博学谷
www.boxuegu.com

第四章 Spring Cloud Alibaba Sentinel 高可用防护

一样的在线教育，不一样的教学品质

目录 Contents

Sentinel 环境搭建

小节导学

Sentinel 的用法很简单，服务只需要添加 Sentinel 的依赖，就具有限流、降级等防护功能了。

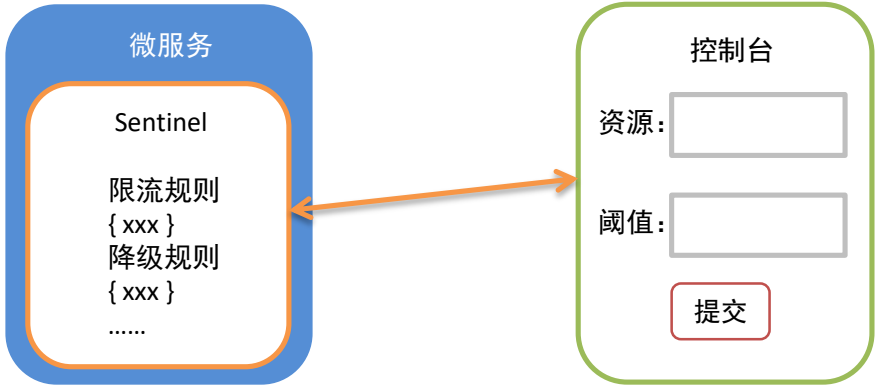
但**具体如何防护呢？**，例如希望此服务的某个接口的 QPS 达到 5 时就限流，这就需要设定一个 Sentinel 规则。

有了规则之后，Sentinel 便会根据规则来保护服务。

那么**规则怎么设置呢？**手写肯定不现实，Sentinel 提供了一个管理控制台，界面化设置规则，规则会被自动推送给服务。

本节我们就学习一下服务如何整合 Sentinel 以及控制台。

- 服务整合 Sentinel
- 搭建 Sentinel 控制台



1. 服务整合 Sentinel

步骤：

1. 服务添加 Sentinel 依赖

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

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

2. 开启属性配置

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

3. 验证

访问 Sentinel 端点信息：

```
http://localhost:8001/actuator/sentinel
```

2. 搭建 Sentinel 控制台

步骤：

1. 下载编译源码

下载源码：

```
https://github.com/alibaba/Sentinel
```

编译源码：

```
> cd sentinel-dashboard
> mvn clean package
```

2. 启动

```
java -jar sentinel-dashboard-1.7.0.jar
```

3. 服务整合控制台

```
spring:
  application:
    name: service-provider
  cloud:
    sentinel:
      transport:
        dashboard: localhost:8080
        port: 8719
```

Sentinel 环境搭建-总结

重难点

1. 理解微服务、Sentinel、Sentinel Console 的关联关系
2. 服务整合 Sentinel 的方式
3. Sentinel Console 的搭建方法
4. 服务整合 Sentinel Console 的方式

目录 Contents

URL限流与资源限流

小节导学

URL限流比较好理解，就是根据访问路径进行限流控制。

例如有一个接口：/user/get，就可以对其添加限流规则。

“**资源**”是 Sentinel 中的重要概念，可以被调用的都可以视为资源，例如接口、Service。

对资源设置一个名称，就可以对其添加限流规则。

本节我们就学习一下服务如何根据URL进行限流，以及如何设置一个资源并对其限流。

- 根据 URL 进行限流
- 设置资源，对资源进行限流

1. 根据 URL 进行限流

目标：

创建一个测试接口，在 Sentinel Console 中对其设置限流规则，验证限流生效。

步骤：

1. 创建一个项目，一个测试接口
2. 启动 Sentinel Console
3. 项目整合 Sentinel 以及 Console
4. 在 Console 中添加测试接口的限流规则
5. 验证规则生效

1. 根据 URL 进行限流

添加限流规则界面

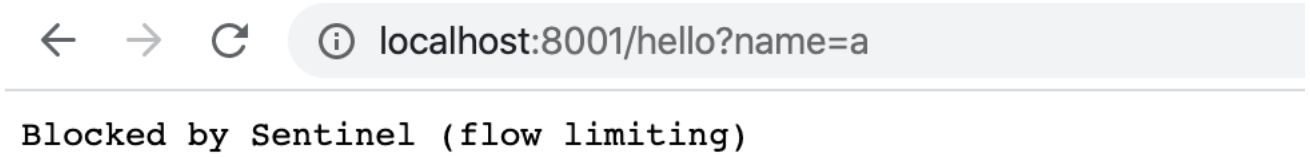
1. 根据 URL 进行限流

流控效果验证：

快速多几次访问 /hello 这个接口。

因为设置的限流是 QPS 阈值为 1，所以很容易触发限流规则。

被限流后的效果：



1. 根据 URL 进行限流

流控规则项说明：

- **资源名**：这里默认使用的接口地址
- **针对来源**：可以指定来自哪个服务的请求才适用这条规则，默认是default，也就相当于不区分来源
- **阈值类型**：可以根据 QPS，也就是每秒请求数，进行设置，也可以根据已经使用的线程数来设置
- **单机阈值**：设置此限流规则的阈值数。例如阈值类型选择了 QPS，阈值设置为1，意思就是当每秒请求数达到 1 以后就限流，阈值类型如果选择了线程数，意思就是处理请求的线程数达到 1 以上就限流。
- **流控模式**：直接，就是针对这个接口本身
- **流控效果**：快速失败，直接失败，抛异常

2. 根据资源进行限流

目标：

对一个测试接口设置为资源，在 Sentinel Console 中对其设置限流规则，验证限流生效。

步骤：

1. 通过注解的方式对一个测试接口设置为资源

```
@GetMapping("/hello")
@SentinelResource(value = "res_hello")
public String hello(@RequestParam String name) {
    return "hello " + name + "!";
}
```

2. 在 Console 中添加测试接口的限流规则
3. 验证规则生效

2. 根据资源进行限流

添加限流规则界面

与之前的对 URL 添加流控规则的界面一致。

只是“资源名”这项不再是接口的 URL，变成了我们定义的资源名。

验证方式和被限流后的效果都与之前URL方式一致

URL限流与资源限流-总结

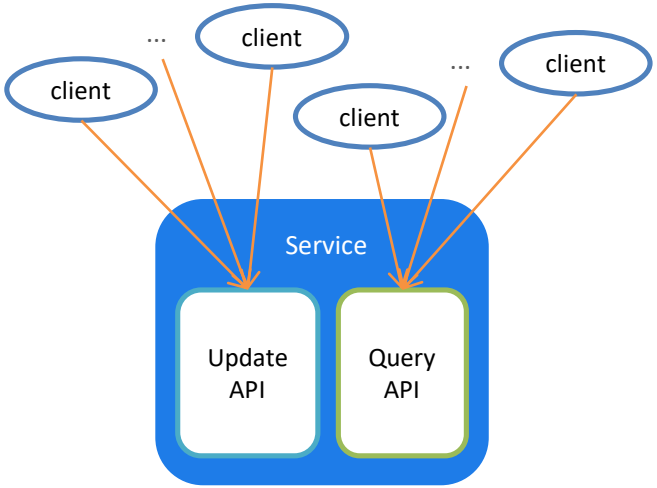
重难点

1. 限流界面中各设置项的含义
2. 基于URL限流的设置方式，与验证方式
3. 代码中设置资源的方式
4. 基于资源限流的设置方式，与验证方式

目录 Contents

关联限流与链路限流

小节导学



假设一个 Service 中有2个接口：查询、修改。

当查询接口访问量特别大时，必然要影响修改接口，可能会导致修改接口无法处理请求。

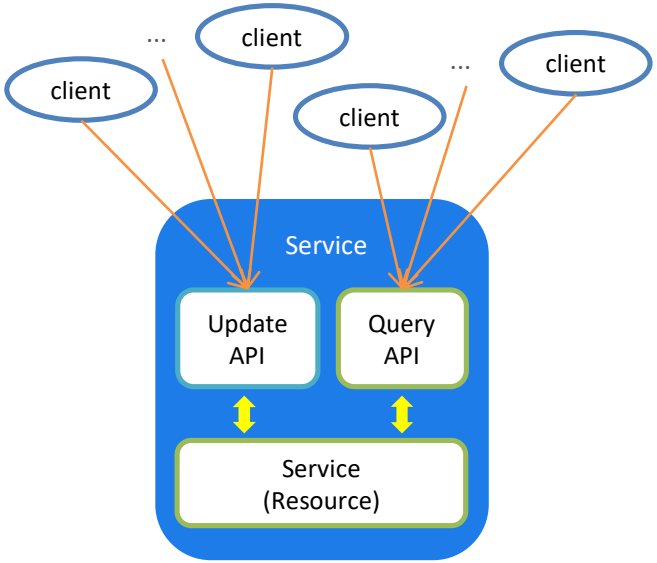
如果业务上，修改功能优先于查询功能，这就出现了一种限流的需求：

“当修改接口的访问量达到一定程度时，就对查询接口限流”。

这就是“关联限流”，限流规则不是自己设置的，而是被别人影响的。

关联限流与链路限流

小节导学



这个微服务中，2个API都需要调用这个Service资源，需要做限流保护。

如果查询接口请求多，导致这个资源被限流，那么修改接口也就用不了了，很冤枉。

这就出现了一种限流的需求：

“**这个Service资源只对查询接口限流，不影响修改接口**”。

这就是“**链路限流**”，执行限流时要判断流量是从哪儿来的。

本节就解决这2个限流问题，学习如何实现关联限流与链路限流。

- 关联限流
- 链路限流

1. 关联限流

目标：

创建2个测试接口，在 Sentinel Console 中对接口1设置限流规则，当接口2的QPS达到1时触发限流，并验证限流生效。

步骤：

1. 创建一个项目，构建2个测试接口
2. 整合 Sentinel 及 Sentinel Console
3. Sentinel Console 中对接口1设置关联接口2的限流规则
4. 验证限流生效

1. 关联限流

添加关联限流规则界面

1. 关联限流

流控效果验证：

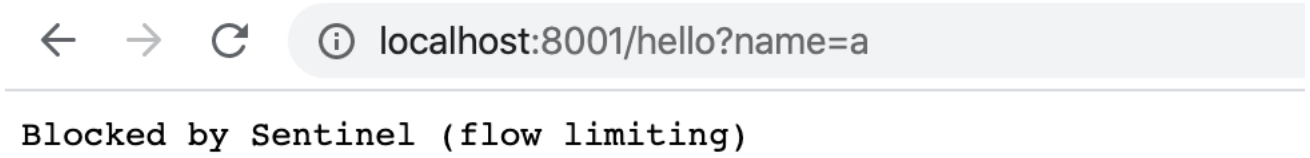
1. 持续大量访问 /hi 接口，使其触发阈值

例如通过脚本持续访问 /hi：

```
while true; \
do curl -X GET "http://localhost:8001/hi?name=a" ;\
done;
```

2. 访问接口 /hello，应已经被限流

被限流后的效果：



2. 链路限流

目标：

创建2个测试接口（/testa，/testb），都调用同一个Service（设置为 SentinelResource）。在 Sentinel Console 中对Service设置限流规则，指定入口资源为 /testa。验证 /testa 被限流时，/testb 正常。

步骤：

1. 构建2个测试接口，1个Service，指定其为 SentinelResource，2个接口都调用此Service
2. Sentinel Console 中对Service设置链路限流，入口资源指定 /testa
3. 验证限流生效

1. 关联限流

添加链路限流规则界面

2. 链路限流

流控效果验证：

1. 持续大量访问 /testa 接口，使其触发阈值，应产生被限流效果

例如通过脚本持续访问 /testa :

```
while true; \
do curl -X GET "http://localhost:8081/testa" ;\
done;
```

2. 访问接口 /testb，应可以正常访问

关联限流与链路限流-总结

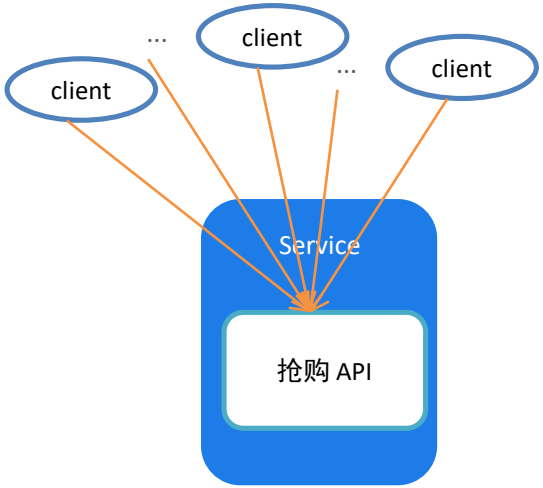
重难点

1. 关联限流、链路限流的使用场景与区别
2. 关联限流的设置方式，与验证方式
3. 链路限流的设置方式，与验证方式

目录 Contents

预热与排队等待

小节导学



秒杀开始！海量用户同时请求购买接口。

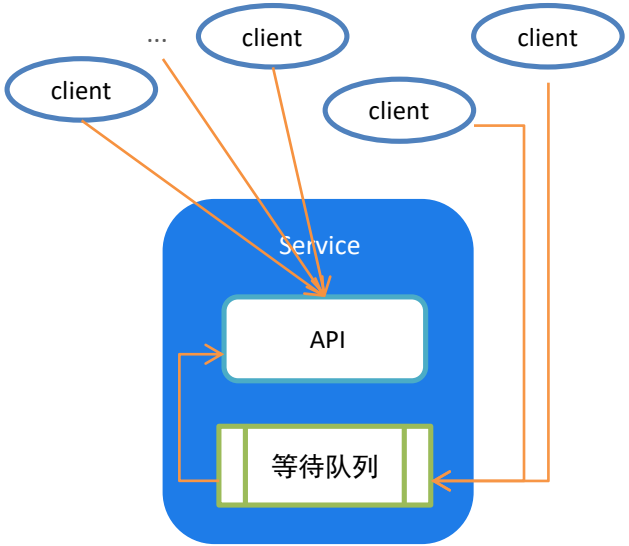
这一**瞬间冲击**非常大，即使接口设置了限流阈值，一瞬间打满也是很危险的。

在秒杀开始之前，这个接口的请求量是比较少的，由空闲状态切换为繁忙状态，我们希望这个过程是**逐步**的，而不是突然的。

这种场景就需要使用“**预热 Warm Up**”的流控方式。

预热与排队等待

小节导学



假设有个接口的特点是**间歇性突发流量**，在某一秒有大量的请求到来，而接下来的几秒则处于空闲状态，属于**时忙时闲。**

正常限流方式会导致大量请求被丢弃。

我们希望系统能够在接下来的**空闲期间逐渐处理**这些请求，而不是在第一秒直接拒绝超出阈值的请求。

这种场景就需要使用“**排队等待**”的流控方式。

本节我们就学习如何使用预热与排队等待。

- 预热 Warm Up
- 排队等待

1. 预热 Warm Up

预热的处理机制：

预热流控会设置一个初始阈值，默认是设置阈值的 **1/3**，在经过预期的时间后，达到设置的阈值。

例如设置：阈值=10，预期时间=10秒。则初始阈值为 3，10秒内，阈值逐步增加，10秒后增加到10。

使用的是 **令牌桶算法**。

实践步骤：

1. 创建一个项目，构建1个测试接口
2. 集成 Sentinel 与 Sentinel Console
3. Sentinel Console 中对接口设置预热 Warm Up 限流，阈值设为 10 QPS，预热时间设为 10 秒
4. 验证限流生效

1. 预热 Warm Up

预热限流设置界面

1. 预热 Warm Up

流控效果验证：

持续大量访问 /hi 接口，应有限流效果，但通过的请求数会逐渐增加

例如通过脚本持续访问 /hello :

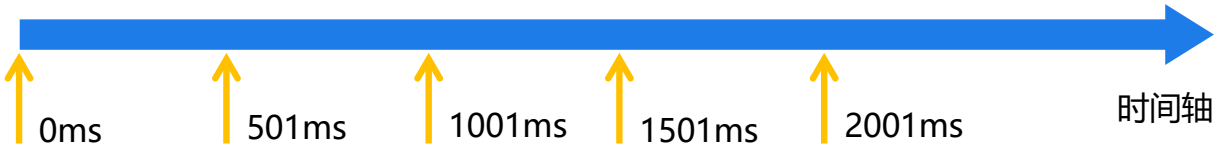
```
while true; \
do curl -X GET "http://localhost:8001/hi" ;\
done;
```

2. 排队等待

排队等待的处理机制：

此方式会严格控制请求通过的间隔时间，让请求以**均匀**的速度通过，对应的是漏桶算法。

例如阈值 QPS = 2，那么就是 500ms 才允许通过下一个请求。



步骤：

1. 构建1个测试接口
2. Sentinel Console 中对接口设置“排队等待”限流，阈值设为 1 QPS，超时时间设为 1000 毫秒
3. 验证限流生效

2. 排队等待

排队等待限流设置界面

2. 排队等待

流控效果验证：

1. 持续访问 /hello 接口，会看到匀速通过的效果

例如通过脚本持续访问 /hello：

```
while true; \
do curl -X GET "http://localhost:8001/hello?name=a" ;\
done;
```

预热与排队等待-总结

重难点

1. 预热限流、排队等待限流的使用场景与区别
2. 预热限流的设置方式，与验证方式
3. 排队等待限流的设置方式，与验证方式

目录 Contents

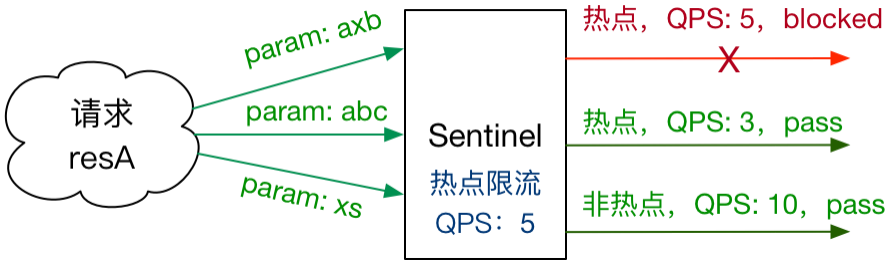
热点限流与系统限流

小节导学

/product/query?id=1&name=phone&price=100

这个API中包括3个参数，假设其中“name”参数是使用最多的，这个接口的绝大部分请求都含有“name”参数，可以称其为“**热点参数**”。

我们可以针对这个热点参数设置接口的限流，当请求带有此参数时，如果达到阈值就触发限流，否则不限流。



热点限流与系统限流

小节导学

之前的限流规则都是针对接口资源的，如果每个资源的阈值都没有达到，但 **系统能力不足了怎么办?**
所以，我们需要针对系统情况来设置一定的规则，系统保护规则是应用整体维度的，而不是资源维度的。

系统保护规则是从应用级别的入口流量进行控制，从单台机器的 load、CPU 使用率、平均 RT、入口 QPS 和并发线程数等几个维度监控应用指标，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

本节我们学习一下如何做热点限流，系统限流中各种维度的设置方式。

- 热点限流
- 系统限流

1. 热点限流

热点规则机制

```
@GetMapping("hotparam")
@SentinelResource(value = "hotparam")
public String hotparam(String type, String name){ ... }
```

参数索引从0开始，按照接口中声明的顺序，此例子中“0”指“type”，“1”指“name”。
可以对资源 hotparam 设置热点限流，例如参数 0 的阈值为 1，窗口时长 1秒，一旦在时间窗口内，带有指定索引的参数的 QPS 达到阈值后，会触发限流。
还可以对此参数的值进行特殊控制，例如参数值为“5”或者“10”时的阈值，根据值独立控制阈值。

实践步骤：

1. 构建1个测试接口
2. Sentinel Console 中对接口设置“热点规则”，索引索引为0，阈值为 1 QPS，统计窗口时长 1秒
3. 验证限流生效

1. 热点限流

热点规则机制

参数索引从0开始，按照接口中声明的顺序

```
public String hotparam(String type, String name){ ... }
```

限流目标：

- 某个参数
- 某个参数的值

注意：

- 热点限流只支持“QPS 限流模式”
- 针对参数值时，参数类型必须是基本类型（byte int long float double boolean char）或者 String

1. 热点限流

热点规则设置界面

1. 热点限流

流控效果验证：

1. 持续访问接口 /hotparam?type=1，会看到限速的效果

例如通过脚本持续访问 /hello :

```
while true; \
do curl -X GET "http://localhost:8001/hotparam?type=1" ;\
done;
```

2. 访问接口 /hotparam?name=a，应可以正常访问

2. 系统限流

系统规则模式

- **Load 自适应**（仅对 Linux/Unix-like 机器生效）：系统的 load1 作为限流指标，进行自适应系统保护。当系统 load1 超过设定的阈值，且系统当前的并发线程数超过估算的系统容量时才会触发系统保护。

  系统容量由系统的 maxQps * minRt 估算得出。

  设定参考值： **CPU cores * 2.5**。
- **CPU 使用率**：当系统 CPU 使用率超过阈值 即触发系统保护（取值范围 0.0-1.0），比较灵敏。
- **平均 RT**：当单台机器上所有入口流量的平均 RT 达到阈值 即触发系统保护，单位是毫秒。
- **并发线程数**：当单台机器上所有入口流量的并发线程数达到阈值 即触发系统保护。
- **入口 QPS**：当单台机器上所有入口流量的 QPS 达到阈值 即触发系统保护。

2. 系统限流

系统规则设置界面

2. 系统限流

RT 效果验证：

1. 阈值设置为 10
2. 持续访问接口 /test-a（其中有睡眠时间），应看到限流的效果

例如通过脚本持续访问 /test-a：

```
while true; \
do curl -X GET "http://localhost:8001/test-a";\
done;
```

2. 系统限流

入口QPS 效果验证：

1. 阈值设置为 3
2. 批量持续访问3个测试接口，应看到限流的效果

2. 系统限流

并发线程数 效果验证：

1. 阈值设置为 3
2. 编写多线程测试代码访问接口，应看到限流效果

```
public static void main(String[] args) {
    RestTemplate restTemplate = new RestTemplate();
    ExecutorService executor = Executors.newFixedThreadPool(10);
    for (int i = 0; i < 50; i++) {
        executor.submit(() -> {
            try {
                System.out.println(restTemplate.getForObject("http://localhost:8001/test-
b", String.class));
            }catch (Exception e){ System.out.println("exception: " + e.getMessage());}
        });
    }
}
```

热点限流与系统限流-总结

重难点

1. 热点限流、系统限流的概念与使用场景
2. 热点限流的设置方式，与验证方式
3. 系统限流中 Load、线程数、RT、CPU使用率、入口QPS 的设置方式，与验证方式

目录 Contents

降级规则

小节导学

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。

Sentinel 熔断降级会在调用链路中某个资源不正常时，对这个资源的调用进行限制，让请求**快速失败**，避免导致级联错误。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**。

本节我们学习如何设置降级规则。

- RT 降级策略
- 异常比例降级策略
- 异常数降级策略

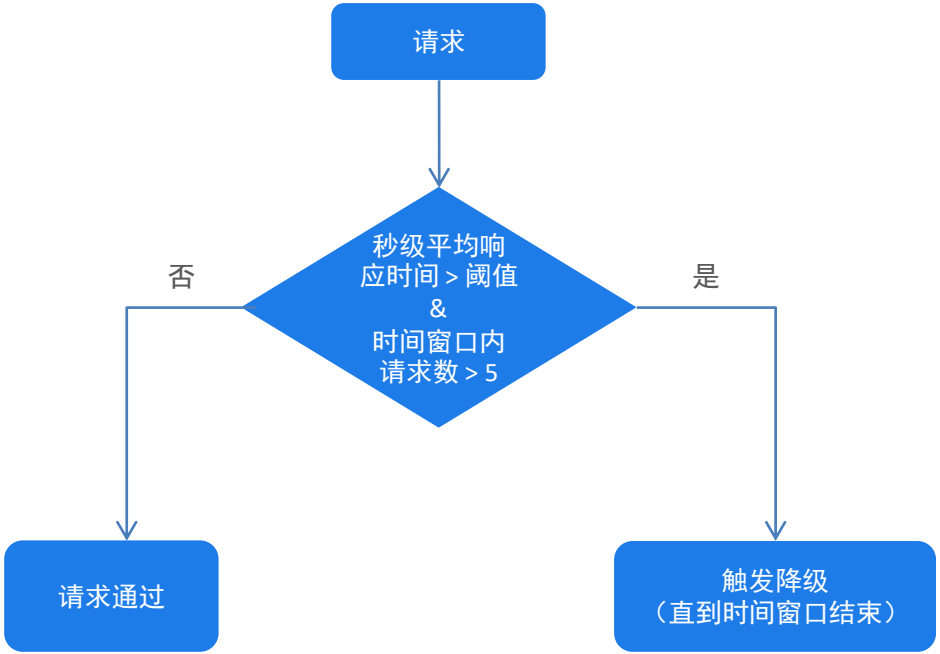
限流

控制上桥人数，防止压塌

降级

桥上有坑，禁止上桥

1. RT 降级策略



1. RT 降级策略

降级规则设置界面

1. RT 降级策略

效果验证：

1. 创建测试接口

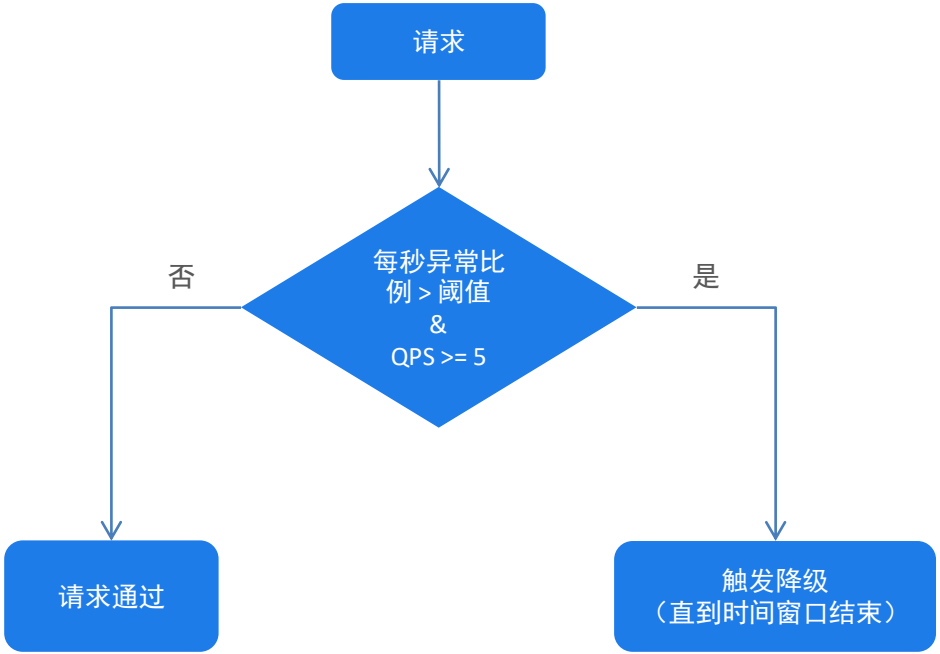
```
@GetMapping("/degrade-rt")
public String test_degrade_rt(){
    try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
    return "ok";
}
```

2. 设置降级规则，阈值 1ms，时间窗口 3秒

3. 持续请求“/degrade-rt”，应显示降级效果

```
while true; \
do curl -X GET "http://localhost:8081/degrade-rt";\
done;
```

2. 异常比例降级策略



2. 异常比例降级策略

降级规则设置界面

2. 异常比例降级策略

效果验证：

1. 创建测试接口

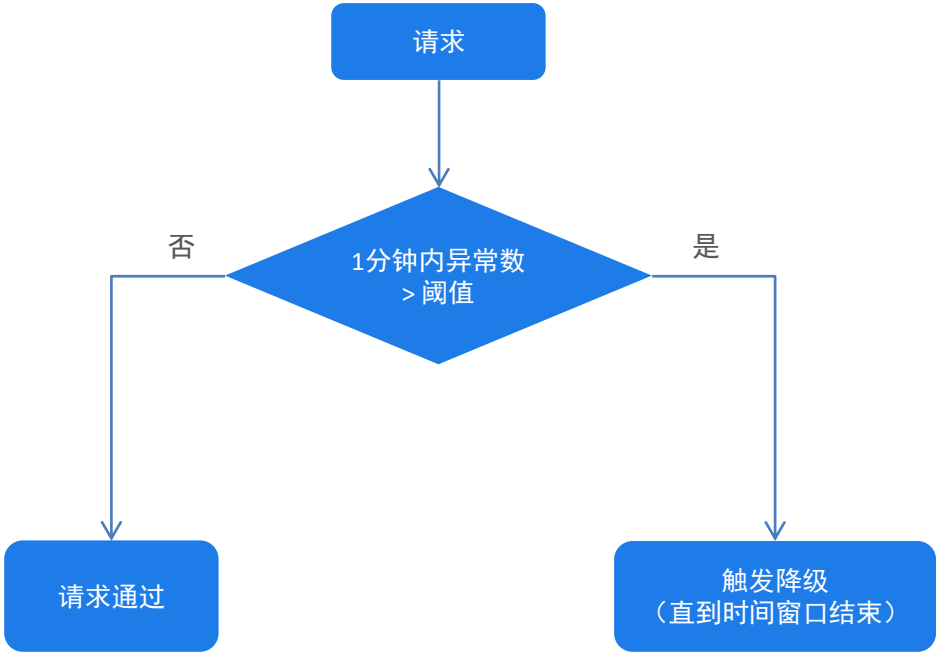
```
@GetMapping("/degrade-exception-rate")
public String test_degrade_exception_rate() throws Exception {
    throw new Exception("rate");
}
```

2. 设置降级规则，阈值 0.1，时间窗口 60秒

3. 持续请求“/degrade-exception-rate”，应显示降级效果

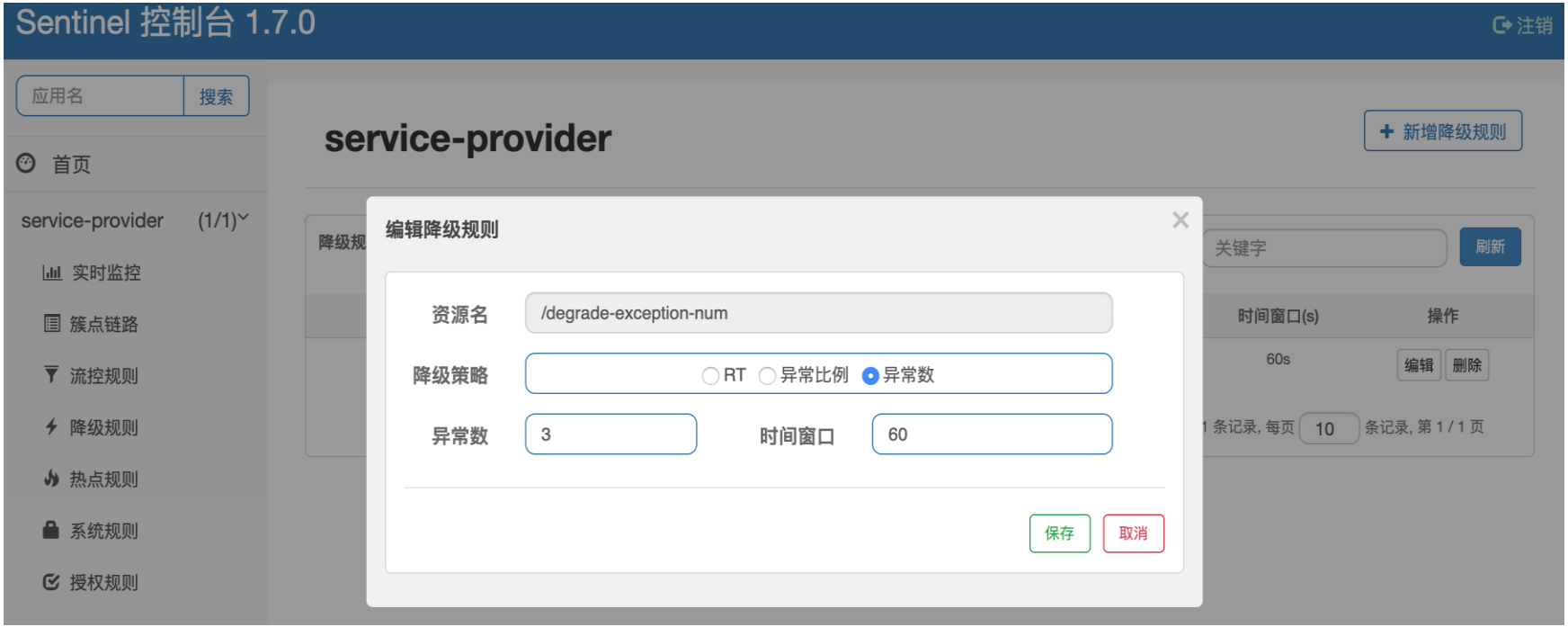
```
while true; \
do curl -X GET "http://localhost:8081/degrade-exception-rate";\
done;
```

3. 异常数降级策略



3. 异常数降级策略

降级规则设置界面



3. 异常数降级策略

效果验证：

1. 创建测试接口

```
@GetMapping("/degrade-exception-num")
public String test_degrade_exception_num() throws Exception {
    throw new Exception("rate");
}
```

2. 设置降级规则，异常数 3，时间窗口 60秒

3. 持续请求“/degrade-exception-num”，应显示降级效果

```
while true; \
do curl -X GET "http://localhost:8001/degrade-exception-num";\
done;
```

降级规则-总结

重难点

1. 降级规则与限流规则的区别
2. RT 降级策略的思路与验证方式
3. 异常比例降级策略的思路与验证方式
4. 异常数降级策略的思路与验证方式

目录 Contents

RestTemplate 与 Feign 整合 Sentinel

小节导学

服务会调用其他服务，调用方式常用的是 RestTemplate 与 Feign，对于 Sentinel 来讲他们都是可以被保护的资源，所以我们需要学习一下 RestTemplate 与 Feign 如何与 Sentinel 整合，还有如何处理限流与降级后的异常。

- RestTemplate 整合 Sentinel
- RestTemplate 限流与降级异常处理
- Feign 整合 Sentinel
- Feign 限流与降级异常处理

1. RestTemplate 整合 Sentinel

整合要点

1. 代码添加 @SentinelRestTemplate 注解

```
@Configuration
public class ServiceConfig {
    @Bean
    @SentinelRestTemplate
    public RestTemplate restTemplate(){ return new RestTemplate(); }
}
```

2. 属性开关

```
resttemplate.sentinel.enabled # true/false（默认 true）
```

1. RestTemplate 整合 Sentinel

整合流程

1. RestTemplate 整合 Sentinel

验证步骤

1. 创建接口，调用 RestTemplate

```
@Autowired
private RestTemplate restTemplate;
@GetMapping("resttemplate_sentinel")
public String resttemplate_sentinel(){
    return restTemplate.getForObject("http://localhost:8001/test-b", String.class);
}
```

2. Sentinel Console 中针对此 RestTemplate 调用添加限流，阈值设为 1
3. 快速多刷几次接口 “/resttemplate_sentinel”，应显示限流效果

2. RestTemplate 限流与降级异常处理

整合要点

1. 定义异常处理类

```
public class ExceptionUtil {
    public static SentinelClientHttpResponse handleException(HttpRequest request,
                                                             byte[] body, ClientHttpRequestExecution execution,
                                                             BlockException ex) {
        System.err.println("Oops: " + ex.getClass().getCanonicalName());
        return new SentinelClientHttpResponse( blockResponse: "my block info");
    }

    public static SentinelClientHttpResponse handleFallback(HttpRequest request,
                                                            byte[] body, ClientHttpRequestExecution execution,
                                                            BlockException ex) {
        System.err.println("fallback: " + ex.getClass().getCanonicalName());
        return new SentinelClientHttpResponse( blockResponse: "my fallback info");
    }
}
```

2. @SentinelRestTemplate 注解中添加异常处理的配置

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

2. RestTemplate 限流与降级异常处理

验证步骤

1. Sentinel Console 中对此 RestTemplate 调用添加限流，阈值设为 1
2. 快速多次访问“/resttemplate_sentinel”，应看到自定义的限流信息

```
my block info
```

3. Sentinel Console 中对此 RestTemplate 调用添加 RT 类型的降级规则，RT 设为 1，时间窗口设为 1
4. 快速多次访问“/resttemplate_sentinel”，应看到自定义的降级信息

```
my fallback info
```

3. Feign 整合 Sentinel

整合要点

1. 属性配置 Feign Sentinel 开关

```
feign:
  sentinel:
    enabled: true
```

验证步骤

1. 创建 service-provider 与 service-consumer，整合 nacos、Sentinel、Feign
1. Sentinel Console 中针对此 Feign 调用添加限流，阈值设为 1
3. 快速多刷几次接口 “/hellofeign”，应显示限流效果

3. Feign 整合 Sentinel

整合流程

2. Feign 限流与降级异常处理

整合要点

1. 定义异常处理类

```
@Component
public class FeignClientServiceFallbackFactory implements FallbackFactory<FeignClientService> {
    @Override
    public FeignClientService create(Throwable throwable) {
        return new FeignClientService() {
            @Override
            public String hello(String name) {
                System.out.println("远程调用出错了：");
                System.out.println(throwable);
                return "fallback default name";
            }
        };
    }
}
```

2. @FeignClient 注解中添加异常处理的配置

```
@FeignClient(name = "service-provider",
        fallbackFactory = FeignClientServiceFallbackFactory.class
)
```

RestTemplate 与 Feign 整合 Sentinel-总结

重难点

1. RestTemplate 整合 Sentinel 的方式，Sentinel Console 设置 RestTemplate 限流与降级的方式
2. RestTemplate 限流降级异常处理方式
3. Feign 整合 Sentinel 的方式，Sentinel Console 设置 Feign 限流与降级的方式
4. Feign 限流降级异常处理方式

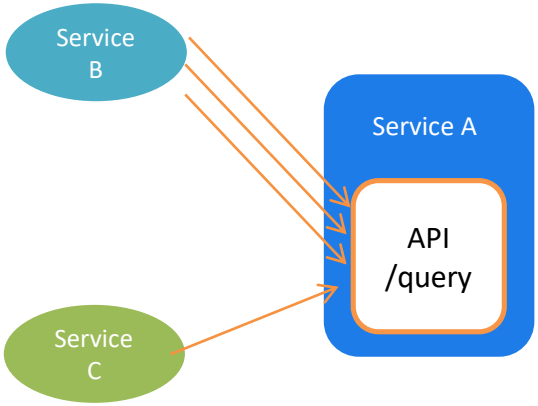
目录 Contents

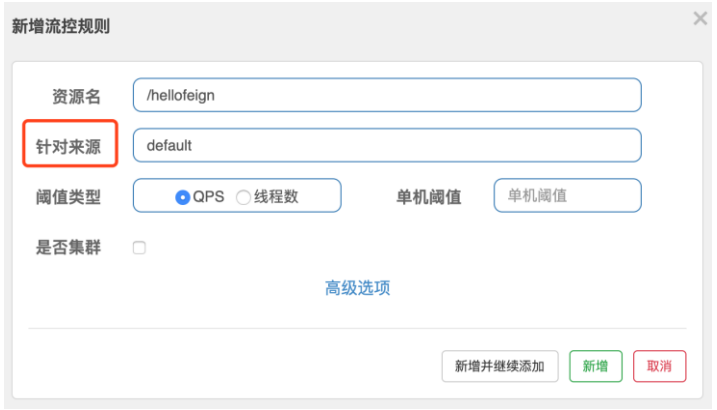
错误信息自定义与区分来源

小节导学

之前我们添加限流规则时，有一项“针对来源”一直都是使用的默认“default”，就是不区分来源。

假设一个场景，service A 的API“/query”的主要调用者是 service B，如果不区分来源，肯定会对 Service C 不公平。需要**对 Service B 特殊关照**，这就可以使用“**针对来源**”。





错误信息自定义与区分来源

小节导学

之前我们被限流或者降级后，看到的提示信息都是：

```
Blocked by Sentinel (flow limiting)
```

效果演示时没问题，但实际环境下这种信息会让我们比较困扰，**到底是被限流了，还是被降级了呢？**如果能针对不同的情况给出不同的信息就好了，这个需求比较好实现。

本节我们就解决这两个问题，学习如何判别来源，和自定义异常信息。

- 定义针对来源
- 根据错误类型自定义错误信息

1. 针对来源

实现方法

Sentinel 提供了“RequestOriginParser”，我们只需要实现其中的“parseOrigin”方法，提供获取来源的方法即可，例如：

```
@Component
public class MyRequestOriginParser implements RequestOriginParser {
    @Override
    public String parseOrigin(HttpServletRequest httpServletRequest) {
        return httpServletRequest.getParameter("from");
    }
}
```

验证步骤

1. Sentinel Console 中为“/hellofeign”设置限流，针对来源设置为“chrome”，阈值设为 1
2. 访问接口 “/hellofeign?name=a&from=chrome”，应显示限流效果

1. 针对来源

授权规则

“针对来源”还有一个用处：用于实现 **授权规则**。

“流控应用”项与“针对来源”是一样的，“白名单”表示允许访问，“黑名单”表示不允许访问。

例如：“流控应用”设为“service-user”，“授权类型”选择“黑名单”，表示“service-user”这个应用不允许访问此资源。

2. 根据错误类型自定义错误信息

Sentinel 已经定义了不同类型的异常，包括 FlowException、DegradeException、ParamFlowException、SystemBlockException、AuthorityException。

我们只需要定义一个异常处理类，针对不同的异常做不同的处理即可，例如：

```
@Component
public class MyUrlBlockHandler implements UrlBlockHandler {
    @Override
    public void blocked(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse,
                        BlockException e) throws IOException {
        String errmsg = null;
        if(e instanceof FlowException){
            errmsg = "限流了";
        }else if (e instanceof DegradeException){
            errmsg = "降级了";
        }else if (e instanceof ParamFlowException){
            errmsg = "热点参数限流了";
        }else if (e instanceof SystemBlockException){
            errmsg = "超出系统规则";
        }else if (e instanceof AuthorityException){
            errmsg = "授权不允许";
        }
        httpServletResponse.setStatus(500);
        httpServletResponse.setCharacterEncoding("utf-8");
        httpServletResponse.setHeader( s: "Content-Type",  s1: "application/json;charset=utf-8");
        httpServletResponse.setContentType("application/json;charset=utf-8");
        new ObjectMapper().writeValue(httpServletResponse.getWriter(), errmsg);
    }
}
```

错误信息自定义与区分来源-总结

重难点

1. Sentinel Console 中设置限流时，“针对来源”这项的意义
2. 代码中解析请求来源
3. 根据错误类型自定义错误处理逻辑

目录 Contents

规则持久化

小节导学

之前我们实践 Sentinel 时，每次我们重启应用 Sentinel 控制台中的规则就都没有了，需要重新设置，这是为什么？

因为应用的 Sentinel 规则是**保存在内存**中的。生产环境中，我们需要做好**规则持久化**。

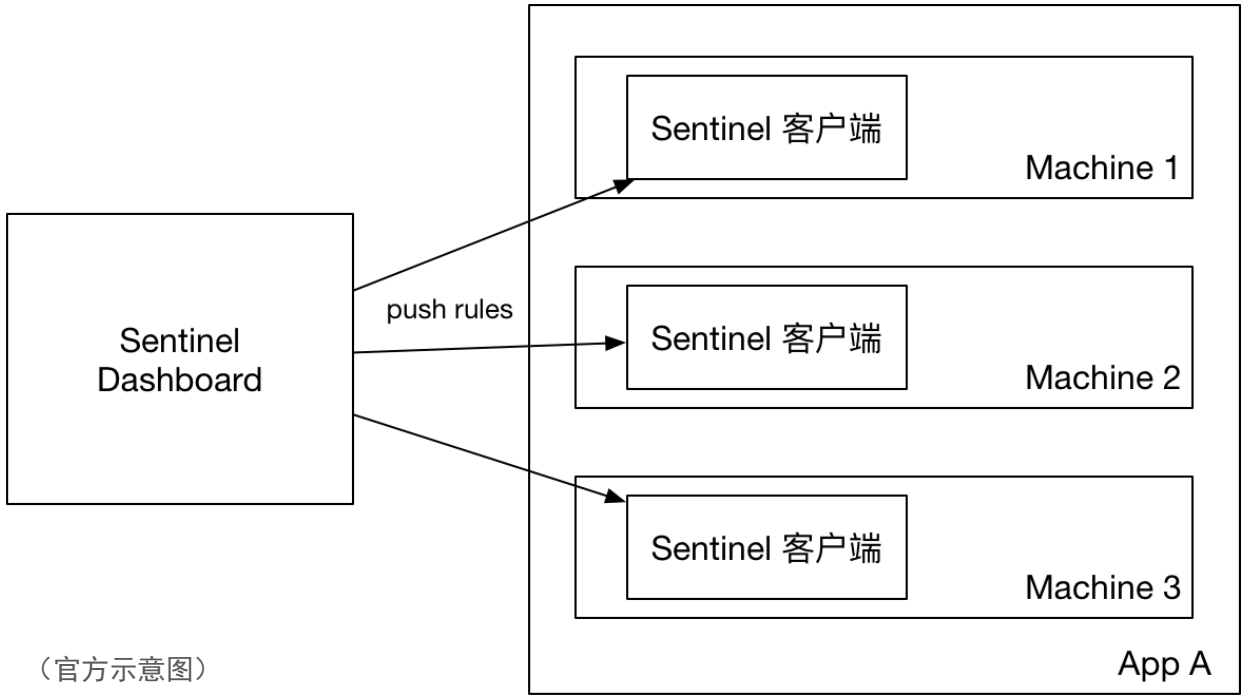
Sentinel 为我们提供了丰富的数据源工具包，便于集成各类数据源，我们需要自己开发代码进行对接。

本节我们分析一下 Sentinel 规则的推送模式，以及如何实现规则的持久化。

- Sentinel 规则推送模式
- Sentinel 整合 Nacos 实现规则持久化

1. Sentinel 规则推送模式

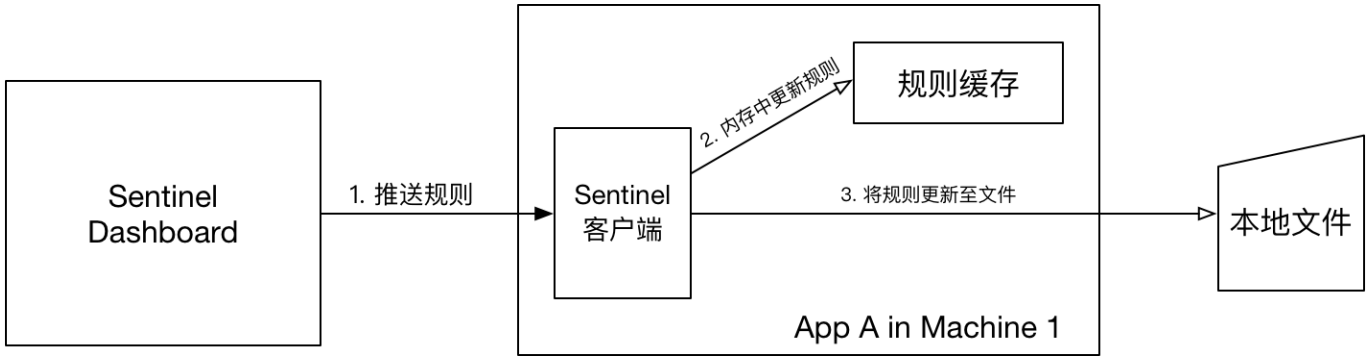
原始模式



（官方示意图）

1. Sentinel 规则推送模式

拉模式



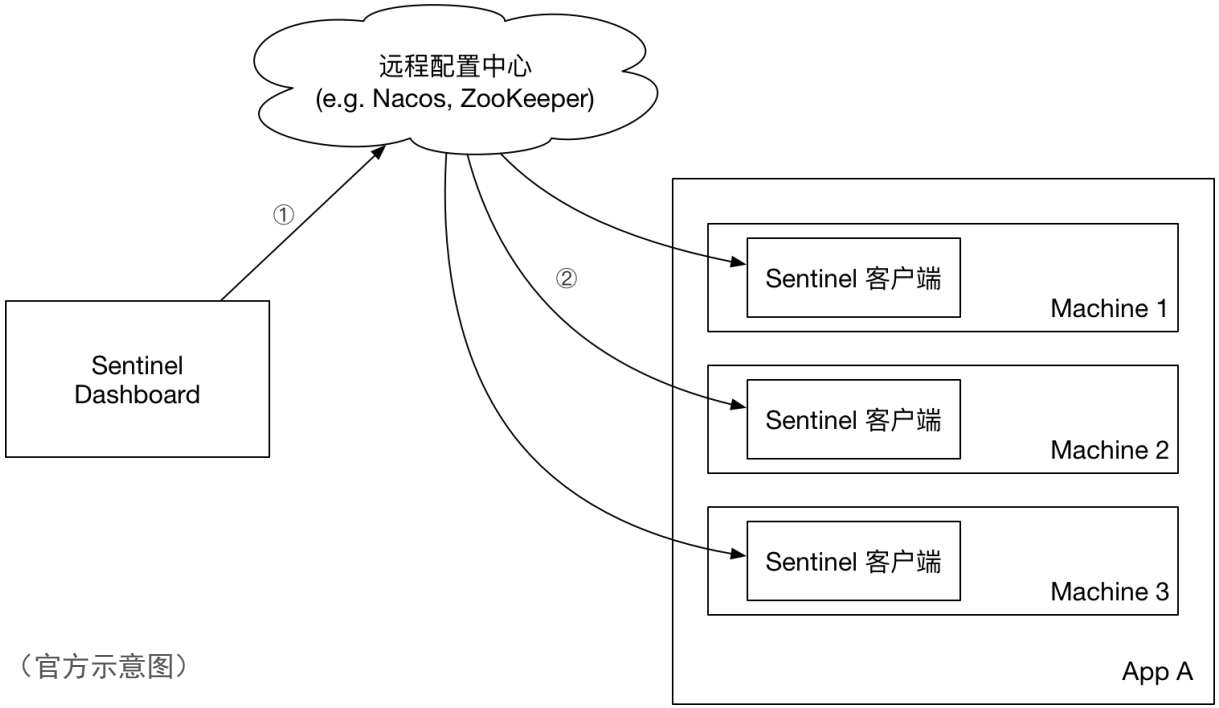
（官方示意图）

支持的数据源

- 文件
- Consul

1. Sentinel 规则推送模式

推模式



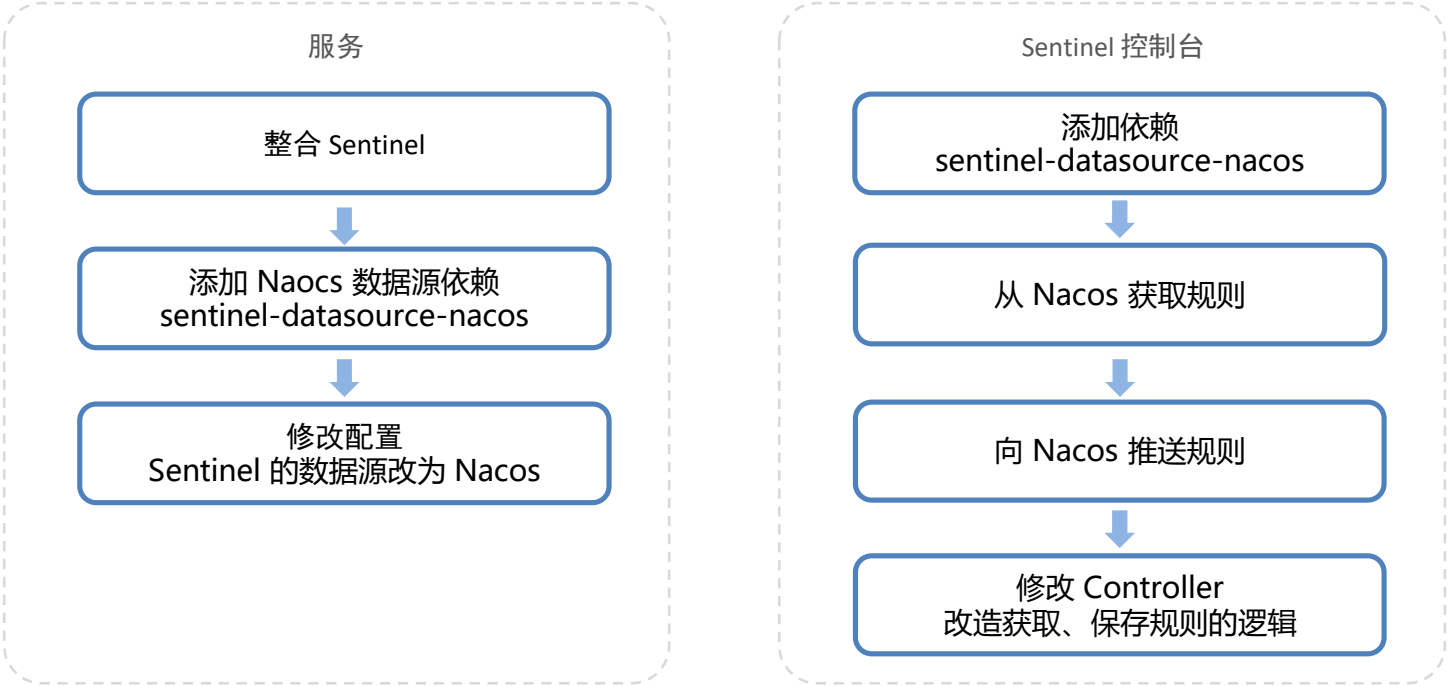
（官方示意图）

支持的数据源

- ZooKeeper
- Redis
- Nacos
- Apollo
- etcd

2. Sentinel 整合 Nacos 实现规则持久化

改造流程



规则持久化-总结

重难点

1. Sentinel 持久化的结构
2. 持久化改造的思路
3. 各类型规则的改造步骤

一样的在线教育，不一样的教学品质