

# 15.精确掌控并发：滑动时间窗口算法在分布式环境下并发流量控制的设计与实现\_V20240116

---

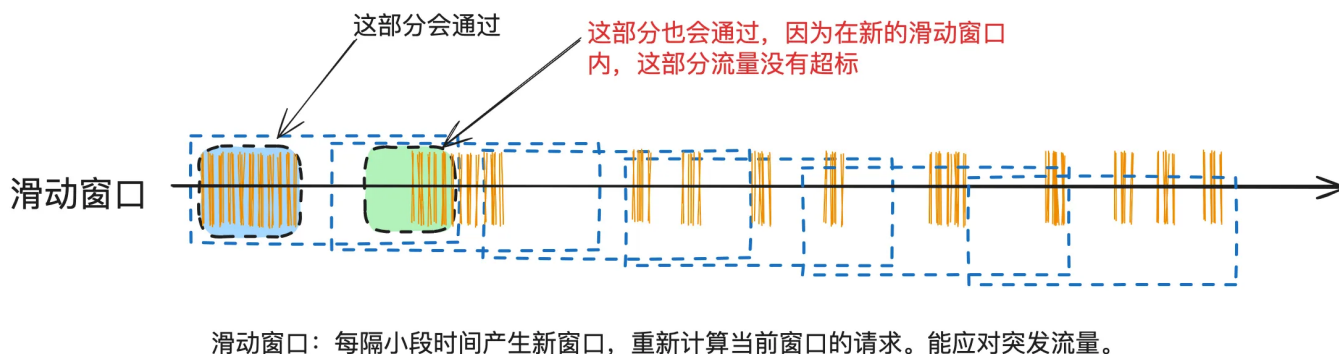
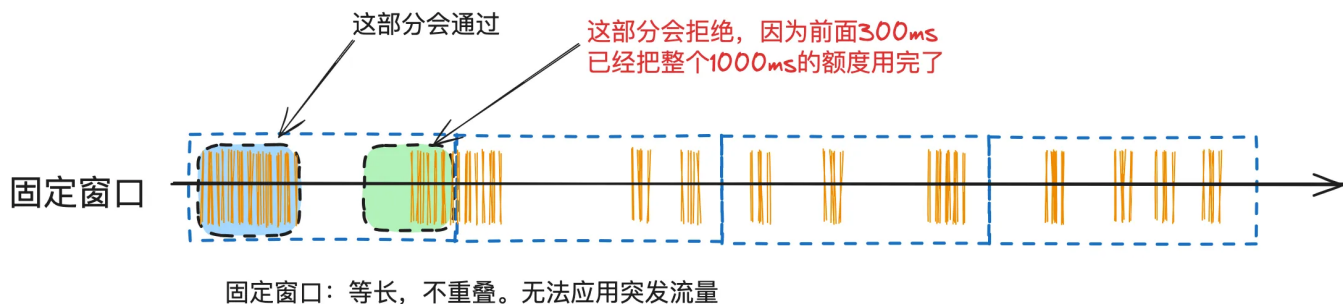
- 1. 滑动时间窗口原理
- 2. 滑动时间窗口在支付系统中的应用场景
- 3. 使用redis实现的核心代码
- 4. 注意事项
- 5. 结束语

上一篇介绍了【固定时间窗口算法】在支付渠道限流的应用以及使用redis实现的核心代码。

本篇重点讲清楚分布式环境下【滑动时间窗口算法】原理和应用场景，以及使用reids实现的核心代码。

## 1. 滑动时间窗口原理

滑动窗口算法是一种更为灵活的流量控制方案，它比固定窗口算法能更平滑地处理突发流量。在滑动窗口中，时间窗口是重叠的，这意味着流量的计算是基于过去的一段连续时间内发生的事件。



#### 工作流程：

1. 窗口定义：确定窗口的大小，例如1秒钟，并设置窗口的滑动间隔，比如100毫秒。
2. 计数与滑动：每个窗口都有自己的计数器。当一个新请求到达时，增加当前时间窗口及其前面相邻的窗口的计数。
3. 限制检查：如果任何连续时间段内的请求总数超过阈值，则拒绝新的请求。
4. 窗口更新：随着时间的推移，不断向前滑动窗口，并更新相应的计数器。

## 2. 滑动时间窗口在支付系统中的应用场景

滑动时间窗口在支付系统中的应用场景主要也是各种精确限流，比如把上一篇讲的固定时间窗口算法中，我们对外部渠道请求会做限流，那么就可以升级到滑动时间窗口，以提高精度。

只要是API限流，都可以使用。

## 3. 使用redis实现的核心代码

滑动窗口可以通过队列或循环数组来实现。每个窗口对应队列中的一个元素，记录该窗口期间的请求数。当时间滑动时，更新队列头部的元素，并可能将旧的元素出队。

在Redis中，可以使用列表或有序集合来模拟这种滑动窗口。下面是一个Redis实现的示例，使用有序集合（sorted set）来实现了滑动时间窗口算法：

```
Java |  
  
1  /**  
2   * redis限流操作类  
3   */  
4   @Component  
5   public class RedisLimitUtil {  
6       @Autowired  
7       private RedisTemplate<String, Object> redisTemplate;  
8       // 滑动时间窗口大小  
9       private static final long WINDOW_SIZE_IN_SECONDS = 1000;  
10  
11      /**  
12       * 判断是否限流  
13       * 这里不考虑超过long最大值的情况，系统在达到long最大值前就奔溃了。  
14       */  
15      public boolean isLimited(String key, String requestId, long countLimit) {  
16          // 使用Redis的多个命令来实现滑动窗口  
17          redisTemplate.zremrangeByScore(key, 0, currentTimeMillis - WINDOW_SIZE_IN_SECONDS);  
18          long count = redisTemplate.zcard(key);  
19  
20          if (countLimit >= count) {  
21              redisTemplate.zadd(key, currentTimeMillis, requestId);  
22              return true;  
23          } else {  
24              return false;  
25          }  
26      }  
27  }
```

每个请求都以其发生的时间戳作为分数(SCORE)存储在集合中。通过移除旧于当前时间窗口的请求来维护滑动窗口。通过检查集合中的元素数量，以确定是否超过了设定的最大请求数。

- zremrangeByScore 用于移除窗口之外的旧请求。
- zcard 获取当前窗口内的请求数量。
- zadd 将新请求添加到集合中。

使用：PayServiceImpl

```
1  /**
2   * 支付服务示例
3   */
4  public class PayServiceImpl implements PayService {
5      @Autowired
6      private RedisLimitUtil redisLimitUtil;
7
8      @Override
9      public PayOrder pay(PayRequest request) {
10         if (isLimited(request)) {
11             throw new RequestLimitedException(buildExceptionMessage(request
12         t));
13         }
14         // 其它业务处理
15         ... ..
16     }
17
18     /**
19     * 限流判断
20     */
21     private boolean isLimited(PayRequest request) {
22         // 限流KEY, 这里以[业务类型 + 渠道]举例
23         String key = request.getBizType() + request.getChannel();
24         // 限流值
25         Long countLimit = countLimitMap.get(key);
26
27         // 如果key对应的限流值没有配置, 或配置为-1, 说明不限流
28         if (null == countLimit || -1 == countLimit) {
29             return false;
30         }
31
32         return redisLimitUtil.isLimited(key, request.getRequestId(), count
33         Limit);
34     }
35 }
```

需要注意一点的是，这次需要传入requestId进去，用于保存这个requestId在redis有序队列里的分数，用于计数和清理。

其它的注释写得比较清楚，没什么补充的。

## 4. 注意事项

一些分布式服务框架，为了更高的可靠性，他们使用的是本地计算。比如接口限流1000TPS，一共有20台应用服务器，框架就会把计算出每台机器是50个TPS，下发给所有的应用服务器，在服务器上线、下线过程中，可能会有一段时间是不准确的。

但在渠道限流应该中，因为每个渠道的流量都不太高，所以可以使用这种redis方案。且精度更高，不受应用服务器的上、下线影响。

另外，在分布式系统中，**需要确保不同节点之间的时间同步**，以保证流量计算的准确性。如果应用服务器之间的时间不同步，那么流量就会计算错误。

## 5. 结束语

分布式流控有很多实现方案，通过把固定时间窗口算法升级为滑动时间窗口算法，我们对流量控制的精度会大幅提升。

下一篇会介绍漏桶原理及实现。漏桶和令牌桶的特点是请求进来先保存起来，然后按一定的速度发送出，而不是超过阈值就拒绝。

这是《百图解码支付系统设计与实现》专栏系列文章中的第（15）篇。和墨哥（隐墨星辰）一起深入解码支付系统的方方面面。

欢迎转载。

系列文章PDF合集，不定时更新：

Github: <https://github.com/yinmo-sc/Decoding-Payment-System-Book>

百度网盘: [https://pan.baidu.com/s/1l5dlR8SoGH\\_Iy\\_8Hbv8mXQ?pwd=0000](https://pan.baidu.com/s/1l5dlR8SoGH_Iy_8Hbv8mXQ?pwd=0000)

公众号：隐墨星辰。



# 微信搜一搜



## 隐墨星辰

有个小群不定时解答一些问题或知识点，有兴趣的同学可先加微信（yinmo\_sc）后进入，添加微信请备注：加支付系统设计与实现讨论群。



隐墨星辰



扫一扫上面的二维码图案，加我为朋友。