Alibaba Summer of Code Proposal

Proposal for Distributed Concurrency Control

About Project

*Abstract:

Now Sentinel can support cluster flow control by QPS and thread number, but It's not enough. We want to enrich cluster flow control capabilities by adding cluster concurrency control. We want to control the number of API calls at a certain moment, which is different from QPS and thread number. At the same time, providing a set of extension mechanisms allows users to implement their own concurrency control logic.

In order to achieve cluster concurrent flow control, we use the token recovery mechanism to calculate the current concurrency, and use ConcurrentLinkedHashMap to store the tokenId. To deal with the situation of token client disconnection and resource call timeout, we decide to detect its status regularly. In order to allow users to quickly implement their own cluster concurrent flow control rules, we have implemented an interface-based framework.

*Implementation plan:

一、并发限流框架设计

最大并发量可以用符号 concurrencyLevel 表示，将当前执行数量 nowCalls 和 concurrencyLevel 作比较，似乎只需要保证 nowCalls 小于等于 concurrencyLevel 就行了，不需要存储额外的信息。但是我们需要考虑到 token client 可能出现掉线或者调用资源超时的情况，这会导致 nowCalls 的统计误差，经过时间的积累甚至会造成 nowCalls 永远大于 concurrencyLevel 的情况发生。同时，如果只是去统计 nowCalls 并不利于我们的 Dashboard 去展示相关信息（哪些调用正在进行、调用时长、调用失效数量等），这些数据对于我们监控系统是否正常运行至关重要。

为了实现刚才所提及的功能，我们必须存储每个调用的信息，发现并清除那些不正常的资源调用。具体的讲，当 token client 发起调用时会获得 token Server 颁发的身份标识 tokenId, token server 会存储这个 tokenId 值和相关的信息。当调用结束时，token client 会携带着该次调用被分配的 tokenId 前往 token server 请求释放资源，token server 会删除这个调用对应的 tokenId。通过这样的操作，我们能够实时的统计当前正在进行的调用数量，即并发量。如果出现资源调用超时或者 token client 掉线的情况，token server 会尝试着去发现删除对应调用并清除存储的 tokenId，从而获得准确的并发量 nowCalls。

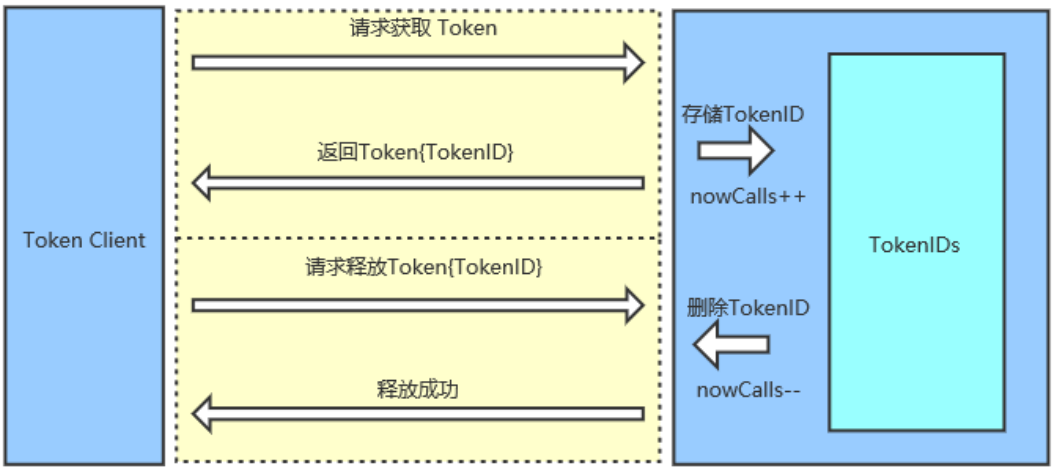

图 1. 请求通过时流程

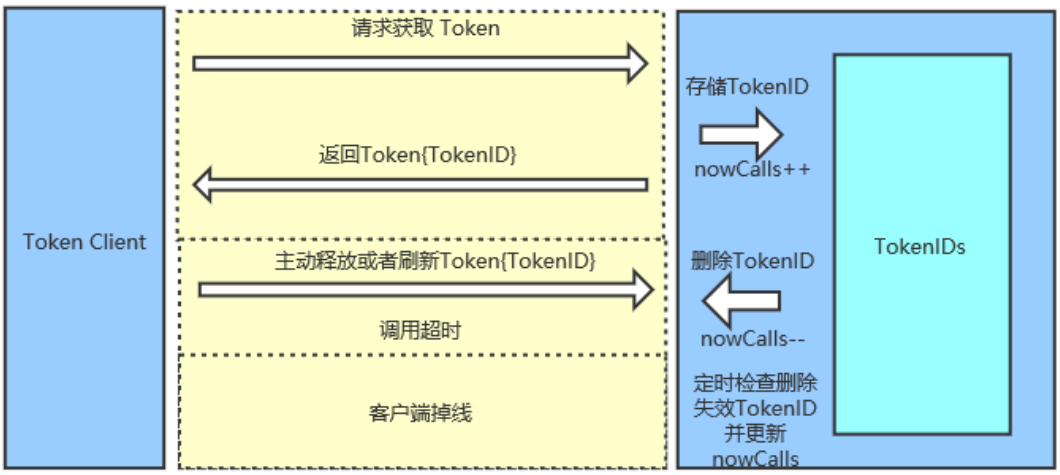

图 2. 连接断开或者调用超时流程

二、并发限流规则源设计

我们使用一个 ConcurrentHashMap<Long, AtomicInteger>类型的结构存储各个 rule 所对应的并发量 nowCalls，其中键为 flowId，值为 nowCalls，由于 nowCalls 可能会被多个线程访问修改，我们考虑将其设计成原子类或者是被 volaitle 关键字修饰。每新建一个并发流控类型的 rule 都会向这个 map 中新增 nowCalls 对象。如果一个 rule 所对应的并发量发生了改变我们会实时去更新所对应的 nowCalls，每次获取 token 的请求都会对 nowCalls 进行加操作；释放 token 的请求都会对 nowCalls 进行减操作

```
public class NowCallsMap {
    // 定义存储结构
    // static ConcurrentHashMap<Long, AtomicInteger> map = new
ConcurrentHashMap<Long, AtomicInteger>();
    // 使用原子操作保证线程安全
    // private static void update(Long flowId, Integer
  count) {

map.get(flowId).getAndAdd(count);     }
    // 获取nowCalls
    private static Integer get(Long flowId)
{         map.get(flowId);
```

```
    }
    // 删除nowCalls
    private static Boolean remove(Long flowId)
{       map.remove(flowId);     }
    // 新增nowCalls
    private static Boolean put(Long flowId, Integer nowCalls)
{       map.put(flowId, new AtomicInteger(nowCalls));
    } }
```

为了实现并发控制，我们要对流控规则类 FlowRule 进行改造，concurrencyLevel 是我们规定的最大并发数，clientTimeout 是 token client 掉线检测时间，也是判定 token 过期的标志，不同的 FlowRule 所对应的 clientTimeout 往往是相等的。sourceTimeout 是 client 端调用资源超时检测时间，这个值会根据资源的实际情况取值，不同的 FlowRule 所对应的 sourceTimeout 往往是不相等的。每次获取 token 的请求生成一个 tokenId 存储到一个底层存储结构为 ConcurrentLinkedHashMap 的 LocalCache 中；每个释放 token 的请求都会从 LocalCache 中删除该 tokenId。之所以会选择 ConcurrentLinkedHashMap 是因为其是线程安全的，能够按照访问顺序进行排序，这个特点对我们设计过期 tokenId 的删除（过期判定根据 clientTimeout 而定）非常有利。同时这个存储结构读写效率很高，和 ConcurrentHashMap 基本相同，为了防止 ConcurrentLinkedHashMap 超过设定的最大容量出现 LRU 会导致键的顺序发生变化，我们可以将最大容量设定为 Integer.MAX_VALUE。

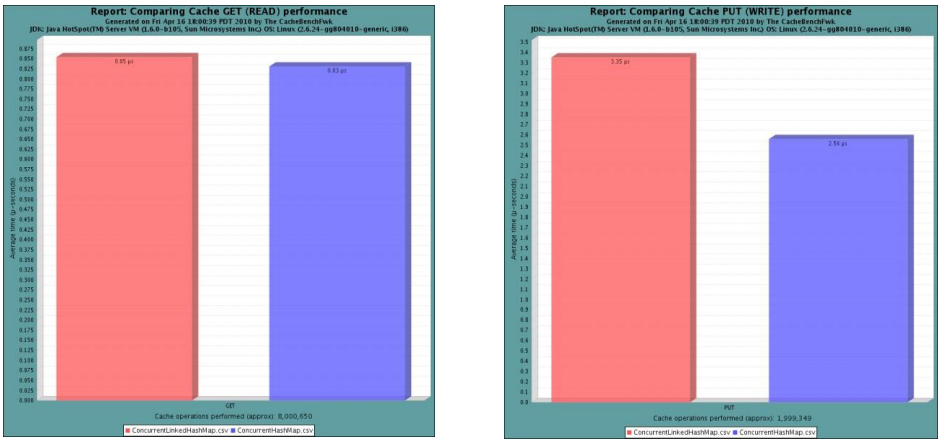

图 3.ConcurrentHashMap 和 ConcurrentLinkedHashMap get 和 put 性能比较

我们需要考虑 token client 掉线和资源调用超时的情况。可以通过设置过期时间来检测，分别是资源调用超时检测时间 sourceTimeout 和掉线检测时间 clientTimeout，超时检测在 client 端触发，如果调用资源超时 token client 会向 token server 请求释放或者继续保持 token。掉线检测在 server 端进行，如果超过掉线检测时间我们会触发对 token client 在线情况的检测，如果 token client 掉线则删除相应的 tokenId，如果未掉线则继续保存，当 token server 发现 tokenId 保存时间远大于超时时间 sourceTimeout（2-3 倍）则判定 client 端调用超时且无响应直接删除 tokenId。

同时如果不同 rule 设置的 tokenId 的过期时间（clientTimeout）相差很大，我们可以按照过期时间存储大 2-3 个不同的 LocalCache 中，这样做的目的是提高过期键的清除效率，

```
public class FlowRule {
  // 规则序列号
  private Long flowId;
  // 最大并发量
  private int concurrencyLevel;
  // client 端掉线检测时间，根据这个值判定节点过期
  private Long clientTimeout;
  // client 端调用超时检测时间
  private Long sourceTimeout;

  // 其他的常规属性
  //something others.....     public boolean
canPass(int aquireCount) {
        return NowCallsMap.get(flowId) + aquireCount >= concurrencyLevel;
    }           public  boolean  tryPass(int
aquireCount)  {                  int  now  =
NowCallsMap.get(flowId);
        // 尝试去增加  nowCalls
        if (canPass(aquireCount) && NowCallsMap.update(flowId, aquireCount)) {
            // 生成一个 TokenId
            Long TokenId = generateTokenId();
            // 生成存储节点
            cacheNode node = generateCacheNode(aquireCount);
            // 将 cacheNode 加入到 LocalCache(LinkedHashMAP 类型)
            LocalCache.putValue(node);
            // 期间异常返回  false
            return true;
        } else
{             return
false;
        }
    }
    public boolean relase(Long TokenId) {
        int aquireCount = LocalCache.getKey(TokenId).getaquireCout();
        // 删除本地存储中对应的节点
        LocalCache.removeKey(TokenId);
        // 更新当前并发量
        NowCallsMap.update(flowId, -aquireCount);
        // 期间异常返回 false
        return true;
    }
    // 其他的函数
   // something
else }
```

我们使用底层存储结构为 ConcurrentLinkedHashMap 的 LocalCache 对象存储 tokenId，其结构如下图所示，其存储节点为 CacheNode。为了能够在定期删除过期 tokenId 时对其对应 FLowRule 的 nowCalls 值进行操作，我们需要把 flowId 存到 CacheNode 里面去。

```
public class CacheNode {
  // 该存储节点对应 Token 的 TokenId
  private Long tokenId;
  // 这个节点的过期时间
  private Long clientTimeout;
  // 这个节点的过期时间
  private Long sourceTimeout;
  // 这个 Token 所对应的规则
  private Long flowId;
```

```
    // 该节点占用数量并发量
    private int aquireCount;
        // 一些其他的属性
      // something else;
    }
```

为了能够让用户自定义过期删除策略，我们在 LocalCache 类中设计了一个接口类型的属性 ExpireStrategy，表示过期删除策略，默认的策略是定期删除。为了保证线程安全，我们需要考虑锁或者其他的同步机制。

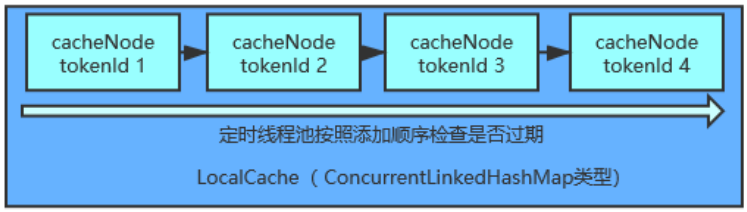

图 4. LocalCache 结构

```
public class LocalCache<K, V> {
    // 底层存储结构
    // private ConcurrentLinkedHashMap<K, CacheNode<K, V>>
    localCache;
    // 过期清理策略
    // private ExpireStrategy
    ExpireStrategy;
    // 构造函数
    // public LocalCache(ExpireStrategy
    expireStrategy) {
            this.localCache = new ConcurrentLinkedHashMap<>();
this.ExpireStrategy = expireStrategy;
      //启动定时清除过期键任务
        ExpireStrategy.removeExpireKey(localCache, null);
        }   //
获取缓存节点
    public V getValue(long TokenId) {}
    //      tokenId
    public V putValue(K key, V value) {}
    // 缓存 key-value，包含过期时间
    public V putValue(K key, V value, long expireTime) {}
    // 删除
    // public V removeKey(K key)
    {}
    // 清空所有内容
    // public void clear()
    {}

}
```

我们通过判定 CacheNode 过期（根据 CacheNode 的 clientTimeout 属性）来检测 token client 是否掉线。如果出现大面积失效的情况我们可以考虑降级策略。删除过期键时我们充分的使用了 ConcurrentLinkedHashMap 键有序的特性，能够最大限度提高我们效率，因为我们设置的过期时间为 token client 掉线检测时间，对 client 端来说这个时间一般情况下是相等的，因此过期的键往往会沉淀到底部。同时我们还能够控制删除时间和最大删除数、线程池大小，用来对整个清理过程进行精确的控制。同时在遍历的时候如果发现保存时间远大于 token client 调用资源超时时间（CacheNode 的 SourceTimeout 属性）则会判定超时且 token client 无响应，工作线程会删除对应的 tokenId。

```
public class RegularExpireStrategy {
  // 定期任务每次执行删除操作的次数,每次删除过期键值对的数量不超过这个数
  private long executeCount = 1000;
  // 定期任务执行时时间长度 【1 分钟】
  private long executeDuration = 1000 * 60;
  // 定期任务执行的频率
  private long executeRate = 60;


    public V removeExpireKey(ConcurrentLinkedHashMap< K, CacheNode<K, V>> localCache, K key) {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
        //定时周期任务, executeRate分钟之后执行
       executor.scheduleAtFixedRate(new MyTask(localCache), 0, executeRate, TimeUnit.MINUTES);
        return null;
    }



    private class MyTask implements Runnable {
        private ConcurrentLinkedHashMap<Long, CacheNode> localCache;

        public MyTask(ConcurrentLinkedHashMap<Long, CacheNode> localCache)
{           this.localCache = localCache;
        }

        @Override
public void run() {
            long start = System.currentTimeMillis();
            List<Long> keyList = localCache.keySet().stream().collect(Collectors.toList());
for (int i = 0; i < executeCount && i < keyList.size(); i++) {
        // 这个顺序遍历是按照 ConcurrentLinkedHashMap加入顺序遍历的, 能够增加检查效率
        Long key = keyList.get(i);
        // 如果发现保存时间远大于 client 端调用资源超时时间则
        // 会判定超时且 client 端无响应删除对应的 tokenId。
        if (localCache.get(key).getExpireTime()  - System.currentTimeMillis() <  0
             || isSourceTimeout()) {
          //和client  通信确认掉线
          if (clientShoutDown) {
                        // 找到对应的 FlowRule 同步 nowCalls
                        NowCallsMap.update(flowId, aquireCount);
                        // 删除存储的  flowId
                        localCache.remove(key);
                    }
                }
        //超时执行退出
        if (System.currentTimeMillis() - start >
        executeDuration) {
                    break;
                }
            }
        }
    }
}
```

三、自定义规则源扩展框架

能够发现，在第一种实现方式中我们提出的基于 ConcurrentLinkedHashMap 的存储结构类似于缓存，目前常用的缓存工具有 redis、guava LocalCache 等。我们只需将在第一种实现方式中的存储结构替换为上述的工具即可。 当然集群并发流控也能够通过其它的方式实现，为了能够让用户定制集群流控方案，我们应该将集群并发流控设计成可扩展的。

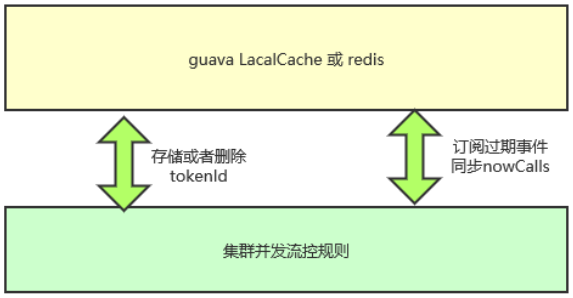

图 5. 其他可能的设计

为了尽可能小的改动现有的代码，本方案兼容了现有的集群限流的设计模式，将完全按照现在的设计思路补充流控规则。在集群流控类型中增加 CONCURRENCY_CONTROL 这个类型用以标识集群并发限流方式，同时在 RequestProcessorProvider 新增 Concurrency Request Processor，如果 hander 发现 tokenRequest 的类型是 CONCURRENCY_CONTROL，Sentinel 将从 RequestProcessorProvider 获取 Concurrency Request Processor 处理器处理并发限流请求，并发限流逻辑将以规则源的形式存在。如果集群并发流控不可用，将以本地并发流控兜底。

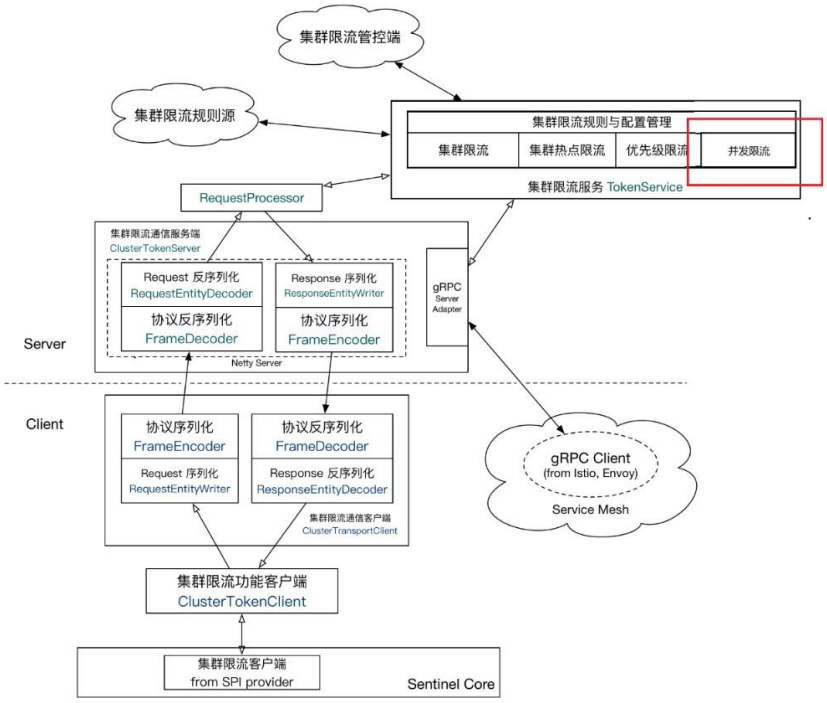

图 6. 集群限流方案架构

同时，为了将集群流控规则和整个限流框架解耦，我们规定了并发流控接口并且使用默认规则源实现它。用户在定制化自己的规则源的时候可以去实现这些接口中的方法，同时我们的默认规则源也会采用这种思路，让用户能够快速修改默认规则。

```
public interface TokenService {
    // 获取 Token
    TokenResult requestToken(Long ruleId, int acquireCount, boolean prioritized);
    // 释放 Token
    TokenResult releaseToken(Long tokenId);
```

```
    // 保持Token
    TokenResult keepToken(Long tokenId);
}
```