Alibaba Summer of Code Proposal

Proposal for Distributed Concurrent Flow Control

About Project

***Problem Description and Work Abstract**

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

In order to achieve cluster concurrent flow control, we use the token recovery mechanism to calculate the current concurrency, and use *ConcurrentLinkedHashMap t*o 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 developers to quickly implement their own cluster concurrent flow control rules, we have implemented an interface-based framework for developers.

***Implementation plan:**

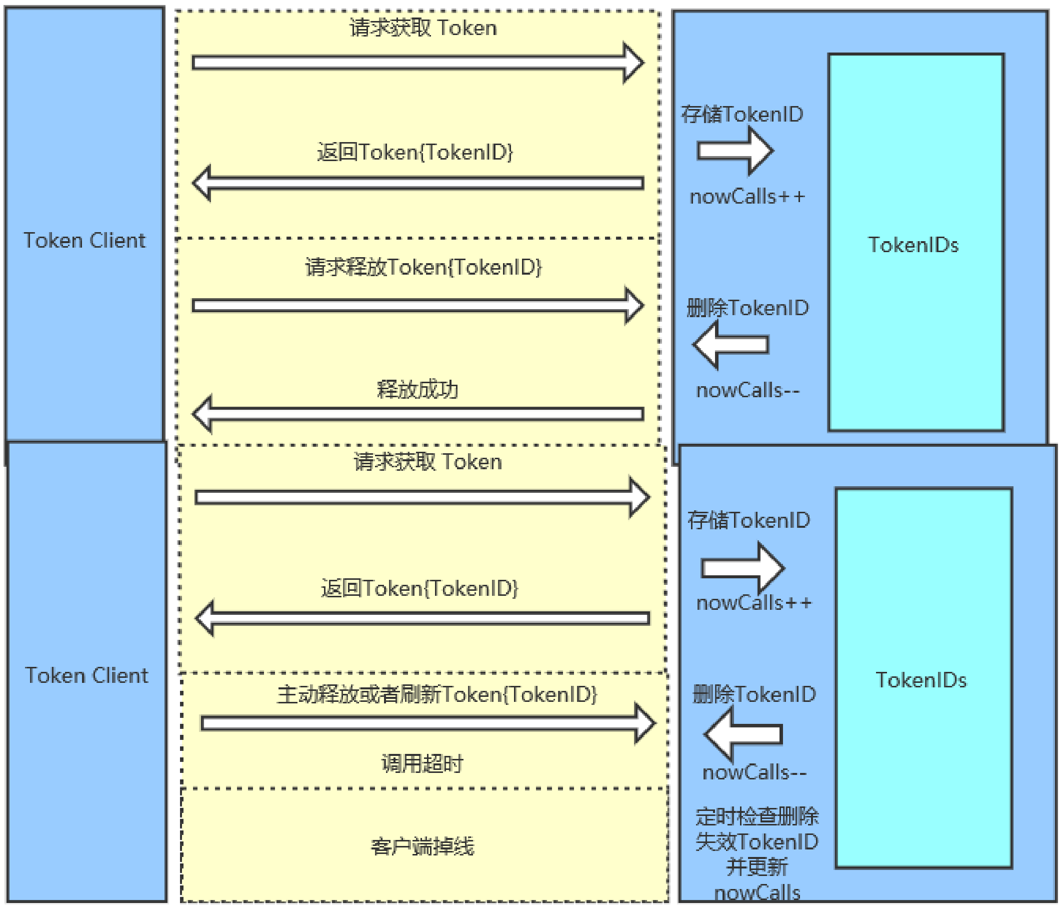
1. Cluster Concurrent Flow Control Principle

We can use *concurrencyLevel* to express the maximum concurrency. To achieve the goals, we should ensure the current concurrency *nowCalls* is less than *concurrencyLevel*. It seems that no other information needs to be stored. However, the token client may be offline or the resource may be called over time, which will cause statistical error of *nowCalls*. At the same time, if we only store *nowCalls*, it is not easy for the Dashboard to display relevant information (which calls are in progress, how long does this call take, etc.). These data are very important for us to monitor the system.

In order to realize cluster flow control, we must store the information of each call, find and clear those abnormal calls. **When token client acquires a token, it will obtain the *tokenId* issued by the token server. Token server will store the *tokenId* and related information in a local cache. When the call ends, the token client will carry the *tokenId* to the token server to release the resources, and the token server will delete the *tokenId* corresponding to the call.** Through such operations, we can count the *nowCalls*, that is, the concurrent volume. **If the resource is called time out or the token client goes offline, the token server will try to find and delete the corresponding *tokenId*, so as to obtain the accurate concurrency *nowCalls*.**

P 1. acquiring the token successfully

P2. situations of the resource called time out or the token client offline



2. Cluster Concurrent Flow Rule Design

We use a *ConcurrentHashMap<Long, AtomicInteger>* type structure to store *nowCalls* corresponding to rules, where the key is *flowId* and the value is *nowCalls*. Because *nowCalls* may be accessed and modified by multiple threads, we consider to design it as an *AtomicInteger* class or modified by the *valotile* keyword. Each newly created rule will add a *nowCalls* object to this map. If the concurrency corresponding to a rule changes, we will update the corresponding *nowCalls* in real time. **Each request to obtain a token**

will increase the *nowCalls*; and the request to release the token will reduce the *nowCalls*.

```
public class NowCallsMap {
    // define the data structure
  static ConcurrentHashMap<Long,  AtomicInteger>  map = new ConcurrentHashMap<Long,
AtomicInteger>();
    // ensure the concurrent safe by AtomicInteger
  private static void update(Long flowId, Integer count) {
        map.get(flowId).getAndAdd(count);
    }
    // acquire nowCalls
    private static Integer get(Long flowId) {
        map.get(flowId);
    }
    // reduce nowCalls
    private static Boolean remove(Long flowId) {
        map.remove(flowId);
    }
    // add nowCalls
    private static Boolean put(Long flowId, Integer nowCalls) {
        map.put(flowId, new AtomicInteger(nowCalls));
    }
}
```

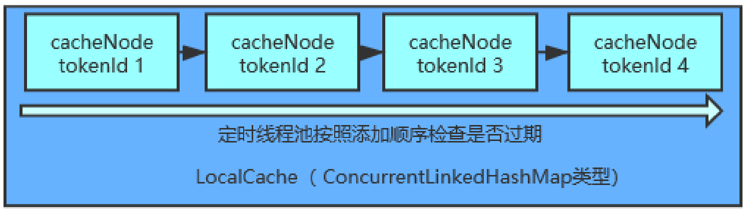
In order to achieve concurrent flow control, we need to reform *FlowRule*. *concurrencyLevel* is the maximum number of concurrency, *clientTimeout* is the token client disconnection detection time, and it is also a sign to determine a token expires. The *clientTimeout* of different *FlowRules* are often equivalent. *sourceTimeout* is the client's call resource timeout detection time. This value will be based on the actual situation of the resource. The *sourceTimeout* of different FlowRules is often not equal. **Each request to obtain a token generates a *tokenId* and stores it in a *LocalCache*, whose underlying storage structure is *ConcurrentLinkedHashMap*; Each request to release a token deletes the *tokenId* from the *LocalCache*.**

```
public class FlowRule {
  // rule's flueId
  private Long flowId;
  // the max concurrency
  private int concurrencyLevel;
  // client offline detection time, which is the token expiration time
  private Long clientTimeout;
  // client call resource overtime detection time
  private Long sourceTimeout;
  // something others......
    public boolean canPass(int acquireCount) {
        return NowCallsMap.get(flowId) + acquireCount >= concurrencyLevel;
    }

    public boolean tryPass(int acquireCount) {
        int now = NowCallsMap.get(flowId);
        // try to add the nowCalls
        if (canPass(acquireCount) && NowCallsMap.update(flowId, acquireCount)) {
            // generate a TokenId
            Long TokenId = generateTokenId();
            // generate a cacheNode
            cacheNode  node  =  generateCacheNode(acquireCount);
            // put cacheNode to LocalCache(ConcurrentLinkedHashMAP)
            LocalCache.putValue(node);
            // if happen exception, return false
            return true;
        } else {
            return false;
        }
    }

    public boolean relase(Long TokenId) {
        int acquireCount = LocalCache.getKey(TokenId).getacquireCout();
```

```
            // delete the cacheNode from the LocalCache
            LocalCache.removeKey(TokenId);
            // update the nowCalls
            NowCallsMap.update(flowId,  -acquireCount);
            // if happen exception, return false
            return true;
        }
      // something else
}
```

We need to consider the situation that the token client goes offline or the resource call times out. **It can be detected by *sourceTimeout* and *clientTimeout*. The resource calls timeout detection is triggered on the token client. If the resource is called over time, the token client will request the token server to release token or refresh the token. The client offline detection is triggered on the token server. If the offline detection time is exceeded, token server will trigger the detection token client's status. If the token client is offline, token server will delete the corresponding *tokenId*. If it is not offline, token server will continue to save it.**

We use *LocalCache* to store the *tokenId*, whose underlying storage structure is *ConcurrentLinkedHashMap*, Its structure is shown in the following figure. Its storage node is CacheNode. **When the expired *tokenId* is deleted regularly, token server will synchronize *nowCalls* in time**



P4.LocalCache structure

```
public class CacheNode {
  // the TokenId of the token
  private Long tokenId;
  // the client goes offline detection time
  private Long clientTimeout;
  // the resource called over time detection time
  private Long sourceTimeout;
  // the flow rule id  corresponding to the token
  private Long flowId;
  // the number this token occupied
  private int acquireCount;
  // something else;
}
```

In order to allow users to customize the expired deletion strategy, we designed an interface type property *ExpireStrategy* in the *LocalCache* class. The default strategy is to delete expire token regularly. In order to ensure thread safety, we need to **consider locks or other synchronization mechanisms in *the LocalCache* design. The reason I choose *ConcurrentLinkedHashMap* is that it is thread-safe and can be stored according to the order of access.** This feature is very beneficial to delete of expired *tokenId* (expiry judgment is based on *clientTimeout).* At the same time, this storage is efficiency, which is basically the same as *ConcurrentHashMap*. But LRU will cause the key sequence to change when it exceeds the set maximum capacity. In order to prevent *ConcurrentLinkedHashMap* from LRU, we can set its maximum capacity as *Integer.MAX_VALUE*.

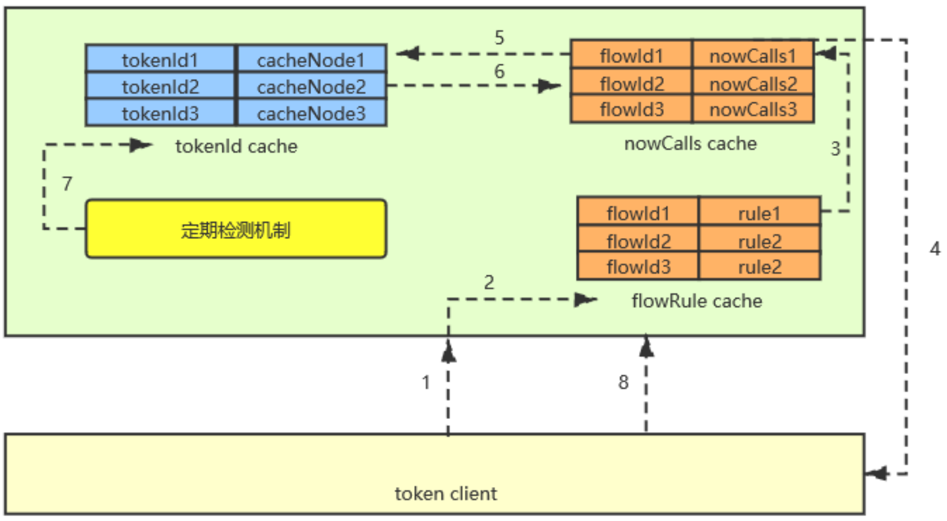
```
public class LocalCache<K, V> {
  // storage structure
  private ConcurrentLinkedHashMap<K,  CacheNode<K,  V>> localCache;
  // expire strategy
  private  ExpireStrategy  ExpireStrategy;
  public LocalCache(ExpireStrategy expireStrategy) {
        this.localCache = new ConcurrentLinkedHashMap<>();
        this.ExpireStrategy = expireStrategy;
    // Start the task of regularly clearing expired keys
    ExpireStrategy.removeExpireKey(localCache,  null);
    }
  public V getValue(long  TokenId)  {}
  public V putValue(K key, V value) {}
  public V putValue(K  key, V value, long expireTime)  {}
  public V removeKey(K  key) {}
  public void clear() {}

}
```

We detect whether the token client is offline by the *CacheNode*'s expiration strategy (according to the *CacheNode*'s *clientTimeout* attribute). **When deleting the expired key, we fully used the *ConcurrentLinkedHashMap*'s key ordering feature, which can maximize the detection efficiency. Because the expiration time we set is the token client disconnection detection time, which are the same for token clients. So expired keys will settle to the bottom.** At the same time, we can also control the deletion time, the maximum token number of deletions and the size of the thread pool, which will help token server precisely control the entire cleaning process. If the token server finds that token's save time is much greater than *sourceTimeout*, token server will determine that the resource is called timeout and the token client can't respond. The worker thread will delete the corresponding *tokenId*.

```
public class RegularExpireStrategy {
  // The max number of token deleted each time,
  // the number of expired key-value pairs deleted each time does not exceed this number
  private long executeCount = 1000;
  // Length of time for task execution
  private  long executeDuration  = 1000 * 60;
  // Frequency of task execution
  private  long executeRate  = 60;

    public V removeExpireKey(ConcurrentLinkedHashMap<K, CacheNode<K, V>> localCache, K key) {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
        // start of a periodic task
       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++) {
        // use ConcurrentLinkedHashMap to improve the expiration detection progress
        Long key = keyList.get(i);
        // If  we find that token's save time is much longer than the client's
        // call resource timeout time, token will be determined to
        // timeout and the client go wrong
        if (localCache.get(key).getExpireTime()  - System.currentTimeMillis()  < 0) {
           //  communicate with the client to confirm disconnection
          if (clientShoutDown)  {
```

```
                        // find the corresponding FlowRule to sync nowCalls
                        NowCallsMap.update(flowId, acquireCount);
                        // remove the token
                        localCache.remove(key);
                    }
                }
        // time out execution exit
        if (System.currentTimeMillis() - start > executeDuration) {
                    break;
                }
            }
        }
    }
}
```

In short, the token server runs like the following picture.

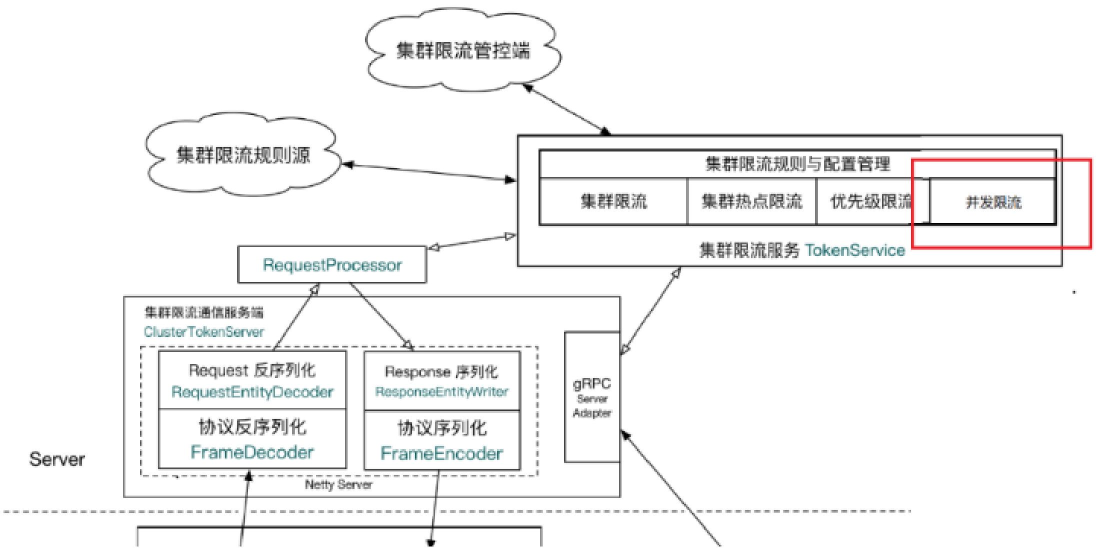


```
    1. Token client requests a token/token client
    2. Locate nowCalls through the flowId bound to the rule
    3. Compare nowCalls with the maximum concurrency of this rule
    4. Reach the maximum concurrency and reject the token request
    5. Distribute tokenId, package CacheNode and store in tokenId cache
    6. Add and delete operations in tokenId cache will be synchronized to nowCalls
    7. The tokenId cache regularly detects the disconnection of token clients and the timeout
of resource calls through the expiration mechanism
    8. The resource called timeout and released the token by token client
```

3. Extensible Framework for Flow Control

It can be found that, the storage structure based on *ConcurrentLinkedHashMap* is similar to cache. So we can use zookeeper, redis and guava *LocalCache* to implement flow rule. We only need to replace *ConcurrentLinkedHashMap* with the caches described above. Of course, cluster concurrent flow control can also be achieved in other ways. In order to allow users to customize cluster flow control solutions, we should design cluster concurrent flow control to be scalable.

In order to modify the existing code as little as possible, my solution will be compatible with the existing cluster current limit design mode. We will add the CONCURRENCY_CONTROL type to identify the cluster concurrent flow control mode. At the same time, add a *ConcurrencyRequestProcessor* to the *RequestProcessorProvider*. If the handler finds that the type of *tokenRequest* is CONCURRENCY_CONTROL, Sentinel will obtain the *ConcurrencyRequestProcessor* from the *RequestProcessorProvider* to handle flow control. **Concurrent flow control logic will exist in the form of rule source. If cluster concurrent flow control is not available, local concurrent flow control will be used.**



P6. cluster flow control scheme architecture

At the same time, in order to decompress cluster concurrent flow control and the entire flow-control framework, we specify concurrent flow control interfaces and use flow control rule to implement it. Developers can implement the methods in these interfaces when customizing their own flow control rule. At the same time, our default rule will also adopt this idea, allowing users to quickly modify the default rules.

```
public interface TokenService {
    // acquire Token
    TokenResult requestToken(Long ruleId, int acquireCount, boolean prioritized);

    // release Token
    TokenResult releaseToken(Long tokenId);

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

* Proposal in Chinese

一、并发流控思路

最大并发量可以用符号 concurrencyLevel 表示，把当前执行数量 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。**

二、流控规则设计

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**

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。

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。

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。**

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。

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。

三、可扩展并发限流框架

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。

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。**

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。