主讲老师： Fox

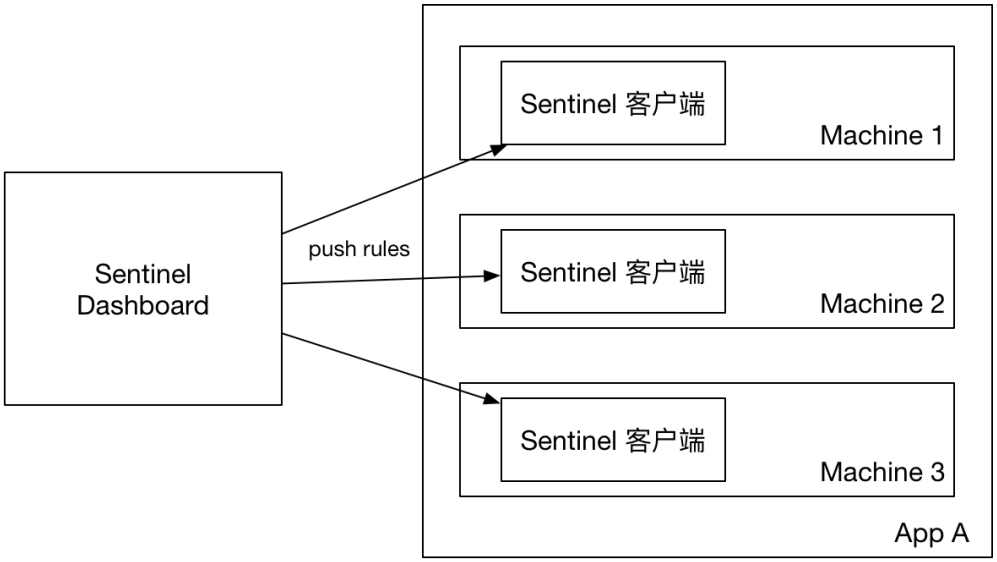
# 1. Sentinel规则持久化模式

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

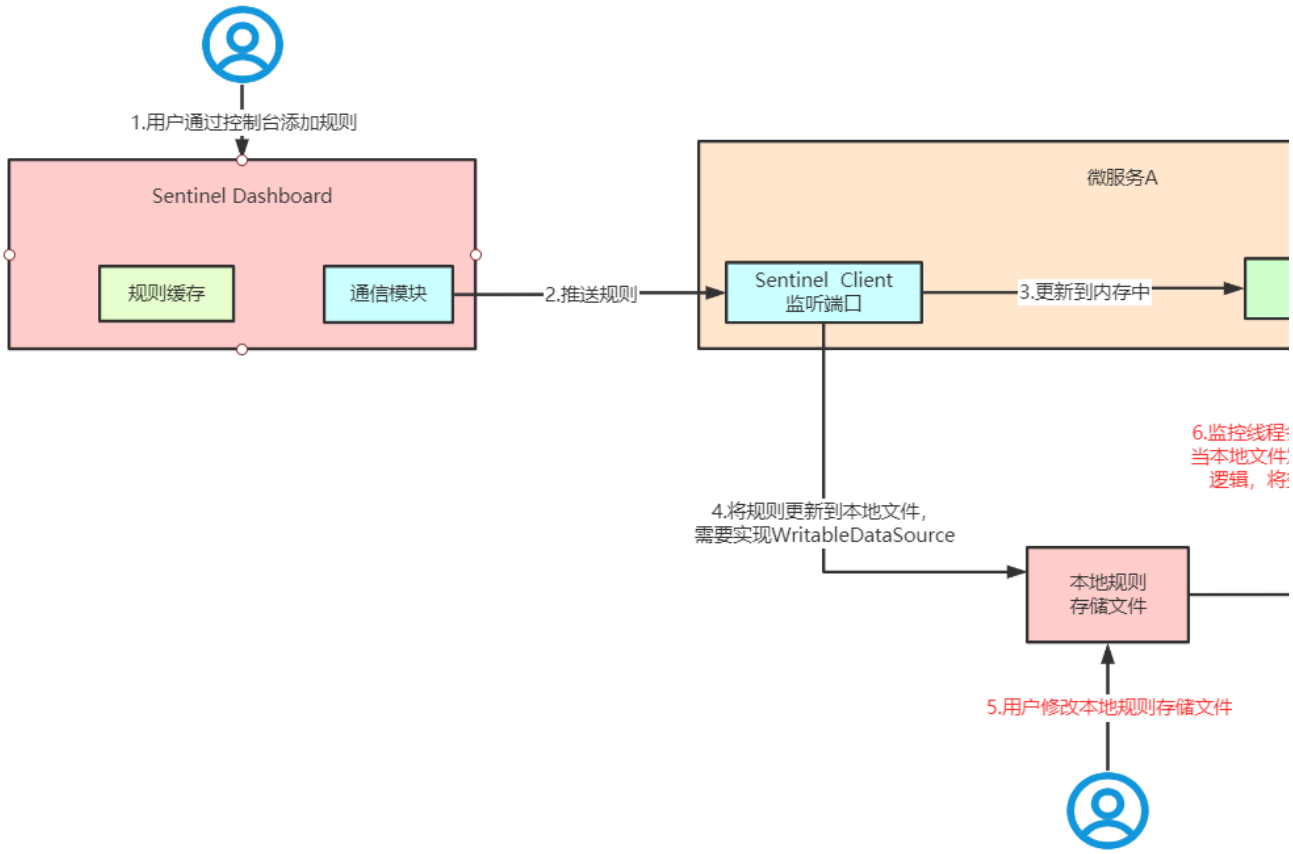
如果不做任何修改，Dashboard 的推送规则方式是通过 API 将规则推送至客户端并直接更新到内存中：




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

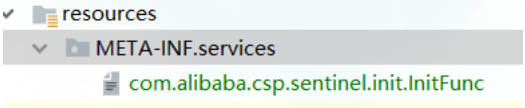
```
<dependency>
 <groupId>com.alibaba.csp</groupId>
 <artifactId>sentinel-datasource-extension</artifactId>
 <version>1.8.0</version>
</dependency>
```

核心代码：

```
// 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。
ReadableDataSource<String, List<FlowRule>> ds = new FileRefreshableDataSource<>(
 flowRulePath, source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {})
 );
// 将可读数据源注册至 FlowRuleManager.
FlowRuleManager.register2Property(ds.getProperty());

WritableDataSource<List<FlowRule>> wds = new FileWritableDataSource<>(flowRulePath, this::encodeJson);
// 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
 // 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.
 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

```
public class FileDataSourceInit implements InitFunc {

```

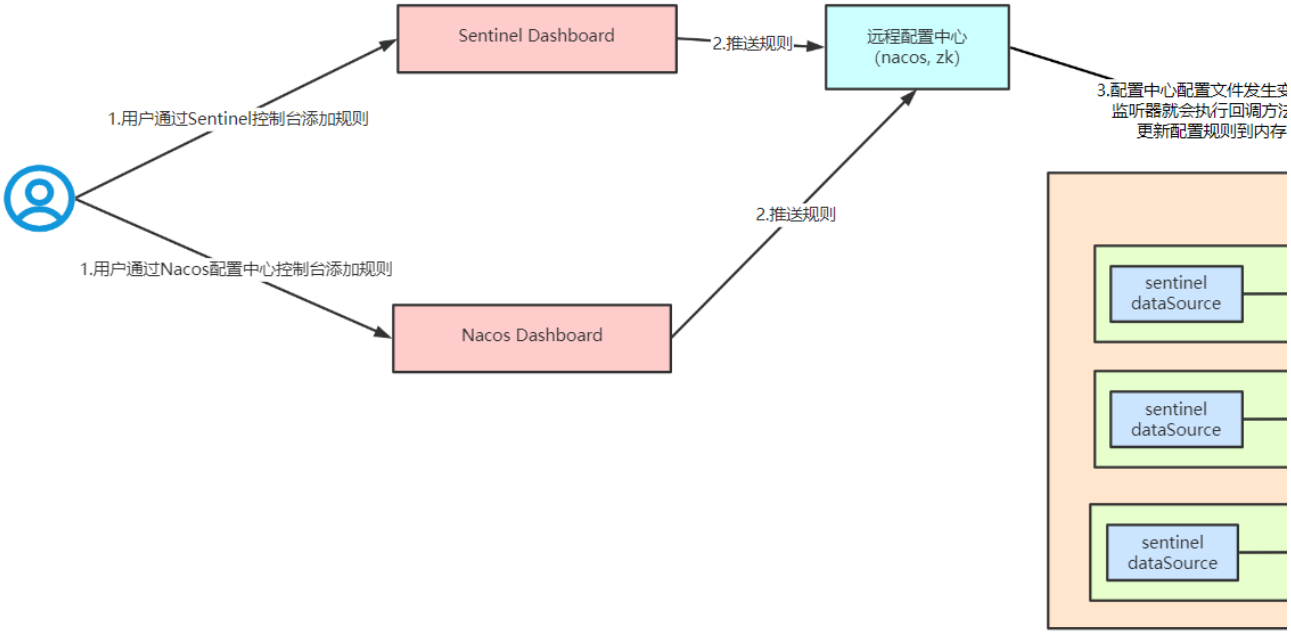
```
@Override
public void init() throws Exception {
//创建文件存储目录
RuleFileUtils.mkdirIfNotExits(PersistenceRuleConstant.storePath);

//创建规则文件
RuleFileUtils.createFileIfNotExits(PersistenceRuleConstant.rulesMap);

 //处理流控规则逻辑
 dealFlowRules();
 // 处理降级规则
 dealDegradeRules();
 // 处理系统规则
 dealSystemRules();
 // 处理热点参数规则
 dealParamFlowRules();
 // 处理授权规则
 dealAuthRules();
 }


 private void dealFlowRules() throws FileNotFoundException {
 String ruleFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.FLOW_RULE_PATH).toString();

 //创建流控规则的可读数据源
 ReadableDataSource<String, List<FlowRule>> flowRuleRDS = new FileRefreshableDataSource(
 ruleFilePath, RuleListConverterUtils.flowRuleListParser
 );

 // 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
 FlowRuleManager.register2Property(flowRuleRDS.getProperty());


 WritableDataSource<List<FlowRule>> flowRuleWDS = new FileWritableDataSource<List<FlowRule>>(
 ruleFilePath, RuleListConverterUtils.flowFuleEnCoding
 );

 // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
 // 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.
 WritableDataSourceRegistry.registerFlowDataSource(flowRuleWDS);
 }

 private void dealDegradeRules() throws FileNotFoundException {
 //讲解规则文件路径
 String degradeRuleFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.DEGRAGE_RULE_PATH).toStr
ing();

 //创建流控规则的可读数据源
 ReadableDataSource<String, List<DegradeRule>> degradeRuleRDS = new FileRefreshableDataSource(
 degradeRuleFilePath, RuleListConverterUtils.degradeRuleListParse
 );

 // 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
 DegradeRuleManager.register2Property(degradeRuleRDS.getProperty());


 WritableDataSource<List<DegradeRule>> degradeRuleWDS = new FileWritableDataSource<>(
 degradeRuleFilePath, RuleListConverterUtils.degradeRuleEnCoding
 );

 // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
 // 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.
```

```
WritableDataSourceRegistry.registerDegradeDataSource(degradeRuleWDS);
}

private void dealSystemRules() throws FileNotFoundException {
//讲解规则文件路径
String systemRuleFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.SYSTEM_RULE_PATH).toStrin
g();

//创建流控规则的可读数据源
ReadableDataSource<String, List<SystemRule>> systemRuleRDS = new FileRefreshableDataSource(
systemRuleFilePath, RuleListConverterUtils.sysRuleListParse
);

// 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
SystemRuleManager.register2Property(systemRuleRDS.getProperty());


WritableDataSource<List<SystemRule>> systemRuleWDS = new FileWritableDataSource<>(
systemRuleFilePath, RuleListConverterUtils.sysRuleEnCoding
);

// 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
// 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.
WritableDataSourceRegistry.registerSystemDataSource(systemRuleWDS);
}


private void dealParamFlowRules() throws FileNotFoundException {
//讲解规则文件路径
String paramFlowRuleFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.HOT_PARAM_RULE).toStri
ng();

//创建流控规则的可读数据源
ReadableDataSource<String, List<ParamFlowRule>> paramFlowRuleRDS = new FileRefreshableDataSource(
paramFlowRuleFilePath, RuleListConverterUtils.paramFlowRuleListParse
);

// 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
 ParamFlowRuleManager.register2Property(paramFlowRuleRDS.getProperty());


 WritableDataSource<List<ParamFlowRule>> paramFlowRuleWDS = new FileWritableDataSource<>(
 paramFlowRuleFilePath, RuleListConverterUtils.paramRuleEnCoding
 );

 // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
 // 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.
 ModifyParamFlowRulesCommandHandler.setWritableDataSource(paramFlowRuleWDS);
 }

 private void dealAuthRules() throws FileNotFoundException {
 //讲解规则文件路径
 String authFilePath = PersistenceRuleConstant.rulesMap.get(PersistenceRuleConstant.AUTH_RULE_PATH).toString();

 //创建流控规则的可读数据源
 ReadableDataSource<String, List<AuthorityRule>> authRuleRDS = new FileRefreshableDataSource(
 authFilePath, RuleListConverterUtils.authorityRuleParse
 );

 // 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
 AuthorityRuleManager.register2Property(authRuleRDS.getProperty());

```

```
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>
```

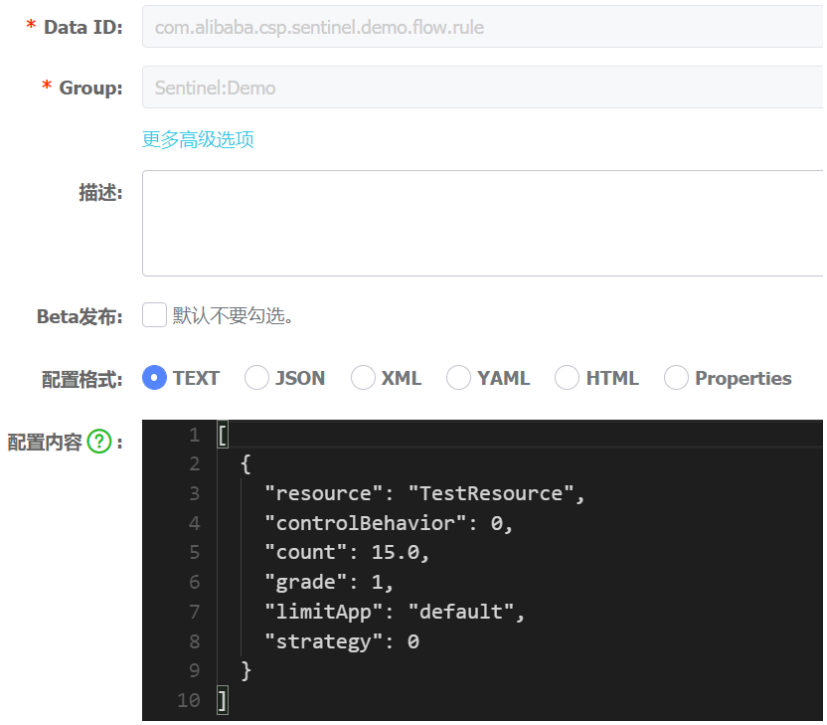
核心代码

```
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,
dataId,source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {}));
8 FlowRuleManager.register2Property(flowRuleDataSource.getProperty());
```

nacos配置中心中配置流控规则

```
1 [
```

```
{
"resource": "TestResource",
"controlBehavior": 0,
"count": 10.0,
"grade": 1,
"limitApp": "default",
"strategy": 0
}
]
```




**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

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

2）yml中配置

```
spring:
 application:
 name: mall-user-sentinel-demo
 cloud:
 nacos:
 discovery:
 server-addr: 127.0.0.1:8848

 sentinel:
  transport:
  # 添加sentinel的控制台地址
  dashboard: 127.0.0.1:8080
  # 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
  port: 8719
  datasource:
  ds1:
  nacos:
```

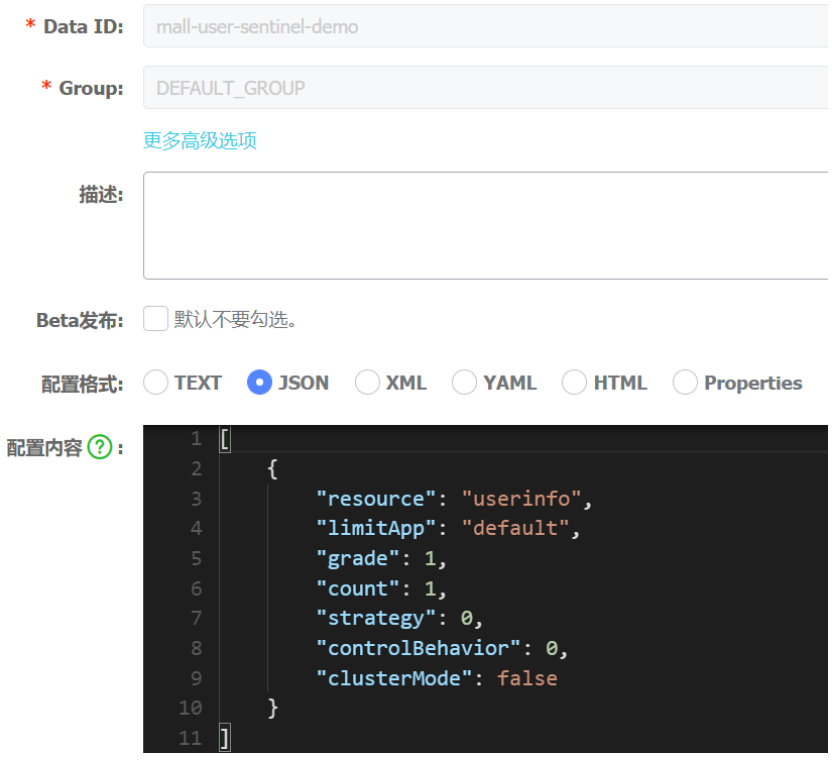
```
server-addr: 127.0.0.1:8848
dataId: ${spring.application.name}
groupId: DEFAULT_GROUP
data-type: json
rule-type: flow
```

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

```
[
 {
 "resource": "userinfo",
 "limitApp": "default",
 "grade": 1,
 "count": 1,
 "strategy": 0,
 "controlBehavior": 0,
 "clusterMode": false
 }
]
```




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

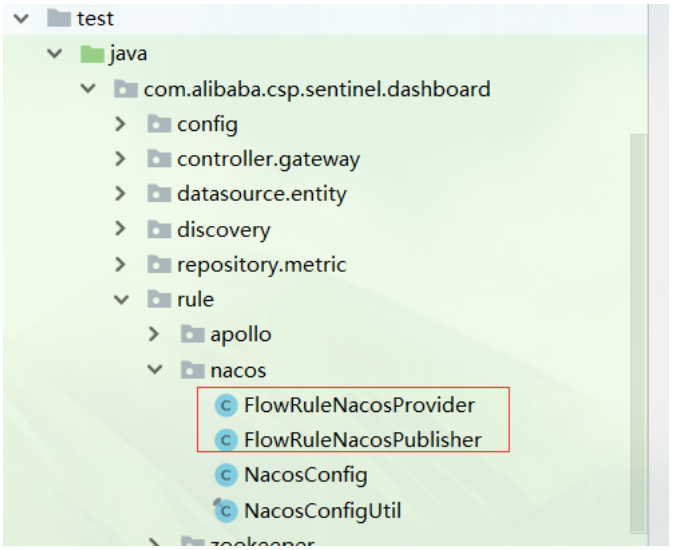
从 Sentinel 1.4.0 开始，Sentinel 控制台提供 `DynamicRulePublisher` 和 `DynamicRuleProvider` 接口用于实现应用维度的规则推送和拉取：

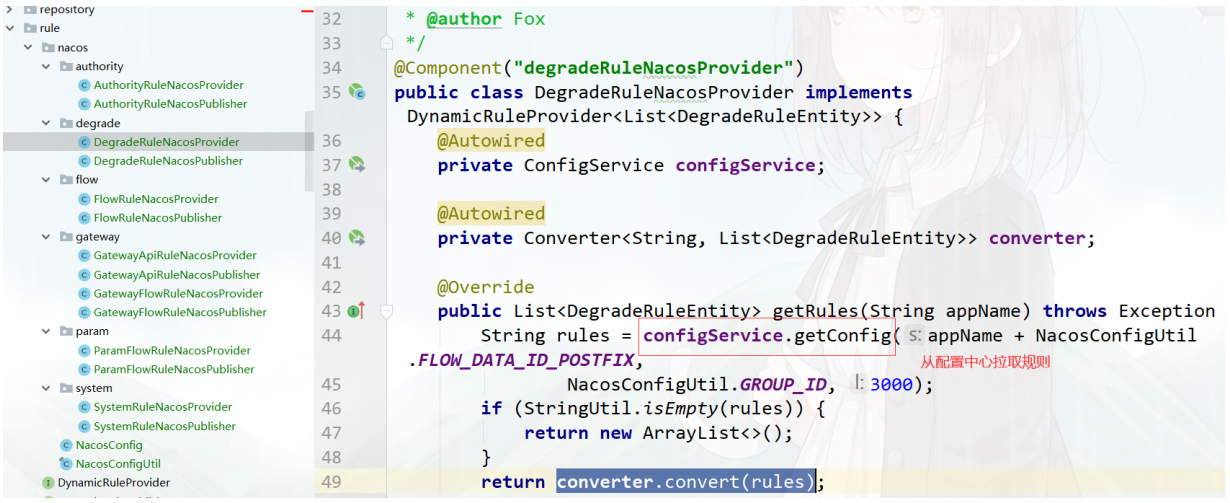
- `DynamicRuleProvider<T>`: 拉取规则
- `DynamicRulePublisher<T>`: 推送规则

**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：

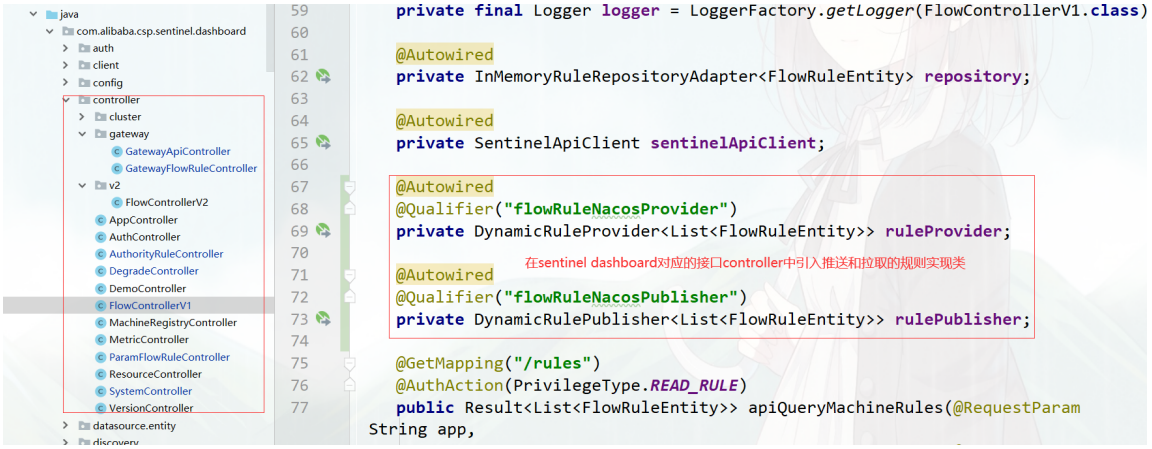



注意：微服务接入Sentinel client，yml配置需要匹配对应的规则后缀

```
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




以流控规则为例，从Nacos配置中心获取所有的流控规则

```
@GetMapping("/rules")
@AuthAction(PrivilegeType.READ_RULE)
public Result<List<FlowRuleEntity>> apiQueryMachineRules(@RequestParam String app,
@RequestParam String ip,
@RequestParam Integer port) {

if (StringUtil.isEmpty(app)) {
return Result.ofFail(-1, "app can't be null or empty");
}
 if (StringUtil.isEmpty(ip)) {
 return Result.ofFail(-1, "ip can't be null or empty");
 }
 if (port == null) {
 return Result.ofFail(-1, "port can't be null");
 }
 try {
 // List<FlowRuleEntity> rules = sentinelApiClient.fetchFlowRuleOfMachine(app, ip, port);
 //从配置中心获取规则配置
```

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

新增流控规则，会推送到nacos配置中心

```
@PostMapping("/rule")
@AuthAction(PrivilegeType.WRITE_RULE)
public Result<FlowRuleEntity> apiAddFlowRule(@RequestBody FlowRuleEntity entity) {
Result<FlowRuleEntity> checkResult = checkEntityInternal(entity);
if (checkResult != null) {
return checkResult;
}
entity.setId(null);
Date date = new Date();
entity.setGmtCreate(date);
entity.setGmtModified(date);
entity.setLimitApp(entity.getLimitApp().trim());
entity.setResource(entity.getResource().trim());
try {
entity = repository.save(entity);

//publishRules(entity.getApp(), entity.getIp(), entity.getPort()).get(5000, TimeUnit.MILLISECONDS);
//发布规则到配置中心
publishRules(entity.getApp());
return Result.ofSuccess(entity);
} catch (Throwable t) {
Throwable e = t instanceof ExecutionException ? t.getCause() : t;
logger.error("Failed to add new flow rule, app={}, ip={}", entity.getApp(), entity.getIp(), e);
return Result.ofFail(-1, e.getMessage());
}
}

/**
* 发布到配置中心
* @param app
* @throws Exception
*/
private void publishRules(/*@NonNull*/ String app) throws Exception {
List<FlowRuleEntity> rules = repository.findAllByApp(app);
rulePublisher.publish(app, rules);
}
```

## 测试：**微服务接入改造后的Sentinel Dashboard**

引入依赖

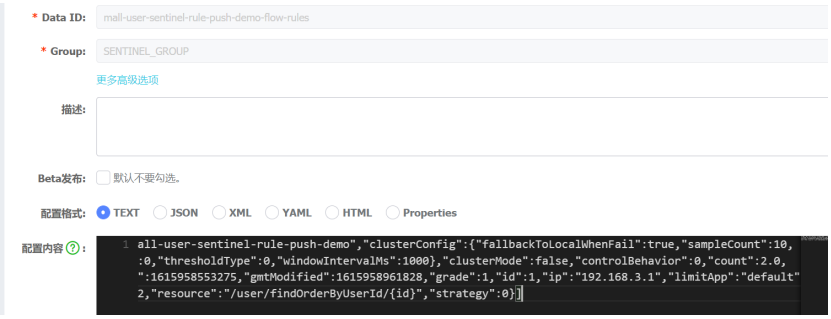
```
<!--sentinel持久化 采用 Nacos 作为规则配置数据源-->
<dependency>
<groupId>com.alibaba.csp</groupId>
<artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

增加yml配置

```
server:
port: 8806

spring:
application:
```

```yaml
  name: mall-user-sentinel-rule-push-demo #微服务名称

 #配置nacos注册中心地址
 cloud:
  nacos:
  discovery:
  server-addr: 127.0.0.1:8848

  sentinel:
  transport:
  # 添加sentinel的控制台地址
  dashboard: 127.0.0.1:8080
  # 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
  #port: 8719
  datasource:
 # ds1: #名称自定义，唯一
 # nacos:
 # server-addr: 127.0.0.1:8848
 # dataId: ${spring.application.name}
 # groupId: DEFAULT_GROUP
 # data-type: json
 # rule-type: flow
  flow-rules:
  nacos:
  server-addr: 127.0.0.1:8848
  dataId: ${spring.application.name}-flow-rules
  groupId: SENTINEL_GROUP # 注意groupId对应Sentinel Dashboard中的定义
  data-type: json
  rule-type: flow
  degrade-rules:
  nacos:
  server-addr: 127.0.0.1:8848
  dataId: ${spring.application.name}-degrade-rules
  groupId: SENTINEL_GROUP
  data-type: json
  rule-type: degrade
  param-flow-rules:
  nacos:
  server-addr: 127.0.0.1:8848
  dataId: ${spring.application.name}-param-flow-rules
  groupId: SENTINEL_GROUP
  data-type: json
  rule-type: param-flow
  authority-rules:
  nacos:
  server-addr: 127.0.0.1:8848
  dataId: ${spring.application.name}-authority-rules
  groupId: SENTINEL_GROUP
  data-type: json
  rule-type: authority
  system-rules:
  nacos:
  server-addr: 127.0.0.1:8848
  dataId: ${spring.application.name}-system-rules
  groupId: SENTINEL_GROUP
  data-type: json
  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。




**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：

```
[{
 "app": "mall-user-sentinel-rule-push-demo",
 "gmtCreate": 1616136838785,
 "gmtModified": 1616136838785,
 "id": 1,
 "ip": "192.168.3.1",
 "port": 8719,
 "rule": {
 "burstCount": 0,
  "clusterConfig": {
  "fallbackToLocalWhenFail": true,
  "sampleCount": 10,
  "thresholdType": 0,
  "windowIntervalMs": 1000
  },
  "clusterMode": false,
  "controlBehavior": 0,
  "count": 1.0,
  "durationInSec": 1,
  "grade": 1,
  "limitApp": "default",
  "maxQueueingTimeMs": 0,
  "paramFlowItemList": [],
  "paramIdx": 1,
  "resource": "hot"
  }
}, {
  "app": "mall-user-sentinel-rule-push-demo",
  "gmtCreate": 1616137178470,
  "gmtModified": 1616658923519,
  "id": 2,
  "ip": "192.168.3.1",
  "port": 8719,
  "rule": {
  "burstCount": 0,
  "clusterConfig": {
  "fallbackToLocalWhenFail": true,
  "sampleCount": 10,
  "thresholdType": 0,
  "windowIntervalMs": 1000
  },
```

```
 "clusterMode": false,
 "controlBehavior": 0,
 "count": 3.0,
 "durationInSec": 1,
 "grade": 1,
 "limitApp": "default",
 "maxQueueingTimeMs": 0,
 "paramFlowItemList": [{
 "classType": "int",
 "count": 1,
 "object": "4"
 }],
 "paramIdx": 0,
 "resource": "findOrderByUserId"
 }
}]
```

我提供两种解决思路：

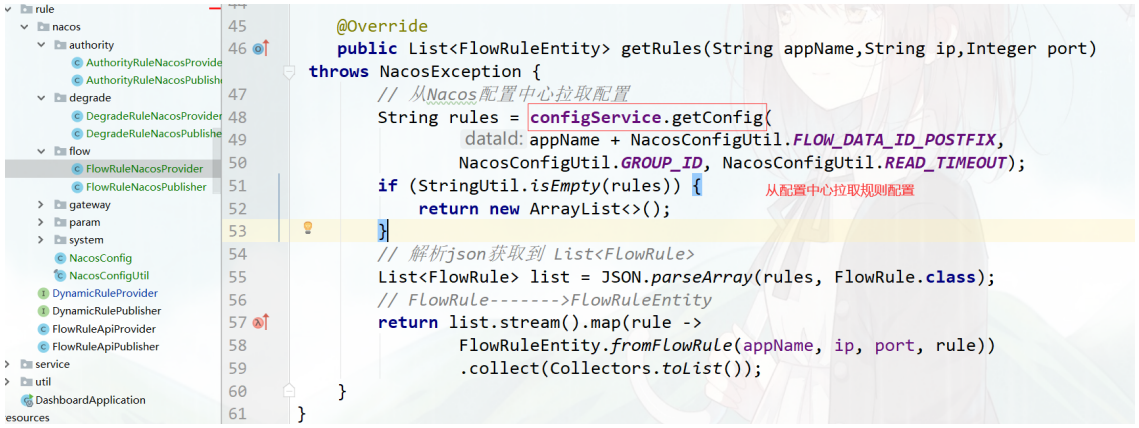
1. 自定义一个解析热点规则配置的解析器FlowParamJsonConverter，继承JsonConverter，重写convert方法。然后利用后置处理器替换beanName为"param-flow-rules-sentinel-nacos-datasource"的converter属性，注入FlowParamJsonConverter。

```
@Configuration
public class ConverterConfig {

 @Bean("sentinel-json-param-flow-converter2")
 @Primary
 public JsonConverter jsonParamFlowConverter() {
 return new FlowParamJsonConverter(new ObjectMapper(), ParamFlowRule.class);
 }
}

@Component
public class FlowParamConverterBeanPostProcessor implements BeanPostProcessor {

 @Autowired
 private JsonConverter jsonParamFlowConverter;

 @Override
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
 if (beanName.equals("param-flow-rules-sentinel-nacos-datasource")) {
 NacosDataSourceFactoryBean nacosDataSourceFactoryBean = (NacosDataSourceFactoryBean) bean;
 nacosDataSourceFactoryBean.setConverter(jsonParamFlowConverter);
 return bean;
 }
 return bean;
 }
}

public class FlowParamJsonConverter extends JsonConverter {
 Class ruleClass;

 public FlowParamJsonConverter(ObjectMapper objectMapper, Class ruleClass) {
 super(objectMapper, ruleClass);
 this.ruleClass = ruleClass;
 }

 @Override
 public Collection<Object> convert(String source) {
 List<Object> list = new ArrayList<>();
```

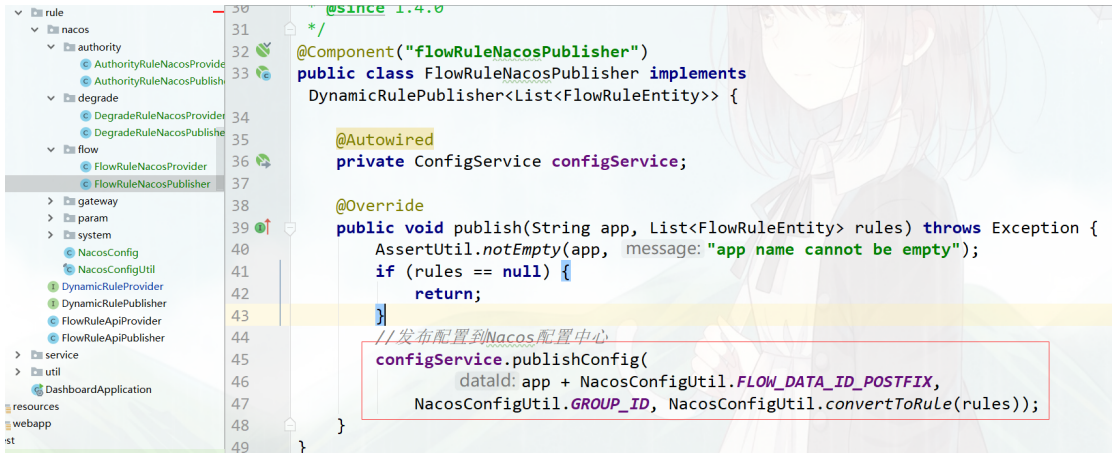
```
39  JSONArray jsonArray = JSON.parseArray(source);
40  for (int i = 0; i < jsonArray.size(); i++) {
41  //解析rule属性
42  JSONObject jsonObject = (JSONObject) jsonArray.getJSONObject(i).get("rule");
43  Object object = JSON.toJavaObject(jsonObject, ruleClass);
44  list.add(object);
45  }
46  return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




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




# 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?id=f87499a0c2e4045c4d94816a37a3d08d&sub=E1F9A1A3AEA9434C8732044893E4CACA