- 在 阿里 sentinel —— core 模块浅析 一文中介绍了 sentinel 组件的功能和一些核心概念，项目中只需依赖 sentinel-core 模块便可实现单体应用的限流、熔断等核心功能
- 本文介绍 seninel-cluster 模块，主要介绍它的功能及应用

# 1 sentinel-cluster 模块的作用

- sentinel-cluster 主要提供微服务中一个应用部署多个实例的全局限流功能（只提供 qps 限流）
- 开启全局限流模式后，client 会向 server 请求 token，限流功能统一有 server 来管理
- server 端提供了两种全局限流策略，每种策略对应一种情景的解决
- 通过 设置 FlowRule.clusterMode = true 对某个资源开启 cluster 模式，默认为 local 模式

---

# 2 全局限流中两种策略讨论

- 通过 设置 FlowRule.clusterConfig.thresholdType = 1 **对某个资源开启** 全局阈值 策略，默认为 阈值均摊 策略

## 2.1 阈值均摊策略

- cluster 模式下，设置 阈值均摊 策略后，全局 qps 的值为：n * v，n 为实例数，v 为资源对应的 qps 阈值
- 解决的情景：local 模式下，多实例中，请求分布不均导致的全局通过流量低于多实例总体的总值

## 2.2 全局阈值策略

- cluster 模式下，设置 全局阈值 策略后，全局的 qps 为定值：v，为资源对应的 qps 阈值
- 解决的情景：local 模式或 cluster 阈值均摊模式下，随着应用实例的增加，全局 qps 也会不断提高，这样可能会导致应用的下游(如数据库)承担不了太多请求

---

# 3 cluster 模式配置

- cluster 模式又可按 server 的性质分为 独立 server 模式、嵌入 server 模式
- 独立 server 模式：server 只用来管理 client 端来的 token 请求，可以管理多个应用

- 嵌入 server 模式：server 端是一个应用中多个实例的一个，可以方便的切换 server，但只能管理它对应的应用，耦合性高

## 3.1 独立 server 模式

**1 server 需要监听如下几个配置**

- **namespaceSet：** 一般每个应用对应一个 namespace，server 需要监听它管理的所有应用
  - namespaceSet 初始化的时候，会去注册每个 namespace 对应的 flowRules 和 paramFlowRules 的监听。注册的关键是：ClusterFlowRuleManager.propertySupplier 属性
  - 所以，需要在 namespaceSet 监听前，设置 ClusterFlowRuleManager.propertySupplier 属性

```
//setPropertySupplier 要在 registerNamespaceSetProperty 之前
ClusterFlowRuleManager.setPropertySupplier(namespace -> {
    ReadableDataSource<String, List<FlowRule>> ds = 配置远端数据源(namespace);
    return ds.getProperty();
});// 同理 paramFlowRules
ReadableDataSource<String, Set<String>> namespaceDs = 配置远端数据源;
ClusterServerConfigManager.registerNamespaceSetProperty(namespaceDs.getProperty());
```

- **ServerTransportConfig**
  - port：设置 server 端接收 client 端 token 的端口
  - idleSeconds：设置服务端定期断开客户端闲置超过 idleSeconds 的连接
  - 只有 port 改变时，才会重启服务

```
ReadableDataSource<String, ServerTransportConfig> transportConfigDs = 配置远端数据源;
ClusterServerConfigManager.registerServerTransportProperty(transportConfigDs.getPro
perty());
```

- **ServerFlowConfig**
  - maxAllowedQps：服务端最大 qps，为了保护服务端，默认为 30000
  - exceedCount：这个参数乘以 全局阈值，才是正真的全局阈值，默认为 1.0
  - maxOccupyRatio：和流量整型模式中的等待请求数有关吧，默认为 1.0
  - intervalMs/sampleCount：窗口大小/窗口的桶个数，用于统计，默认为1000/10

```
ReadableDataSource<String, ServerFlowConfig> ServerFlowConfigs = 配置远端数据源;
ClusterServerConfigManager.registerGlobalServerFlowProperty(ServerFlowConfigs.getPr
operty());
```

**注：** server 监听的 namespace 可以和 client 的 namespace 分开，只要保证对应的 flowId 一样就行。这样就可以避免在全局阈值的情况下，server 不可用时，client 走 local 模式，这时 的 qps 是很大的，导致限流近似失效

**2 client 需要监听如下几个配置**

- **ClusterClientAssignConfig：** 监听 serverHost、serverPort。改变时重启 client connect

```
ReadableDataSource<String, ClusterClientAssignConfig> clientAssignDs = 配置远端数据源;
ClusterClientConfigManager.registerServerAssignProperty(clientAssignDs.getProperty());
```

- **ClusterClientConfig：** 监听 requestTimeout 配置，当 client 请求超过这个配置时，默认切换为 local 流程

```
ReadableDataSource<String, ClusterClientConfig> clientTimeouts = 配置远端数据源;
ClusterClientConfigManager.registerClientConfigProperty(clientTimeouts.getProperty());
```

- **监听和 local 模式一样的规则配置：** flowRules、paramFlowRules、degradeRules、systemRules 等

**注：** 独立 server 模式下，server 端监听的 flowRules、paramFlowRules 可以和 client 的规则不一样(只要flowId一样就可以匹配)，这样就可以避免 在全局阈值模式下 client 端获取的 qps 值是全局的值，当 server 不可用时，会造成所有 client 都会使用 这个很大的 qps 来过滤！！

## 3.2 嵌入 server 模式

- 嵌入 server 模式需要监听 独立 server 模式中 server 和 client 监听的所有配置（除了 namespaceSet 以外）
  - 因为嵌入模式它的有效的 namespace 只有一个，即应用本身的 namespace
- 并且还需要监听：**ClusterState**，当切换 server 时，所有实例的这个状态也相应变化。每个实例根据这个变化，来切换 client 和 server

```
ReadableDataSource<String, Integer> clusterModeDs = 配置远端数据源;
ClusterStateManager.registerProperty(clusterModeDs.getProperty());
```

- 嵌入模式是通过首次启动 server 时，注册 namespace 对应的 flowRules 和 paramFlowRules 的监听

```
private void startServerIfScheduled() throws Exception {
    if (shouldStart.get()) {
        if (server != null) {
            //这个期间，client 请求都会失败，如果是全局阈值策略的话，可能会造成灾难
            server.start();
            ClusterStateManager.markToServer();
            if (embedded) {
                //一直到这
                handleEmbeddedStart();//这里注册监听。独立模式，通过监听 spaceNameSet 来注册的所有不需要这个方法
            }
        }
    }
}
```

**注1：** 嵌入模式在切换 server 中，server 开启一直到 执行完 handleEmbeddedStart() 前的一段时间内，client 请求都会出问题，如果这时设置的是全局阈值策略，就可能发生灾难，因为全局阈值一般很大，造成应用 限流策略可以看成失效。

**注2：** 嵌入模式中，namespaceSet 不是通过监听获取，而是通过 namespaceSupplier 属性获取，默认返回的是 appName，可以通过这个设定，来使得 client 获取的 flowRule 和 server 获取的 serverFlowRule 分开（但 flowId 要对应），这样就可以避免，server 不可用时，整个 限流策略 都近似失效的情况

**注3：** 嵌入和独立模式中，namespaceSet 默认都包含 default 属性。

---

# 4 cluster 模式下 请求/响应 token 的核心流程

## 4.1 client 请求 token 流程

```
//客户端走到限流 slot 中，开始判断是否限流
public void checkFlow(...) throws BlockException {
    if (rules != null) {
        for (FlowRule rule : rules) {
            //这里开始做 限流判断
            if (!canPassCheck(rule, context, node, count, prioritized)) {
                throw new FlowException(rule.getLimitApp(), rule);
            }
        }
    }
}
//在 canPassCheck 中，如果判断是 clusterMode 就会走 cluster 流程
public boolean canPassCheck(...)
    if (rule.isClusterMode()) {
        //如果是 cluster 模式，则走 cluster 流程
        return passClusterCheck(rule, context, node, acquireCount, prioritized);
    }
    return passLocalCheck(rule, context, node, acquireCount, prioritized);
}
private static boolean passClusterCheck(...) {
        //server/client 都持有一个对应的 TokenService，这里根据实例类型来返回相应的
tokenService
        TokenService clusterService = pickClusterService();

        long flowId = rule.getClusterConfig().getFlowId();//全局的 rule 通过 flowId 来区分
        //通过 flowId 来请求 token，每一个 flowRule 对应一个 唯一的 flowId
        TokenResult result = clusterService.requestToken(flowId, acquireCount,
prioritized);
        //根据 token 的类型，作不同的处理
        return applyTokenResult(result, rule, context, node, acquireCount,
prioritized);

    return fallbackToLocalOrPass(rule, context, node, acquireCount, prioritized);
}
```

## 4.2 server 响应 token 流程

- 服务端通过 processRequest() 方法接收并处理请求

```
ClusterResponse<?> response = processor.processRequest(request);
writeResponse(ctx, response);
```

- 提取 request 得到 flowId,count,prioritized，处理并响应

```
public ClusterResponse<FlowTokenResponseData>
processRequest(ClusterRequest<FlowRequestData> request) {
    TokenService tokenService = TokenServiceProvider.getService();
    long flowId = request.getData().getFlowId();
    int count = request.getData().getCount();
    boolean prioritized = request.getData().isPriority();
    TokenResult result = tokenService.requestToken(flowId, count, prioritized);
    return toResponse(result, request);
}
```

- 最终会走到这个方法，处理请求

```
static TokenResult acquireClusterToken(/*@Valid*/ FlowRule rule, int acquireCount,
boolean prioritized) {
    Long id = rule.getClusterConfig().getFlowId();
    //服务端限流保护
    if (!allowProceed(id)) {
        return new TokenResult(TokenResultStatus.TOO_MANY_REQUEST);
    }
    //正真全局限流
    double latestQps = metric.getAvg(ClusterFlowEvent.PASS_REQUEST);
    //通过 calcGlobalThreshold() 方法得到全局阈值，乘以 exceedCount 得到正真的全局阈值
    double globalThreshold = calcGlobalThreshold(rule) *
ClusterServerConfigManager.getExceedCount();
    double nextRemaining = globalThreshold - latestQps - acquireCount;

    if (nextRemaining >= 0) {
       //做 metrix 收集，并返回成功
    } else {
        if (prioritized) {
            //做优先级处理，如果成功则返回
        }
        //做 metrix 收集，并返回阻塞
    }
}
```

### 4.3 全局阈值和阈值均摊处理的核心代码

```
private static double calcGlobalThreshold(FlowRule rule) {
    double count = rule.getCount();
    switch (rule.getClusterConfig().getThresholdType()) {
        case ClusterRuleConstant.FLOW_THRESHOLD_GLOBAL:
            return count;
        case ClusterRuleConstant.FLOW_THRESHOLD_AVG_LOCAL:
        default:
            int connectedCount =
ClusterFlowRuleManager.getConnectedCount(rule.getClusterConfig().getFlowId());
            return count * connectedCount;
    }
}
```

## 参考

sentinel-cluster 1.6.3