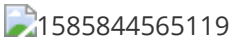
系统设计

网关-》限流/负载均衡-》服务集群（redis-》rabbitmq-》mysql）



短视频评论表设计

首先我们先创建一个用户表

```
CREATE TABLE users (
    id INT PRIMARY KEY auto_increment,
    username VARCHAR(20) UNIQUE NOT NULL
);
```

评论表

将评论拆分为评论表和回复表，评论挂在各种视频下面，而回复挂在评论下面。

评论表设计如下：

表字段	字段说明
id	主键
video_id	所属视频id
content	评论内容
from_uid	评论用户id
time	评论时间

回复表

comment_id字段表示该回复挂在的根评论id，可以直接通过评论id一次性的找出该评论下的所有回复，然后通过程序来编排回复的显示结构。 通过适当的冗余来提高性能也是常用的优化手段之一。

reply_type：表示回复的类型，是针对评论的回复，还是针对回复的回复，通过这个字段来区分两种情景。

reply_id：表示回复目标的id，如果reply_type是comment的话，那么reply_id = commit_id，如果reply_type是reply的话，这表示这条回复的父回复。

表字段	字段说明
id	主键
comment_id	根评论id
reply_type	回复类型
reply_id	回复目标id
content	回复内容
user_id	用户id

视频表

视频表，视频表中有视频自己的id、用户的id、标题内容发表时间，视频url

点赞关系表

因为点赞是用户与视频多对多的关系，所以要建立点赞关系表，有视频id和用户id，点赞状态

以及点赞数量表 视频id + 点赞数

操作结束
接下来是举例

```
注册用户：张三、李四
INSERT INTO users (username) VALUES ('张三');
INSERT INTO users (username) VALUES ('李四');

张三发表一篇文章《SQL的设计》，内容是‘请看本文’
SELECT id FROM users WHERE username = "张三";
INSERT INTO article (user_id, title, content, published_at)
            VALUES (1, "SQL的设计", "请看本文", "2020-2-19 14:10:00");

李四评论了张三的文章《SQL的设计》，内容是“说的对”
SELECT id FROM users WHERE username = "李四";
SELECT id FROM article WHERE title = "SQL的设计";
INSERT INTO comments (user_id, article_id, content, published_at)
            VALUES (2, 1, "说的对", "2020-2-19 14:20:00");

李四点赞了张三的文章
SELECT id FROM users WHERE username = "李四";
SELECT id FROM article WHERE title = "SQL的设计";
INSERT INTO good_relation (user_id, article_id)
            VALUES (2, 1);
```

点赞系统设计

Redis + Mysql

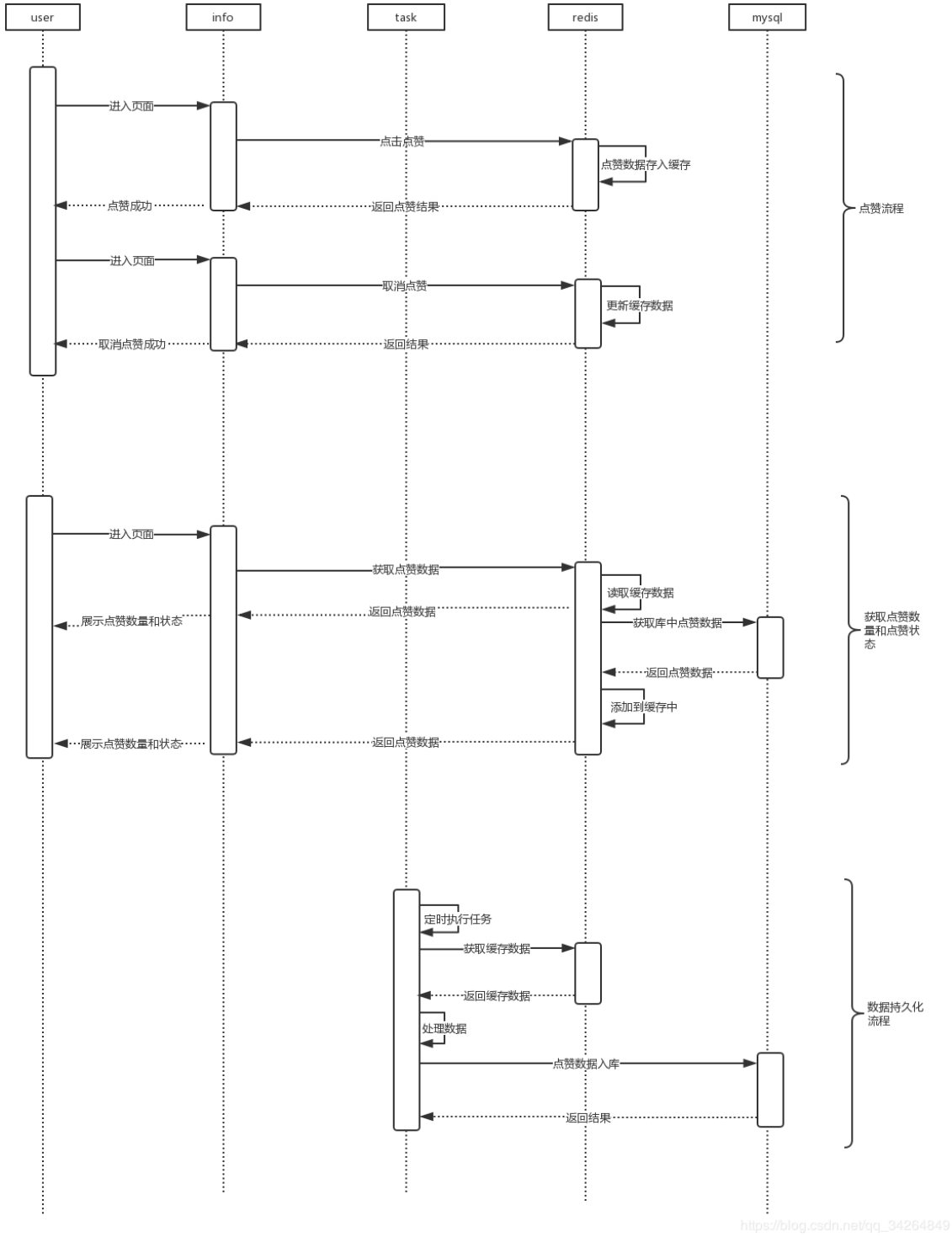
redis:

1.设计缓存数据格式
选择hash数据结构来实现，键值对格式如下：
用户的点赞状态key-value------>{“被点赞的id::用户id” : “点赞状态::点赞时间::点赞类型”}
被点赞id的点赞数量key-value------>{“被点赞id” : “点赞数量”}

2.大key拆分
点赞的数据量比较大的情况下，上面的设计会造成单个key存储的value很大，由于redis是单线程运行，如果一次操作的value很大，会对整个redis的响应时间有影响，所以我们这里在将上面的两个key做拆分。固定key的数量，每次存取时都先在本地计算出落在了哪个key上，这个操作就类似于redis分区、分片。有利于降低单次操作的压力，将压力平分到多个key上。

mysql：需要有一个点赞关系表　主键id ，　点赞人，被点赞人，点赞状态

　　点赞数量表：　被点赞id，　点赞数量。



https://blog.csdn.net/qq_34264849/article/details/84401198

微博

抢票系统

火车票抢票，只有一台服务器，瞬时访问量很大，如何系统的解决？

前端页面限制，url屏蔽，缓存 ，消息队列，负载均衡。

数据库设计

一个数据库数据很多怎么办
10个mysql节点加入一个新节点怎么办，怎么动态调整？
分页查询怎么写？limit 0 offset 1

设计一个多个mysql节点的分页查询方案？怎么排序？

第一种:也是最简单的一种:通过额外的添加一张关联表,属性中必有id属性,至于是否有库id属性和表id属性(既第几个库和第几个表)可有可无,因为这个可以根据id自行取模获取,注意这张表存放的数据是所有数据,但是胜在属性列少,只有提供索引的几个属性列,这样的话我们只需要select * from brand_temp where ... limit 400,10(插叙第41页的数据,每页显示5条数据),然后我们获取了id之后就可以去对应的表中查询了

第二种: 如果我们要查询第2页的数据的时候,需要查询所有库,sql语句为select * from db_x limit 0,10+10 ， 然后再在内存中合并所有表返回的记录然后进行解析,最后取第10开始的记录 ...可以看出这个方案一旦页码数达到n页,而每页显示的记录数为m条记录的时候,每个表需要查询的记录数为:(n-1)*m+m=nm条记录,内存中需要解析的记录数为 t * n * m 条记录,cpu不爆炸算我输

第三种:采取的是基于业务的模式:迫使用户无法进行跳页查询,每次只能点击下一页或者上一页的方式浏览。在查询得到记录数的同时记录下当前唯一id值的最大值,然后再次查询的时候添加where 条件。从而每个库都只返回一页的结果，避免内存开销过大

第四种:传说中的最好的方式,支持跳页查询,这个方式核心在于2次sql查询,具体怎么做呢:
前提条件假设:查询第1001页的数据,每页显示10条记录
1):我们先记录下要查询的记录数的范围:(1001-1)*10=10000 开始,10010结束->10000-10010
单体的sql为:select * from db limit 10000,10;
我们总共有4个表,意味着:每个表的start应该为10000/4=2500,从而sql变成了:
select * from db_x limit 2500,10; //假设是平均分配的,因而我们可以均分,不均分也没关系,后续操作会补齐
我们会得到4个表中的记录:(因为我demo还没写,所以先手写了)
T1:(1,"a"),.....
T2:(2,"b"),.....
T3:(3,"c"),.....
T4:(4,"d"),.....
真实数据第1001页不可能是1开头的,将就着看吧,过几天会一起讲rabbitMQ分布式一致性和这个demo一起发布的
ok,第一阶段的sql查询结束

2):对4个表中返回的记录进行id匹配(id如果非整型,自行用hashCode匹配),因为是升序查询,所以我们只需要比较下每个表的首条记录的id值即可,获得了最小的minId=1,和各个表最大的那个值maxId;ok,转换sql思路,这里我们采用条件查询了(弥补操作第一步):
select * from db_x where id between minId and maxId 这样我们就获取到了遗漏的数据(当然有多余的数据)
这样我们4个表中就返回了可能记录数各不相同的记录,第二步结束

3):
之后记录minId出现的位置,如T1出现的位置为2500,T2出现的位置为2500-2=2048 ,T3出现的位置为2500-3=2047 ,T4出现的位置为2500-3=2047 则最终出现的记录数为:2500+2048+2047+2047=10000-2-3-3=9992,因此我们需要的查询的记录数需要从9992 依次往后取8个开始,然后再取10个就是所求的数据,这种方式能做到数据精确查询,但是唯一的缺点就是每次查询都需要进行二次sql查询
12345678910111213141516171 8
总结:
第一种通过关联表的方式,是大部分所采用的,当然缺点也有那个关联表会变得无比巨大,但是这种方案很好解决了数据的查询问题,第二种方式是效率最低的方案, 适用于小型项目,不过既然是小型项目的话也没必要进行切库切表了,所以第二种方式属于知道即可,第三种方式不支持跳页,但是相比而言是更简洁的一种方式,最后一种方式的唯一缺陷在于需要查询两次sql,最终项目的选择本人还是推荐第一种>第四总>第三种的.

https://blog.csdn.net/Coder_Joker/article/details/82696641

mysql读写分离一致性问题

半同步复制

分布式ID生成

1 UUID （Universally Unique Identifier），通用唯一识别码的缩写。

缺点：不易于存储：UUID太长，16字节128位，通常以36长度的字符串表示，很多场景不适用。
信息不安全：基于MAC地址生成UUID的算法可能会造成MAC地址泄露，暴露使用者的位置。
对MySQL索引不利：如果作为数据库主键，在InnoDB引擎下，UUID的无序性可能会引起数据位置频繁变动，严重影响性能。

2 数据库自增

将分布式系统中数据库的同一个业务表的自增ID设计成不一样的起始值，然后设置固定的步长，步长的值即为分库的数量或分表的数量。

3 redis INCR原子命令

4 雪花算法

以划分命名空间的方式将 64-bit位分割成多个部分，每个部分代表不同的含义。而 Java中64bit的整数是Long类型，所以在 Java 中 SnowFlake 算法生成的 ID 就是 long 来存储的。

第1位占用1bit，其值始终是0，可看做是符号位不使用。
第2位开始的41位是时间戳，41-bit位可表示2^41个数，每个数代表毫秒，那么雪花算法可用的时间年限是(1L<<41)/(1000L360024*365)=69 年的时间。
43- 52位可表示机器数，即2^10 = 1024台机器，但是一般情况下我们不会部署这么台机器。如果我们对IDC（互联网数据中心）有需求，还可以将 10-bit 分 5-bit 给 IDC，分5-bit给工作机器。这样就可以表示32个IDC，每个IDC下可以有32台机器，具体的划分可以根据自身需求定义。
最后12-bit位是自增序列，可表示2^12 = 4096个数。
这样的划分之后相当于在一毫秒一个数据中心的一台机器上可产生4096个有序的不重复的ID。但是我们IDC 和机器数肯定不止一个，所以毫秒内能生成的有序ID数是翻倍的。

秒杀系统

页面静态化放哪，过期时间怎么设计
登录：分布式session，token被盗用了，如何设计防盗?
秒杀流程讲一下：内存标记-》redis-》rabbitmq-》mysql
如何异步下单?
减库存失败怎么回滚?用TCC补偿事务try：减库存 confirm：下订单 cancel：mysql减库存失败，

redis+库存
减库存和下订单是两个微服务，如何保证分布式事务执行成功，怎么实现了解过吗？同步事务变为异步，A事务执行成功通知B事务执行，B事务执行成功则发送ACK，A事务超时未收到ACK就执行回滚。
如果回滚了两次怎么办？用版本号
很多请求要回滚怎么办？
5000个线程在内存中占多大？

负载均衡算法设计

负载均衡有那几种算法？设计一个加权轮询的负载均衡方法。

限流系统设计

普通的计数器算法存在一个问题，用户通过在时间窗口的重置节点处突发请求， 可以瞬间超过我们的速率限制。

有可能通过算法的这个漏洞，瞬间压垮我们的应用。

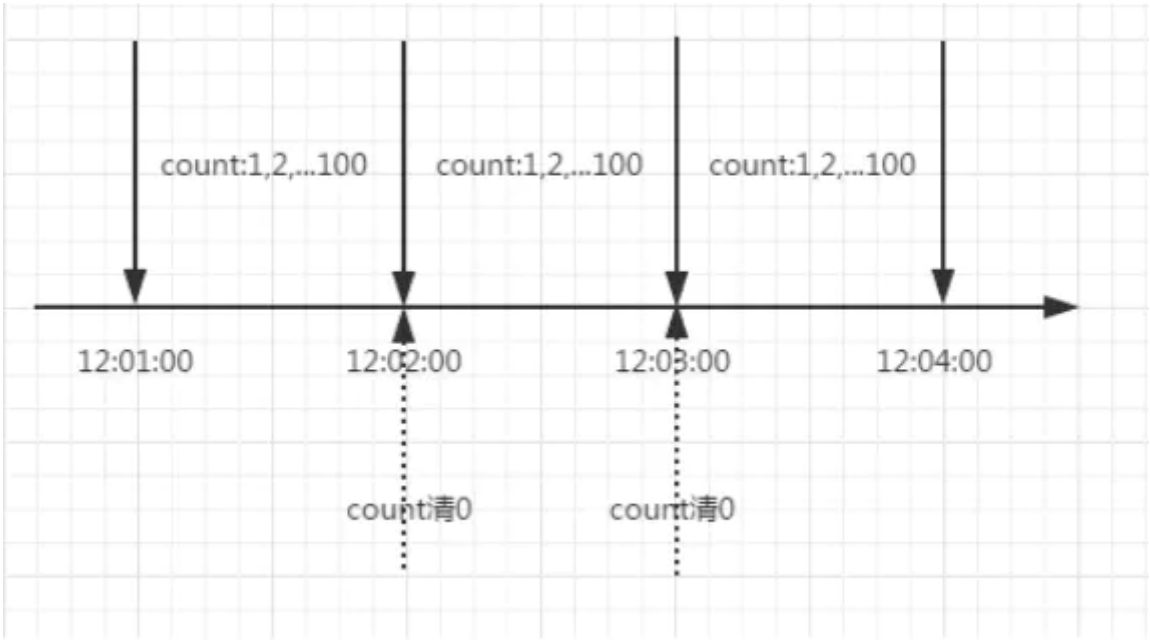
1秒内限制发送5个包

限流的常用方式

限流的常用处理手段有：计数器、滑动窗口、漏桶、令牌。

计数器

计数器是一种比较简单的限流算法，用途比较广泛，在接口层面，很多地方使用这种方式限流。在一段时间内，进行计数，与阀值进行比较，到了时间临界点，将计数器清0。



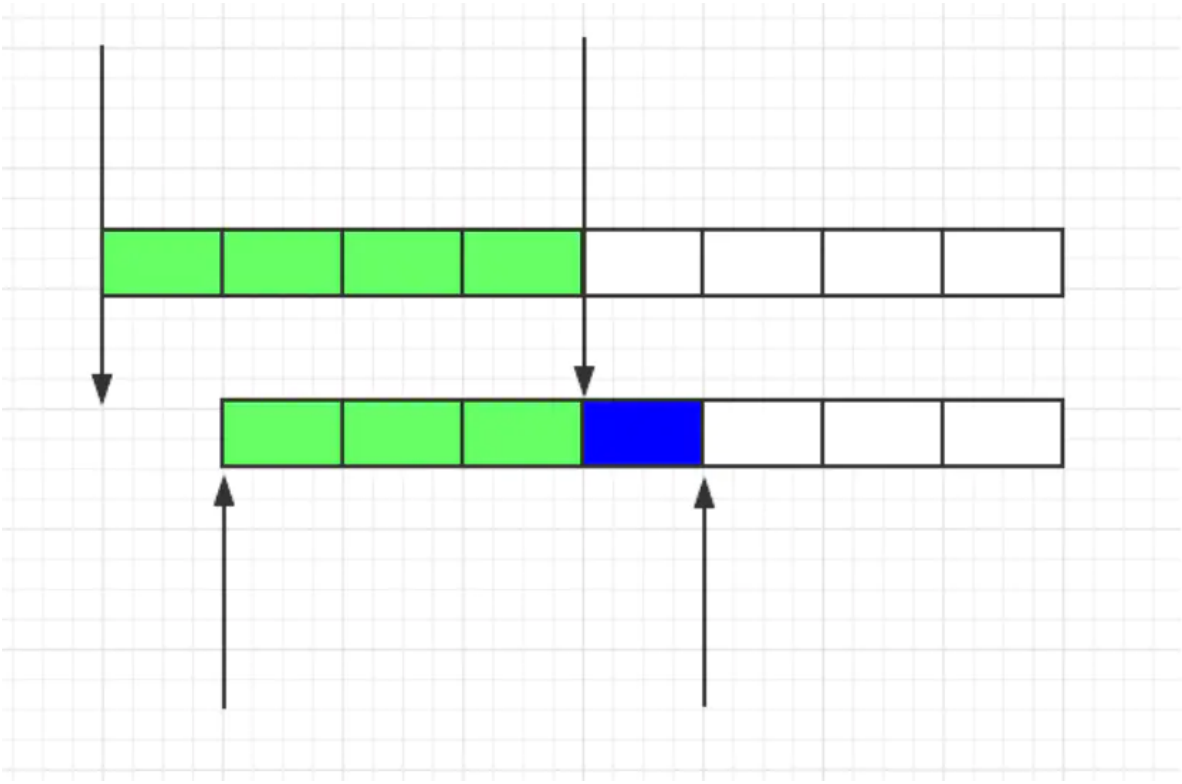
代码实例

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;
public class EnjoyCountLimit {
    private int limtCount = 60;// 限制最大访问的容量
    AtomicInteger atomicInteger = new AtomicInteger(0); // 每秒钟 实际请求的容量
    private long start = System.currentTimeMillis();// 获取当前系统时间
    private int interval = 60*1000;// 间隔时间60秒
    public boolean acquire() {
        long newTime = System.currentTimeMillis();
        if (newTime > (start + interval)) {
            // 判断是否是一个周期
            start = newTime;
            atomicInteger.set(0); // 清理为0
            return true;
        }
        atomicInteger.incrementAndGet();// i++;
        return atomicInteger.get() <= limtCount;
    }
    static EnjoyCountLimit limitService = new EnjoyCountLimit();
    public static void main(String[] args) {
        ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();
        for (int i = 1; i < 100; i++) {
            final int tempI = i;
            newCachedThreadPool.execute(new Runnable() {
                public void run() {
                    if (limitService.acquire()) {
                        System.out.println("你没有被限流,可以正常访问逻辑 i:" + tempI);
                    } else {
                            System.out.println("你已经被限流呢  i:" + tempI);
                        }
                    }
                });
            }
    }
}
```

这里需要注意的是，存在一个时间临界点的问题。举个栗子，在12:01:00到12:01:58这段时间内没有用户请求，然后在12:01:59这一瞬时发出100个请求，OK，然后在12:02:00这一瞬时又发出了100个请求。这里你应该能感受到，在这个临界点可能会承受恶意用户的大量请求，甚至超出系统预期的承受。

滑动窗口

由于计数器存在临界点缺陷，后来出现了滑动窗口算法来解决。

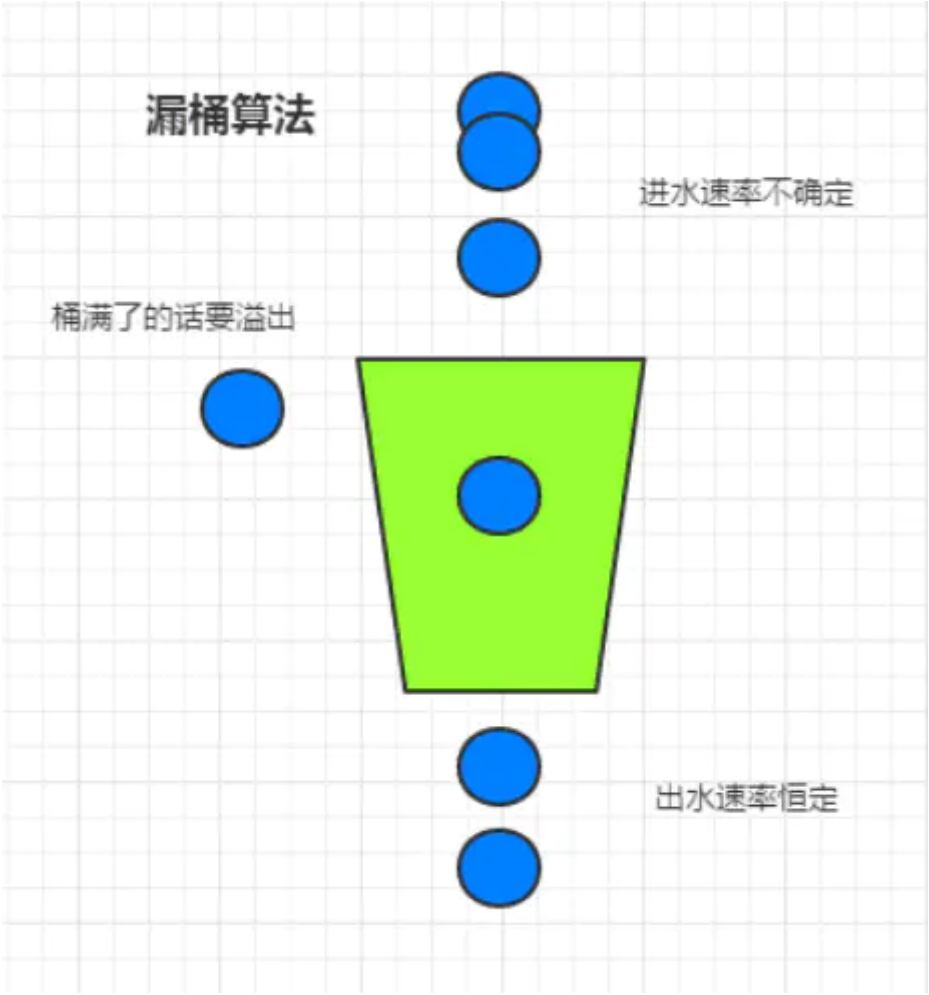


滑动窗口的意思是说把固定时间片，进行划分，并且随着时间的流逝，进行移动，这样就巧妙的避开了计数器的临界点问题。也就是说这些固定数量的可以移动的格子，将会进行计数判断阀值，因此格子的数量影响着滑动窗口算法的精度。

漏桶

虽然滑动窗口有效避免了时间临界点的问题，但是依然有时间片的概念，而漏桶算法在这方面比滑动窗口而言，更加先进。

有一个固定的桶，进水的速率是不确定的，但是出水的速率是恒定的，当水满的时候是会溢出的。



代码实现

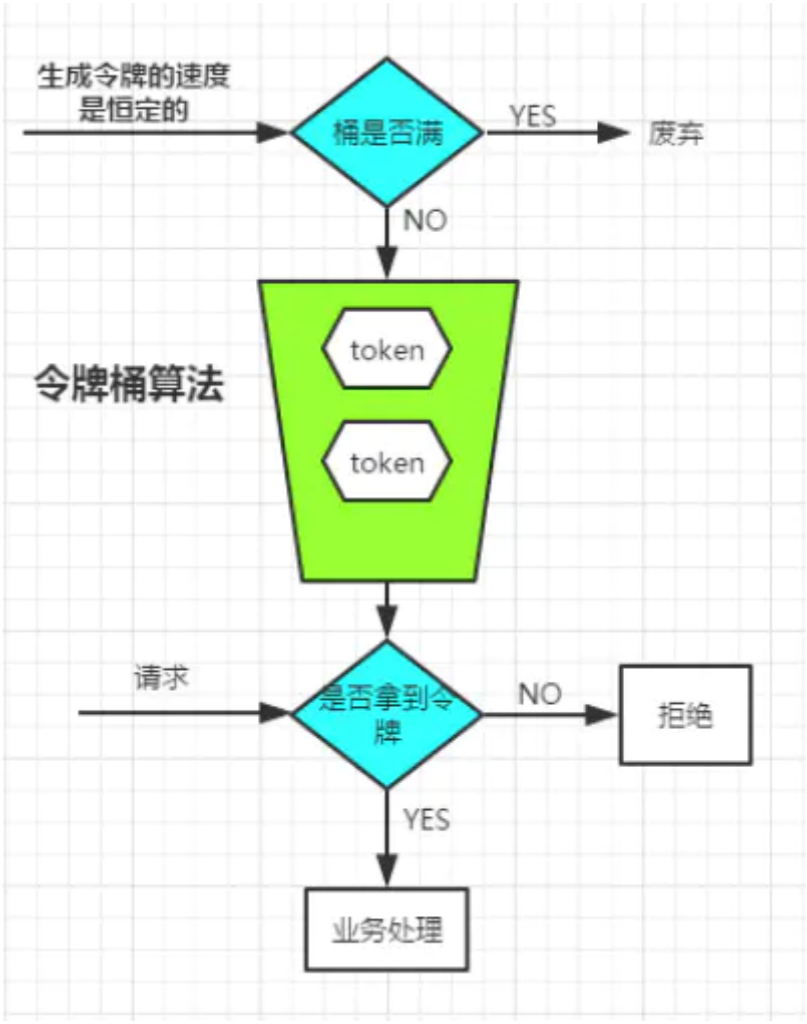
```java
/**
 * Created by zhangfz on 2017/8/9.
 */
public class LeakyBucketDemo {
    //时间刻度
    private static long time = System.currentTimeMillis();
    //桶里面现在的水
    private static int water = 0;
    //桶的大小
    private static int size = 10;
    //出水速率
    private static int rate = 3;
    public static boolean grant(){
        //计算出水的数量
        long now = System.currentTimeMillis();
        int out = (int) ((now - time ) / 700 * rate);
        //漏水后的剩余
        water = Math.max(0,water - out);
        time = now;
        if((water + 1) < size){
            ++water;
            return true;
        }else{
            return  false;
        }

    }
    public static void main(String[] args) {

        for(int i=0 ; i < 500 ; i++){
            new Thread(new Runnable() {
                @Override
                public void run() {
                    if(grant()){
                        System.out.println("执行业务逻辑");
                    }else{
                        System.out.println("限流");
                    }
                }
            }).start();
        }
```

令牌桶

漏桶把流量看作水，出水速度是恒定的，那么意味着如果瞬时大流量的话，将有大部分请求被丢弃掉（也就是所谓的溢出）。为了解决这个问题，令牌桶进行了算法改进。



生成令牌的速度是恒定的，而请求去拿令牌是没有速度限制的。这意味，面对瞬时大流量，该算法可以在短时间内请求拿到大量令牌，而且拿令牌的过程并不是消耗很大的事情。（有一点生产令牌，消费令牌的意味）

不论是对于令牌桶拿不到令牌被拒绝，还是漏桶的水满了溢出，都是为了保证大部分流量的正常使用，而牺牲掉了少部分流量，这是合理的，如果因为极少部分流量需要保证的话，那么就可能导致系统达到极限而挂掉，得不偿失。

代码实现：

```java
 * Created by zhangfz on 2017/8/9.
 */
public class TokenBucketDemo {

    private static long time = System.currentTimeMillis();
    private static int createTokenRate = 3;
    private static int size = 10;
    //当前令牌数
    private static int tokens = 0;

    public static boolean grant(){

        long now = System.currentTimeMillis();
        //在这段时间内需要产生的令牌数量
        int in = (int) ((now - time) / 50 * createTokenRate);
        tokens = Math.min(size,tokens + in);
        time = now;
        if(tokens > 0){
            --tokens;
            return true;
        }else{
            return false;
        }

    }

    public static void main(String[] args) {

        for(int i=0 ; i < 500 ; i++){
            new Thread(new Runnable() {
                @Override
                public void run() {
                    if(grant()){
                        System.out.println("执行业务逻辑");
                    }else{
                        System.out.println("限流");
                    }
                }
            }).start();
        }
    }
```

四、限流神器：Guava RateLimiter

Guava不仅仅在集合、缓存、异步回调等方面功能强大，而且还给我们封装好了限流的API！

Guava RateLimiter基于令牌桶算法，我们只需要告诉RateLimiter系统限制的QPS是多少，那么RateLimiter将以这个速度往桶里面放入令牌，然后请求的时候，通过tryAcquire()方法向RateLimiter获取许可（令牌）。

代码示例

```java
/**
 * Created by zhangfz on 2017/8/10.
 */
public class LimitDemo {

    public static ConcurrentHashMap<String,RateLimiter> resourceRateLimiter =
    new ConcurrentHashMap<String, RateLimiter>();

    static{
        createResourceLimiter("order",50);
    }
    public static void createResourceLimiter(String resource,double qps){

        if(resourceRateLimiter.contains(resource)){
            resourceRateLimiter.get(resource).setRate(qps);
        }else{
            RateLimiter rateLimiter = RateLimiter.create(qps);
            resourceRateLimiter.putIfAbsent(resource,rateLimiter);
        }

    }

    public static void main(String[] args) {

        for(int i = 0; i < 5000; i++){
            new Thread(new Runnable() {
                @Override
                public void run() {
                    if(resourceRateLimiter.get("order").tryAcquire(10, TimeUnit.MILLISECONDS)){
                        System.out.println("执行业务逻辑");
                    }else{
                        System.out.println("限流");
                    }
                }
            }).start();
        }


    }
```

五、分布式场景下的限流

上面所说的限流的一些方式，都是针对单机而言的，其实大部分的场景，单机的限流已经足够了。分布式下限流的手段常常需要多种技术相结合，比如Nginx+Lua，Redis+Lua等去做。本文主要讨论的是单机的限流，这里就不在详细介绍分布式场景下的限流了。

一句话，让系统的流量，先到队列中排队、限流，不要让流量直接打到系统上。

滑动QPS算法，用远端缓存redis来做集群的调用计数，redis key为时间窗口标识，远端value为对应时间窗口剩余份额，用decr操作对份额进行扣减。key可以设置为用户ID+时间ID。

若QPS过大，加入本地缓存，每次从redis中获取一个小份额。

海量数据处理

大数组求和

拿出一个大数组，每个元素都是整数，可以放在内存，如何快速求和。
我说FORK JOIN框架，面试官让自己用多线程处理
怎么用多线程处理？Callable线程，线程池submit，通过future获取计算结果
如何拆分任务呢，每个线程计算范围如何设计。说了一堆面试官都不满意，最后提示用闭包。不会

Bloom filter/Bitmap

- Bloom filter

适用范围：可以用来实现数据字典，进行数据的判重，或者集合求交集。

基本原理：元素加入集合时，通过K个Hash函数将这个元素映射成一个位阵列（Bit array）中的K个点，把它们置为1。查询时，如果这些点有任何一个0，则被检索元素一定不在；如果都是1，则被检索元素很可能在。

Bloom Filter的这种高效是有一定代价的：**有可能会把不属于这个集合的元素误认为属于这个集合**（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

- Bitmap

  Bitmap就是用一个bit位来标记某个元素对应的Value， 而Key即是该元素。由于采用了Bit为单位来表示某个元素是否存在，因此在存储空间方面，可以大大节省。

  Bitmap排序方法
  第一步，将所有的位都置为0，从而将集合初始化为空。
  第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为1。
  第三步，检验每一位，如果该位为1，就输出对应的整数。

Bloom filter可以看做是对Bitmap的扩展。

数据量太大，无法在较短时间内迅速解决，无法一次性装入内存。

处理海量数据，不外乎

1. 分而治之/hash映射 + hash统计 + 堆/快速/归并排序
2. 双层桶划分
3. Bloom filter/Bitmap；
4. Trie树/数据库/倒排索引；
5. 外排序；
6. 分布式处理之Hadoop/Mapreduce。

一亿数据处理

一亿个数据求Top 10，先对1000取模将数据分到1000个小文件中去，保证每种只出现在一个文件中。再对每个小文件中的数据进行HashMap计数统计并进行堆排序，最后堆排序依次处理每个小文件的Top10以得到最后的结果

通用思路：

两种情况：可一次读入内存，不可一次读入

解法：

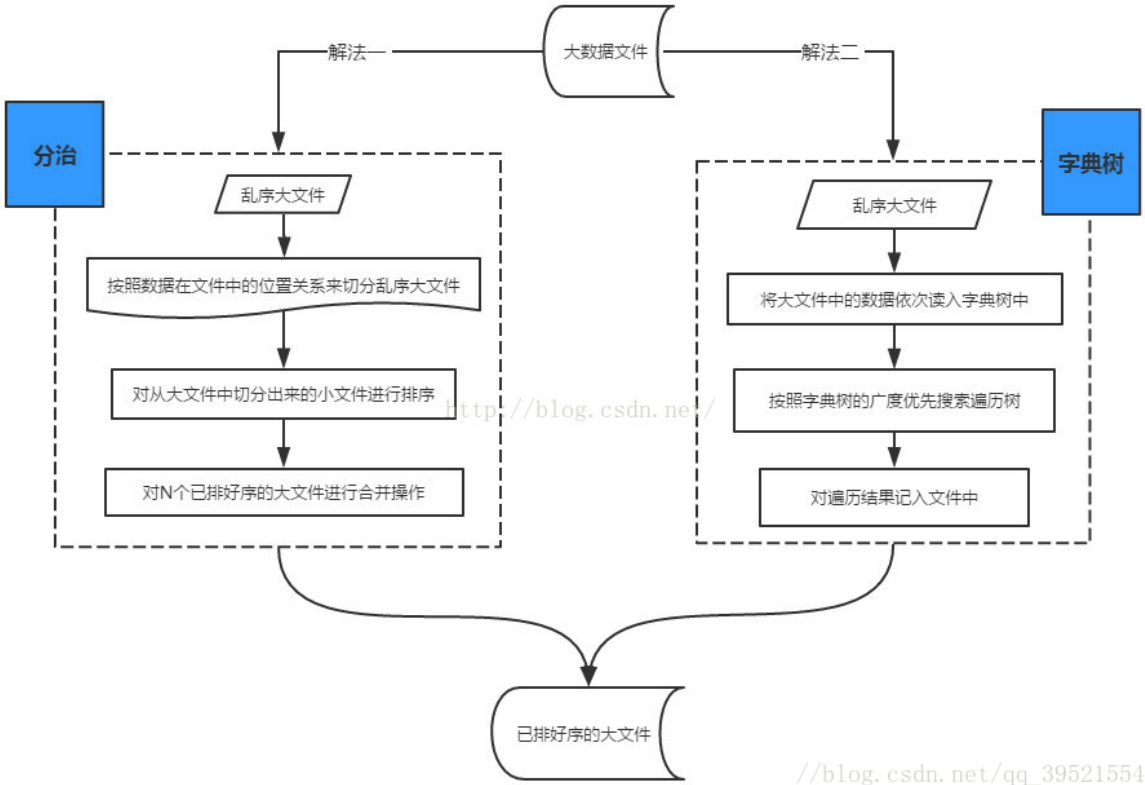
一、区间快速排序（当某个区间的长度=N则输出排序区间）

二、堆排序（维护N个结点的堆结构）

三、哈希映射（用hash将大文件映射为小文件，依次进内存排序后输出）

四、trie树

五、位图 （Bit Map）



频率最高的k个数/最大的k个数

频率最高的10个数需要先分治。先对1000取模将数据分到1000个小文件中去，保证每种只出现在一个文件中。再对每个小文件中的数据进行HashMap计数统计并进行堆排序，最后堆排序依次处理每个小文件的Top10以得到最后的结果。

最大的10个数可以直接堆排序。

堆排序的具体实现：最大的k个数，需要维护一个大小为k的小顶堆。前k个元素建堆。对于后面的每个元素，假如小于堆顶，就不做操作。如果大于堆顶，就和堆顶交换，然后做siftdown操作,sifdown函数len参数始终保持为k。

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> map = new HashMap<>();
        for(int i = 0; i < nums.length; i++){
            map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);
        }
        int[][] arr = new int[map.size()][2];
        int[] res = new int[k];
        int index = 0;
        for(Map.Entry entry : map.entrySet()){
            arr[index][0] = (int)entry.getKey();
            arr[index][1] = (int)entry.getValue();
            index++;
        }
        int t = k >> 1;
        for(;t >= 0; t--){
            siftdown(arr, k, t);
        }
        for(int i = k; i < arr.length; i++){
            if(arr[i][1] > arr[0][1]) {
                if(i != k){
```

```
                    swap(arr, i, k);
                }
                swap(arr, 0, k);
                siftdown(arr, k, 0);
            }
        }
        for(int i = 0; i < k; i++){
            res[i] = arr[i][0];
        }
        return res;
    }
    public void siftdown(int[][] arr, int len, int t){
        int v = arr[t][1];
        int key = arr[t][0];
        while(2 * t + 1 < len){
            int temp = 2 * t + 1;
            if(2 * t + 2 < len && arr[2 * t + 2][1] < arr[2 * t + 1][1]){
                temp++;
            }
            if(v > arr[temp][1]){
                arr[t][1] = arr[temp][1];
                arr[t][0] = arr[temp][0];
            }else{
                break;
            }
            t = temp;
        }
        arr[t][1] = v;
        arr[t][0] = key;
    }
    public void siftup(int[][] arr, int len, int t){
        int v = arr[t][1];
        int key = arr[t][0];
        while(t > 0){
            int temp = (t - 1) >> 1;
            if(arr[temp][1] <= arr[t][1]){
                break;
            }
            arr[t][1] = arr[temp][1];
            arr[t][0] = arr[temp][0];
            t = temp;
        }
        arr[t][1] = v;
        arr[t][0] = key;
    }
    public void swap(int[][] arr, int i, int j){
        int temp0 = arr[i][0];
        int temp1 = arr[i][1];
        arr[i][0] = arr[j][0];
        arr[i][1] = arr[j][1];
        arr[j][0] = temp0;
        arr[j][1] = temp1;
    }
}
```

一亿数据查找是否存在

建字典树

一亿数据排序

根据内存大小计算出单个文件大小，假设是整形数字，每个数字4B，每G内存可以存放2.5亿数据，

100亿数据找中位数

假设是整形数字，每个数字4B，512M内存可以存放1.25亿个数。

将每个数按照首位是0还是1划分到两个文件，第一个文件假设有40亿数据，第二个文件有60亿数据，排除掉第一个文件。此时要在第二个文件找到第10亿个数。

在第二个文件再按照第二位0还是1划分到两个文件，以此类推。

如何设计一个支持高并发的高可用服务?

负载均衡

缓存

消息队列

限流