到 num，发现此时 bitArr[num*2+1]和 bitArr[num*2]已经被设置为 11，就不再做任何设置。遍历完成后，再依次遍历 bitArr，如果发现 bitArr[i*2+1]和 bitArr[i*2]设置为 10，那么 $i$ 就是出现了两次的数。

对于补充问题，用分区间的方式处理，长度为 2MB 的无符号整型数组占用的空间为 8MB，所以将区间的数量定为 4294967295/2M，向上取整为 2148 个区间。第 0 区间为 0~2$M$-1，第 1 区间为 2$M$~4$M$-1，第 $i$ 区间为 2$M$×$i$~2$M$×($i$+1)-1……

申请一个长度为 2148 的无符号整型数组 arr[0..2147]，arr[i]表示第 $i$ 区间有多少个数。arr 必然小于 10MB。然后遍历 40 亿个数，如果遍历到当前数为 num，先看 num 落在哪个区间上（num/2M），然后将对应的进行 arr[num/2M]++操作。这样遍历下来，就得到了每一个区间的数的出现状况，通过累加每个区间的出现次数，就可以找到 40 亿个数的中位数（也就是第 20 亿个数）到底落在哪个区间上。比如，0~$K$-1 区间上数的个数为 19.998 亿，但是发现当加上第 $K$ 个区间上数的个数之后就超过了 20 亿，那么可以知道第 20 亿个数是第 $K$ 区间上的数，并且可以知道第 20 亿个数是第 $K$ 区间上的第 0.002 亿个数。

接下来申请一个长度为 2MB 的无符号整型数组 countArr[0..2M-1]，占用空间 8MB。然后再遍历 40 亿个数，此时只关心处在第 $K$ 区间的数记为 numi，其他的数省略，然后将 countArr[numi-K*2M]++，也就是只对第 $K$ 区间的数做频率统计。这次遍历完 40 亿个数之后，就得到了第 $K$ 区间的词频统计结果 countArr，最后只在第 $K$ 区间上找到第 0.002 亿个数即可。

# 一致性哈希算法的基本原理

## 【题目】

工程师常使用服务器集群来设计和实现数据缓存，以下是常见的策略：

1．无论是添加、查询还是删除数据，都先将数据的 id 通过哈希函数转换成一个哈希值，记为 key。

2．如果目前机器有 $N$ 台，则计算 key%N 的值，这个值就是该数据所属的机器编号，无论是添加、删除还是查询操作，都只在这台机器上进行。

请分析这种缓存策略可能带来的问题，并提出改进的方案。

## 【难度】

尉 ★★☆☆

## 【解答】

题目中描述的缓存策略的潜在问题是如果增加或删除机器时（$N$ 变化）代价会很高，所有的数据都不得不根据 id 重新计算一遍哈希值，并将哈希值对新的机器数进行取模操作，然后进行大规模的数据迁移。

为了解决这些问题，下面介绍一下一致性哈希算法，这是一种很好的数据缓存设计方案。我们假设数据的 id 通过哈希函数转换成的哈希值范围是 $2^{32}$，也就是 0~($2^{32}$)-1 的数字空间中。现在我们可以将这些数字头尾相连，想象成一个闭合的环形，那么一个数据 id 在计算出哈希值之后认为对应到环中的一个位置上，如图 6-3 所示。

接下来想象有三台机器也处在这样一个环中，这三台机器在环中的位置根据机器 id 计算出的哈希值来决定。那么一条数据如何确定归属哪台机器呢？首先把该数据的 id 用哈希函数算出哈希值，并映射到环中的相应位置，然后顺时针找寻离这个位置最近的机器，那台机器就是该数据的归属，如图 6-4 所示。

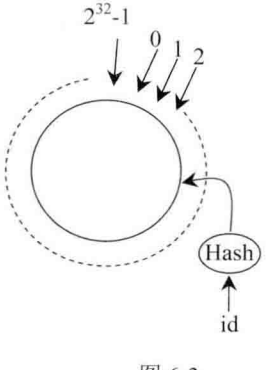


图 6-3

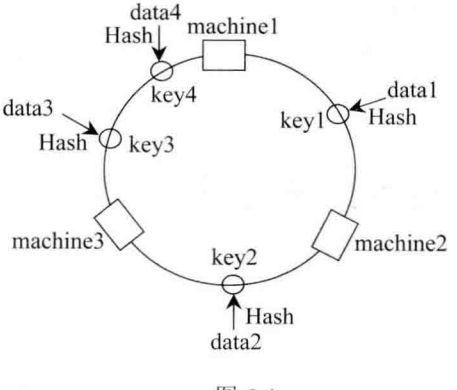


图 6-4

在图 6-4 中，data1 根据其 id 计算出的哈希值为 key1，顺时针的第一台机器是 machine2，所以 data1 归属 machine2；同理，data2 归属 machine3，data3 和 data4 都归属 machine1。

增加机器时的处理。假设有两台机器（m1、m2）和三个数据（data1、data2、data3），数据和机器在环中的结构如图 6-5 所示。

如果此时想加入新的机器 m3，同时算出机器 m3 的 id 在 m1 与 m2 右半侧的环中，那么发生的变化如图 6-6 所示。

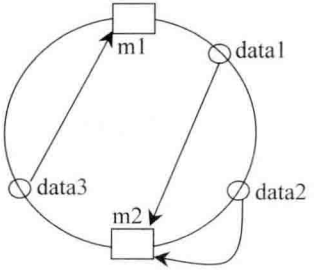


图 6-5

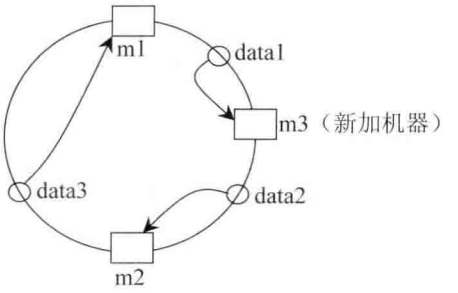


图 6-6

在没有添加 m3 之前，从 m1 到现在 m3 位置上的这一段是 m2 掌管范围的一部分；添加 m3 之后则统一归属于 m3，同时要把这一段旧数据从 m2 迁移到 m3 上。由此可见，添加机器时的调整代价是比较小的。在删除机器时也一样，只要把要删除机器的数据全部复制到顺时针找到的下一台机器上即可。比如，要在图 6-6 中删除机器 m2，m2 上有数据 data2，那么只用把 data2 迁移到 m1 上即可。

机器负载不均时的处理。如果机器较少，很有可能造成机器在整个环上的分布不均匀，从而导致机器之间的负载不均衡，比如，图 6-7 所示的两台机器，m1 可能比 m2 面临更大的负载。

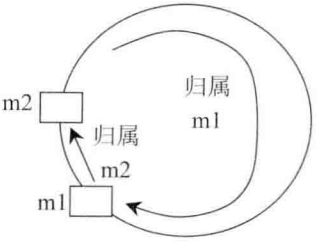


图 6-7

为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一台机器通过不同的哈希函数计算出多个哈希值，对多个位置都放置一个服务节点，称为虚拟节点。具体做法可以在机器 ip 或主机名的后面增加编号或端口号来实现。以图 6-7 的情况，可以为每台机器计算两个虚拟节点，分别计算 m1-1、m1-2、m2-1 和 m2-2 的哈希值，于是形成四个虚拟节点，节点数变多了，根据哈希函数的性质，平衡性自然会变好，如图 6-8 所示。

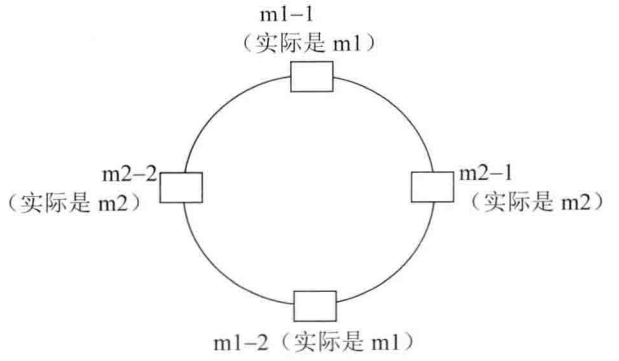


图 6-8

此时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，比如下表：

| 虚拟节点 | 对应的实际节点 |
| --- | --- |
| m1-1 | m1 |
| m1-2 | m1 |
| m2-1 | m2 |
| m2-2 | m2 |

当某一条数据计算出归属于 m1-2 时，再根据上表的转跳，数据将最终归属于实际的 m1 节点。基于一致性哈希的原理有多种具体的实现，包括 Chord 算法、KAD 算法等。有兴趣的读者可以进一步学习，本书由于篇幅所限，在此不再详述。