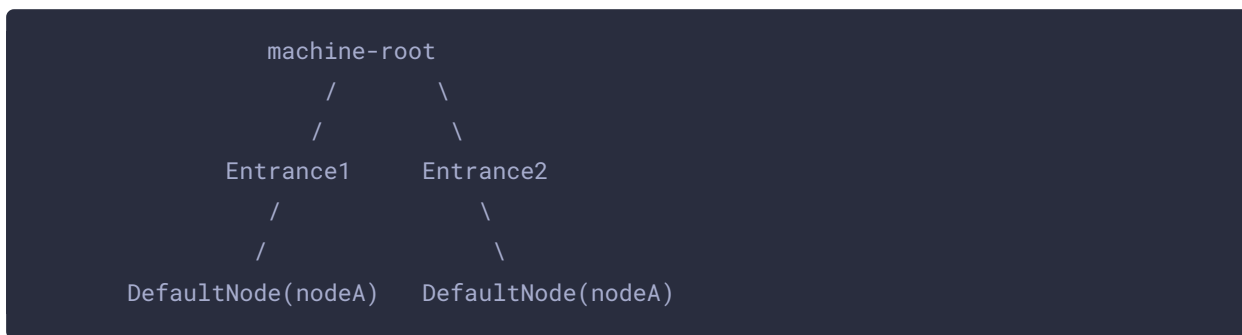
关闭高级选项

新增并继续添加

新增

取消

### 3. 链路



上图中来自入口 `Entrance1` 和 `Entrance2` 的请求都调用到了资源 `NodeA`，Sentinel 允许只根据某个入口的统计信息对资源限流。

比如我们可以设置 `FlowRule.strategy` 为 `RuleConstant.CHAIN`，同时设置 `FlowRule.ref_identity` 为 `Entrance1` 来表示只有从入口 `Entrance1` 的调用才会记录到 `NodeA` 的限流统计当中，而对来自 `Entrance2` 的调用漠不关心。

从 `Entrance1` 请求 `nodeA` 的 QPS 达到 **10次/s** 时，`nodeA` 对来自 `Entrance1` 的请求限流（`nodeA` 限制 `Entrance1` 的请求）



新增流控规则

资源名

nodeA

针对来源

default

阈值类型

☒ QPS
 ☐ 并发线程数

单机阈值

10

是否集群

☐

流控模式

☐ 直接
 ☐ 关联
 ☒ 链路

入口资源

Entrance1

流控效果

☒ 快速失败
 ☐ Warm Up
 ☐ 排队等待

关闭高级选项

新增

取消

machine-root

入口1

Entrance1

入口2

Entrance2

DefaultNode(nodeA)

DefaultNode(nodeA)

4. 在yml在增加 `web-context-unify: false` 配置

```

spring:
  cloud:
    sentinel:
      transport:
        # sentinel控制台
        dashboard: localhost:8000
        # 所有资源不挂在sentinel_spring_web_context资源下
        web-context-unify: false
  
```

5. 配置 `web-context-unify: false` 前后对比

▼ sentinel_spring_web_context		0
/flow/write	配置web-context-unify之前	0
/flow/read		0

资源名	通过QP S	拒绝QP S	并发 数	平均 T
▶ /flow/write	0	0	0	0
▶ /flow/read	0	0	0	0

配置web-context-unify=false 之后

## 6. 代码

```
@RestController
@RequestMapping("/flow")
public class FlowController {

    /** ***** 流控模式：链路模式 start ***** */

    @Autowired
    FlowService flowService;

    @RequestMapping("/entrance1")
    public String entrance1() {
        // 调用nodeA资源
        return flowService.nodeA("entrance1");
    }

    @RequestMapping("/entrance2")
    public String entrance2() {
        // 调用nodeA资源
        return flowService.nodeA("entrance2");
    }

    /** ***** 流控模式：链路模式 end ***** */
}
```

```
@Service
public class FlowService {

    // entrance1和entrance2共同调用的资源nodeA
    @SentinelResource(value = "nodeA", blockHandler = "nodeBlockHandler")
    public String nodeA(String from) {
        System.out.println("from==" + from);
        return from;
    }

    public String nodeBlockHandler(String from, BlockException e) {
        return from + " - 被限流了! ";
    }

}
```

## 7. 规则配置

### 新增流控规则

资源名

nodeA

针对来源

default

阈值类型

☒ QPS ☐ 并发线程数

单机阈值

2

是否集群

☐

流控模式

☐ 直接 ☐ 关联 ☒ 链路

入口资源

/flow/entrance1

nodeA资源对来自/flow/entrance1的请求进行限流  
限流阈值: 2次/2

流控效果

☒ 快速失败 ☐ Warm Up ☐ 排队等待

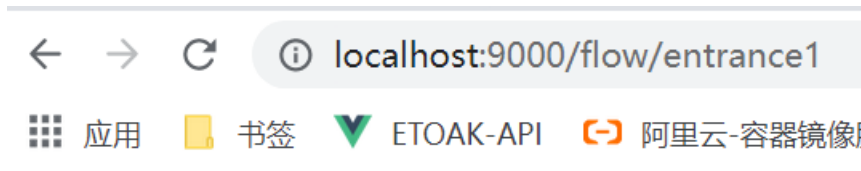
关闭高级选项

新增并继续添加

新增

取消

## 8. 测试结果



entrance1 - 被限流了!

## 9.4 流控效果

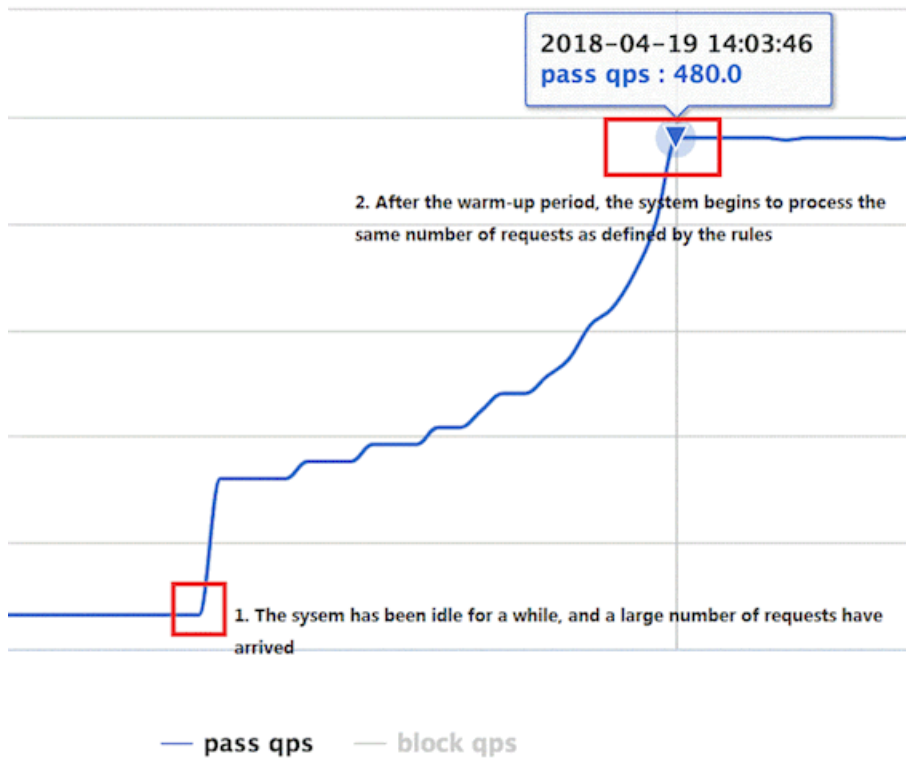
### 1. 直接拒绝 - 快速失败 (RuleConstant.CONTROL\_BEHAVIOR\_DEFAULT)

该方式是默认的流量控制方式，当QPS超过任意规则的阈值后，新的请求就会被立即拒绝，拒绝方式为抛出 `FlowException`。这种方式适用于对系统处理能力确切已知的情况下，比如通过压测确定了系统的准确水位时。

### 2. Warm Up - 冷启动 (RuleConstant.CONTROL\_BEHAVIOR\_WARM\_UP) 方式

该方式主要用于系统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，避免冷系统被压垮的情况。

sentinel客户端的默认冷加载因子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。



## 1、Sentinel控制台设置预热

新增流控规则

资源名: /user/test1

针对来源: default

阈值类型: ☒ QPS ☐ 并发线程数 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

预热时长: 20 → 20秒

冷加载因子: coldFactor为3  
对于/user/test1资源，一开始的QPS阈值为3 (10/3)，经过20秒后，QPS阈值达到10

新增并继续添加 新增 取消

## 2、流控效果



### 3. 排队等待 - 匀速器 (RuleConstant.CONTROL\_BEHAVIOR\_RATE\_LIMITER) 方式

排队等待方式不会拒绝请求，而是严格控制请求通过的间隔时间，也就是让请求匀速通过；

#### 1、在Sentinel控制台设置排队等待

编辑流控规则

资源名

/user/test1

针对来源

default

阈值类型

QPS

并发线程数

单机阈值

5

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

5000 单位ms

关闭高级选项

保存

取消

JMeter测试计划

线程属性

线程数:

100

Ramp-Up时间 (秒):

10

循环次数

☐ 永远

1

100个线程

10秒

10个/秒

1秒通过5个，相当于200ms处理一个

超时时间5000ms(5秒)

也就是排队最多25个(每个处理时间200ms)

之后再进入请求就扔掉

按照这个配置

通过75请求

拒绝25个

Status

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

#### 2、流控效果



## 10. 熔断降级

除了流量控制以外，对调用链路中不稳定的资源进行熔断降级也是保障高可用的重要措施之一。

一个服务常常会调用别的模块，可能是另外的一个远程服务、数据库，或者第三方 API 等。

**例如：支付的时候，可能需要远程调用银联提供的 API；查询某个商品的价格，可能需要进行数据库查询。**

然而，被依赖服务的稳定性是不能保证的。

如果依赖的服务出现了不稳定的情况，请求的响应时间变长，那么调用服务的方法的响应时间也会变长，线程会产生堆积，最终可能耗尽业务自身的线程池，服务本身也变得不可用。

现代微服务架构都是分布式的，由非常多的服务组成。不同服务之间相互调用，组成复杂的调用链路，以上的问题在链路调用中会产生放大的效果。

复杂链路上的某一环不稳定，就可能会层层级联，最终导致整个链路都不可用。因此我们需要对不稳定的弱依赖服务调用进行熔断降级，暂时切断不稳定调用，避免局部不稳定因素导致整体的雪崩。

## • 10.1 熔断策略

### • 慢调用比例 ( SLOW\_REQUEST\_RATIO )

选择以慢调用比例作为阈值，需要设置允许的慢调用 RT（即最大的响应时间），请求的响应时间大于该值则统计为慢调用。当单位统计时长（ statIntervalMs ）内请求数目大于设置的最小请求数目，并且慢调用的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求响应时间小于设置的慢调用 RT 则结束熔断，若大于设置的慢调用 RT 则会再次被熔断。

#### 新增熔断规则

The screenshot shows the 'Add Circuit Breaker Rule' form for the 'Slow Request Ratio' strategy. The resource name is '/user/test4'. The strategy is 'Slow Request Ratio'. The maximum RT is 200ms, the ratio threshold is 0.5, the熔断时长 (circuit break duration) is 20s, and the统计时长 (stat interval) is 1000ms. A red box highlights the '最大响应时间200ms' and '比例阈值 0.5' fields. A yellow box contains the following code snippet:

```
@RequestMapping("/test4")
public String test4() throws Exception {
    // 放慢响应时间
    Thread.sleep(250);
    return "success!";
}
```

A red box at the bottom explains the logic: '1秒内至少发送5个请求，其中请求最大响应超过200ms的比例达到50%时熔断请求' (At least 5 requests must be sent within 1 second, and when the proportion of requests with maximum response time exceeding 200ms reaches 50%, the request is熔断).

### • 异常比例 ( ERROR\_RATIO )

当单位统计时长（ statIntervalMs ）内请求数目大于设置的最小请求数目，并且异常的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求成功完成（没有错误）则结束熔断，否则会再次被熔断。异常比率的阈值范围是 [0.0, 1.0]，代表 0% - 100%。

#### 编辑熔断规则

The screenshot shows the 'Edit Circuit Breaker Rule' form for the 'Error Ratio' strategy. The resource name is '/user/test3'. The strategy is 'Error Ratio'. The比例阈值 (ratio threshold) is 0.2, the熔断时长 (circuit break duration) is 20s, and the统计时长 (stat interval) is 2000ms. A red box highlights the '异常比例' label and the '20' value in the '熔断时长' field. A yellow box contains the following text:

两秒内统计接口异常比例达到20%时熔断，熔断20秒  
但是，两秒内请求数必须大于等于5个  
如果两秒内请求数不到5个，即使100%错误也不会熔断

### • 异常数 ( ERROR\_COUNT )

当单位统计时长内的异常数目超过阈值之后会自动进行熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求成功完成（没有错误）则结束熔断，否则会再次被熔断。

### 新增熔断规则

资源名	<input type="text" value="/user/test3"/>		
熔断策略	<input type="radio"/> 慢调用比例 <input type="radio"/> 异常比例 <input checked="" type="radio"/> 异常数		
异常数	<input type="text" value="3"/>		
熔断时长	<input type="text" value="20"/> s	最小请求数	<input type="text" value="5"/>
统计时长	<input type="text" value="1000"/> ms		

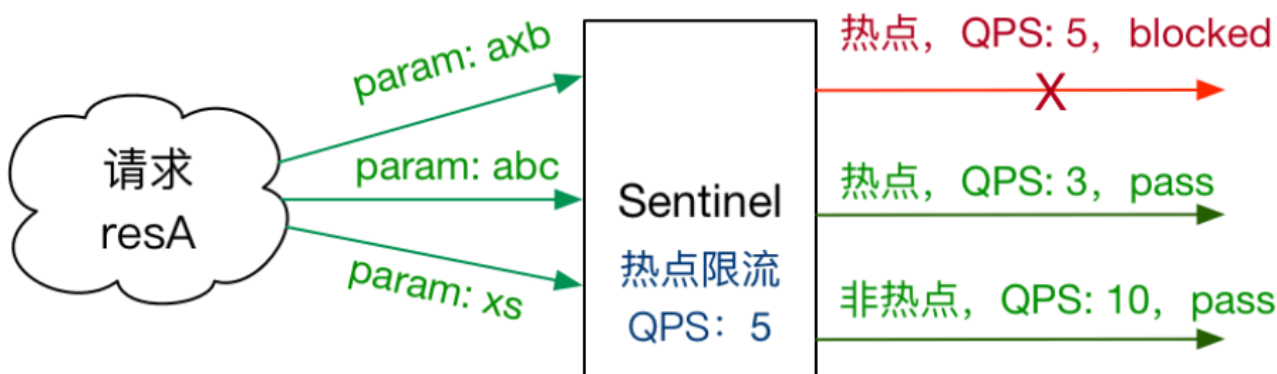
1秒内统计至少5次请求，如果有3次出错就熔断请求，熔断20秒

## 11. 热点参数限流

何为热点？热点即经常访问的数据。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。



### 1. 代码编写

```

@RestController
@RequestMapping("/product")
public class HotParamFlowController {

    @SentinelResource(value = "order")
    @RequestMapping("/{id}")
    public String getProduct(@PathVariable long id) {
        return "product id : " + id;
    }
}

```

## 2. 控制台配置

新增热点规则

资源名

限流模式 QPS 模式

参数索引  **参数下标, 从0开始**

单机阈值  统计窗口时长  秒

是否集群 ☐ **QPS 2次/s**

参数例外项

参数类型

参数值  限流阈值  **+ 添加**

参数值	参数类型	限流阈值	操作
<input type="text" value="2"/>	int	<input type="text" value="5"/>	<input type="button" value="删除"/>

**id=2的参数, 不使用2次/s的QPS  
它采用5次/s的QPS**

## 12. 自定义限流或降级返回结果

1. 对于使用 `@SentinelResource` 注解定义的资源, 使用它的 `blockHandler` 属性即可
2. 对于 `Spring MVC` 请求URL 的资源可以实现 `BlockExceptionHandler` 接口

```

@Service
public class MyBlockHandler implements BlockExceptionHandler {

    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
        BlockException e) throws Exception {

```



```

String uri = request.getRequestURI();
response.setContentType("application/json;charset=utf-8");
ResultVO resultVO = null;

if (e instanceof FlowException) {
    resultVO = ResultVO.failed(uri + "超出最大请求数! ");
} else if (e instanceof DegradeException) {
    resultVO = ResultVO.failed(uri + "降级! ");
} else if (e instanceof ParamFlowException) {
    resultVO = ResultVO.failed(uri + "热点参数限流了! ");
} else {
    resultVO = ResultVO.failed("系统异常! ");
}

String json = JSONUtil.toJsonStr(resultVO);
PrintWriter writer = response.getWriter();
writer.print(json);
writer.flush();
writer.close();
}
}

```

## 13. 在order-service开启Feign对Sentinel的支持

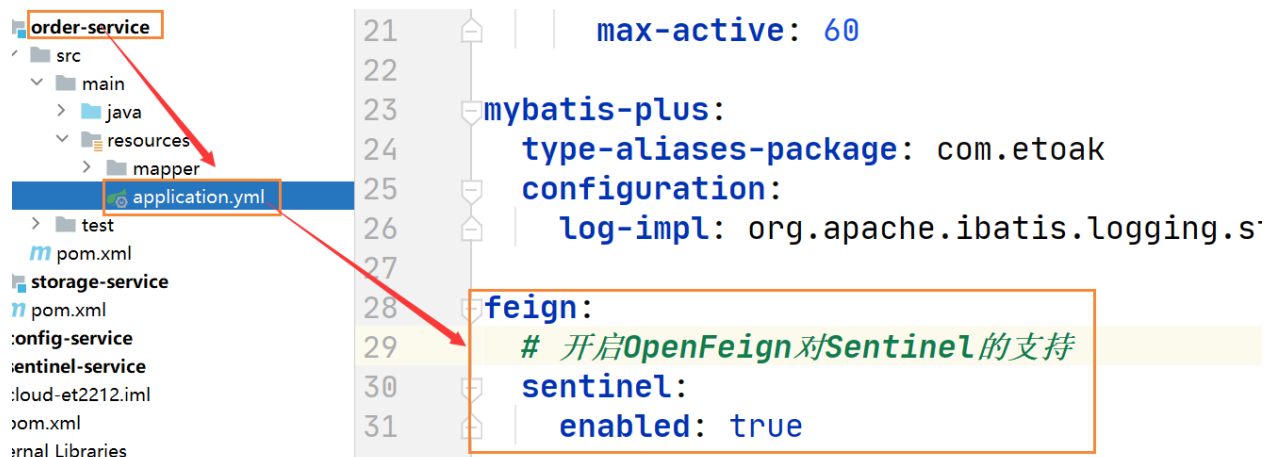
1. 在order-service项目中增加如下Maven依赖：spring-cloud-starter-sentinel

```

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

```

2. 在application.yml中配置sentinel控制台地址，并开启feign对sentinel支持



```

21 | | | max-active: 60
22 |
23 | mybatis-plus:
24 |   type-aliases-package: com.etoak
25 |   configuration:
26 |     log-impl: org.apache.ibatis.logging.s
27 |
28 | feign:
29 |   # 开启OpenFeign对Sentinel的支持
30 |   sentinel:
31 |     enabled: true

```

3. 在Feign接口StorageServiceApi上的@FeignClient注解添加 **fallback属性**（属性值是一个实现了Feign接口的实现类）

```
/**
 * 声明式HTTP客户端
 * name或value属性的值：要调用的服务的服务名称(spring.application.name的值)
 *
 * fallback属性：调用远程服务失败之后执行StorageServiceApiFallback对应的方法
 */
@FeignClient(name = "storage-service", fallback = StorageServiceApiFallback.class)
public interface StorageServiceApi {

    /**
     * 扣减库存接口
     */
    @PostMapping("/storage/deduct")
    ResultVO deduct(@RequestBody Storage storage);

    /**
     * 扣减库存接口
     * 注解@SpringQueryMap：允许客户端使用Java Bean的方式调用远程的http+form的接口
     */
    @PostMapping("/storage/deduct2")
    ResultVO deduct2(@SpringQueryMap Storage storage);

    @PostMapping("/storage/deduct2")
    ResultVO deduct3(@RequestParam("productCode") String productCode,
                     @RequestParam("count") int count);
}
```

```
@Service
public class StorageServiceApiFallback implements StorageServiceApi {

    @Override
    public ResultVO deduct(Storage storage) {
        return ResultVO.failed("库存不足");
    }

    @Override
    public ResultVO deduct2(Storage storage) {
        return ResultVO.failed("库存不足");
    }

    @Override
    public ResultVO deduct3(String productCode, int count) {
        return ResultVO.failed("库存不足");
    }
}
```

## 14. 持久化限流规则

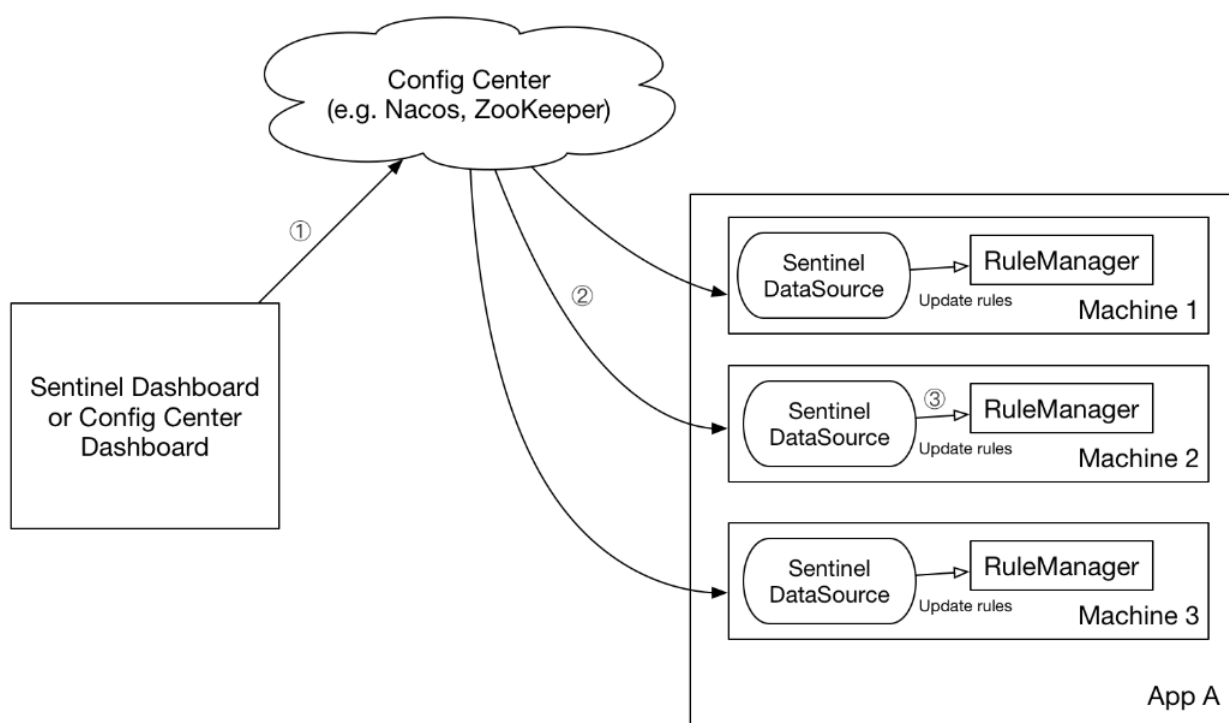
Sentinel 的理念是 **开发者只需要关注资源的定义**，当资源定义成功后可以动态增加各种流控降级规则。

Sentinel 提供两种方式修改规则：

- 通过 API 直接修改 ( `loadRules` )
- 通过 `DataSource` 适配不同数据源修改

手动修改规则（硬编码方式）一般仅用于测试和演示，生产上一般通过动态规则源的方式来动态管理规则。

我们推荐通过控制台设置规则后将规则推送到统一的规则中心，客户端实现 `ReadableDataSource` 接口端监听规则中心实时获取变更，流程如下：



`DataSource` 扩展常见的实现方式有：

- **拉模式**：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，甚至是 VCS 等。这样做的方式是简单，缺点是无法及时获取变更；
- **推模式**：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。

Sentinel 目前支持以下数据源扩展：

- Pull-based: 动态文件数据源、`Consul`、`Eureka`
- Push-based: `ZooKeeper`、`Redis`、`Nacos`、`Apollo`、`etcd`

## • 14.1 使用 Nacos 配置规则实施步骤

Nacos是阿里中间件团队开源的服务发现和动态配置中心。Sentinel 针对 Nacos 作了适配，底层可以采用 Nacos 作为规则配置数据源。

### 1. 添加Maven依赖

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

### 2. 在yaml中配置nacos持久化

```
server:
  port: 8000

spring:
  application:
    name: sentinel-nacos-service

  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
        namespace: et2301

  sentinel:
    transport:
      dashboard: localhost:9999
      eager: true
      web-context-unify: false

  datasource:
    et2301:
      nacos:
        server-addr: ${spring.cloud.nacos.discovery.server-addr}
        namespace: ${spring.cloud.nacos.discovery.namespace}
        # 配置集ID名称
        data-id: sentinel-flow
        # data id类型
        data-type: json
        # 规则类型
        rule-type: flow
```

### 3. 在nacos控制增加配置

\* Data ID:

\* Group:

[更多高级选项](#)

描述:

Beta发布: ☐ 默认不要勾选。

配置格式: ☐ TEXT ☒ JSON ☐ XML ☐ YAML ☐ HTML ☐ Properties

配置内容🔗:

```
1  [
2    {
3      "resource": "/hello",
4      "limitApp": "default",
5      "grade": 1,
6      "count": 2,
7      "strategy": 0,
8      "controlBehavior": 0
9    }
10 ]
```

```
# FlowRule类的属性
[
  {
    "resource": "/hello", // 资源名
    "limitApp": "default", // 针对来源
    "grade": 1, // 阈值类型 1: QPS; 0: 并发线程数
    "count": 2, // 单机阈值
    "strategy": 0, // 流控模式 0:直接, 1:关联, 2:链路
    "controlBehavior": 0 // 流控效果 0: 快速失败, 1: warm up, 2:排队等待
  }
]
```