微服务保护

Sentinel

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

目录

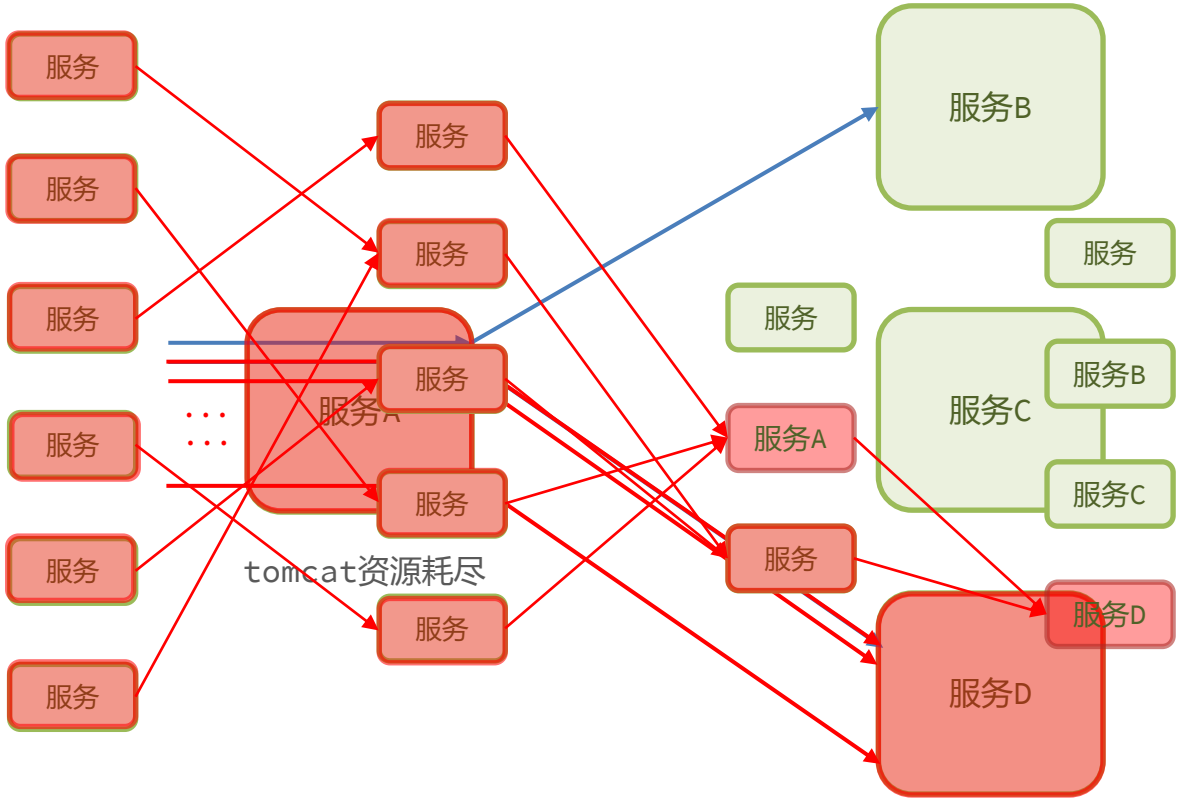
Contents

初识Sentinel

- 雪崩问题及解决方案
- 服务保护技术对比
- Sentinel介绍和安装
- 微服务整合Sentinel

雪崩问题

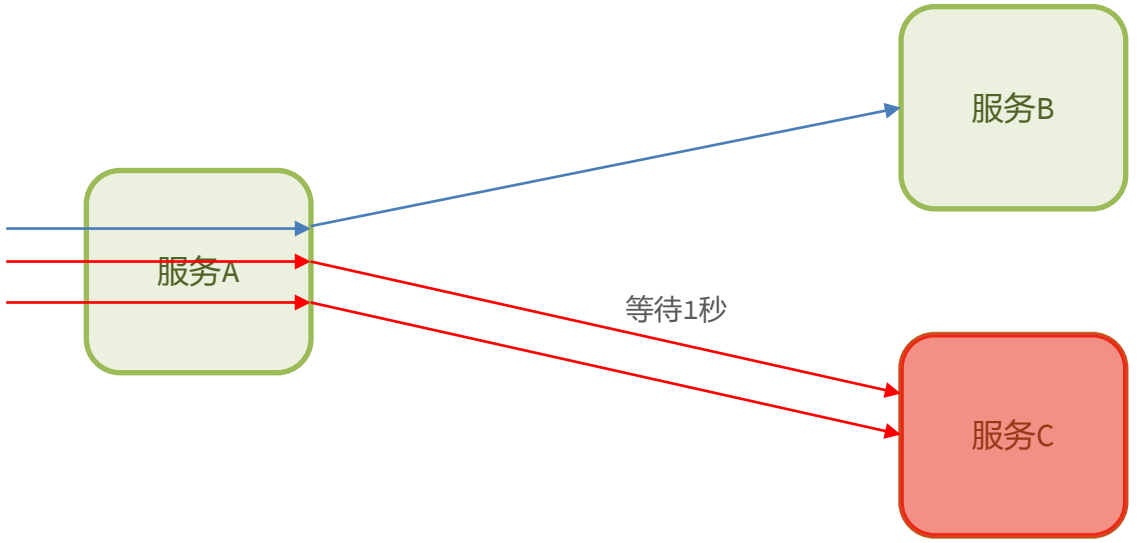
微服务调用链路中的某个服务故障，引起整个链路中的所有微服务都不可用，这就是雪崩。



雪崩问题

解决雪崩问题的常见方式有四种：

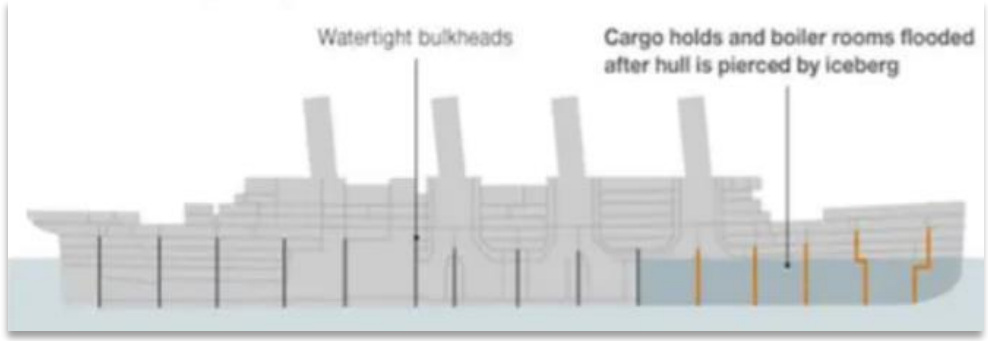
- 超时处理：设定超时时间，请求超过一定时间没有响应就返回错误信息，不会无休止等待

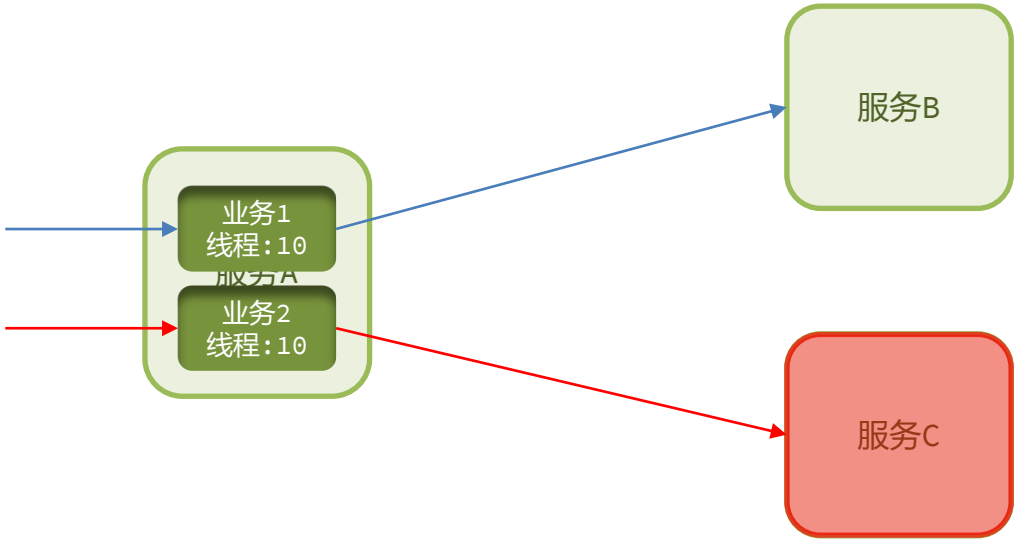


雪崩问题

解决雪崩问题的常见方式有四种：

- 超时处理：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，不会无休止等待
- 舱壁模式：限定每个业务能使用的线程数，避免耗尽整个tomcat的资源，因此也叫线程隔离。

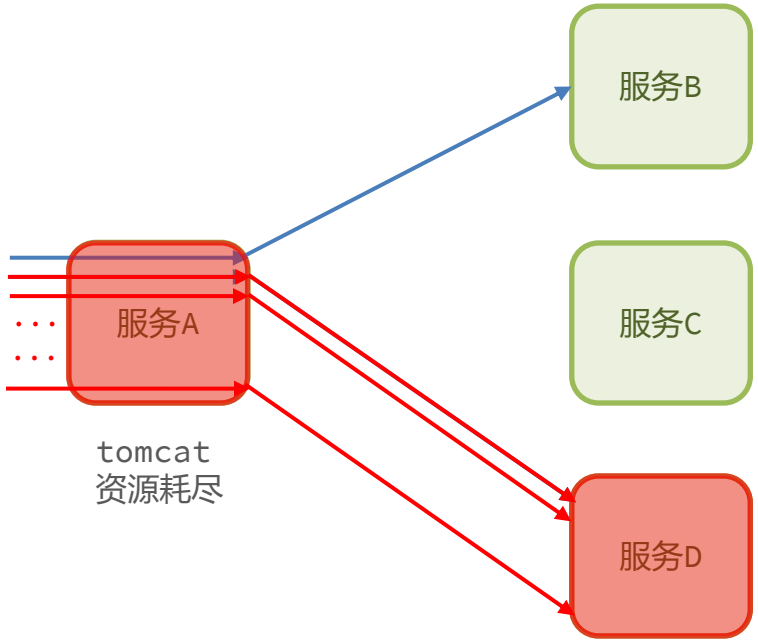


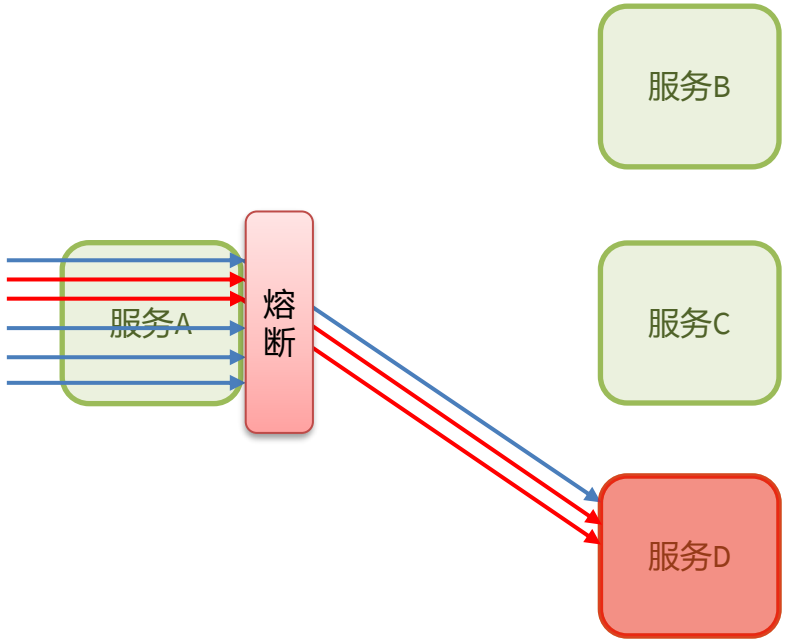


雪崩问题

解决雪崩问题的常见方式有四种：

- 超时处理：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，不会无休止等待
- 舱壁模式：限定每个业务能使用的线程数，避免耗尽整个tomcat的资源，因此也叫线程隔离。
- 熔断降级：由**断路器**统计业务执行的异常比例，如果超出阈值则会**熔断**该业务，拦截访问该业务的一切请求。

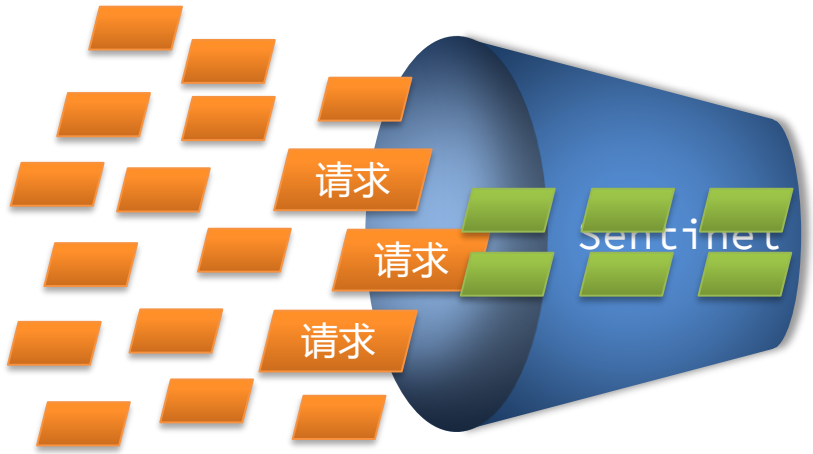


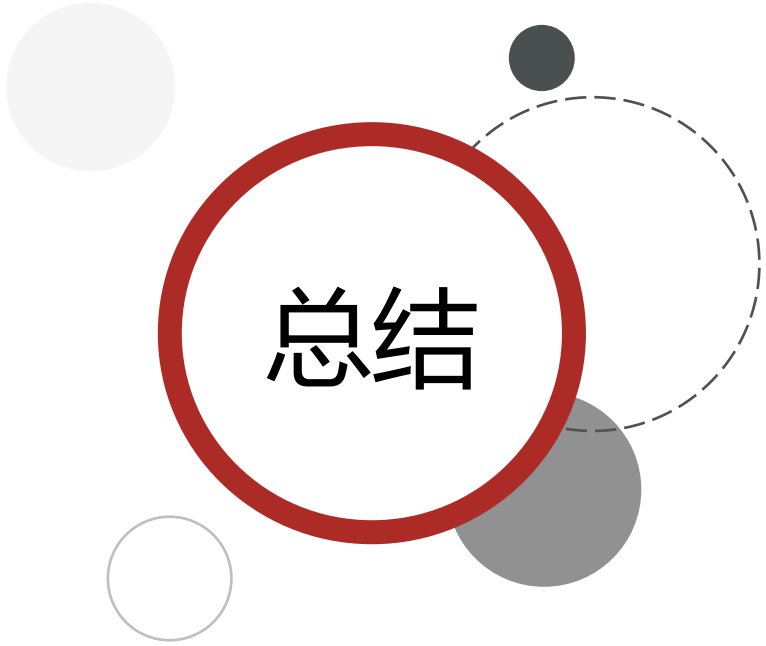


雪崩问题

解决雪崩问题的常见方式有四种：

- 超时处理：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，不会无休止等待
- 舱壁模式：限定每个业务能使用的线程数，避免耗尽整个tomcat的资源，因此也叫线程隔离。
- 熔断降级：由**断路器**统计业务执行的异常比例，如果超出阈值则会**熔断**该业务，拦截访问该业务的一切请求。
- 流量控制：限制业务访问的QPS，避免服务因流量的突增而故障。





什么是雪崩问题？

- 微服务之间相互调用，因为调用链中的一个服务故障，引起整个链路都无法访问的情况。

如何避免因瞬间高并发流量而导致服务故障？

- 流量控制

如何避免因服务故障引起的雪崩问题？

- 超时处理
- 线程隔离
- 降级熔断

服务保护技术对比

	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	支持慢启动、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

认识Sentinel

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。

安装Sentinel控制台

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。课前资料提供了下载好的jar包


```
java -jar sentinel-dashboard-1.8.1.jar
```

2、然后访问：localhost:8080，即可看到控制台页面，默认的账户和密码都是sentinel

安装Sentinel控制台

如果要修改Sentinel的默认端口、账户、密码，可以通过下列配置：

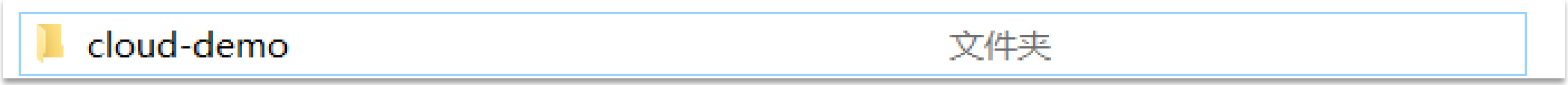
配置项	默认值	说明
server.port	8080	服务端口
sentinel.dashboard.auth.username	sentinel	默认用户名
sentinel.dashboard.auth.password	sentinel	默认密码

举例说明

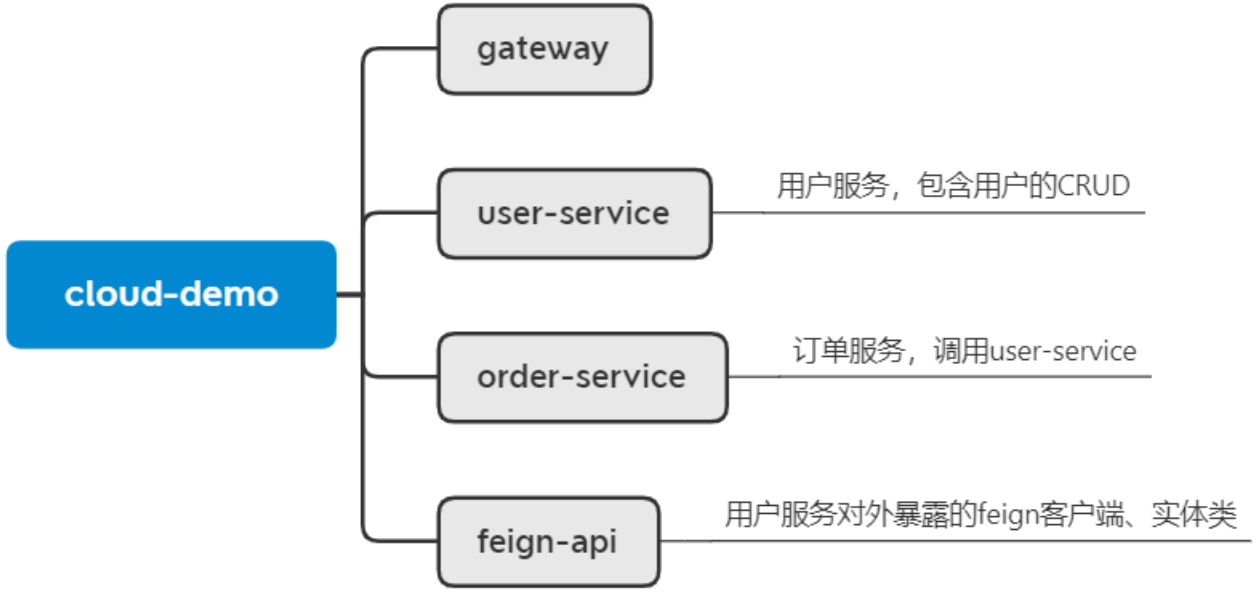
```
java -jar sentinel-dashboard-1.8.1.jar -Dserver.port=8090
```

引入cloud-demo

要使用Sentinel肯定要结合微服务，这里我们使用SpringCloud实用篇中的cloud-demo工程。没有的小伙伴可以在课前资料中找到：



项目结构如下：



微服务整合Sentinel

我们在order-service中整合Sentinel，并且连接Sentinel的控制台，步骤如下：

1. 引入sentinel依赖：

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

2. 配置控制台地址：

```
spring:
  cloud:
    sentinel:
      transport:
        dashboard: localhost:8080
```

3. 访问微服务的任意端点，触发sentinel监控

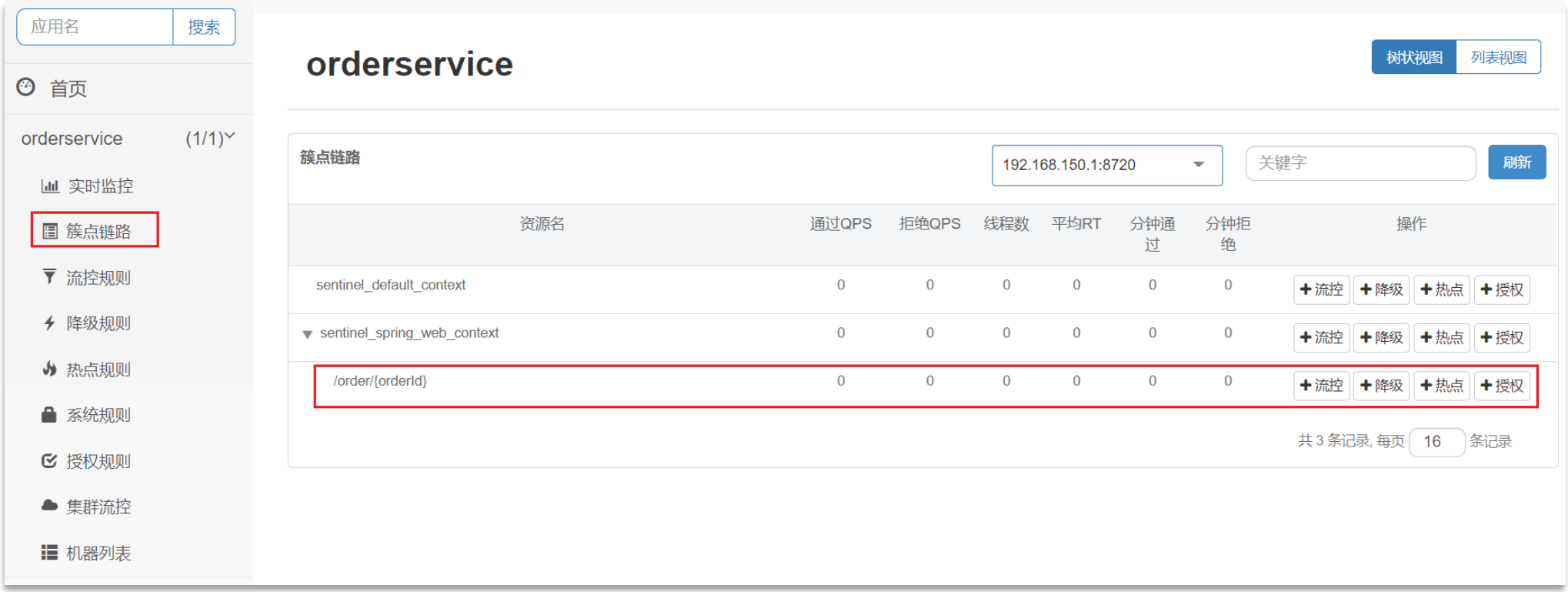
限流规则

- 快速入门
- 流控模式
- 流控效果
- 热点参数限流

簇点链路

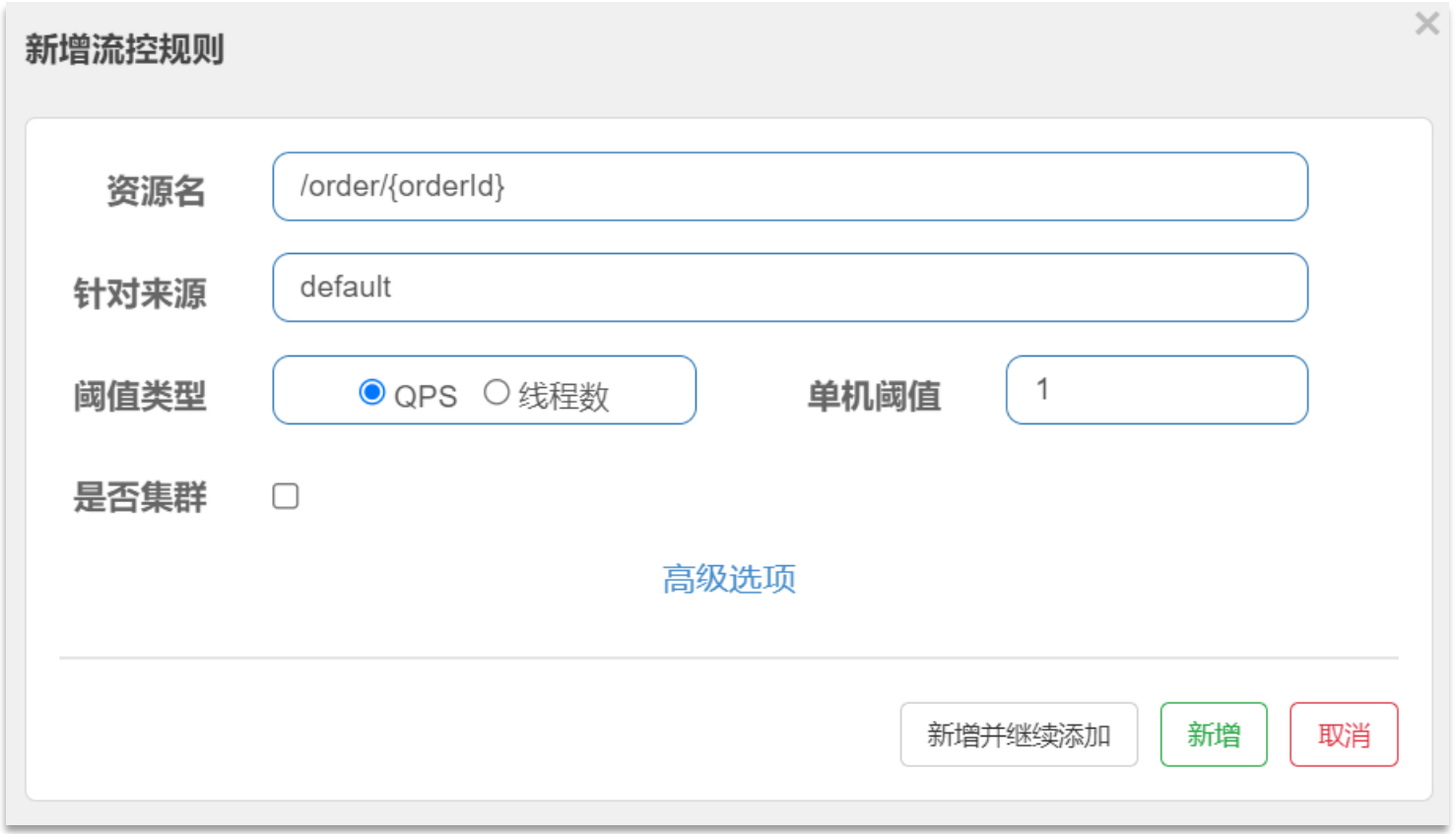
簇点链路：就是项目内的调用链路，链路中被监控的每个接口就是一个资源。默认情况下sentinel会监控SpringMVC的每一个端点（Endpoint），因此SpringMVC的每一个端点（Endpoint）就是调用链路中的一个资源。

流控、熔断等都是针对簇点链路中的资源来设置的，因此我们可以点击对应资源后面的按钮来设置规则：



快速入门

点击资源/order/{orderId}后面的流控按钮，就可以弹出表单。表单中可以添加流控规则，如下图所示：



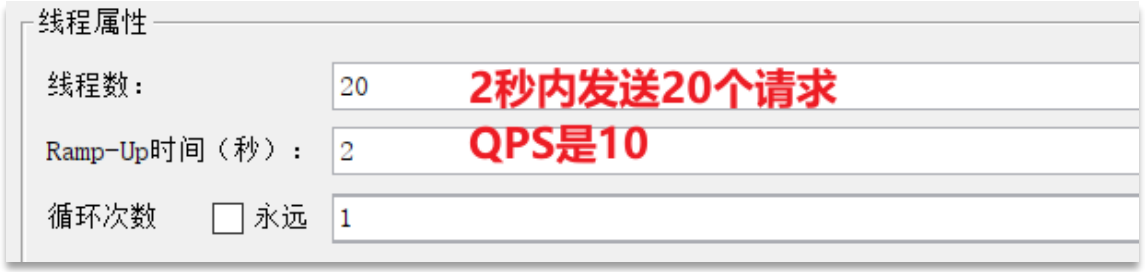
其含义是限制 /order/{orderId}这个资源的单机QPS为1，即每秒只允许1次请求，超出的请求会被拦截并报错。

案例 流控规则入门案例

需求：给 /order/{orderId}这个资源设置流控规则，QPS不能超过 5。然后利用jemeter测试。

1. 设置流控规则：

2. jemeter测试：



流控模式

在添加限流规则时，点击高级选项，可以选择三种流控模式：

- 直接：统计当前资源的请求，触发阈值时对当前资源直接限流，也是默认的模式
- 关联：统计与当前资源相关的另一个资源，触发阈值时，对当前资源限流
- 链路：统计从指定链路访问到本资源的请求，触发阈值时，对指定链路限流

流控模式-关联

- 关联模式：统计与当前资源相关的另一个资源，触发阈值时，对当前资源限流
- 使用场景：比如用户支付时需要修改订单状态，同时用户要查询订单。查询和修改操作会争抢数据库锁，产生竞争。业务需求是有限支付和更新订单的业务，因此当修改订单业务触发阈值时，需要对查询订单业务限流。

当/write资源访问量触发阈值时，就会对/read资源限流，避免影响/write资源。

案例 流控模式-关联

需求：

- 在OrderController新建两个端点：/order/query和/order/update，无需实现业务
- 配置流控规则，当/order/ update资源被访问的QPS超过5时，对/order/query请求限流

小结

满足下面条件可以使用关联模式：

- 两个有竞争关系的资源
- 一个优先级较高，一个优先级较低

流控模式-链路

链路模式：只针对从指定链路访问到本资源的请求做统计，判断是否超过阈值。

例如有两条请求链路：

- /test1 → /common
- /test2 → /common

如果只希望统计从/test2进入到/common的请求，则可以这样配置：

案例

流控模式-链路

需求：有查询订单和创建订单业务，两者都需要查询商品。针对从查询订单进入到查询商品的请求统计，并设置限流。

步骤：

1. 在OrderService中添加一个queryGoods方法，不用实现业务
2. 在OrderController中，改造/order/query端点，调用OrderService中的queryGoods方法
3. 在OrderController中添加一个/order/save的端点，调用OrderService的queryGoods方法
4. 给queryGoods设置限流规则，从/order/query进入queryGoods的方法限制QPS必须小于2

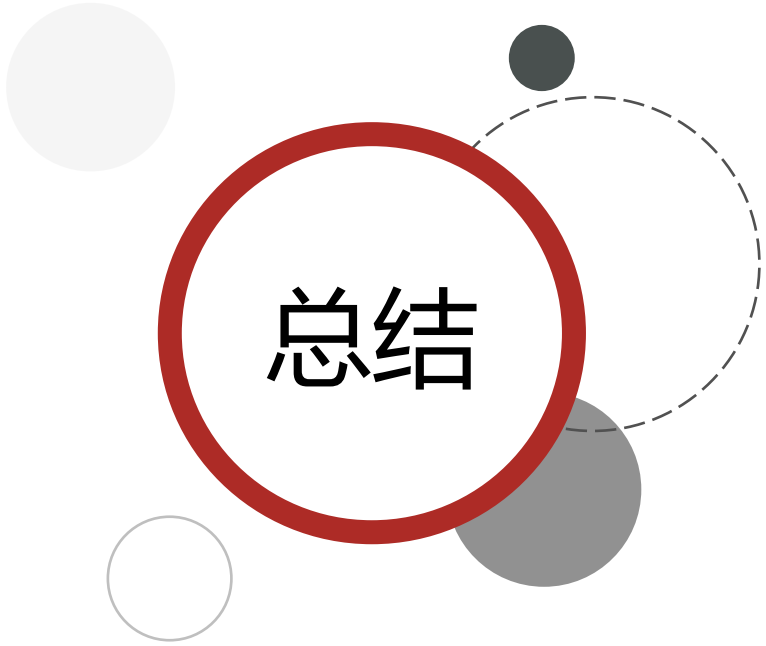
流控模式-链路

- Sentinel默认只标记Controller中的方法为资源，如果要标记其它方法，需要利用@SentinelResource注解，示例：

```
@SentinelResource("goods")
public void queryGoods() {
    System.err.println("查询商品");
}
```

- Sentinel默认会将Controller方法做context整合，导致链路模式的流控失效，需要修改application.yml，添加配置：

```
spring:
  cloud:
    sentinel:
      web-context-unify: false # 关闭context整合
```



流控模式有哪些?

- 直接：对当前资源限流
- 关联：高优先级资源触发阈值，对低优先级资源限流。
- 链路：阈值统计时，只统计从指定资源进入当前资源的请求，是对请求来源的限流

流控效果

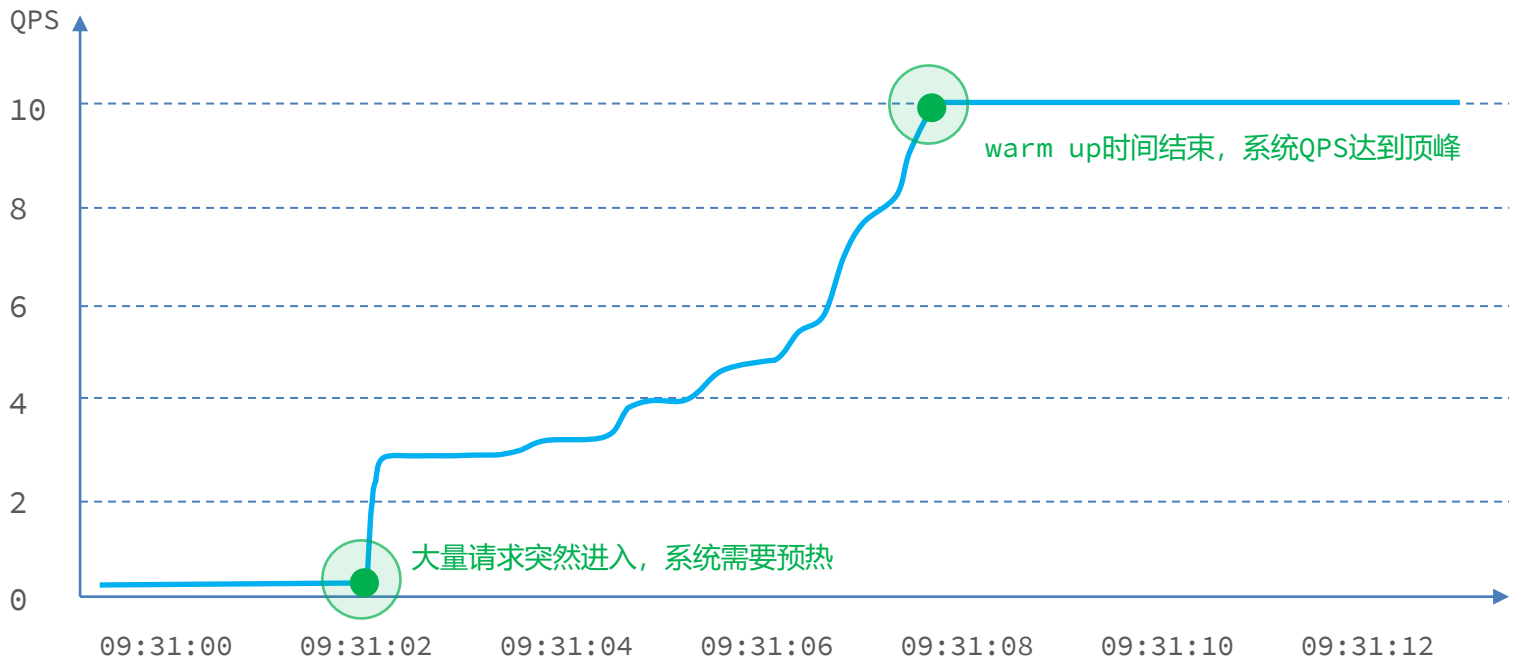
流控效果是指请求达到流控阈值时应该采取的措施，包括三种：

- 快速失败：达到阈值后，新的请求会被立即拒绝并抛出FlowException异常。是默认的处理方式。
- warm up：预热模式，对超出阈值的请求同样是拒绝并抛出异常。但这种模式阈值会动态变化，从一个较小值逐渐增加到最大阈值。
- 排队等待：让所有的请求按照先后次序排队执行，两个请求的间隔不能小于指定时长

流控效果-warm up

warm up也叫预热模式，是应对服务冷启动的一种方案。请求阈值初始值是 threshold / coldFactor，持续指定时长后，逐渐提高到threshold值。而coldFactor的默认值是3.

例如，我设置QPS的threshold为10，预热时间为5秒，那么初始阈值就是 10 / 3 ， 也就是3，然后在5秒后逐渐增长到10.



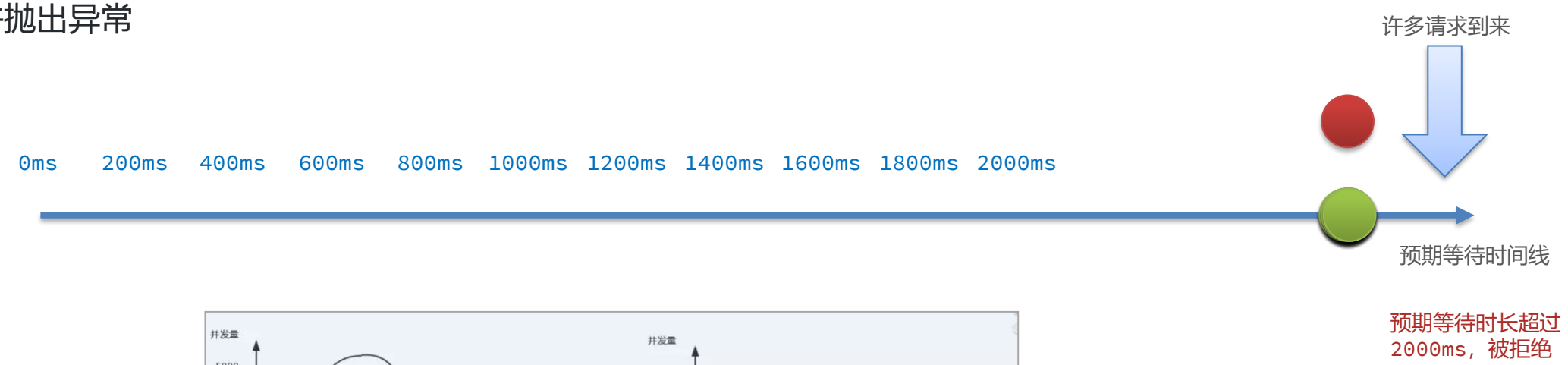
案例 流控效果-warm up

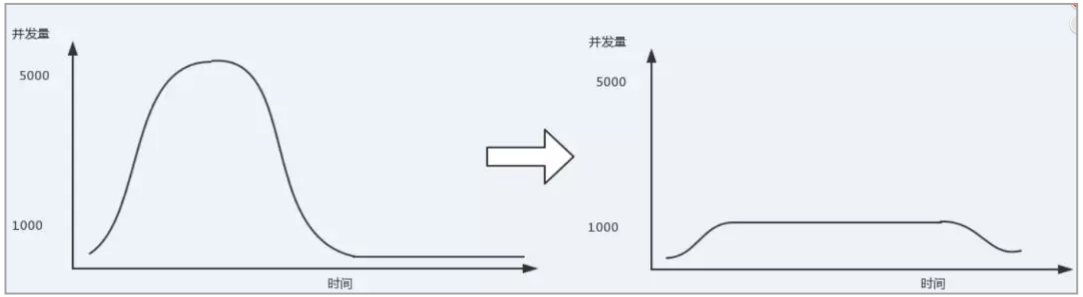
需求：给/order/{orderId}这个资源设置限流，最大QPS为10，利用warm up效果，预热时长为5秒

流控效果-排队等待

当请求超过QPS阈值时，快速失败和warm up 会拒绝新的请求并抛出异常。而排队等待则是让所有请求进入一个队列中，然后按照阈值允许的时间间隔依次执行。后来的请求必须等待前面执行完成，如果请求预期的等待时间超出最大时长，则会被拒绝。

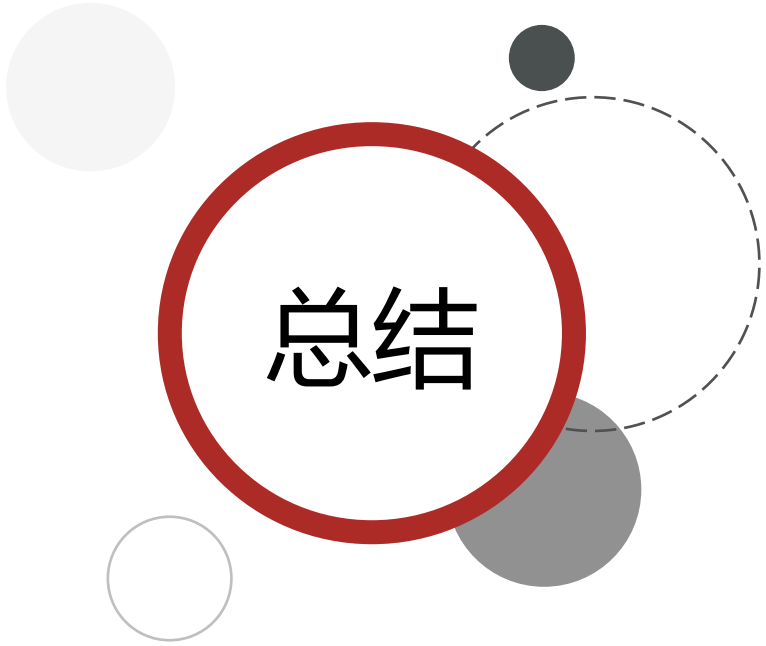
例如：QPS = 5，意味着每200ms处理一个队列中的请求；timeout = 2000，意味着预期等待超过2000ms的请求会被拒绝并抛出异常





案例 流控效果-排队等待

需求：给/order/{orderId}这个资源设置限流，最大QPS为10，利用排队的流控效果，超时时长设置为5s



流控效果有哪些?

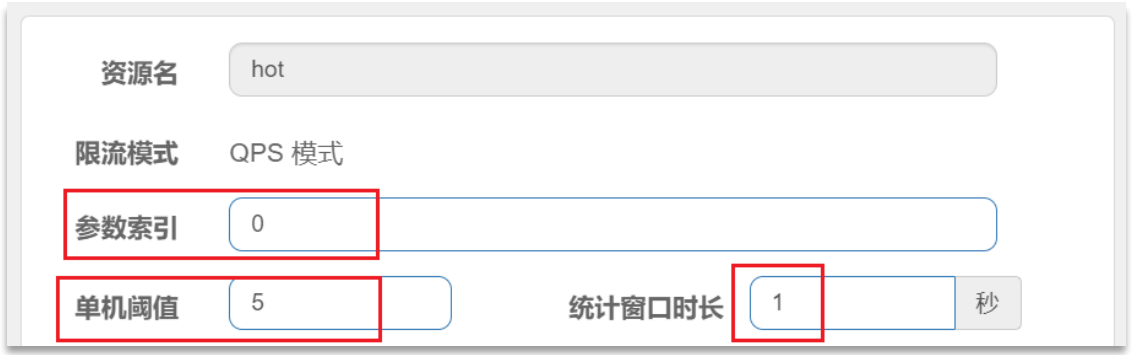
- 快速失败：QPS超过阈值时，拒绝新的请求
- warm up： QPS超过阈值时，拒绝新的请求；QPS阈值是逐渐提升的，可以避免冷启动时高并发导致服务宕机。
- 排队等待：请求会进入队列，按照阈值允许的时间间隔依次执行请求；如果请求预期等待时长大于超时时间，直接拒绝

热点参数限流

之前的限流是统计访问某个资源的所有请求，判断是否超过QPS阈值。而热点参数限流是分别统计参数值相同的请求，判断是否超过QPS阈值。

参数值	QPS
id = 1	3
id = 2	1

配置示例：



代表的含义是：对hot这个资源的0号参数（第一个参数）做统计，每1秒**相同参数值**的请求数不能超过5

热点参数限流

在热点参数限流的高级选项中，可以对部分参数设置例外配置：

结合上一个配置，这里的含义是对0号的long类型参数限流，每1秒相同参数的QPS不能超过5，有两个例外：

- 如果参数值是100，则每1秒允许的QPS为10
- 如果参数值是101，则每1秒允许的QPS为15

案例 热点参数限流

给/order/{orderId}这个资源添加热点参数限流，规则如下：

- 默认的热点参数规则是每1秒请求量不超过2
- 给102这个参数设置例外：每1秒请求量不超过4
- 给103这个参数设置例外：每1秒请求量不超过10

注意

热点参数限流对默认的SpringMVC资源无效

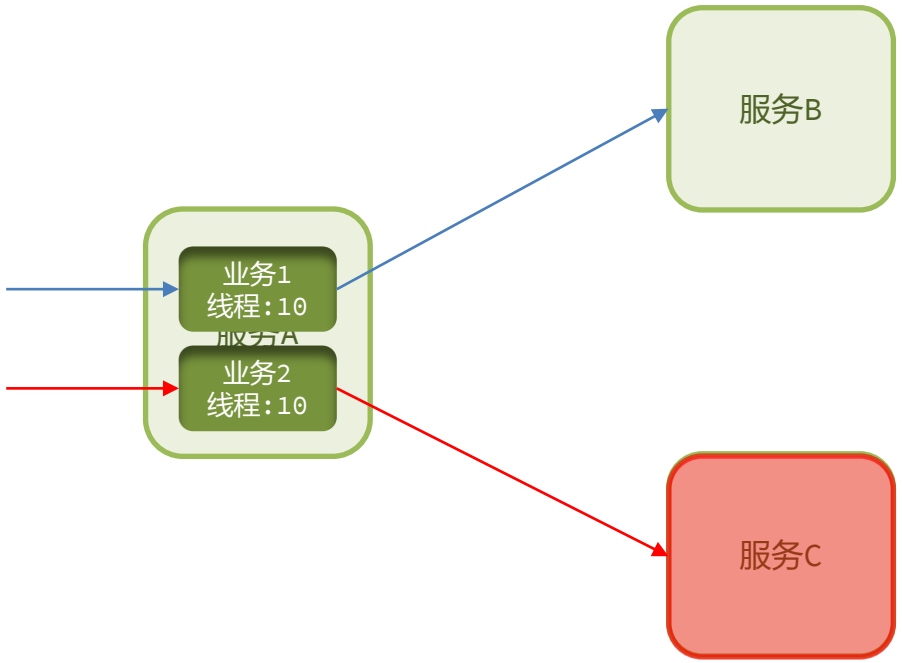
隔离和降级

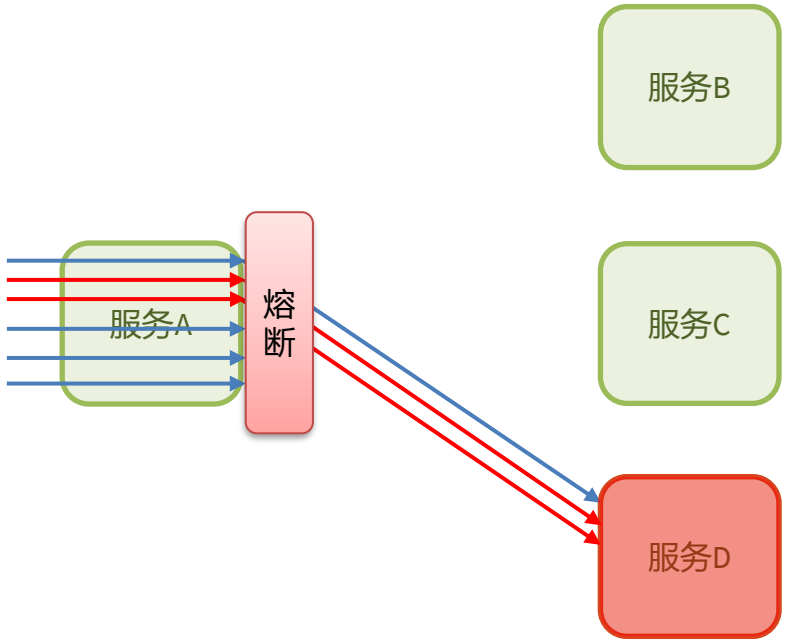
- FeignClient整合Sentinel
- 线程隔离（舱壁模式）
- 熔断降级

隔离和降级

虽然限流可以尽量避免因高并发而引起的服务故障，但服务还会因为其它原因而故障。而要将这些故障控制在一定范围，避免雪崩，就要靠线程隔离（舱壁模式）和熔断降级手段了。

不管是线程隔离还是熔断降级，都是对客户端（调用方）的保护。





Feign整合Sentinel

SpringCloud中，微服务调用都是通过Feign来实现的，因此做客户端保护必须整合Feign和Sentinel。

1. 修改OrderService的application.yml文件，开启Feign的Sentinel功能

```yaml
feign:
  sentinel:
    enabled: true # 开启Feign的Sentinel功能
```

2. 给FeignClient编写失败后的降级逻辑
   ① 方式一：FallbackClass，无法对远程调用的异常做处理
   ② 方式二：FallbackFactory，可以对远程调用的异常做处理，我们选择这种

Feign整合Sentinel

步骤一：在feing-api项目中定义类，实现FallbackFactory：

```
@Slf4j
public class UserClientFallbackFactory implements FallbackFactory<UserClient> {
    @Override
    public UserClient create(Throwable throwable) {
        // 创建UserClient接口实现类，实现其中的方法，编写失败降级的处理逻辑
        return new UserClient() {
            @Override
            public User findById(Long id) {
                // 记录异常信息
                log.error("查询用户失败", throwable);
                // 根据业务需求返回默认的数据，这里是空用户
                return new User();
            }
        };
    }
}
```

Feign整合Sentinel

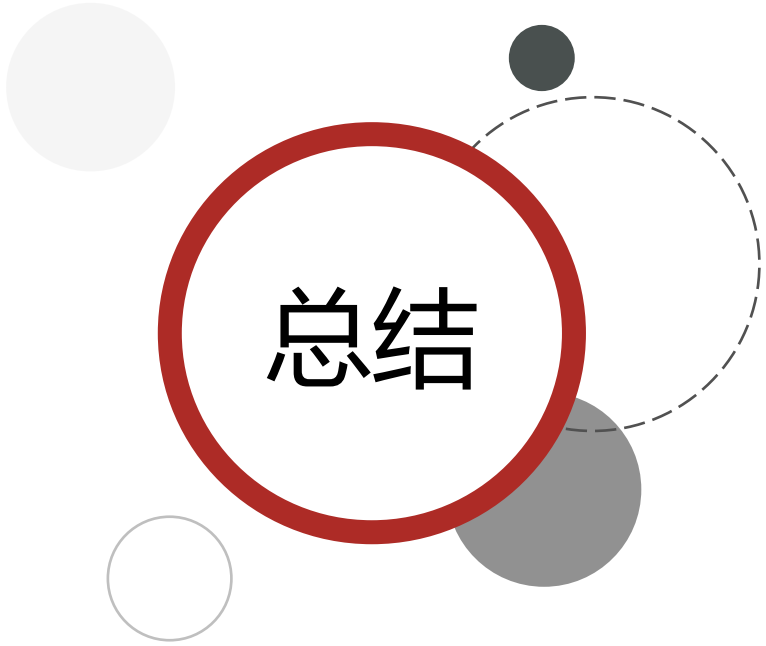
步骤二：在feing-api项目中的DefaultFeignConfiguration类中将UserClientFallbackFactory注册为一个Bean：

```
@Bean
public UserClientFallbackFactory userClientFallback(){
    return new UserClientFallbackFactory();
}
```

步骤三：在feing-api项目中的UserClient接口中使用UserClientFallbackFactory：

```
@FeignClient(value = "userservice", fallbackFactory = UserClientFallbackFactory.class)
public interface UserClient {

    @GetMapping("/user/{id}")
    User findById(@PathVariable("id") Long id);
}
```



Sentinel支持的雪崩解决方案：

- 线程隔离（仓壁模式）
- 降级熔断

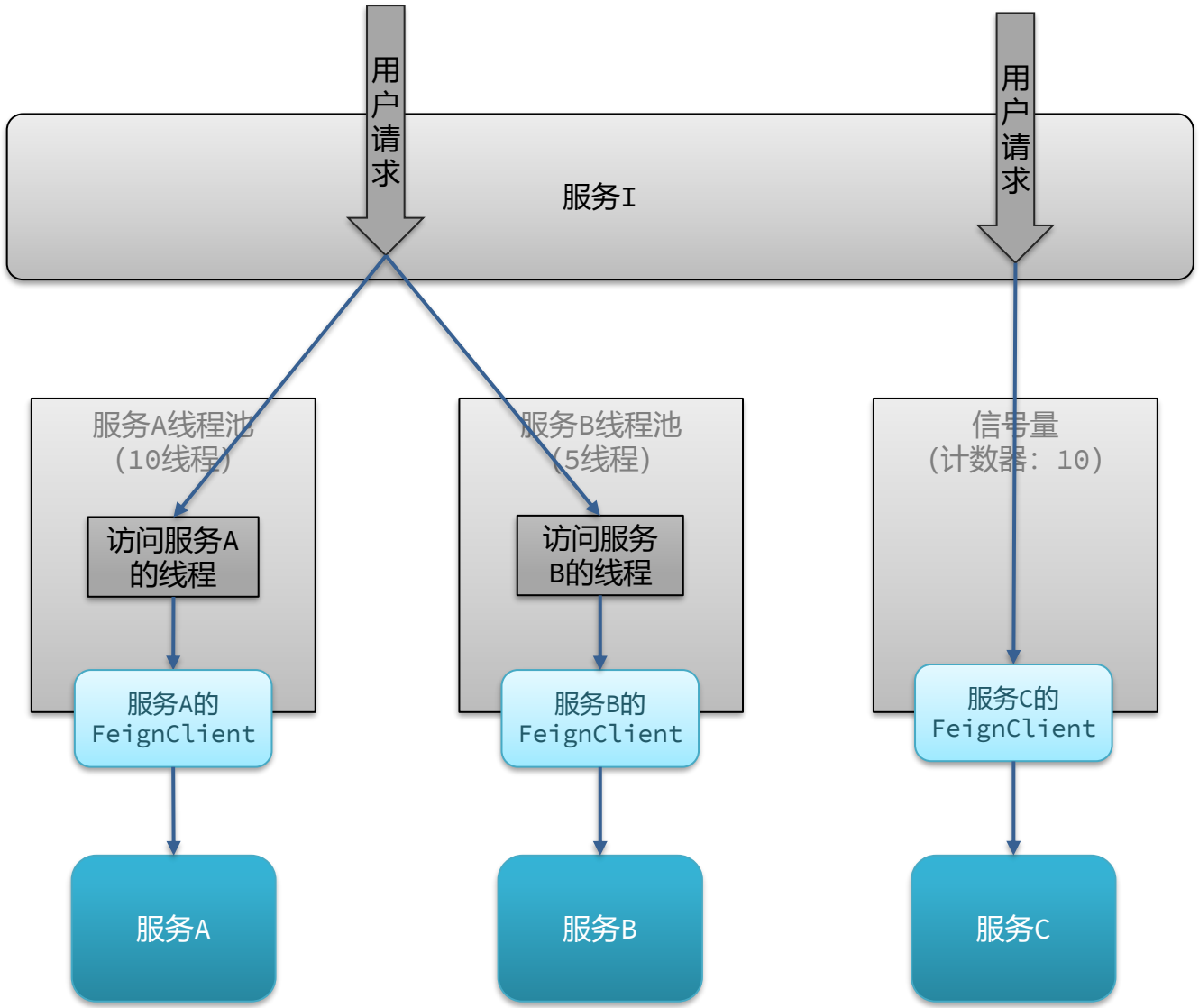
Feign整合Sentinel的步骤：

- 在application.yml中配置：feign.sentienl.enable=true
- 给FeignClient编写FallbackFactory并注册为Bean
- 将FallbackFactory配置到FeignClient

线程隔离

线程隔离有两种方式实现：

- 线程池隔离
- 信号量隔离（Sentinel默认采用）



线程隔离

信号量隔离

优点

轻量级，无额外开销

缺点

不支持主动超时
不支持异步调用

场景

高频调用
高扇出

线程池隔离

优点

支持主动超时
支持异步调用

缺点

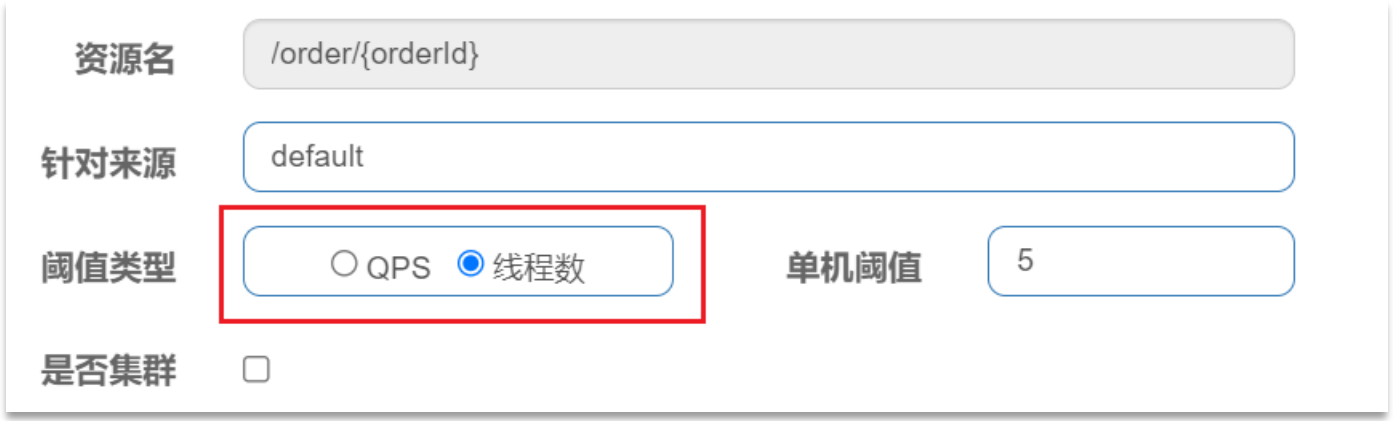
线程的额外开销比较大

场景

低扇出

线程隔离（舱壁模式）

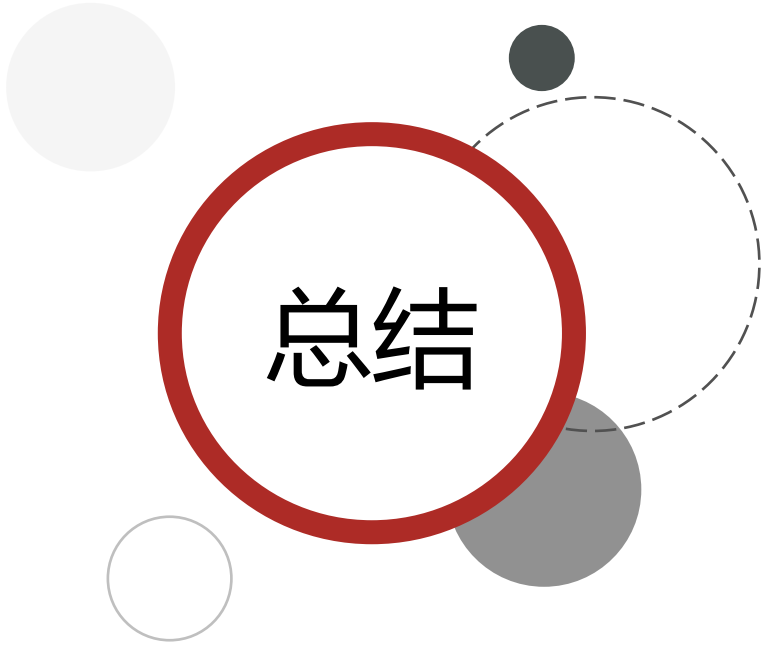
在添加限流规则时，可以选择两种阈值类型：



- QPS：就是每秒的请求数，在快速入门中已经演示过
- 线程数：是该资源能使用用的tomcat线程数的最大值。也就是通过限制线程数量，实现舱壁模式。

案例 线程隔离（舱壁模式）

需求：给 UserClient的查询用户接口设置流控规则，线程数不能超过 2。然后利用jemeter测试。



线程隔离的两种手段是?

- 信号量隔离
- 线程池隔离

信号量隔离的特点是?

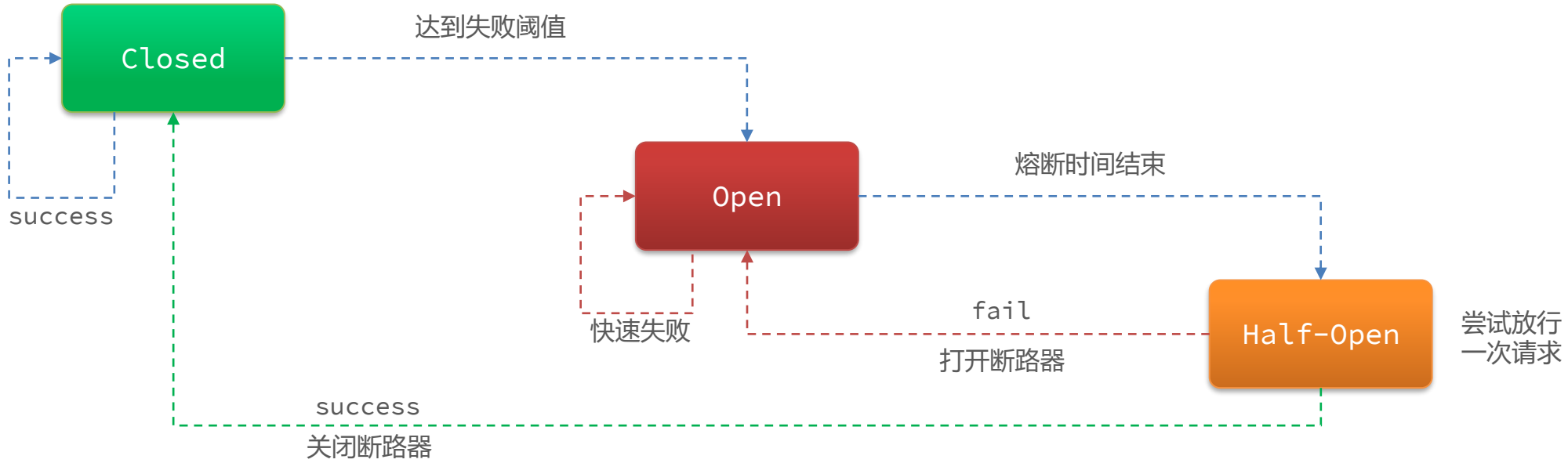
- 基于计数器模式，简单，开销小

线程池隔离的特点是?

- 基于线程池模式，有额外开销，但隔离控制更强

熔断降级

熔断降级是解决雪崩问题的重要手段。其思路是由**断路器**统计服务调用的异常比例、慢请求比例，如果超出阈值则会**熔断**该服务。即拦截访问该服务的一切请求；而当服务恢复时，断路器会放行访问该服务的请求。



熔断策略-慢调用

断路器熔断策略有三种：慢调用、异常比例、异常数

- 慢调用：业务的响应时长（RT）大于指定时长的请求认定为慢调用请求。在指定时间内，如果请求数量超过设定的最小数量，慢调用比例大于设定的阈值，则触发熔断。例如：

解读：RT超过500ms的调用是慢调用，统计最近10000ms内的请求，如果请求量超过10次，并且慢调用比例不低于0.5，则触发熔断，熔断时长为5秒。然后进入half-open状态，放行一次请求做测试。

案例 熔断策略-慢调用

需求：给 UserClient的查询用户接口设置降级规则，慢调用的RT阈值为50ms，统计时间为1秒，最小请求数量为5，失败阈值比例为0.4，熔断时长为5

提示：为了触发慢调用规则，我们需要修改UserService中的业务，增加业务耗时：

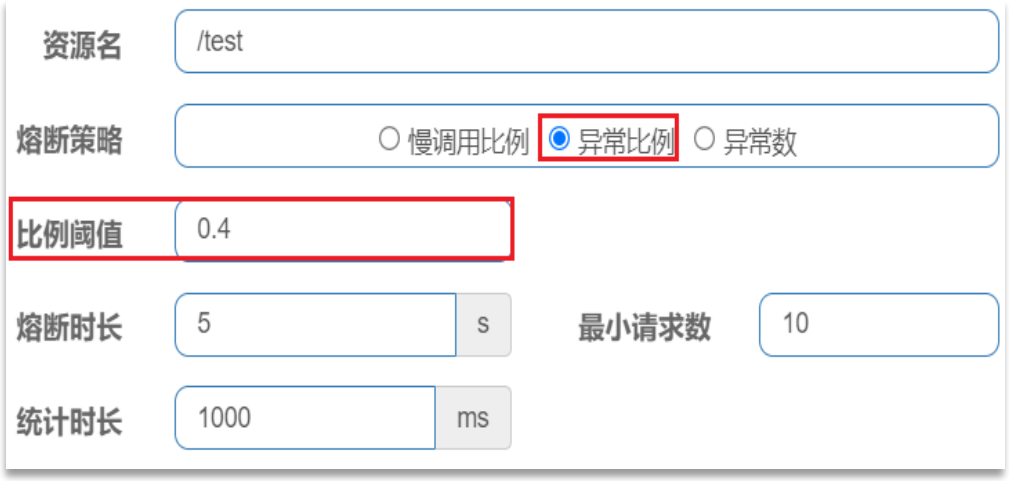
UserController.java

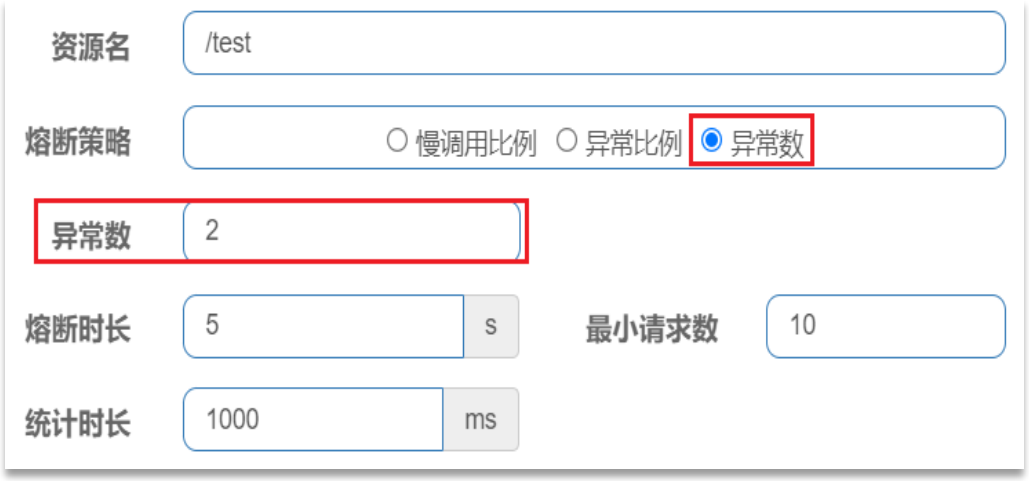
```java
/**
 * @param id 用户id
 */
@GetMapping("/{id}")
public User queryById(@PathVariable("id") Long id) throws InterruptedException {
    if(id == 1){
        // id为1时，触发慢调用
        Thread.sleep( millis: 60);
    }
    return userService.queryById(id);
}
```

熔断策略-异常比例、异常数

断路器熔断策略有三种：慢调用、异常比例或异常数

- 异常比例或异常数：统计指定时间内的调用，如果调用次数超过指定请求数，并且出现异常的比例达到设定的比例阈值（或超过指定异常数），则触发熔断。例如：





解读：统计最近1000ms内的请求，如果请求量超过10次，并且异常比例不低于0.5，则触发熔断，熔断时长为5秒。然后进入half-open状态，放行一次请求做测试。

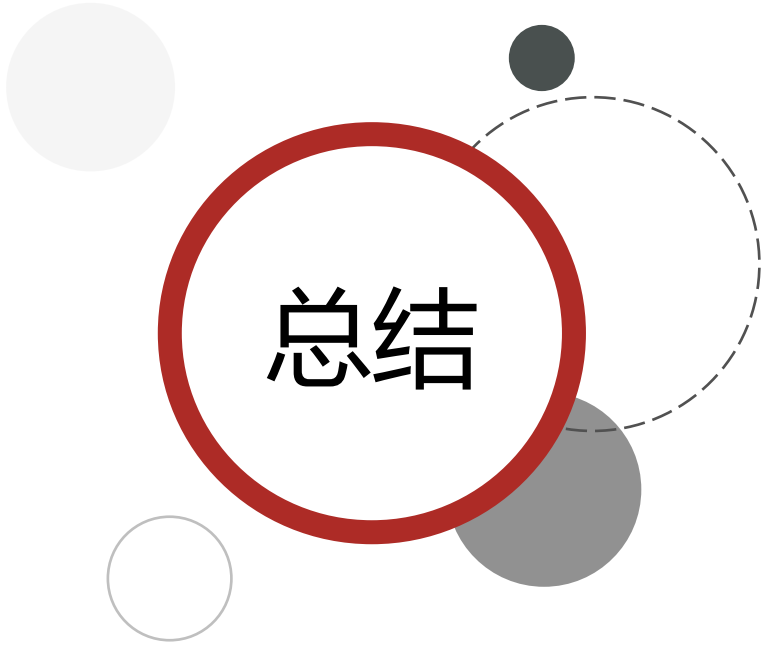
案例 熔断策略-异常比例

需求：给 UserClient的查询用户接口设置降级规则，统计时间为1秒，最小请求数量为5，失败阈值比例为0.4，熔断时长为5s

提示：为了触发异常统计，我们需要修改UserService中的业务，抛出异常：

UserController.java

```
@GetMapping("/{id}")
public User queryById(@PathVariable("id") Long id) throws InterruptedException {
    if(id == 1){
        // id为1时，触发慢调用
        Thread.sleep( millis: 60);
    }else if(id == 2){
        throw new RuntimeException("故意抛出异常，触发异常比例熔断");
    }
    return userService.queryById(id);
}
```



Sentinel熔断降级的策略有哪些?

- 慢调用比例：超过指定时长的调用为慢调用，统计单位时长内慢调用的比例，超过阈值则熔断
- 异常比例：统计单位时长内异常调用的比例，超过阈值则熔断
- 异常数：统计单位时长内异常调用的次数，超过阈值则熔断

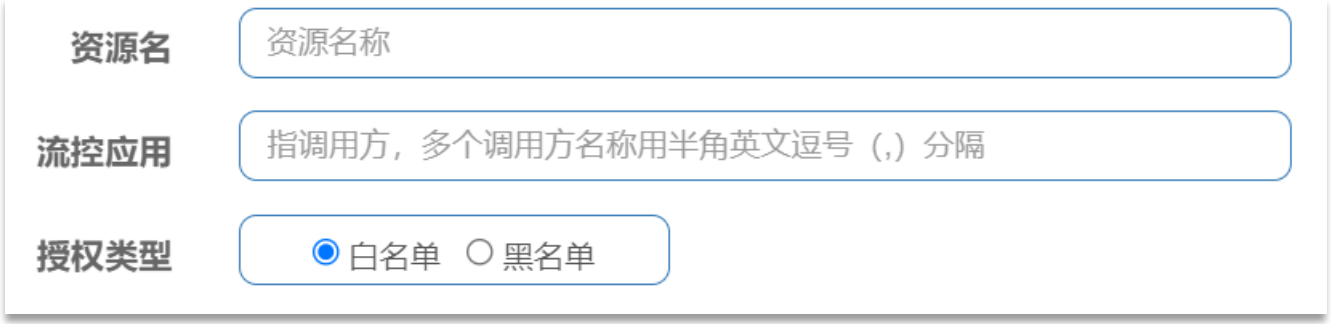
授权规则

- 授权规则
- 自定义异常结果

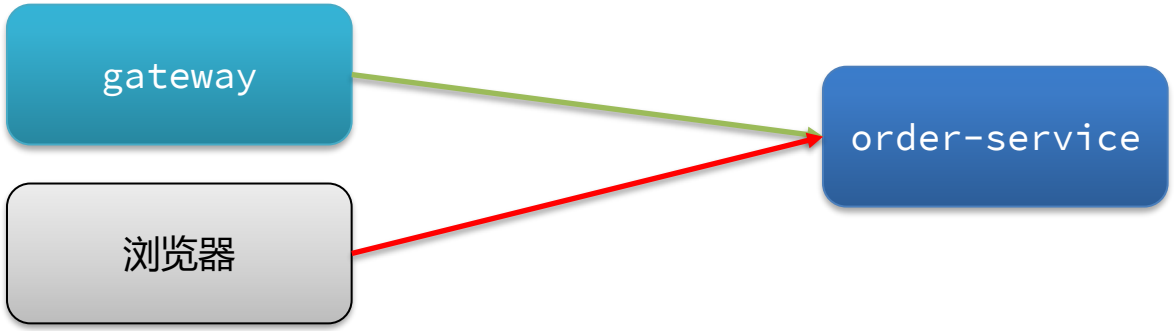
授权规则

授权规则可以对调用方的来源做控制，有白名单和黑名单两种方式。

- 白名单：来源（origin）在白名单内的调用者允许访问
- 黑名单：来源（origin）在黑名单内的调用者不允许访问



例如，我们限定只允许从网关来的请求访问order-service，那么流控应用中就填写网关的名称



授权规则

Sentinel是通过RequestOriginParser这个接口的parseOrigin来获取请求的来源的。

```
public interface RequestOriginParser {

    /**
     * 从请求request对象中获取origin，获取方式自定义
     */
    String parseOrigin(HttpServletRequest request);
}
```

例如，我们尝试从request中获取一个名为origin的请求头，作为origin的值：

```
@Component
public class HeaderOriginParser implements RequestOriginParser {
    @Override
    public String parseOrigin(HttpServletRequest request) {
        String origin = request.getHeader("origin");
        if(StringUtils.isEmpty(origin)){
            return "blank";
        }
        return origin;
    }
}
```

授权规则

我们还需要在gateway服务中，利用网关的过滤器添加名为gateway的origin头：

```yaml
spring:
  cloud:
    gateway:
      default-filters:
        - AddRequestHeader=origin,gateway # 添加名为origin的请求头，值为gateway
```

给/order/{orderId} 配置授权规则：

自定义异常结果

默认情况下，发生限流、降级、授权拦截时，都会抛出异常到调用方。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;

}
```

自定义异常结果

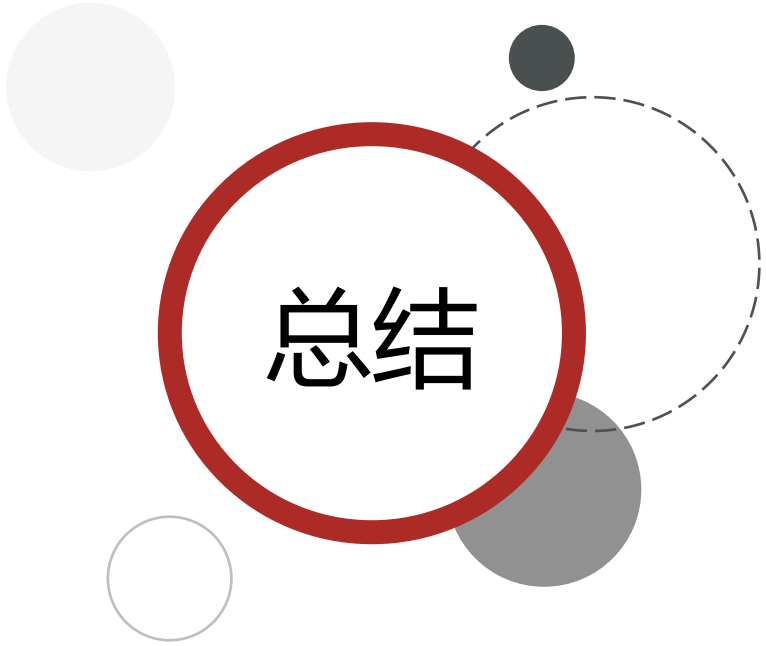
而BlockException包含很多个子类，分别对应不同的场景：

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

自定义异常结果

我们在order-service中定义类，实现BlockExceptionHandler接口：

```
@Component
public class SentinelBlockHandler implements BlockExceptionHandler {
    @Override
    public void handle(
            HttpServletRequest httpServletRequest,
            HttpServletResponse httpServletResponse, 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 DegradeException) {
            msg = "请求被降级了！";
        } else if (e instanceof ParamFlowException) {
            msg = "热点参数限流！";
        } else if (e instanceof AuthorityException) {
            msg = "请求没有权限！";
            status = 401;
        }
        httpServletResponse.setContentType("application/json;charset=utf-8");
        httpServletResponse.setStatus(status);
        httpServletResponse.getWriter().println("{\"message\": \"" + msg + "\", \"status\": " + status + "}");
    }
}
```



获取请求来源的接口是什么?

- `RequestOriginParser`

处理BlockException的接口是什么?

- `BlockExceptionHandler`

规则持久化

- 规则管理模式
- 实现push模式

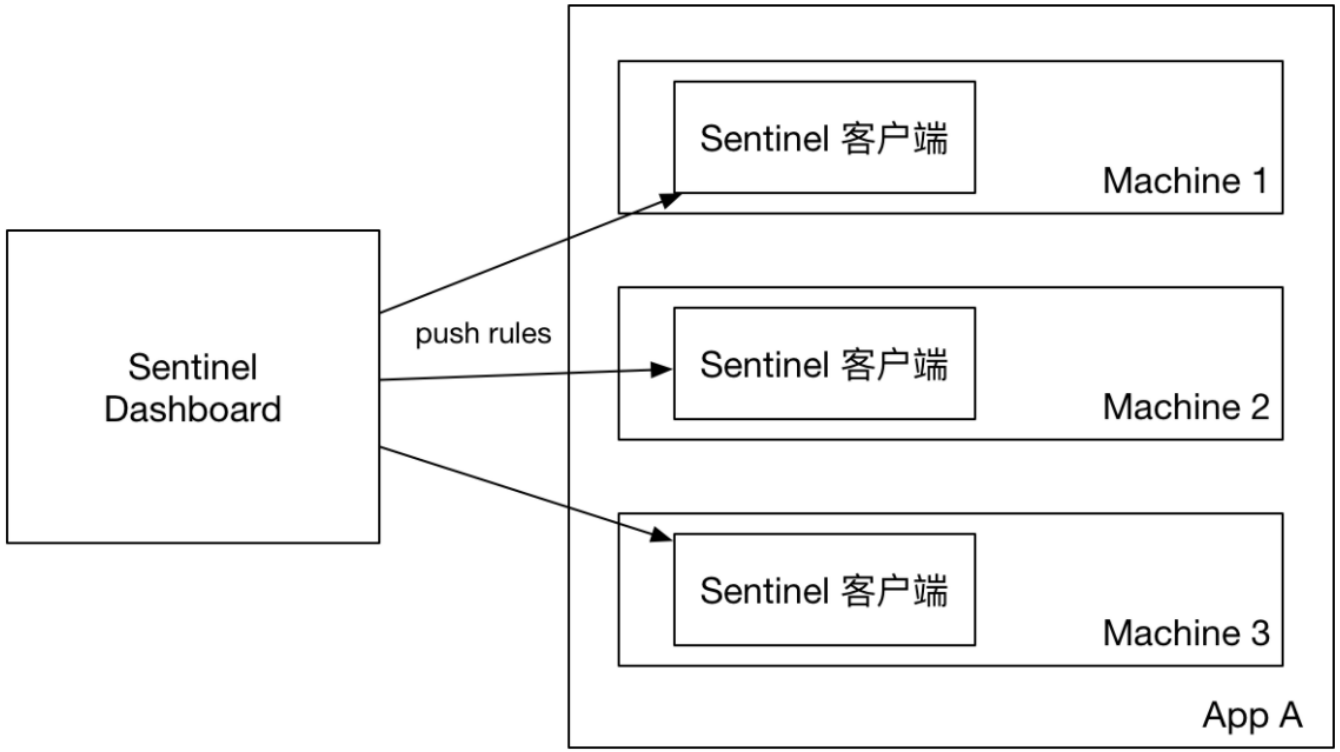
规则管理模式

Sentinel的控制台规则管理有三种模式：

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），默认就是这种	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；	引入第三方依赖

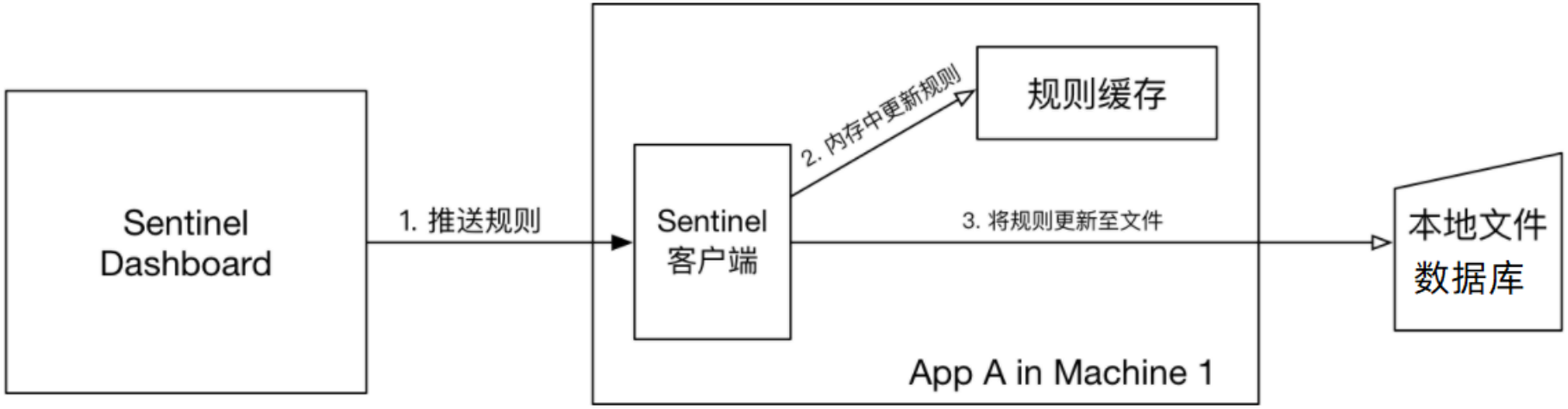
规则管理模式-原始模式

原始模式：控制台配置的规则直接推送到Sentinel客户端，也就是我们的应用。然后保存在内存中，服务重启则丢失



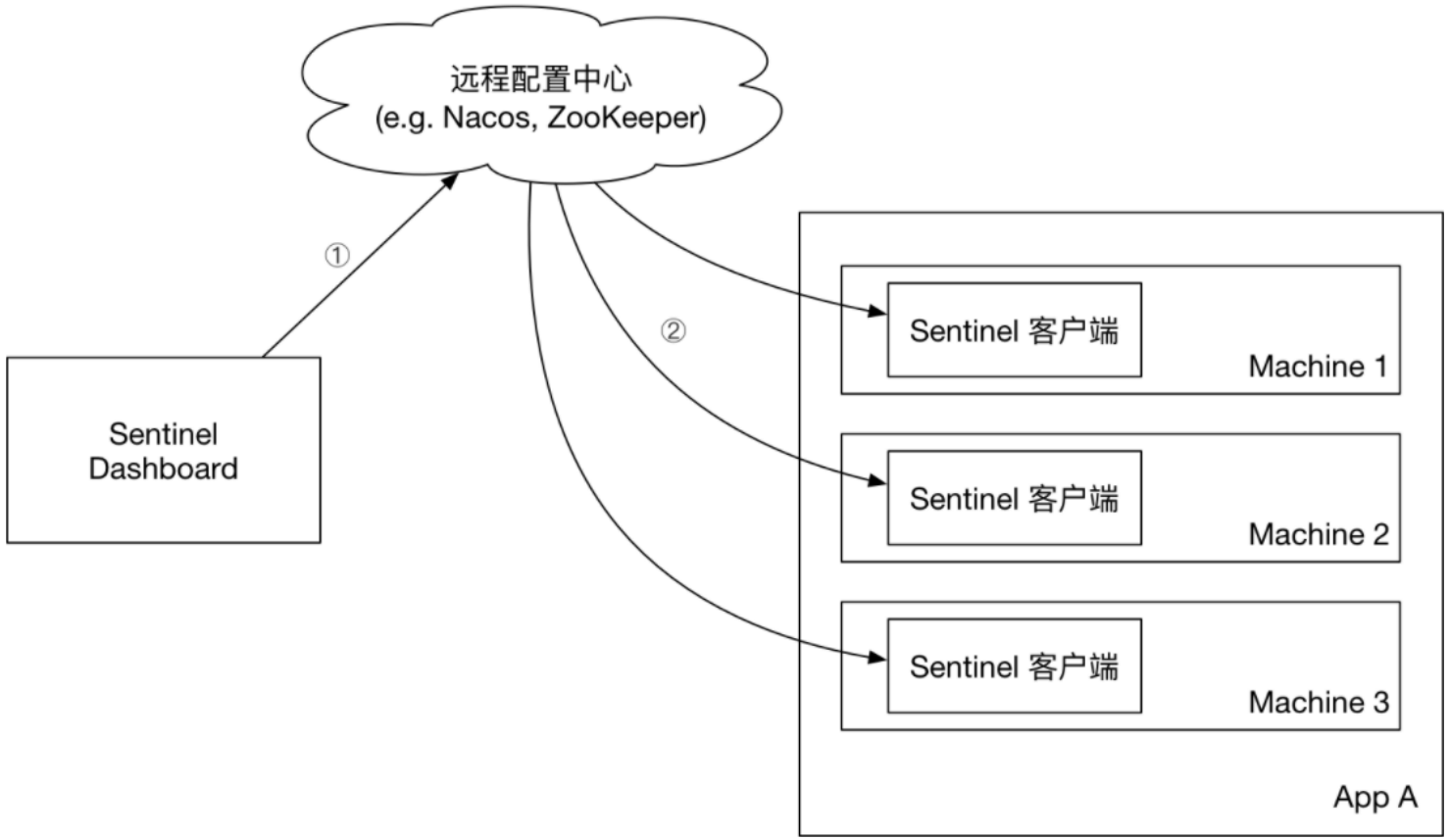
规则管理模式-pull模式

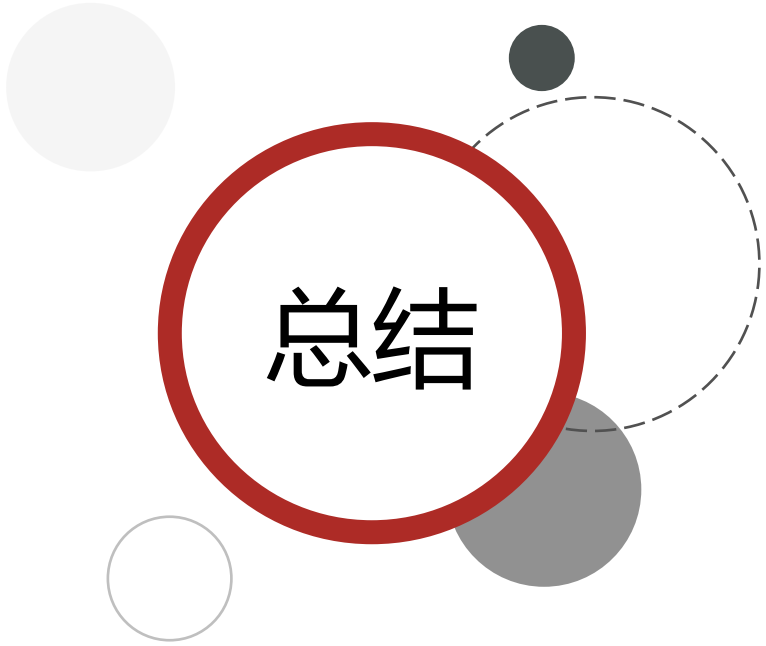
pull模式：控制台将配置的规则推送到Sentinel客户端，而客户端会将配置规则保存在本地文件或数据库中。以后会定时去本地文件或数据库中查询，更新本地规则。



规则管理模式-push模式

push模式：控制台将配置规则推送到远程配置中心，例如Nacos。Sentinel客户端监听Nacos，获取配置变更的推送消息，完成本地配置更新。





Sentinel的三种配置管理模式是什么？

- 原始模式：保存在内存
- pull模式：保存在本地文件或数据库，定时去读取
- push模式：保存在nacos，监听变更实时更新

实现push模式

push模式实现最为复杂，依赖于nacos，并且需要改在Sentinel控制台。整体步骤如下：

1. 修改order-service服务，使其监听Nacos配置中心
2. 修改Sentinel-dashboard源码，配置nacos数据源
3. 修改Sentinel-dashboard源码，修改前端页面
4. 重新编译、打包-dashboard源码

详细步骤可以参考课前资料的《sentinel规则持久化》：

sentinel规则持久

步骤

步骤一：修改order-service服务，使其监听Nacos配置中心

具体步骤如下：

1. 引入依赖

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

2. 配置nacos地址

```
spring:
  cloud:
    sentinel:
      datasource:
        flow:
          nacos:
            server-addr: localhost:8848 # nacos地址
            dataId: orderservice-flow-rules
            groupId: SENTINEL_GROUP
            rule-type: flow # 还可以是：degrade、authority、param-flow
```

步骤

步骤二：修改sentinel-dashboard源码，添加nacos数据源

具体步骤如下：

1. 解压课前资料中的sentinel源码包，并用IDEA打开：

Sentinel-1.8.1.zip

2. 修改sentinel-dashboard源码的pom文件，将sentinel-datasource-nacos依赖的scope去掉

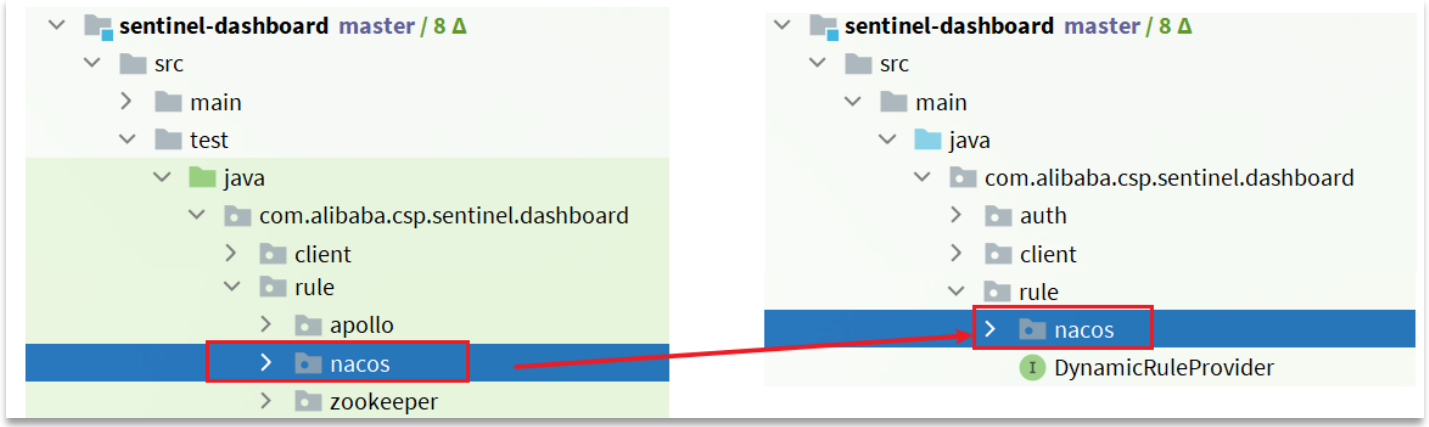
```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
    <!--<scope>test</scope>-->
</dependency>
```

步骤

步骤三：修改sentinel-dashboard的源码，配置nacos数据源

具体步骤如下：

1. 拷贝test目录下的nacos代码到main下的com.alibaba.csp.sentinel.dashboard.rule包：



2. 修改刚刚拷贝的nacos包下的NacosConfig类，修改其中的nacos地址：

```
@Bean
public ConfigService nacosConfigService() throws Exception {
    return ConfigFactory.createConfigService("localhost:8848");
}
```

步骤

步骤三：修改sentinel-dashboard的源码，配置nacos数据源

3. 修改 com.alibaba.csp.sentinel.dashboard.controller.v2包下的FlowControllerV2类：

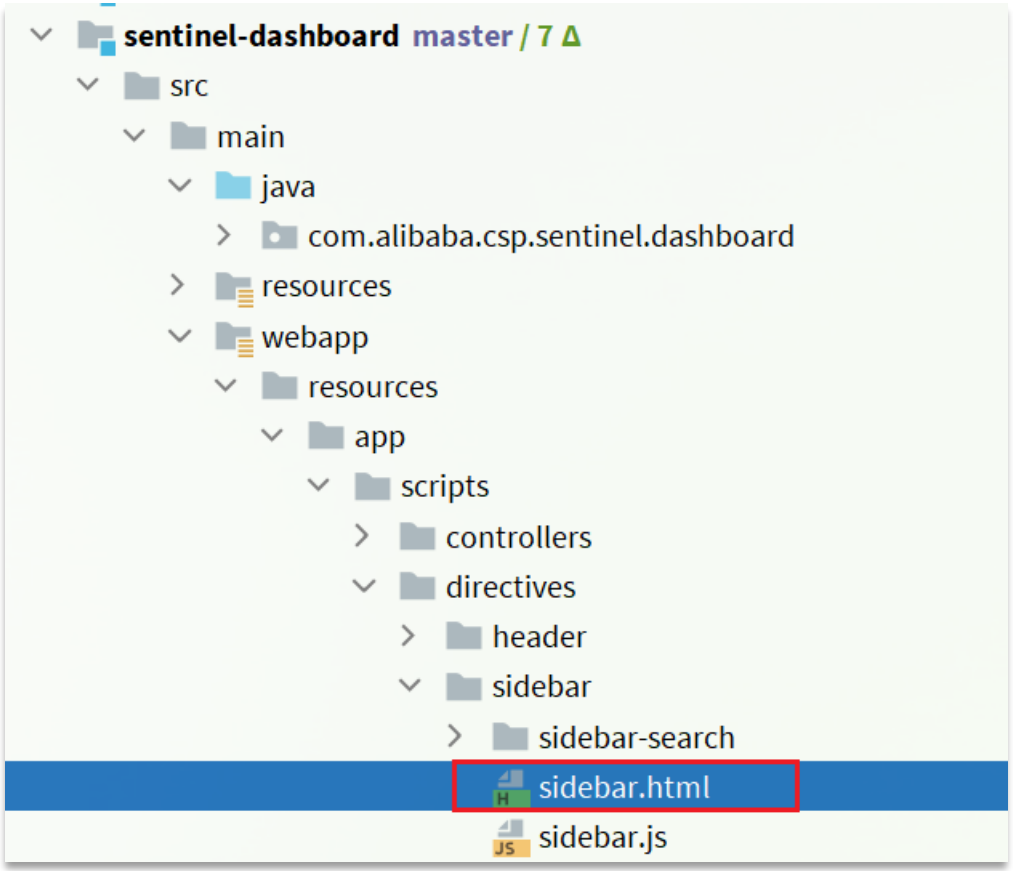
```
@Autowired
@Qualifier("flowRuleDefaultProvider")
private DynamicRuleProvider<List<FlowRuleEntity>> ruleProvider;
@Autowired
@Qualifier("flowRuleDefaultPublisher")
private DynamicRulePublisher<List<FlowRuleEntity>> rulePublisher;


@Autowired
@Qualifier("flowRuleNacosProvider")
private DynamicRuleProvider<List<FlowRuleEntity>> ruleProvider;
@Autowired
@Qualifier("flowRuleNacosPublisher")
private DynamicRulePublisher<List<FlowRuleEntity>> rulePublisher;
```

步骤

步骤四：修改sentinel-dashboard的源码，修改前端页面：

修改src/main/webapp/resources/app/scripts/directives/sidebar/目录下的sidebar.html文件：



步骤

步骤四：修改sentinel-dashboard的源码，修改前端页面：

修改src/main/webapp/resources/app/scripts/directives/sidebar/目录下的sidebar.html文件。

将其中的这部分注释打开：

```
<!--<li ui-sref-active="active" ng-if="entry.appType==0">
  <a ui-sref="dashboard.flow({app: entry.app})">
    <i class="glyphicon glyphicon-filter"></i>  流控规则-V1</a>
</li>-->
```

修改其中的文本：

```
<li ui-sref-active="active" ng-if="entry.appType==0">
  <a ui-sref="dashboard.flow({app: entry.app})">
    <i class="glyphicon glyphicon-filter"></i>  流控规则-NACOS</a>
</li>
```

传智教育旗下高端IT教育品牌