

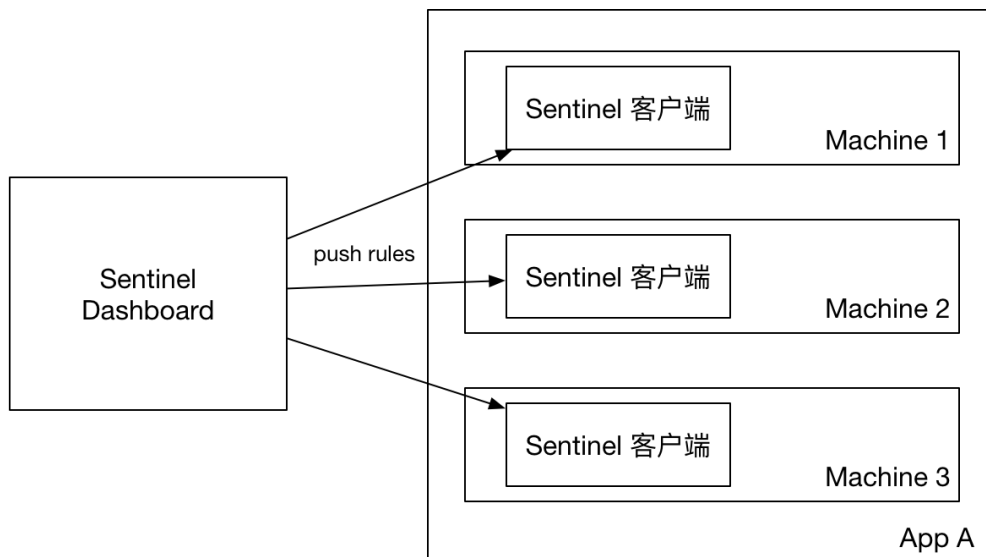
1. Sentinel规则持久化模式

Sentinel规则的推送有下面三种模式:

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)	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。生产环境下一一般采用 push 模式的数据源。	规则持久化; 一致性; 快速	引入第三方依赖

1.1 原始模式

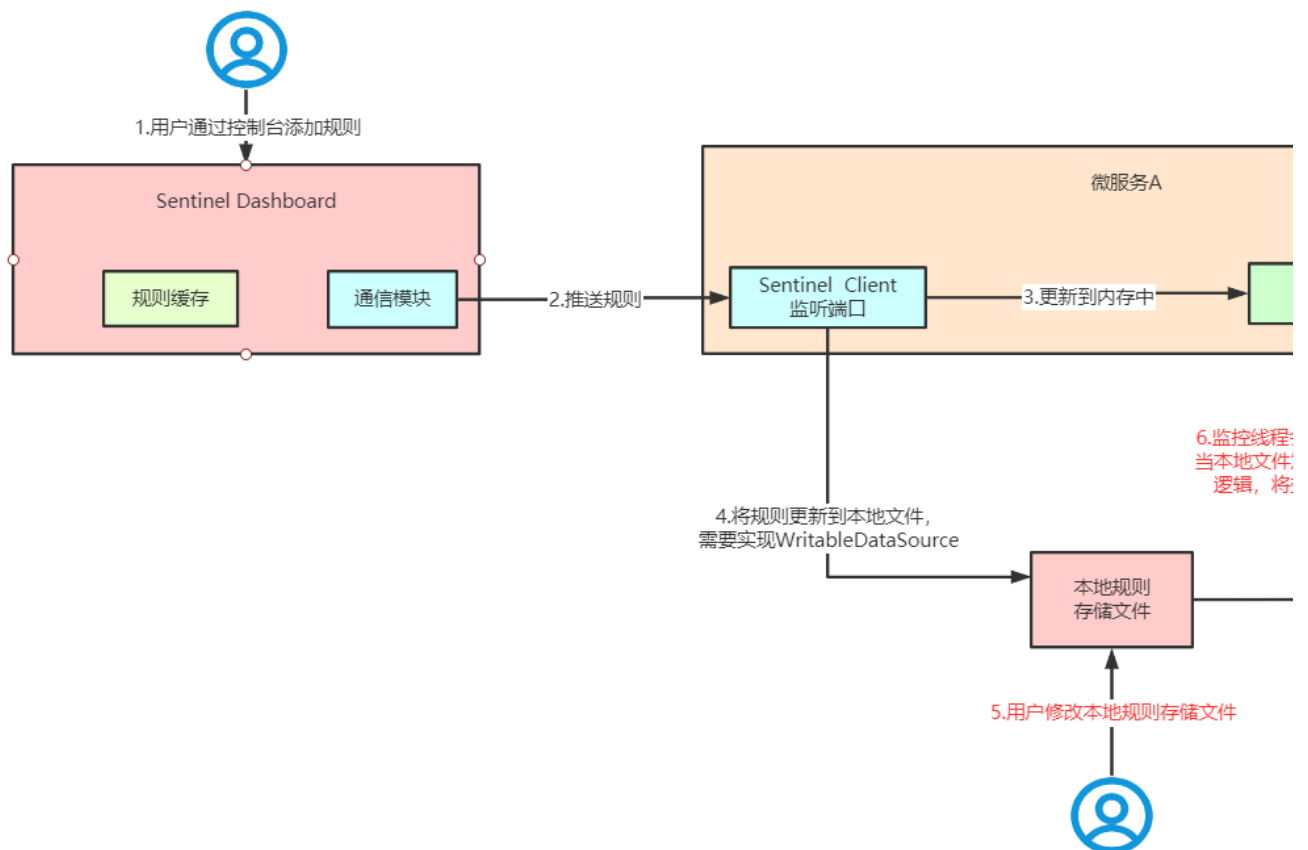
如果不做任何修改, Dashboard 的推送规则方式是通过 API 将规则推送至客户端并直接更新到内存中:



这种做法的好处是简单, 无依赖; 坏处是应用重启规则就会消失, 仅用于简单测试, 不能用于生产环境。

1.2 拉模式

pull 模式的数据源 (如本地文件、RDBMS 等) 一般是可写入的。使用时需要在客户端注册数据源: 将对应的读数据源注册至对应的 RuleManager, 将写数据源注册至 transport 的 WritableDataSourceRegistry 中。



首先 Sentinel 控制台通过 API 将规则推送至客户端并更新到内存中，接着注册的写数据源会将新的规则保存到本地的文件中。使用 pull 模式的数据源时一般不需要对 Sentinel 控制台进行改造。这种实现方法好处是简单，坏处是无法保证监控数据的一致性。

官方demo: [sentinel-demo/sentinel-demo-dynamic-file-rule](#)

引入依赖

```

1 <dependency>
2   <groupId>com.alibaba.csp</groupId>
3   <artifactId>sentinel-datasource-extension</artifactId>
4   <version>1.8.0</version>
5 </dependency>

```

核心代码:

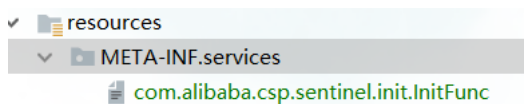
```

1 // FileRefreshableDataSource 会周期性的读取文件以获取规则，当文件有更新时会及时发现，并将规则更新到内存中。
2 ReadableDataSource<String, List<FlowRule>> ds = new FileRefreshableDataSource<>(
3   flowRulePath, source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {})
4 );
5 // 将可读数据源注册至 FlowRuleManager.
6 FlowRuleManager.register2Property(ds.getProperty());
7
8 WritableDataSource<List<FlowRule>> wds = new FileWritableDataSource<>(flowRulePath, this::encodeJson);
9 // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
10 // 这样收到控制台推送的规则时，Sentinel 会先更新到内存，然后将规则写入到文件中.
11 WritableDataSourceRegistry.registerFlowDataSource(wds);

```

拉模式改造

实现InitFunc接口，在init中处理DataSource初始化逻辑，并利用spi机制实现加载。



其中部分核心代码

```

1 public class FileDataSourceInit implements InitFunc {
2

```

```

3  @Override
4  public void init() throws Exception {
5  //创建文件存储目录
6  RuleFileUtils.mkdirIfNotExists(PersistenceRuleConstant.storePath);
7
8  //创建规则文件
9  RuleFileUtils.createFileIfNotExists(PersistenceRuleConstant.rulesMap);
10
11 //处理流控规则逻辑
12 dealFlowRules();
13 // 处理降级规则
14 dealDegradeRules();
15 // 处理系统规则
16 dealSystemRules();
17 // 处理热点参数规则
18 dealParamFlowRules();
19 // 处理授权规则
20 dealAuthRules();
21 }
22
23
24 private void dealFlowRules() throws FileNotFoundException {
25 String ruleFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.FLOW_RULE_PATH).toString();
26
27 //创建流控规则的可读数据源
28 ReadableDataSource<String, List<FlowRule>> flowRuleRDS = new FileRefreshableDataSource(
29 ruleFilePath, RuleListConverterUtils.flowRuleListParser
30 );
31
32 // 将可读数据源注册至FlowRuleManager 这样当规则文件发生变化时, 就会更新规则到内存
33 FlowRuleManager.register2Property(flowRuleRDS.getProperty());
34
35
36 WritableDataSource<List<FlowRule>> flowRuleWDS = new FileWritableDataSource<List<FlowRule>>(
37 ruleFilePath, RuleListConverterUtils.flowRuleEncoding
38 );
39
40 // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
41 // 这样收到控制台推送的规则时, Sentinel 会先更新到内存, 然后将规则写入到文件中.
42 WritableDataSourceRegistry.registerFlowDataSource(flowRuleWDS);
43 }
44
45 private void dealDegradeRules() throws FileNotFoundException {
46 //讲解规则文件路径
47 String degradeRuleFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.DEGRADE_RULE_PATH).toString();
48
49 //创建流控规则的可读数据源
50 ReadableDataSource<String, List<DegradeRule>> degradeRuleRDS = new FileRefreshableDataSource(
51 degradeRuleFilePath, RuleListConverterUtils.degradeRuleListParser
52 );
53
54 // 将可读数据源注册至FlowRuleManager 这样当规则文件发生变化时, 就会更新规则到内存
55 DegradeRuleManager.register2Property(degradeRuleRDS.getProperty());
56
57
58 WritableDataSource<List<DegradeRule>> degradeRuleWDS = new FileWritableDataSource<>(
59 degradeRuleFilePath, RuleListConverterUtils.degradeRuleEncoding
60 );
61
62 // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
63 // 这样收到控制台推送的规则时, Sentinel 会先更新到内存, 然后将规则写入到文件中.

```

```

64 WritableDataSourceRegistry.registerDegradeDataSource(degradeRuleWDS);
65 }
66
67 private void dealSystemRules() throws FileNotFoundException {
68 //讲解规则文件路径
69 String systemRuleFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.SYSTEM_RULE_PATH).toString();
70
71 //创建流控规则的可读数据源
72 ReadableDataSource<String, List<SystemRule>> systemRuleRDS = new FileRefreshableDataSource(
73 systemRuleFilePath, RuleListConverterUtils.sysRuleListParse
74 );
75
76 // 将可读数据源注册至FlowRuleManager 这样当规则文件发生变化时, 就会更新规则到内存
77 SystemRuleManager.register2Property(systemRuleRDS.getProperty());
78
79
80 WritableDataSource<List<SystemRule>> systemRuleWDS = new FileWritableDataSource<>(
81 systemRuleFilePath, RuleListConverterUtils.sysRuleEnCoding
82 );
83
84 // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
85 // 这样收到控制台推送的规则时, Sentinel 会先更新到内存, 然后将规则写入到文件中.
86 WritableDataSourceRegistry.registerSystemDataSource(systemRuleWDS);
87 }
88
89
90 private void dealParamFlowRules() throws FileNotFoundException {
91 //讲解规则文件路径
92 String paramFlowRuleFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.HOT_PARAM_RULE).toString();
93
94 //创建流控规则的可读数据源
95 ReadableDataSource<String, List<ParamFlowRule>> paramFlowRuleRDS = new FileRefreshableDataSource(
96 paramFlowRuleFilePath, RuleListConverterUtils.paramFlowRuleListParse
97 );
98
99 // 将可读数据源注册至FlowRuleManager 这样当规则文件发生变化时, 就会更新规则到内存
100 ParamFlowRuleManager.register2Property(paramFlowRuleRDS.getProperty());
101
102
103 WritableDataSource<List<ParamFlowRule>> paramFlowRuleWDS = new FileWritableDataSource<>(
104 paramFlowRuleFilePath, RuleListConverterUtils.paramRuleEnCoding
105 );
106
107 // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
108 // 这样收到控制台推送的规则时, Sentinel 会先更新到内存, 然后将规则写入到文件中.
109 ModifyParamFlowRulesCommandHandler.setWritableDataSource(paramFlowRuleWDS);
110 }
111
112 private void dealAuthRules() throws FileNotFoundException {
113 //讲解规则文件路径
114 String authFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.AUTH_RULE_PATH).toString();
115
116 //创建流控规则的可读数据源
117 ReadableDataSource<String, List<AuthorityRule>> authRuleRDS = new FileRefreshableDataSource(
118 authFilePath, RuleListConverterUtils.authorityRuleParse
119 );
120
121 // 将可读数据源注册至FlowRuleManager 这样当规则文件发生变化时, 就会更新规则到内存
122 AuthorityRuleManager.register2Property(authRuleRDS.getProperty());
123

```

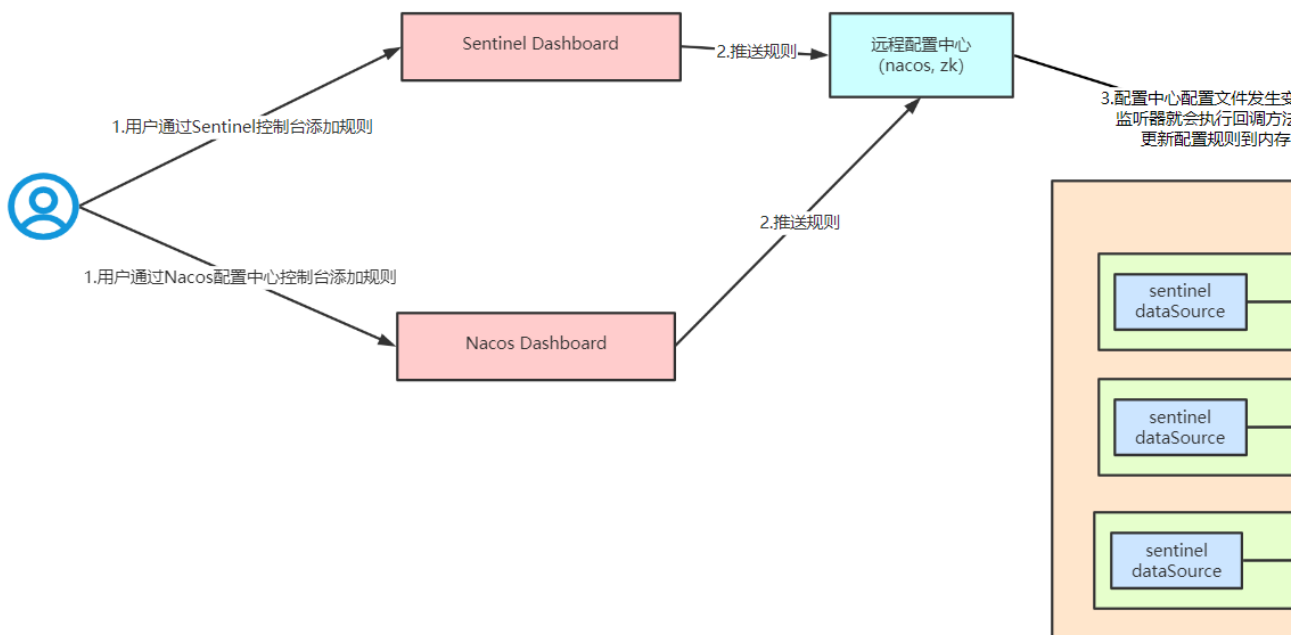
```

124
125 WritableDataSource<List<AuthorityRule>> authRuleWDS = new FileWritableDataSource<>{
126     authFilePath, RuleListConverterUtils.authorityEncoding
127 };
128
129 // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中。
130 // 这样收到控制台推送的规则时, Sentinel 会先更新到内存, 然后将规则写入到文件中。
131 WritableDataSourceRegistry.registerAuthorityDataSource(authRuleWDS);
132 }
133
134 }

```

1.3 推模式

生产环境下一般更常用的是 push 模式的数据源。对于 push 模式的数据源,如远程配置中心 (ZooKeeper, Nacos, Apollo 等等), 推送的操作不应由 Sentinel 客户端进行, 而应该经控制台统一进行管理, 直接进行推送, 数据源仅负责获取配置中心推送的配置并更新到本地。因此推送规则正确做法应该是 **配置中心控制台/Sentinel 控制台** → **配置中心** → **Sentinel 数据源** → **Sentinel**, 而不是经 Sentinel 数据源推送至配置中心。这样的流程就非常清晰了:



1.3.1 基于Nacos配置中心控制台实现推送

官方demo: sentinel-demo-nacos-datasource

引入依赖

```

1 <dependency>
2 <groupId>com.alibaba.csp</groupId>
3 <artifactId>sentinel-datasource-nacos</artifactId>
4 <version>1.8.0</version>
5 </dependency>

```

核心代码

```

1 // nacos server ip
2 private static final String remoteAddress = "localhost:8848";
3 // nacos group
4 private static final String groupId = "Sentinel:Demo";
5 // nacos dataId
6 private static final String dataId = "com.alibaba.csp.sentinel.demo.flow.rule";
7 ReadableDataSource<String, List<FlowRule>> flowRuleDataSource = new NacosDataSource<>(remoteAddress, groupId,
8     dataId, source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {}));
9 FlowRuleManager.register2Property(flowRuleDataSource.getProperty());

```

nacos配置中心中配置流控规则

```

1 [

```

```
2 {
3   "resource": "TestResource",
4   "controlBehavior": 0,
5   "count": 10.0,
6   "grade": 1,
7   "limitApp": "default",
8   "strategy": 0
9 }
10 ]
```

* 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": "TestResource",
4     "controlBehavior": 0,
5     "count": 15.0,
6     "grade": 1,
7     "limitApp": "default",
8     "strategy": 0
9   }
10 ]
```

微服务中通过yml配置实现

SentinelProperties 内部提供了 TreeMap 类型的 datasource 属性用于配置数据源信息

1) 引入依赖

```
1 <!--sentinel持久化 采用 Nacos 作为规则配置数据源-->
2 <dependency>
3   <groupId>com.alibaba.csp</groupId>
4   <artifactId>sentinel-datasource-nacos</artifactId>
5 </dependency>
```

2) yml中配置

```
1 spring:
2   application:
3     name: mall-user-sentinel-demo
4   cloud:
5     nacos:
6     discovery:
7       server-addr: 127.0.0.1:8848
8
9   sentinel:
10    transport:
11      # 添加sentinel的控制台地址
12    dashboard: 127.0.0.1:8080
13    # 指定应用与Sentinel控制台交互的端口, 应用本地会起一个该端口占用的HttpServer
14    port: 8719
15    datasource:
16      ds1:
17        nacos:
```

```
18 server-addr: 127.0.0.1:8848
19 dataId: ${spring.application.name}
20 groupId: DEFAULT_GROUP
21 data-type: json
22 rule-type: flow
```

源码参考 [com.alibaba.cloud.sentinel.datasource.config.AbstractDataSourceProperties#postRegister](#)

```
public void postRegister(AbstractDataSource dataSource) {
    switch (this.getRuleType()) {
        case FLOW:
            FlowRuleManager.register2Property(dataSource.getProperty());
            break;
        case DEGRADE:
            DegradeRuleManager.register2Property(dataSource.getProperty());
            break;
        case PARAM_FLOW:
            ParamFlowRuleManager.register2Property(dataSource.getProperty());
            break;
        case SYSTEM:
            SystemRuleManager.register2Property(dataSource.getProperty());
            break;
        case AUTHORITY:
            AuthorityRuleManager.register2Property(dataSource.getProperty());
            break;
        case GW_FLOW:
            GatewayRuleManager.register2Property(dataSource.getProperty());
            break;
        case GW_API_GROUP:
            GatewayApiDefinitionManager.register2Property(dataSource.getProperty());
    }
}
```

3) nacos配置中心中添加

```
1 [
2 {
3   "resource": "userinfo",
4   "limitApp": "default",
5   "grade": 1,
6   "count": 1,
7   "strategy": 0,
8   "controlBehavior": 0,
9   "clusterMode": false
10 }
11 ]
```

* 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": "userinfo",
4   "limitApp": "default",
5   "grade": 1,
6   "count": 1,
7   "strategy": 0,
8   "controlBehavior": 0,
9   "clusterMode": false
10 }
11 ]
```

缺点: 直接在Sentinel Dashboard中修改规则配置, 配置中心的配置不会发生变化

思考: 如何实现将通过sentinel控制台设置的规则直接持久化到nacos配置中心?

扩展改造的思路：

Sentinel Dashboard监听Nacos配置的变化，如发生变化就更新本地缓存。在Sentinel Dashboard端新增或修改规则配置在保存到内存的同时，直接发布配置到nacos配置中心；Sentinel Dashboard直接从nacos拉取所有的规则配置。sentinel Dashboard和sentinel client 不直接通信，而是通过nacos配置中心获取到配置的变更。

1.3.2 基于Sentinel控制台实现推送

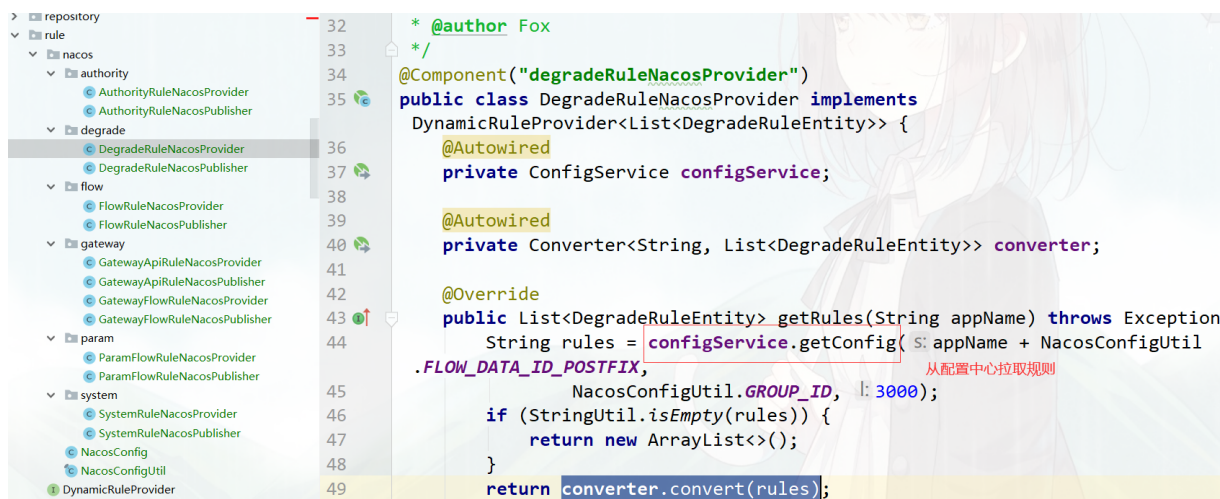
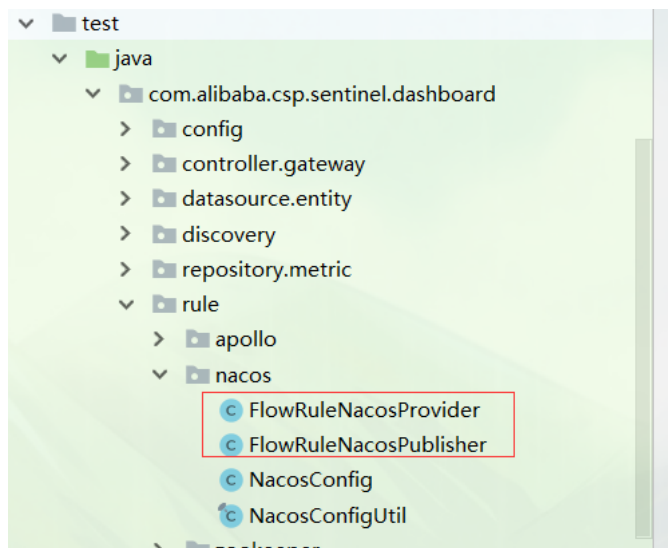
从 Sentinel 1.4.0 开始，Sentinel 控制台提供 `DynamicRulePublisher` 和 `DynamicRuleProvider` 接口用于实现应用维度的规则推送和拉取：

- `DynamicRuleProvider<T>`: 拉取规则
- `DynamicRulePublisher<T>`: 推送规则

Sentinel Dashboard改造

第1步：在`com.alibaba.csp.sentinel.dashboard.rule`包下创建`nacos`包，然后把各种场景的配置规则拉取和推送的实现类写到此包下

可以参考Sentinel Dashboard test包下的流控规则拉取和推送的实现逻辑：



```

36 @Autowired
37 private ConfigService configService;
38
39 @Autowired
40 private Converter<List<DegradeRuleEntity>, String> converter;
41
42 @Override
43 public void publish(String app, List<DegradeRuleEntity> rules) throws
44 Exception {
45     AssertUtil.notEmpty(app, message: "app name cannot be empty");
46     if (rules == null) {
47         return;
48     }
49     configService.publishConfig(:"app + NacosConfigUtil
50     .DEGRADE_DATA_ID_POSTFIX,
51     NacosConfigUtil.GROUP_ID, converter.convert(rules));

```

注意：微服务接入Sentinel client, yml配置需要匹配对应的规则后缀

```

public final class NacosConfigUtil {
    public static final String GROUP_ID = "SENTINEL_GROUP"; // 对应groupid
    // 对应dataId的后缀: ${spring.application.name}-flow-rules
    public static final String FLOW_DATA_ID_POSTFIX = "-flow-rules";
    public static final String PARAM_FLOW_DATA_ID_POSTFIX = "-param-flow-rules";
    public static final String DEGRADE_DATA_ID_POSTFIX = "-degrade-rules";
    public static final String SYSTEM_DATA_ID_POSTFIX = "-system-rules";
    public static final String AUTHORITY_DATA_ID_POSTFIX = "-authority-rules";
    public static final String GATEWAY_FLOW_DATA_ID_POSTFIX = "-gateway-flow-rules";
    public static final String GATEWAY_API_DATA_ID_POSTFIX = "-gateway-api-rules";
}

```

第2步：进入com.alibaba.csp.sentinel.dashboard.controller包下修改对应的规则controller实现类

```

59 private final Logger logger = LoggerFactory.getLogger(FlowControllerV1.class);
60
61 @Autowired
62 private InMemoryRuleRepositoryAdapter<FlowRuleEntity> repository;
63
64 @Autowired
65 private SentinelApiClient sentinelApiClient;
66
67 @Autowired
68 @Qualifier("flowRuleNacosProvider")
69 private DynamicRuleProvider<List<FlowRuleEntity>> ruleProvider;
70
71 @Autowired //在sentinel dashboard对应的接口Controller中引入推送和拉取的规则实现类
72 @Qualifier("flowRuleNacosPublisher")
73 private DynamicRulePublisher<List<FlowRuleEntity>> rulePublisher;
74
75 @GetMapping("/rules")
76 @AuthAction(PrivilegeType.READ_RULE)
77 public Result<List<FlowRuleEntity>> apiQueryMachineRules(@RequestParam
78 String app,

```

以流控规则为例，从Nacos配置中心获取所有的流控规则

```

1 @GetMapping("/rules")
2 @AuthAction(PrivilegeType.READ_RULE)
3 public Result<List<FlowRuleEntity>> apiQueryMachineRules(@RequestParam String app,
4 @RequestParam String ip,
5 @RequestParam Integer port) {
6
7     if (StringUtil.isEmpty(app)) {
8         return Result.ofFail(-1, "app can't be null or empty");
9     }
10    if (StringUtil.isEmpty(ip)) {
11        return Result.ofFail(-1, "ip can't be null or empty");
12    }
13    if (port == null) {
14        return Result.ofFail(-1, "port can't be null");
15    }
16    try {
17        // List<FlowRuleEntity> rules = sentinelApiClient.fetchFlowRuleOfMachine(app, ip, port);
18        //从配置中心获取规则配置

```

```

19 List<FlowRuleEntity> rules = ruleProvider.getRules(app, ip, port);
20 rules = repository.saveAll(rules);
21 return Result.ofSuccess(rules);
22 } catch (Throwable throwable) {
23     logger.error("Error when querying flow rules", throwable);
24     return Result.ofThrowable(-1, throwable);
25 }
26 }

```

新增流控规则，会推送到nacos配置中心

```

1 @PostMapping("/rule")
2 @AuthAction(PrivilegeType.WRITE_RULE)
3 public Result<FlowRuleEntity> apiAddFlowRule(@RequestBody FlowRuleEntity entity) {
4     Result<FlowRuleEntity> checkResult = checkEntityInternal(entity);
5     if (checkResult != null) {
6         return checkResult;
7     }
8     entity.setId(null);
9     Date date = new Date();
10    entity.setGmtCreate(date);
11    entity.setGmtModified(date);
12    entity.setLimitApp(entity.getLimitApp().trim());
13    entity.setResource(entity.getResource().trim());
14    try {
15        entity = repository.save(entity);
16
17        //publishRules(entity.getApp(), entity.getIp(), entity.getPort()).get(5000, TimeUnit.MILLISECONDS);
18        //发布规则到配置中心
19        publishRules(entity.getApp());
20        return Result.ofSuccess(entity);
21    } catch (Throwable t) {
22        Throwable e = t instanceof ExecutionException ? t.getCause() : t;
23        logger.error("Failed to add new flow rule, app={}, ip={}", entity.getApp(), entity.getIp(), e);
24        return Result.ofFail(-1, e.getMessage());
25    }
26 }
27
28 /**
29  * 发布到配置中心
30  * @param app
31  * @throws Exception
32  */
33 private void publishRules(/*@NonNull*/ String app) throws Exception {
34     List<FlowRuleEntity> rules = repository.findAllByApp(app);
35     rulePublisher.publish(app, rules);
36 }

```

测试：微服务接入改造后的Sentinel Dashboard

引入依赖

```

1 <!--sentinel持久化 采用 Nacos 作为规则配置数据源-->
2 <dependency>
3 <groupId>com.alibaba.csp</groupId>
4 <artifactId>sentinel-datasource-nacos</artifactId>
5 </dependency>

```

增加yml配置

```

1 server:
2   port: 8806
3
4 spring:
5   application:

```

```
6 name: mall-user-sentinel-rule-push-demo #微服务名称
7
8 #配置nacos注册中心地址
9 cloud:
10 nacos:
11 discovery:
12 server-addr: 127.0.0.1:8848
13
14 sentinel:
15 transport:
16 # 添加sentinel的控制台地址
17 dashboard: 127.0.0.1:8080
18 # 指定应用与Sentinel控制台交互的端口，应用本地会起一个该端口占用的HttpServer
19 #port: 8719
20 datasource:
21 # ds1: #名称自定义，唯一
22 # nacos:
23 # server-addr: 127.0.0.1:8848
24 # dataId: ${spring.application.name}
25 # groupId: DEFAULT_GROUP
26 # data-type: json
27 # rule-type: flow
28 flow-rules:
29 nacos:
30 server-addr: 127.0.0.1:8848
31 dataId: ${spring.application.name}-flow-rules
32 groupId: SENTINEL_GROUP # 注意groupId对应Sentinel Dashboard中的定义
33 data-type: json
34 rule-type: flow
35 degrade-rules:
36 nacos:
37 server-addr: 127.0.0.1:8848
38 dataId: ${spring.application.name}-degrade-rules
39 groupId: SENTINEL_GROUP
40 data-type: json
41 rule-type: degrade
42 param-flow-rules:
43 nacos:
44 server-addr: 127.0.0.1:8848
45 dataId: ${spring.application.name}-param-flow-rules
46 groupId: SENTINEL_GROUP
47 data-type: json
48 rule-type: param-flow
49 authority-rules:
50 nacos:
51 server-addr: 127.0.0.1:8848
52 dataId: ${spring.application.name}-authority-rules
53 groupId: SENTINEL_GROUP
54 data-type: json
55 rule-type: authority
56 system-rules:
57 nacos:
58 server-addr: 127.0.0.1:8848
59 dataId: ${spring.application.name}-system-rules
60 groupId: SENTINEL_GROUP
61 data-type: json
62 rule-type: system
```

以流控规则测试，当在sentinel dashboard配置了流控规则，会在nacos配置中心生成对应的配置。

* Data ID: mall-user-sentinel-rule-push-demo-flow-rules


* Group: SENTINEL_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 all-user-sentinel-rule-push-demo", "clusterConfig": {"fallbackToLocalWhenFail": true, "sampleCount": 10,
2 :0, "thresholdType": 0, "windowIntervalMs": 1000}, "clusterMode": false, "controlBehavior": 0, "count": 2.0,
3 "161595853275, "gmtModified": 1615958961828, "grade": 1, "id": 1, "ip": "192.168.3.1", "limitApp": "default"
4 2, "resource": "/user/findOrderByUserId/{id}", "strategy": 0}]
```

热点参数规则失效和解决思路

注意：控制台改造后有可能出现规则不生效的情况，比如热点参数规则因为Converter解析json错误的原因会导致不生效。

参见源码：com.alibaba.csp.sentinel.datasource.AbstractDataSource#loadConfig(S) 会解析配置规则。

原因是：改造dashboard，提交到nacos配置中心的数据是ParamFlowRuleEntity类型，微服务拉取配置要解析的是ParamFlowRule类型，会导致规则解析丢失数据，造成热点规则不生效。其他的规则原理也是一样，存在失效的风险。

nacos配置中心保存的数据格式：

```
1 [{
2   "app": "mall-user-sentinel-rule-push-demo",
3   "gmtCreate": 1616136838785,
4   "gmtModified": 1616136838785,
5   "id": 1,
6   "ip": "192.168.3.1",
7   "port": 8719,
8   "rule": {
9     "burstCount": 0,
10    "clusterConfig": {
11      "fallbackToLocalWhenFail": true,
12      "sampleCount": 10,
13      "thresholdType": 0,
14      "windowIntervalMs": 1000
15    },
16    "clusterMode": false,
17    "controlBehavior": 0,
18    "count": 1.0,
19    "durationInSec": 1,
20    "grade": 1,
21    "limitApp": "default",
22    "maxQueueingTimeMs": 0,
23    "paramFlowItemList": [],
24    "paramIdx": 1,
25    "resource": "hot"
26  }
27 }, {
28   "app": "mall-user-sentinel-rule-push-demo",
29   "gmtCreate": 1616137178470,
30   "gmtModified": 1616658923519,
31   "id": 2,
32   "ip": "192.168.3.1",
33   "port": 8719,
34   "rule": {
35     "burstCount": 0,
36     "clusterConfig": {
37       "fallbackToLocalWhenFail": true,
38       "sampleCount": 10,
39       "thresholdType": 0,
40       "windowIntervalMs": 1000
41     },
```

```

42 "clusterMode": false,
43 "controlBehavior": 0,
44 "count": 3.0,
45 "durationInSec": 1,
46 "grade": 1,
47 "limitApp": "default",
48 "maxQueueingTimeMs": 0,
49 "paramFlowItemList": [{
50 "classType": "int",
51 "count": 1,
52 "object": "4"
53 }],
54 "paramIdx": 0,
55 "resource": "findOrderByUserId"
56 }
57 }]

```

我提供两种解决思路：

1. 自定义一个解析热点规则配置的解析器FlowParamJsonConverter，继承JsonConverter，重写convert方法。然后利用后置处理器替换beanName为"param-flow-rules-sentinel-nacos-datasource"的converter属性，注入FlowParamJsonConverter。

```

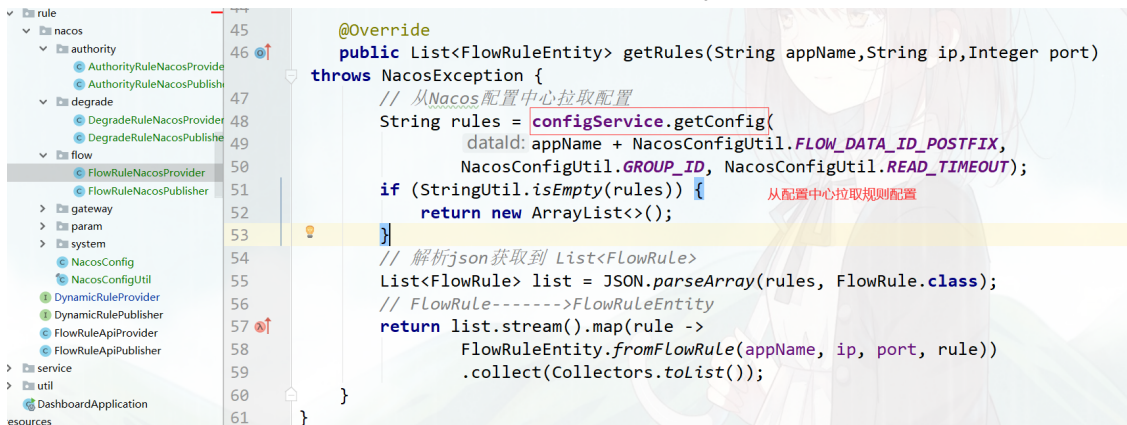
1 @Configuration
2 public class ConverterConfig {
3
4     @Bean("sentinel-json-param-flow-converter2")
5     @Primary
6     public JsonConverter jsonParamFlowConverter() {
7         return new FlowParamJsonConverter(new ObjectMapper(), ParamFlowRule.class);
8     }
9 }
10
11 @Component
12 public class FlowParamConverterBeanPostProcessor implements BeanPostProcessor {
13
14     @Autowired
15     private JsonConverter jsonParamFlowConverter;
16
17     @Override
18     public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
19         if (beanName.equals("param-flow-rules-sentinel-nacos-datasource")) {
20             NacosDataSourceFactoryBean nacosDataSourceFactoryBean = (NacosDataSourceFactoryBean) bean;
21             nacosDataSourceFactoryBean.setConverter(jsonParamFlowConverter);
22             return bean;
23         }
24         return bean;
25     }
26 }
27
28 public class FlowParamJsonConverter extends JsonConverter {
29     Class ruleClass;
30
31     public FlowParamJsonConverter(ObjectMapper objectMapper, Class ruleClass) {
32         super(objectMapper, ruleClass);
33         this.ruleClass = ruleClass;
34     }
35
36     @Override
37     public Collection<Object> convert(String source) {
38         List<Object> list = new ArrayList<>();

```

```
39 JSONArray jsonArray = JSON.parseArray(source);
40 for (int i = 0; i < jsonArray.size(); i++) {
41 //解析rule属性
42 JSONObject jsonObject = (JSONObject) jsonArray.getJSONObject(i).get("rule");
43 Object object = JSON.toJavaObject(jsonObject, ruleClass);
44 list.add(object);
45 }
46 return list;
47 }
48 }
```

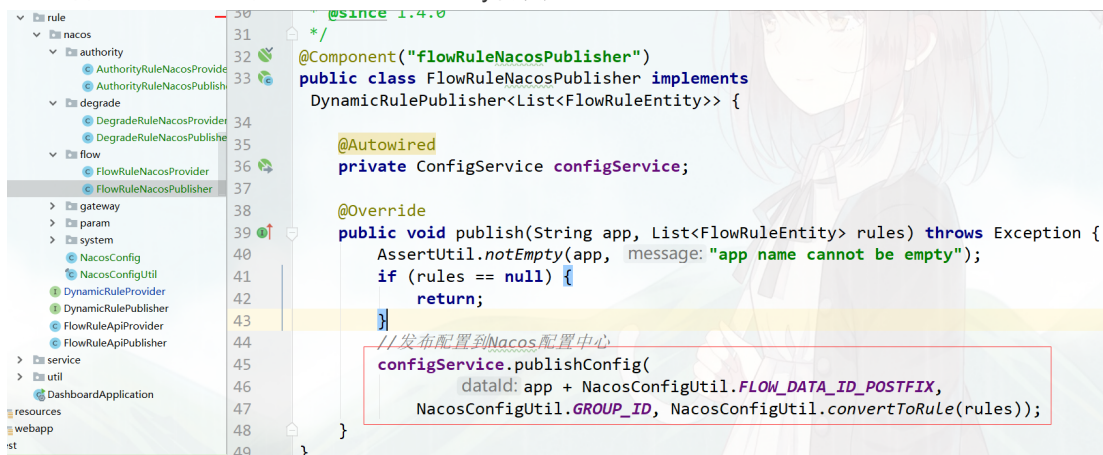
2. 改造Sentinel Dashboard控制台，发布配置时将ParamFlowRuleEntity转成ParamFlowRule类型，再发布到Nacos配置中心。从配置中心拉取配置后将ParamFlowRule转成ParamFlowRuleEntity。

从配置中心拉取配置到控制台时，FlowRule转换为FlowRuleEntity



```
45 @Override
46 public List<FlowRuleEntity> getRules(String appName,String ip,Integer port)
    throws NacosException {
47 // 从Nacos配置中心拉取配置
48 String rules = configService.getConfig(
49     dataId: appName + NacosConfigUtil.FLOW_DATA_ID_POSTFIX,
50     NacosConfigUtil.GROUP_ID, NacosConfigUtil.READ_TIMEOUT);
51 if (StringUtil.isEmpty(rules)) { // 从配置中心拉取规则配置
52     return new ArrayList<>();
53 }
54 // 解析json获取到 List<FlowRule>
55 List<FlowRule> list = JSON.parseArray(rules, FlowRule.class);
56 // FlowRule----->FlowRuleEntity
57 return list.stream().map(rule ->
58     FlowRuleEntity.fromFlowRule(appName, ip, port, rule))
59     .collect(Collectors.toList());
60 }
61 }
```

从控制台发布配置到配置中心时，FlowRuleEntity转换为FlowRule



```
31 @Component("flowRuleNacosPublisher")
32 public class FlowRuleNacosPublisher implements
    DynamicRulePublisher<List<FlowRuleEntity>> {
33
34
35     @Autowired
36     private ConfigService configService;
37
38     @Override
39     public void publish(String app, List<FlowRuleEntity> rules) throws Exception {
40         AssertUtil.notEmpty(app, message: "app name cannot be empty");
41         if (rules == null) {
42             return;
43         }
44         //发布配置到Nacos配置中心
45         configService.publishConfig(
46             dataId: app + NacosConfigUtil.FLOW_DATA_ID_POSTFIX,
47             NacosConfigUtil.GROUP_ID, NacosConfigUtil.convertToRule(rules));
48     }
49 }
```

2. sentinel规则持久化部分源码分析

<https://www.processon.com/view/link/607fef267d9c08283ddc2f8d>

文档: 10 Sentinel规则持久化实战及其源码分?..

链接: [http://note.youdao.com/noteshare?](http://note.youdao.com/noteshare?id=f87499a0c2e4045c4d94816a37a3d08d&sub=E1F9A1A3AEA9434C8732044893E4CACA)

[id=f87499a0c2e4045c4d94816a37a3d08d&sub=E1F9A1A3AEA9434C8732044893E4CACA](http://note.youdao.com/noteshare?id=f87499a0c2e4045c4d94816a37a3d08d&sub=E1F9A1A3AEA9434C8732044893E4CACA)