<https://mp.weixin.qq.com/s/EF1cgBlJB3U37oZm3KgQvQ>

<https://www.jianshu.com/p/55defda6dcd2>

# Reids(4)——神奇的HyperLoglog解决统计问题

# **一、HyperLogLog 简介**

**HyperLogLog** 是最早由 Flajolet 及其同事在 2007 年提出的一种 **估算基数的近似最优算法**。但跟原版论文不同的是，好像很多书包括 Redis 作者都把它称为一种 **新的数据结构(new datastruct)** *(算法实现确实需要一种特定的数据结构来实现)*。

## **关于基数统计**

**基数统计(Cardinality Counting)** 通常是用来统计一个集合中不重复的元素个数。

**思考这样的一个场景：** 如果你负责开发维护一个大型的网站，有一天老板找产品经理要网站上每个网页的 **UV(独立访客，每个用户每天只记录一次)**，然后让你来开发这个统计模块，你会如何实现？

如果统计 **PV(浏览量，用户没点一次记录一次)**，那非常好办，给每个页面配置一个独立的 Redis 计数器就可以了，把这个计数器的 key 后缀加上当天的日期。这样每来一个请求，就执行 INCRBY 指令一次，最终就可以统计出所有的 **PV** 数据了。

但是 **UV** 不同，它要去重，**同一个用户一天之内的多次访问请求只能计数一次**。这就要求了每一个网页请求都需要带上用户的 ID，无论是登录用户还是未登录的用户，都需要一个唯一 ID 来标识。

你也许马上就想到了一个 *简单的解决方案*：那就是 **为每一个页面设置一个独立的 set 集合** 来存储所有当天访问过此页面的用户 ID。但这样的 **问题** 就是：

1. **存储空间巨大：** 如果网站访问量一大，你需要用来存储的 set 集合就会非常大，如果页面再一多.. 为了一个去重功能耗费的资源就可以直接让你 **老板打死你**；
2. **统计复杂：** 这么多 set 集合如果要聚合统计一下，又是一个复杂的事情；

## **基数统计的常用方法**

对于上述这样需要 **基数统计** 的事情，通常来说有两种比 set 集合更好的解决方案：

### **第一种：B 树**

**B 树最大的优势就是插入和查找效率很高**，如果用 B 树存储要统计的数据，可以快速判断新来的数据是否存在，并快速将元素插入 B 树。要计算基础值，只需要计算 B 树的节点个数就行了。

不过将 B 树结构维护到内存中，能够解决统计和计算的问题，但是 **并没有节省内存**。

### **第二种：bitmap**

**bitmap** 可以理解为通过一个 bit 数组来存储特定数据的一种数据结构，**每一个 bit 位都能独立包含信息**，bit 是数据的最小存储单位，因此能大量节省空间，也可以将整个 bit 数据一次性 load 到内存计算。如果定义一个很大的 bit 数组，基数统计中 **每一个元素对应到 bit 数组中的一位**，例如：



bitmap 还有一个明显的优势是 **可以轻松合并多个统计结果**，只需要对多个结果求异或就可以了，也可以大大减少存储内存。可以简单做一个计算，如果要统计 **1 亿** 个数据的基数值，**大约需要的内存**：100\_000\_000/ 8/ 1024/ 1024 ≈ 12 M，如果用 **32 bit** 的 int 代表 **每一个** 统计的数据，**大约需要内存**：32 \* 100\_000\_000/ 8/ 1024/ 1024 ≈ 381 M

可以看到 bitmap 对于内存的节省显而易见，但仍然不够。统计一个对象的基数值就需要 12 M，如果统计 1 万个对象，就需要接近 120 G，对于大数据的场景仍然不适用。

## **概率算法**

实际上目前还没有发现更好的在 **大数据场景** 中 **准确计算** 基数的高效算法，因此在不追求绝对精确的情况下，使用概率算法算是一个不错的解决方案。

概率算法 **不直接存储** 数据集合本身，通过一定的 **概率统计方法预估基数值**，这种方法可以大大节省内存，同时保证误差控制在一定范围内。目前用于基数计数的概率算法包括:

* **Linear Counting(LC)**：早期的基数估计算法，LC 在空间复杂度方面并不算优秀，实际上 LC 的空间复杂度与上文中简单 bitmap 方法是一样的（但是有个常数项级别的降低），都是 O(Nmax)
* **LogLog Counting(LLC)**：LogLog Counting 相比于 LC 更加节省内存，空间复杂度只有 O(log2(log2(Nmax)))
* **HyperLogLog Counting(HLL)**：HyperLogLog Counting 是基于 LLC 的优化和改进，在同样空间复杂度情况下，能够比 LLC 的基数估计误差更小

其中，**HyperLogLog** 的表现是惊人的，上面我们简单计算过用 **bitmap** 存储 **1 个亿** 统计数据大概需要 12 M 内存，而在 **HyperLoglog** 中，只需要不到 **1 K** 内存就能够做到！在 Redis 中实现的 **HyperLoglog** 也只需要 **12 K** 内存，在 **标准误差 0.81%** 的前提下，**能够统计 264 个数据**！

# **二、HyperLogLog 原理**

我们来思考一个抛硬币的游戏：你连续掷 n 次硬币，然后说出其中**连续掷为正面的最大次数，我来猜你一共抛了多少次**。

这很容易理解吧，例如：你说你这一次 *最多连续出现了 2 次* 正面，那么我就可以知道你这一次投掷的次数并不多，所以 *我可能会猜是 5* 或者是其他小一些的数字，但如果你说你这一次 *最多连续出现了 20 次* 正面，虽然我觉得不可能，但我仍然知道你花了特别多的时间，所以 *我说 GUN...*。

这期间我可能会要求你重复实验，然后我得到了更多的数据之后就会估计得更准。**我们来把刚才的游戏换一种说法**：



这张图的意思是，我们给定一系列的随机整数，**记录下低位连续零位的最大长度 K**，即为图中的 maxbit，**通过这个 K 值我们就可以估算出随机数的数量 N**。

## **代码实验**

我们可以简单编写代码做一个实验，来探究一下 K 和 N 之间的关系：