14.精确掌控并发：固定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

这是《百图解码支付系统设计与实现》专栏系列文章中的第（14）篇，也是流量控制系列的第（1）篇。点击上方关注，深入了解支付系统的方方面面。

本篇主要介绍分布式场景下常用的并发流量控制方案，包括固定时间窗口、滑动时间窗口、漏桶、令牌桶、分布式消息中间件等，并重点讲清楚固定时间窗口应用原理和应用场景，以及使用reids实现的核心代码。

在非支付场景，也常常需要用到这些并发流量控制方案。

1. 前言

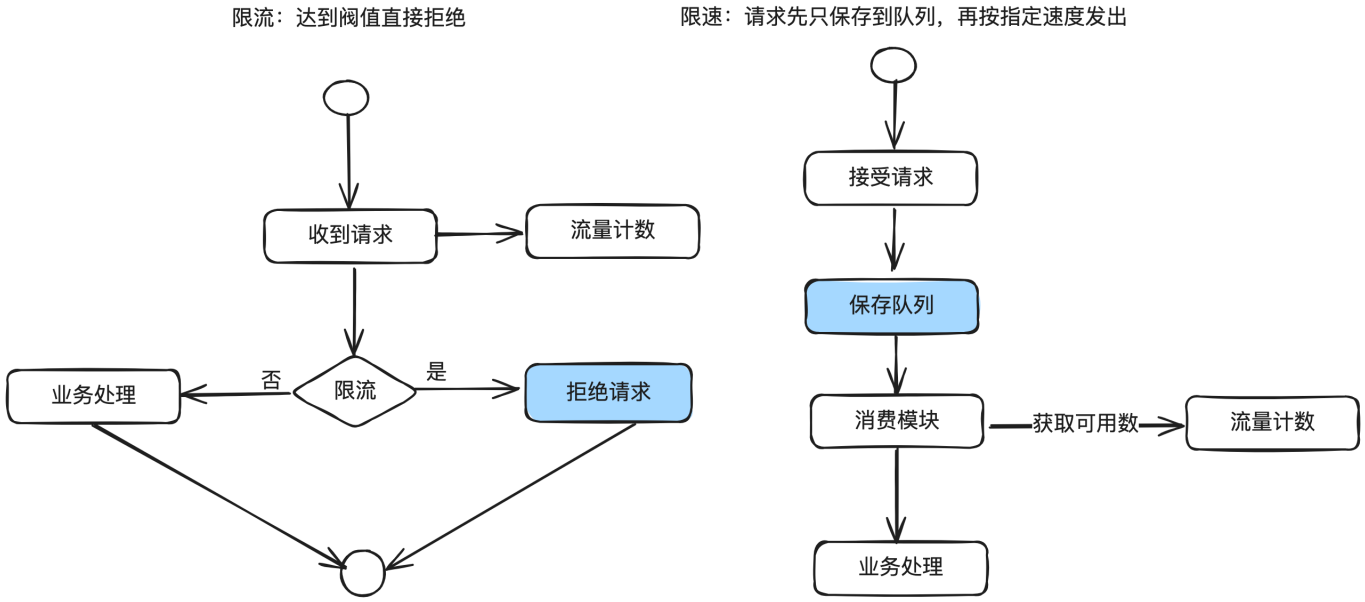
在互联网应用里面，并发流量控制无所不在。在支付系统中，流量控制同样是一个关键的技术方面，主要用于确保系统的稳定性和可靠性，尤其在高流量的情况下。以下是一些主要使用流量控制的场景：

1. **对外API限流**：对外提供的API（如支付接口）需要限流来保护后端服务不会过载。
2. **保护外部渠道**：大促时，对下流渠道的支付流量要做削峰填谷，避免突发流量把渠道打挂。
3. **保护内部应用**：大促时，内部各应用要根据流量模型配置限流值，避免形成雪崩。
4. **满足外部退款限流要求**：电商批量提交退款时，支付系统内部要在分布式集群环境下对某个渠道实现低至1TPS的退款并发，避免超过渠道退款并发导致大批量失败。

特别说明的是，流量控制通常包括**限流**和**限速**。

限流：就是流量达到一定程度，超过的流量会全部立即拒绝掉，也就是快速失败。比如上面的API限流。

限速：一般是指接收流量后，先保存到队列中，然后按指定的速度发出去，如果超过队列最大值，才会拒绝。比如上面的支付流量和退款流量打到外部渠道。



另外，支付和退款流量控制虽然都是流量控制，但有一些细小的区别：

1. 支付的限流TPS通常比较高，从十几TPS到几百TPS都有，排队时效性要求很高，秒级内就要付出去。
2. 退款的限流TPS通常比较低，在国外的基础设施建设很差，甚至部分渠道要求退款1TPS。但是排队时效性要求很低，几天内退出去就行。

2. 几种方案对比

固定窗口：算法简单，对突然流量响应不够灵活。超过流量的会直接拒绝，通常用于限流。

滑动窗口： 算法简单，对突然流量响应比固定窗口灵活。超过流量的会直接拒绝，通常用于限流。

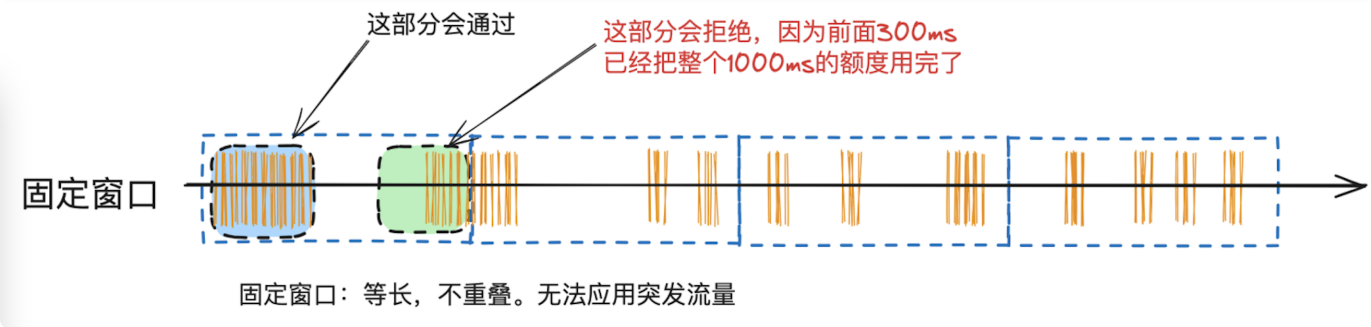
漏桶算法：在固定窗口的基础之上，使用队列缓冲流量。提供了稳定的流量输出，适用于对流量平滑性有严格要求的场景。后面会介绍如何应用到外部渠道退款场景。

令牌桶算法：在滑动窗口的基础之上，使用队列缓冲流量。能够允许一定程度的突发性流量，但实现较为复杂。

分布式消息中间件：如Kafka和RabbitMQ等，能够有效地对消息进行缓冲和管理，增加系统复杂性，且如果需要精确控制流量还需要引入额外的机制。后面会介绍如何应用到外部渠道支付场景。

Sentinel：阿里开源的流控与熔断利器，提供实时的监控、熔断、降级、限流等功能。后面会单独介绍。

3. 固定时间窗口原理



固定窗口算法，也称为时间窗口算法，是一种流量控制和速率限制策略。此算法将时间轴分割成等长、不重叠的时间段，称为“窗口”。每个窗口都有一个独立的计数器，用于跟踪窗口期间的事件数量（如API调用、数据包传输等）。

固定窗口算法的好处是简单，缺点也很明显，就是无法应对突发流量，比如每秒30并发，如果前100ms来了30个请求，那么在10ms内就会把30个请求打出去，后面的900ms的请求全部拒绝。

工作流程：

1. 窗口定义：首先确定窗口大小，比如1秒钟。
2. 计数：每当发生一个事件（比如一个请求到达），就在当前窗口的计数器上加一。
3. 限制检查：如果当前窗口的计数器达到预设阀值，则拒绝新的请求。直到下一个窗口开始。
4. 窗口重置：当前窗口结束时，计算数器重置为零，开始下一个窗口计数。

4. 固定时间窗口在支付系统中的应用场景

主要用于简单的**限流**。比如在渠道网关做限流，发送渠道的请求最大不能超过测算出来的值，避免渠道侧过载，可能会导致支付请求批量失败。

是有损服务的一种实现方式。

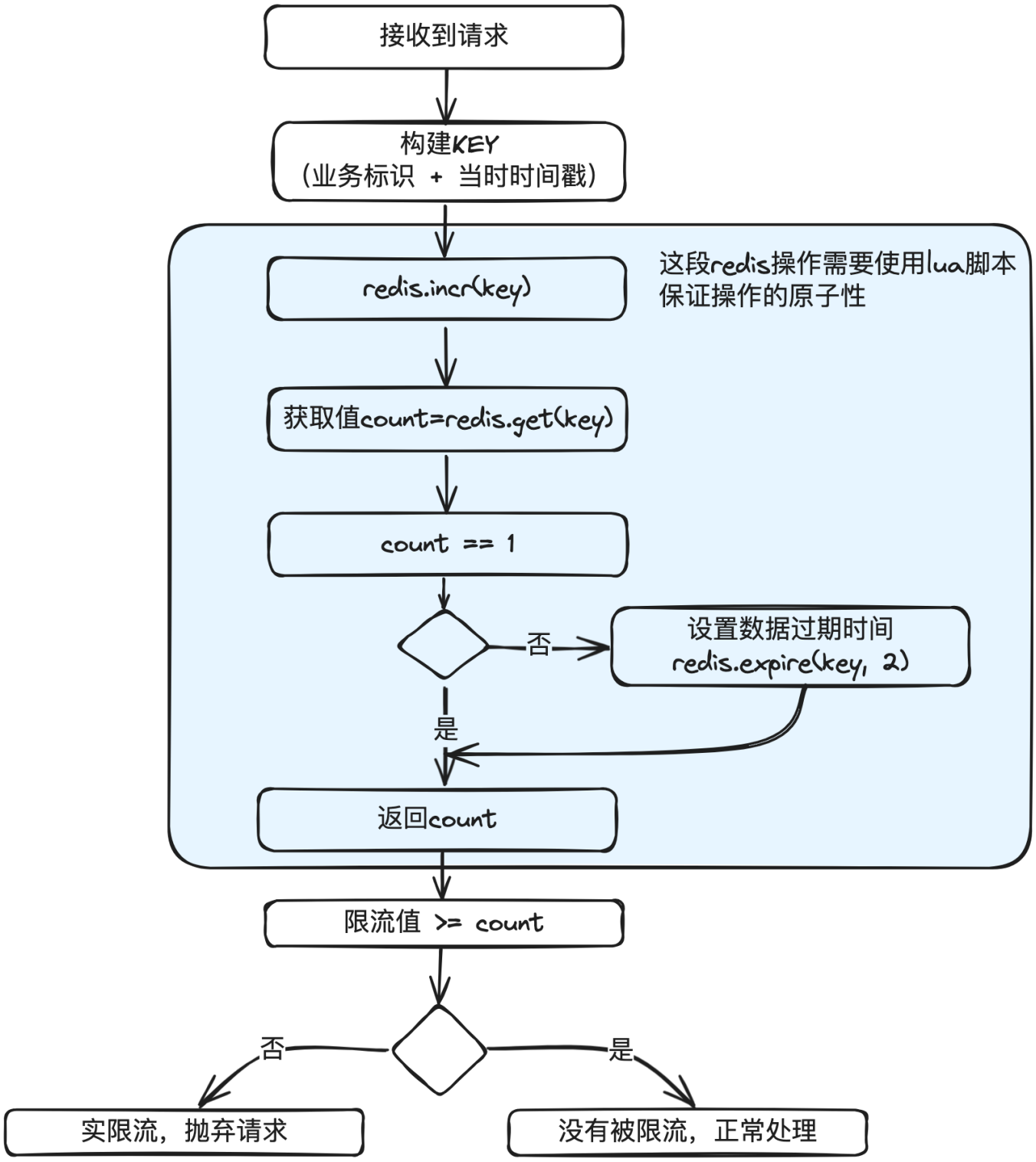
5. 使用redis实现的核心代码

为什么选择redis?因为在分布式场景下，限流需要有一个集群共用的计算数来保存当前时间窗口的请求量，redis是一个比较优的方案。

场景示例：WPG渠道的支付每秒不能超过20TPS。

那么设计key=“WPG–PAY” + 当前时间戳（精确到S），数据过期时间为2S（这个过期时间主要是兼容各服务器的时间差）。

下面是流程图：



lua脚本：limit.lua

```lua
local key = KEYS[1]
-- 默认为2S超期，精确到S级。也可以改造成由外面传进来 --
local expireTime = 2
-- 先自增，如果不存在就自动创建 --
redis.incr(key);
local count = tonumber(redis.call("get", key))
-- 如果结果为1，说明是新增的，设置超时时间 --
if count == 1 then
    redis.call("expire", key, expireTime)
end
return count;
```

redis操作类：RedisLimitUtil

Java

```java
/**
 * redis限流操作类
 */
@Component
public class RedisLimitUtil {
    // 限流脚本
    private static final String LIMIT_SCRIPT_LUA = "limit.lua";
    @Autowired
    private RedisTemplate<String, Object> redisTemplate;
    private DefaultRedisScript<Long> limitScript;

    /**
     * 缓存脚本
     */
    @PostConstruct
    public void cacheScript() {
        limitScript = new DefaultRedisScript();
        limitScript.setScriptSource(new ResourceScriptSource(new ClassPathResource(LIMIT_SCRIPT_LUA)));
        limitScript.setResultType(Long.class);

        List<Boolean> cachedScripts = redisTemplate.getConnectionFactory().getConnection().scriptExists(
            limitScript.getSha1());
        // 需要缓存
        if (CollectionUtils.isEmpty(cachedScripts) || !cachedScripts.get(0)) {
            redisTemplate.getConnectionFactory().getConnection().
            scriptLoad(redisTemplate.getStringSerializer().serialize(limitScript.getScriptAsString()));
        }
    }

    /**
     * 判断是否限流
     * 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。
     */
    public boolean isLimited(String key, long countLimit) {
        Long count = redisTemplate.execute(limitScript, Lists.newArrayList(key));
        return countLimit >= count;
    }

}
```

使用：PayServiceImpl

```java
/**
 * 支付服务示例
 */
public class PayServiceImpl implements PayService {
    @Autowired
    private RedisLimitUtil redisLimitUtil;

    @Override
    public PayOrder pay(PayRequest request) {
        if (isLimited(request)) {
            throw new RequestLimitedException(buildExceptionMessage(request));
        }

        // 其它业务处理
        ... ...
    }

    /*
     * 限流判断
     */
    private boolean isLimited(PayRequest request) {
        // 限流KEY，这里以[业务类型 + 渠道]举例
        String key = request.getBizType() + request.getChannel();
        // 限流值
        Long countLimit = countLimitMap.get(key);

        // 如果key对应的限流值没有配置，或配置为-1，说明不限流
        if (null == countLimit || -1 == countLimit) {
            return false;
        }

        return redisLimitUtil.isLimited(key + buildTime(), countLimit);
    }
}
```

注释写得比较清楚，没有什么需要补充的。

6. 结束语

分布式流控有很多实现方案，使用redis实现的固定时间窗口是最简单的方案，而且也非常实用，应付一般的场景已经足够使用。

下一篇会介绍滑动时间窗口算法及实现。

这是《百图解码支付系统设计与实现》专栏系列文章中的第（14）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。

Github（PDF文档全集，不定时更新）：https://github.com/yinmo-sc/Decoding-Payment-System-Book

公众号：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。