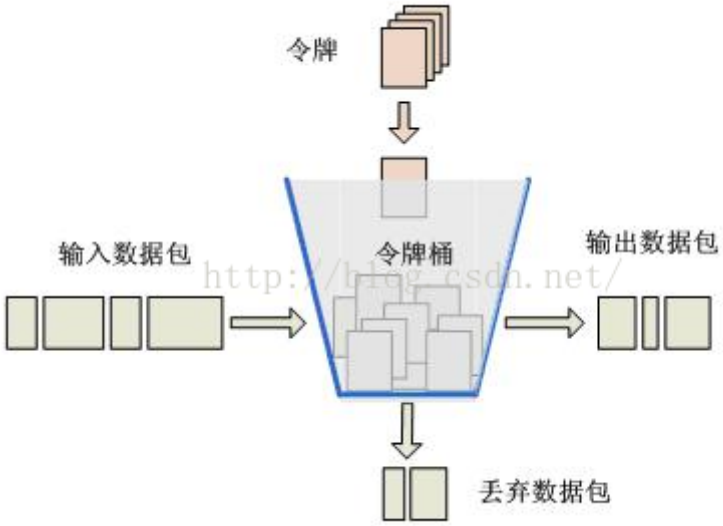
常见算法/协议

1.令牌桶限流算法

> 令牌桶算法最初来源于计算机网络。在网络传输数据时，为了防止网络拥塞，需限制流出网络的流量，使流量以比较均匀的速度向外发送。令牌桶算法就实现了这个功能，可控制发送到网络上数据的数目，并允许突发数据的发送。
>
> 大小固定的令牌桶可自行以恒定的速率源源不断地产生令牌。如果令牌不被消耗，或者被消耗的速度小于产生的速度，令牌就会不断地增多，直到把桶填满。后面再产生的令牌就会从桶中溢出。最后桶中可以保存的最大令牌数永远不会超过桶的大小。
>
> 
>

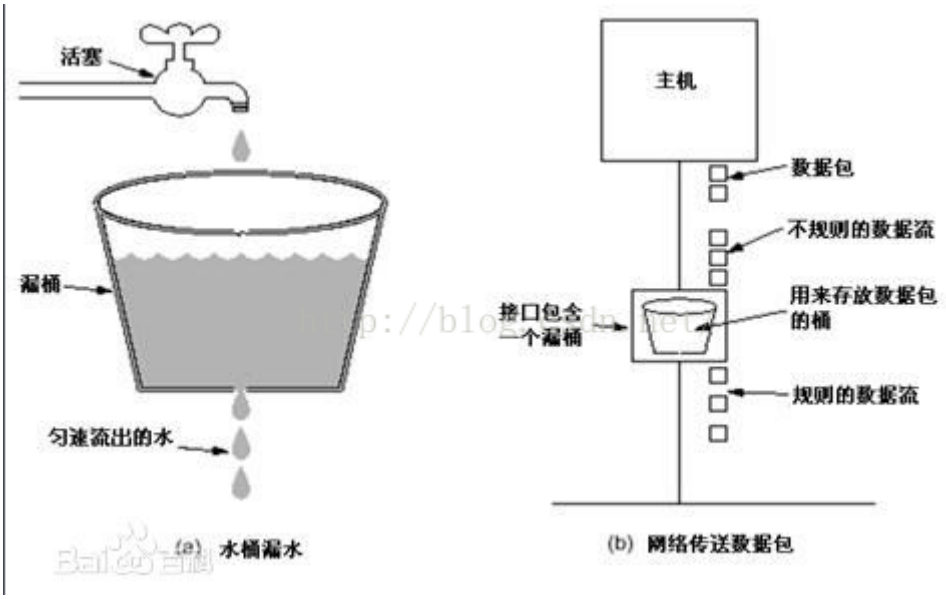
令牌桶限流算法可以限制流量，使得流量尽可能均匀发送。而且只要令牌桶存在令牌，就允许某一时刻突发的流量。

Guava的RateLimiter实现了令牌桶算法

> ```
> //速率是每秒两个许可
> final RateLimiter rateLimiter = RateLimiter.create(2.0);
> 
> void submitTasks(List tasks, Executor executor) {
>     for (Runnable task : tasks) {
>         rateLimiter.acquire(); // 也许需要等待
>         executor.execute(task);
>     }
> }
> ```
>
> RateLimiter 并不提供公平性的保证。

2.漏桶算法限流

> 漏桶算法（Leaky Bucket）：主要目的是控制数据注入到网络的速率，平滑网络上的突发流量。漏桶算法提供了一种机制，通过它，突发流量可以被整形以便为网络提供一个稳定的流量。漏桶算法的示意图如下：



> 请求先进入到漏桶里，漏桶以一定的速度出水，当水请求过大会直接溢出，可以看出漏桶算法能强行限制数据的传输速率。

漏桶算法与令牌桶算法的区别在于漏桶算法能够强行限制数据的传输速率，而令牌桶算法在能够限制数据的平均传输速率外，还允许某种程度的突发传输。在“令牌桶算法”中，只要令牌桶中存在令牌，那么就允许突发地传输数据直到达到用户配置的门限，所以它适合于具有突发特性的流量。

> 其他的限流算法还有**计数器（固定窗口）算法** 和 **滑动窗口算法**
>
> 常见四种限流算法： https://blog.csdn.net/weixin_41846320/article/details/95941361

3.分布式一致性协议 (https://zhuanlan.zhihu.com/p/130974371)

> 最终目的：维护集群中各个节点的数据一致性

按照是单主还是多主可以分为

单主协议：2PC, Paxos, Raft

多主：Pow，Gossip

> 单主协议由一个主节点发出数据，传输给其余从节点，能保证数据传输的有序性。而多主协议则是从多个主节点出发传输数据，传输顺序具有随机性，因而数据的有序性无法得到保证，只保证最终数据的一致性。

Gossip算法

> Gossip又被称为流行病算法，它与流行病毒在人群中传播的性质类似，由初始的几个节点向周围互相传播，到后期的大规模互相传播，最终达到一致性。
>
> 初始由几个节点发起消息，这几个节点会将消息的更新内容告诉自己周围的节点，收到消息的节点再将这些信息告诉周围的节点。依照这种方式，获得消息的节点会越来越多，总体消息的传输规模会越来越大，消息的传偶速度也越来越快。虽然不是每个节点都能在相同的时间达成一致，但是最终集群中所有节点的信息必然是一致的。
>
> ***Gossip 协议确保的是分布式集群的最终一致性。***
>
> 著名的Redis数据库便是使用Gossip传播协议保持一致性
>
> 大名鼎鼎的 Bitcoin 则是使用了 Gossip 协议来传播交易和区块信息
>
> 优势：

> 易扩展，新增的节点的状态最终会跟其他节点达成一致；
>
> 容错性高，任何节点的宕机和重启都不会影响 Gossip 消息的传播，Gossip 协议具有天然的分布式系统容错特性；
>
> 去中心化：Gossip 协议不要求任何中心节点，所有节点都可以是对等的；
>
> 一致性收敛：Gossip 协议中的消息会以一传十、十传百一样的指数级速度在网络中快速传播，因此系统状态的不一致可以在很快的时间内收敛到一致
>
> 缺陷：
>
> 延迟，消息最终是通过多个轮次的散播而到达全网的，因此使用 Gossip 协议会造成不可避免的消息延迟；
>
> 冗余，节点会定期随机选择周围节点发送消息，而收到消息的节点也会重复该步骤，因此就不可避免的存在消息重复发送给同一节点的情况，造成了消息的冗余

Proof-of-work（Pow）算法

> 工作量证明算法，大量的节点参与竞争，通过自身的工作量大小来证明自己的能力，最终能力最大的节点获得优胜，其他节点的信息需要与该节点统一。Pow最为人所熟知的应用是比特币。
>
> 比特币塑造的是一个去中心化的交易平台，最重要的一点就是该平台的可信度。要达到高可信度，要求整个系统中没有固定的leader，且为了防止外界篡改，必然要设定一些特殊的机制，比如让图谋不轨的一方无法篡改或者必须付出与收获完全不等称的代价才有修改的可能，以这样的方式打消其修改的念头。这时候比特币引入了Pow算法，在Pow机制下，所有参与者共同求解数学问题，这些数学问题往往需要经过大量枚举才能求解，因此需要参与者消耗大量的硬件算力。成功求解数学问题的参与者将获得记账权，并获得比特币作为奖励。其余所有参与者需要保持和获得记账权节点的区块一致，由此达到最终的一致性。

在比特币的应用中，使用**Pow算法确定竞争者的记账权**，尽可能地解决"拜占庭将军"问题，再将可信的结果由传播速度极强，**节点数目量大的Gossip协议去进行传输**，最终达成全网一致，可谓很好地利用这两大算法的特点，将二者优劣互补并巧妙地用于区块链领域，这一点令人非常赞叹！

Paxos 和 Raft(https://zhuanlan.zhihu.com/p/147691282?from_voters_page=true)

> https://www.cnblogs.com/zhang-qc/p/8688258.html
>
> Paxos算法

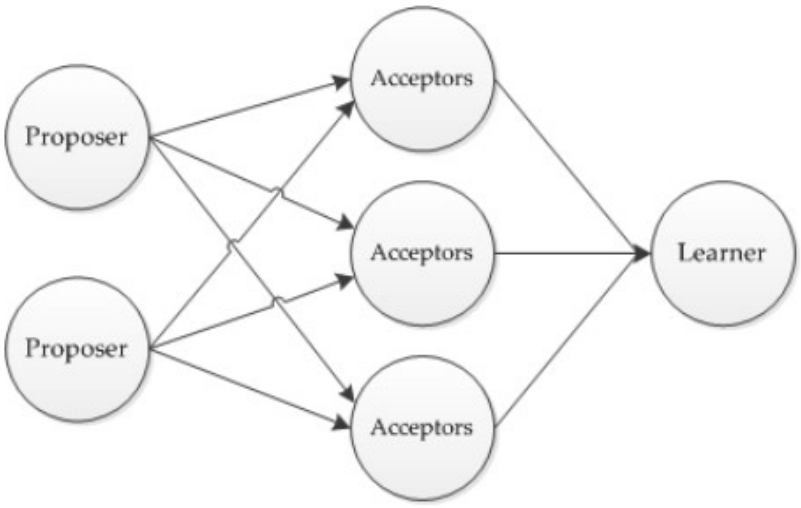


Figure 1: Basic Paxos architecture. A number of proposers make proposals to acceptors. When an acceptor accepts a value it sends the result to learner nodes.

主要有三类节点：

1. 提议者（Proposer）：提议一个值；
2. 接受者（Acceptor）：对每个提议进行投票；
3. 告知者（Learner）：被告知投票的结果，不参与投票过程。

规定一个提议包含两个字段：[n, v]，其中 n 为序号（具有唯一性），v 为提议值。

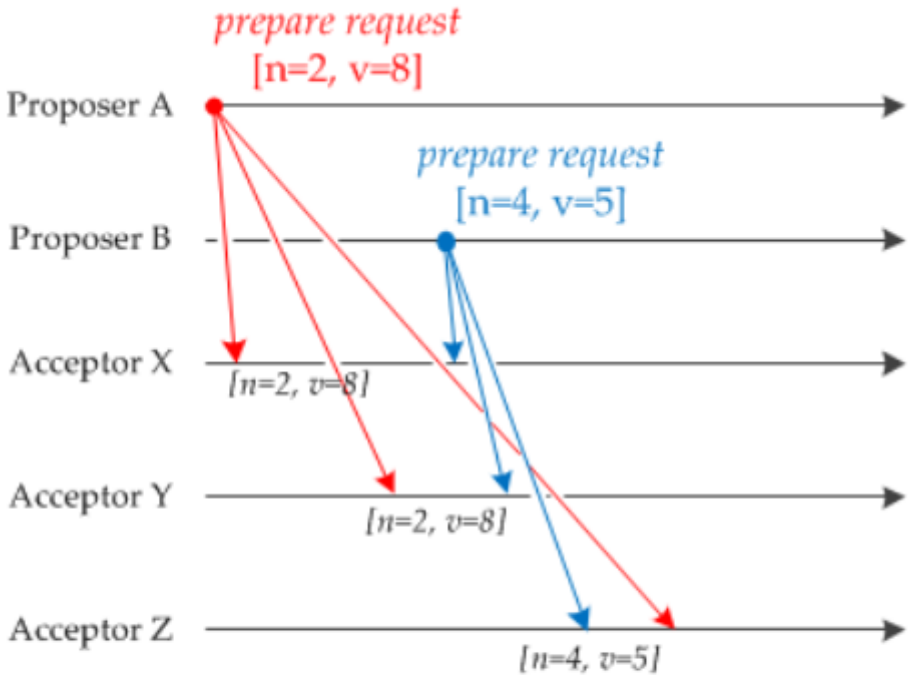


Figure 2: Paxos. Proposers A and B each send prepare requests to every acceptor. In this example proposer A's request reaches acceptors X and Y first, and proposer B's request reaches acceptor Z first.

当 Acceptor 接收到一个提议请求，包含的提议为 [n1, v1]，并且之前还未接收过提议请求，那么发送一个提议响应，设置当前接收到的提议为 [n1, v1]，并且保证以后不会再接受序号小于 n1 的提议。

如果 Acceptor 接受到一个提议请求，包含的提议为 [n2, v2]，并且之前已经接收过提议 [n1, v1]。如果 n1 > n2，那么就丢弃该提议请求；否则，发送提议响应，该提议响应包含之前已经接收过的提议 [n1, v1]，设置当前接收到的提议为 [n2, v2]，并且保证以后不会再接受序号小于 n2 的提议。

当一个 Proposer 接收到超过一半 Acceptor 的提议响应时，就可以发送接受请求。

Acceptor 接收到接受请求时，如果序号大于等于该 Acceptor 承诺的最小序号，那么就发送通知给所有的 Learner。当 Learner 发现有大多数的 Acceptor 接收了某个提议，那么该提议的提议值就被 Paxos 选择出来。

Raft算法 ETCD使用该协议实现节点之前的数据一致性

节点类型：Follower、Candidate 和 Leader

Leader 会周期性的发送心跳包给 Follower。每个 Follower 都设置了一个随机的竞选超时时间，一般为 150ms~300ms，如果在这个时间内没有收到 Leader 的心跳包，就会变成 Candidate，进入竞选阶段。

> 1) 分布式系统初始阶段：
>
> 此时只有 Follower，没有 Leader。Follower A 等待一个随机的竞选超时时间之后，没收到 Leader 发来的心跳包，因此进入竞选阶段。此时变为Candidate 节点
>
> 此时 A 发送投票请求给其它所有节点。
>
> 其它节点会对请求进行回复，如果超过一半的节点回复了，那么该 Candidate 就会变成 Leader。
>
> 之后 Leader 会周期性地发送心跳包给 Follower，Follower 接收到心跳包，会重新开始计时。
>
> 2 ）分布式系统Leader节点异常，多个Follower节点成为Candidate
>
> 多个Candidate 节点请求投票，投票规则：
>
> - 在任一任期内，单个节点最多只能投一票
> - 候选人知道的信息不能比自己的少（这一部分，后面介绍log replication和safety的时候会详细介绍）
> - first-come-first-served 先来先得

> https://www.jianshu.com/p/3fec1f8bfc5f ZooKeeper ZAB协议

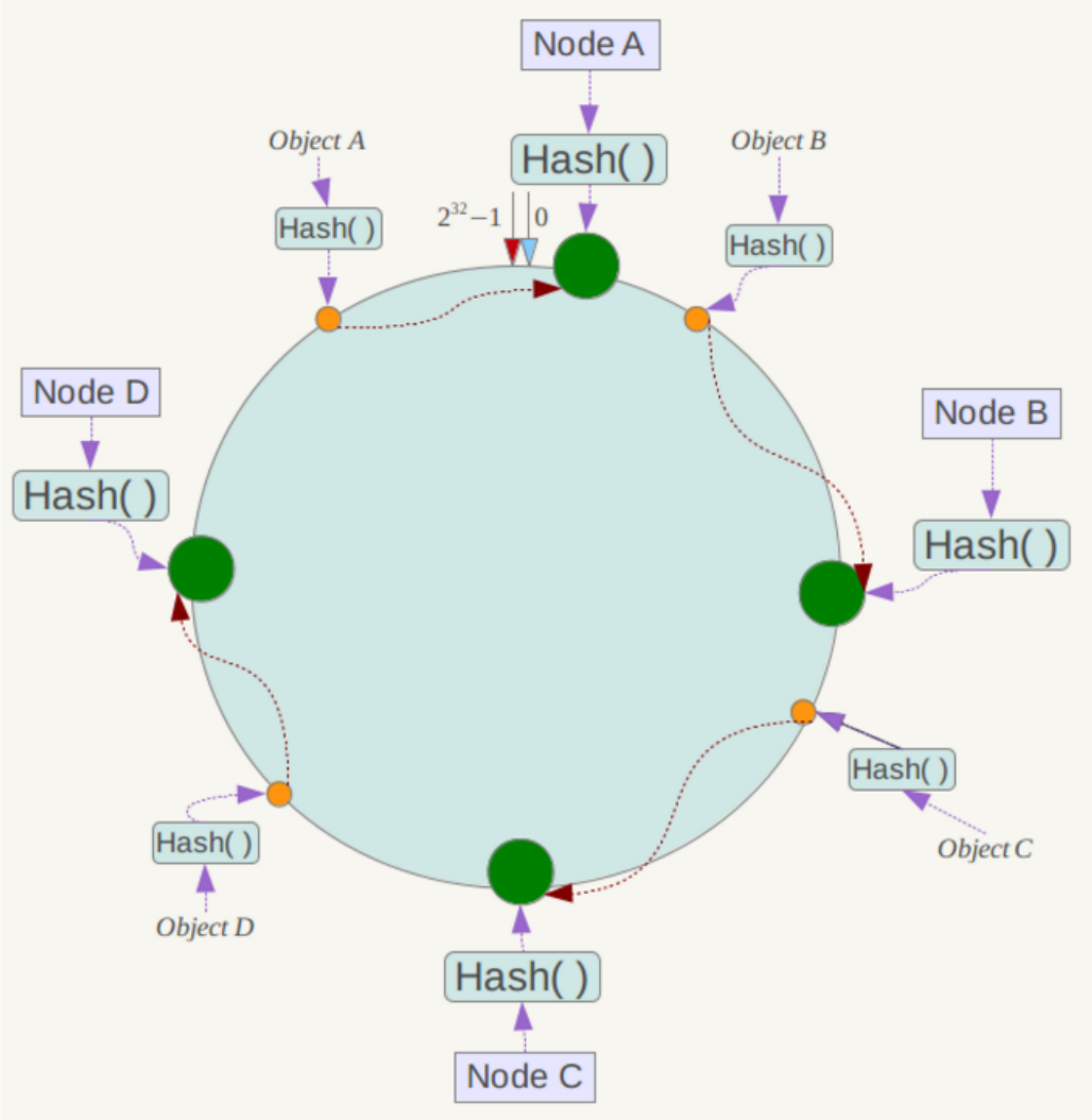
4.一致性Hash算法

> 一致性哈希算法最初的目的是为了解决因特网中的热点问题，现在也在分布式系统中广泛应用。

一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0-2^32-1，整个空间按顺时针方向组织，0和2^32-1在零点中方向重合。

下一步将各个服务器使用Hash进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置。

接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针"行走"，第一台遇到的服务器就是其应该定位到的服务器。



一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器。

如果环上的节点数目过少或者分布不均匀，可以造成数据倾斜的问题，这种情况可以引入虚拟节点，例如找到节点在圆环上的对称节点作为虚拟节点。

5.负载均衡算法(https://juejin.im/post/6844903793012768781)

随机

随机就是没有规律的，随便从负载中获得一台，又分为完全随机和加权随机：

轮询

轮询又分为三种，1.完全轮询 2.加权轮询 3.平滑加权轮询

哈希

负载均衡算法中的哈希算法，就是根据某个值生成一个哈希值，然后对应到某台服务器上去，当然可以根据用户，也可以根据请求参数，或者根据其他

最小压力

所以的最小压力负载均衡算法就是 选择一台当前最“悠闲”的服务器

6.Bitmap

> Bit-map的基本思想就是用一个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为单位来存储数据，因此在存储空间方面，可以大大节省。（PS：划重点 **节省存储空间**）
>
> 假设有这样一个需求：在20亿个随机整数中找出某个数m是否存在其中，并假设32位操作系统，4G内存
>
> 在Java中，int占4字节，1字节=8位（1 byte = 8 bit）
>
> 如果每个数字用int存储，那就是20亿个int，因而占用的空间约为(2000000000*4/1024/1024/1024)≈**7.45**G
>
> 如果按位存储就不一样了，20亿个数就是20亿位，占用空间约为(2000000000/8/1024/1024/1024)≈**0.233**G
>
> image-20201119195721277
>
> 
>
>
> 1个int占32位，那么我们只需要申请一个int数组长度为 int tmp[1+N/32] 即可存储，其中N表示要存储的这些数中的最大值，于是乎：
>
> tmp[0]：可以表示0~31
>
> tmp[1]：可以表示32~63
>
> tmp[2]：可以表示64~95
>
> 。。。
>
> 如此一来，给定任意整数M，那么**M/32就得到下标，M%32就知道它在此下标的哪个位置**
>
> 如何添加：

添加

这里有个问题，我们怎么把一个数放进去呢？例如，想把5这个数字放进去，怎么做呢？

首先，5/32=0，5%32=5，也是说它应该在tmp[0]的第5个位置，那我们把1向左移动5位，然后按位或

不影响其它位

即应该在数组中的下标

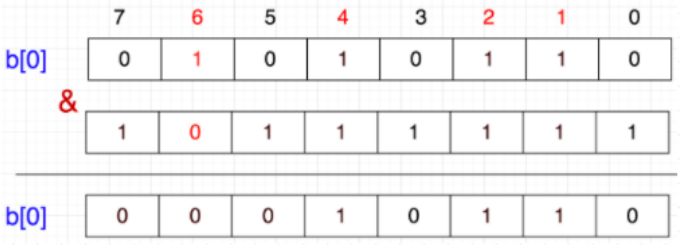
换成二进制就是

```
0101 0110
0010 0000
0111 0110
```

清除

以上是添加，那如果要清除该怎么做呢？

还是上面的例子，假设我们要6移除，该怎么做呢？



从图上看，只需将该数所在的位置为0即可

1左移6位，就到达6这个数字所代表的位，然后按位取反，最后与原数按位与，这样就把该位置为0了

```
b[0] = b[0] & (~(1<<6))
```

```
b[0] = b[0] & (~(1<<(i%8)))
```

查找

前面我们也说了，每一位代表一个数字，1表示有（或者说存在），0表示无（或者说不存在）。通过把该为置为1或者0来达到添加和清除的小伙，那么判断一个数存不存在就是判断该数所在的位是0还是1

假设，我们想知道3在不在，那么只需判断 `b[0] & (1<<3)` 如果这个值是0，则不存在，如果是1，就表示存在

2、Bitmap有什么用

大量数据的快速排序、查找、去重

快速排序

假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复），我们就可以采用Bit-map的方法来达到排序的目的。

要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0，然后将对应位置为1。

最后，遍历一遍Bit区域，将该位是一的位的编号输出（2，3，4，5，7），这样就达到了排序的目的，时间复杂度O(n)。

优点：

- 运算效率高，不需要进行比较和移位；
- 占用内存少，比如N=10000000；只需占用内存为N/8=1250000Byte=1.25M

缺点：

- 所有的数据不能重复。即不可对重复的数据进行排序和查找。
- 只有当数据比较密集时才有优势

快速去重

20亿个整数中找出不重复的整数的个数，内存不足以容纳这20亿个整数。

首先，根据"内存空间不足以容纳这05亿个整数"我们可以快速的联想到Bit-map。下边关键的问题就是怎么设计我们的Bit-map来表示这20亿个数字的状态了。其实这个问题很简单，一个数字的状态只有三种，分别为不存在，只有一个，有重复。因此，我们只需要2bits就可以对一个数字的状态进行存储了，假设我们设定一个数字不存在为00，存在一次01，存在两次及其以上为11。那我们大概需要存储空间2G左右。

接下来的任务就是把这20亿个数字放进去（存储），如果对应的状态位为00，则将其变为01，表示存在一次；如果对应的状态位为01，则将其变为11，表示已经有一个了，即出现多次；如果为11，则对应的状态位保持不变，仍表示出现多次。

最后，统计状态位为01的个数，就得到了不重复的数字个数，时间复杂度为O(n)。

快速查找

这就是我们前面所说的了，int数组中的一个元素是4字节占32位，那么除以32就知道元素的下标，对32求余数（%32）就知道它在哪一位，如果该位是1

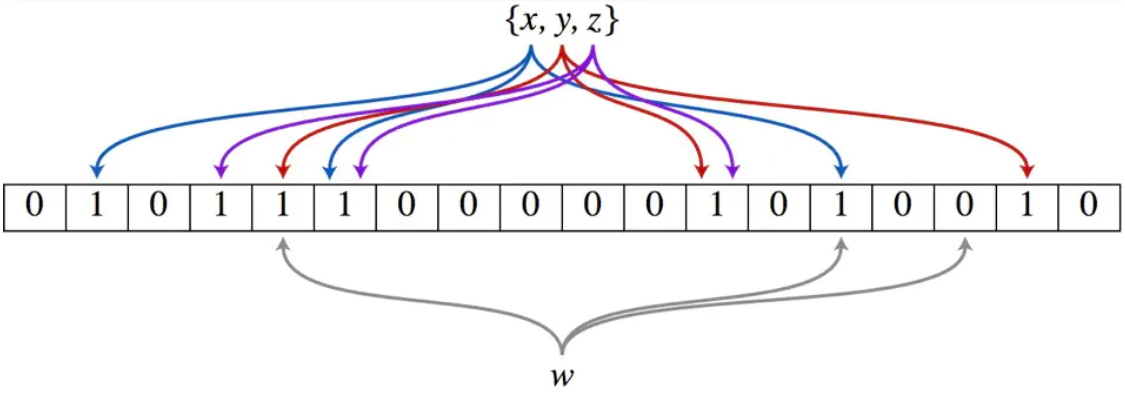
> 大量数据的排序，查找，去重

7. Bloom filter (布隆过滤器)

> Bloom filter 是一个数据结构，它可以用来判断某个元素是否在集合内，具有运行快速，内存占用小的特点。
>
> 而高效插入和查询的代价就是，Bloom Filter 是一个基于概率的数据结构：**它只能告诉我们一个元素绝对不在集合内或可能在集合内。**
>
> 布隆过滤器可以确认一个元素肯定不在集合内，但是无法确定元素肯定在集合内；
>
> 主要应用于大规模数据下不需要精确过滤的场景，如检查垃圾邮件地址，爬虫URL地址去重，解决缓存穿透问题等；
>
> Bloom filter 的基础数据结构是一个 比特向量（可理解为位数组）。
>
> 布隆过滤器的原理是，当一个元素被加入集合时，通过 K 个散列函数将这个元素映射成一个位数组（Bit array）中的 K 个点，把它们置为 1 。检索时，只要看看这些点是不是都是1就知道元素是否在集合中；如果这些点有任何一个 0，则被检元素一定不在；如果都是1，则被检元素很可能在（之所以说“可能”是误差的存在）

布隆过滤器原理

布隆过滤器（Bloom Filter）的核心实现是一个超大的位数组和几个哈希函数。假设位数组的长度为m，哈希函数的个数为k



以上图为例，具体的操作流程：假设集合里面有3个元素{x, y, z}，哈希函数的个数为3。首先将位数组进行初始化，将里面每个位都设置位0。对于集合里面的每一个元素，将元素依次通过3个哈希函数进行映射，每次映射都会产生一个哈希值，这个值对应位数组上面的一个点，然后将位数组对应的位置标记为1。查询W元素是否存在集合中的时候，同样的方法将W通过哈希映射到位数组上的3个点。如果3个点的其中有一个点不为1，则可以判断该元素一定不存在集合中。反之，如果3个点都为1，则该元素可能存在集合中。注意：此处不能判断该元素是否一定存在集合中，可能存在一定的误判率。可以从图中可以看到：假设某个元素通过映射对应下标为4，5，6这3个点。虽然这3个点都为1，但是很明显这3个点是不同元素经过哈希得到的位置，因此这种情况说明元素虽然不在集合中，也可能对应的都是1，这是误判率存在的原因。