HyperLogLog 算法的原理讲解以及 Redis 是如何应用它的

> 作者：林冠宏 / 指尖下的幽灵

目录

问题原形

如果要实现这么一个功能：

> 统计 APP或网页 的一个页面，每天有多少用户点击进入的次数。同一个用户的反复点击进入记为 1 次。

聪明的你可能会马上想到，用 `HashMap` 这种数据结构就可以了，也满足了去重。的确，这是一种解决方法，除此之外还有其它的解决方案。

问题虽不难，但当参与问题中的变量达到一定数量级的时候，再简单的问题都会变成一个难题。假设 APP 中日活用户达到 `百万` 或 `千万以上级别` 的话，我们采用 `HashMap` 的做法，就会导致程序中占用大量的内存。

我们下面尝试估算下 `HashMap` 的在应对上述问题时候的内存占用。假设定义 `HashMap` 中 `Key` 为 `string` 类型， `value` 为 `bool` 。 `key` 对应用户的 `Id` , `value` 是 `是否点击进入` 。明显地，当百万不同用户访问的时候。此 `HashMap` 的内存占用空间为： `100万 * (string + bool)` 。

条件选择

可以说，在上述问题目前现有的解决方案中，`HashMap` 是内存占用量最多的一种。如果统计量不多，那么可以使用这种方法解决问题，实现起来也简单。

除此之外还有 `B+ 树`，`Bitmap 位图`，以及该文章主要介绍的 `HyperLogLog` 算法解决方案。

在一定条件允许下，如果允许统计在巨量数据面前的误差率在可接受的范围内，1000万浏览量允许最终统计出少了一两万这样子，那么就可以采用 `HyperLogLog` 算法来解决上面的计数类似问题。

HyperLogLog

`HyperLogLog`，下面简称为 `HLL`，它是 `LogLog` 算法的升级版，作用是能够提供不精确的去重计数。存在以下的特点：

- 代码实现较难。
- 能够使用极少的内存来统计巨量的数据，在 `Redis` 中实现的 `HyperLogLog`，只需要 `12K` 内存就能统计 `2^64` 个数据。
- 计数存在一定的误差，误差率整体较低。标准误差为 0.81% 。
- 误差可以被设置 `辅助计算因子` 进行降低。

稍微对编程中的基础数据类型内存占用有了解的同学，应该会对其只需要 `12K` 内存就能统计 `2^64` 个数据而感到惊讶。为什么这样说呢，下面我们举下例子：

取 `Java` 语言来说，一般 `long` 占用8字节，而一字节有8位，即：1 byte = 8 bit，即 `long` 数据类型最大可以表示的数是：`2^63-1`。对应上面的 `2^64` 个数，假设此时有 `2^63-1` 这么多个数，从 `0 ~ 2^63-1`，按照 `long` 以及 `1k = 1024字节` 的规则来计算内存总数，就是：`((2^63-1) * 8/1024)K`，这是很庞大的一个数，存储空间远远超过 `12K`。而 `HyperLogLog` 却可以用 `12K` 就能统计完。

伯努利试验

在认识为什么 `HyperLogLog` 能够使用极少的内存来统计巨量的数据之前，要先认识下 `伯努利试验`。

> `伯努利试验` 是数学 `概率论` 中的一部分内容，它的典故来源于 `抛硬币`。

硬币拥有正反两面，一次的上抛至落下，最终出现正反面的概率都是50%。假设一直抛硬币，直到它出现正面为止，我们记录为一次完整的试验，间中可能抛了一次就出现了正面，也可能抛了4次才出现正面。无论抛了多少次，只要出现了正面，就记录为一次试验。这个试验就是 `伯努利试验`。

那么对于多次的 `伯努利试验`，假设这个多次为 `n` 次。就意味着出现了 `n` 次的正面。假设每次 `伯努利试验` 所经历了的抛掷次数为 `k`。第一次 `伯努利试验`，次数设为 `k1`，以此类推，第 `n` 次对应的是 `kn`。

其中，对于这 n 次 伯努利试验 中，必然会有一个最大的抛掷次数 k，例如抛了12次才出现正面，那么称这个为 k_max，代表抛了最多的次数。

伯努利试验 容易得出有以下结论：

1. n 次伯努利过程的投掷次数都不大于 k_max。
2. n 次伯努利过程，至少有一次投掷次数等于 k_max

最终结合极大似然估算的方法，发现在 n 和 k_max 中存在估算关联： n = 2^(k_max) 。这种通过局部信息预估整体数据流特性的方法似乎有些超出我们的基本认知，需要用概率和统计的方法才能推导和验证这种关联关系。

例如下面的样子：

```
复制代码
第一次试验：抛了3次才出现正面，此时 k=3，n=1
第二次试验：抛了2次才出现正面，此时 k=2，n=2
第三次试验：抛了6次才出现正面，此时 k=6，n=3
第n 次试验：抛了12次才出现正面，此时我们估算， n = 2^12
```

假设上面例子中实验组数共3组，那么 k_max = 6，最终 n=3，我们放进估算公式中去，明显： 3 ≠ 2^6 。也即是说，当试验次数很小的时候，这种估算方法的误差是很大的。

估算的优化

在上面的3组例子中，我们称为一轮的估算。如果只是进行一轮的话，当 n 足够大的时候，估算的误差率会相对减少，但仍然不够小。

那么是否可以进行多轮呢？例如进行 100 轮或者更多轮次的试验，然后再取每轮的 k_max，再取平均数，即: k_mx/100 。最终再估算出 n。下面是 LogLog 的估算公式：

$$DV_{LL} = \text{constant} * m * 2^{\overline{R}}$$

上面公式的 DVLL 对应的就是 n， constant 是修正因子，它的具体值是不定的，可以根据实际情况而分支设置。 m 代表的是试验的轮数。头上有一横的 R 就是平均数： (k_max_1 + ... + k_max_m)/m 。

这种通过增加试验轮次，再取 k_max 平均数的算法优化就是 LogLog 的做法。而 HyperLogLog 和 LogLog 的区别就是，它采用的不是 平均数 ，而是 调和平均数 。 调和平均数 比 平均数 的好处就是不容易受到大的数值的影响。下面举个例子：

> 求平均工资:

> A的是1000/月，B的30000/月。采用平均数的方式就是： (1000 + 30000) / 2 = 15500

> 采用调和平均数的方式就是： 2/(1/1000 + 1/30000) ≈ 1935.484

明显地， `调和平均数` 比 `平均数` 的效果是要更好的。下面是 `调和平均数` 的计算方式， `∑` 是累加符号。

$$H_n = \frac{1}{\frac{1}{n}\sum_{i=1}^{n}\frac{1}{x_i}} = \frac{n}{\sum_{i=1}^{n}\frac{1}{x_i}}$$

扯上关系

上面的内容我们已经知道，在抛硬币的例子中，可以通过一次伯努利试验中出现的 `k_max` 来估算 `n` 。

那么这种估算方法如何和下面问题有所关联呢?

> 统计 APP或网页 的一个页面，每天有多少用户点击进入的次数。同一个用户的反复点击进入记为 1 次

`HyperLogLog` 是这样做的。对于输入的数据，进行下面几个步骤：

1.比特串

通过 `hash` 函数，将数据转为 `比特串` ，例如输入5，便转为：101。为什么要这样转化呢?

是因为要和抛硬币对应上， `比特串` 中，0 代表了反面，1 代表了正面，如果一个数据最终被转化了 `10010000` ，那么从右往左，从低位往高位看，我们可以认为，首次出现 1 的时候，就是正面。

那么基于上面的估算结论，我们可以通过多次抛硬币实验的最大抛到正面的次数来预估总共进行了多少次实验，同样也就可以根据存入数据中，转化后的出现了 1 的最大的位置 k_max 来估算存入了多少数据。

2.分桶

分桶就是分多少轮。抽象到计算机存储中去，就是存储的是一个以单位是比特(bit)，长度为 L 的大数组 S ，将 S 平均分为 m 组，注意这个 m 组，就是对应多少轮，然后每组所占有的比特个数是平均的，设为 P。容易得出下面的关系：

- L = S.length
- L = m * p
- 以 K 为单位，S 占用的内存 = L / 8 / 1024

在 `Redis` 中， `HyperLogLog` 设置为：m=16834，p=6，L=16834 * 6。占用内存为=16834 * 6 / 8 / 1024 = 12K

形象化为：

```
  第0组     第1组                       .... 第16833组
[000 000] [000 000] [000 000] [000 000] .... [000 000]
```

3. 对应

现在回到我们的原始APP页面统计用户的问题中去。

- 设 APP 主页的 key 为： main
- 用户 id 为：idn , n->0,1,2,3....

在这个统计问题中，不同的用户 id 标识了一个用户，那么我们可以把用户的 id 作为被 `hash` 的输入。即：

> hash(id) = 比特串

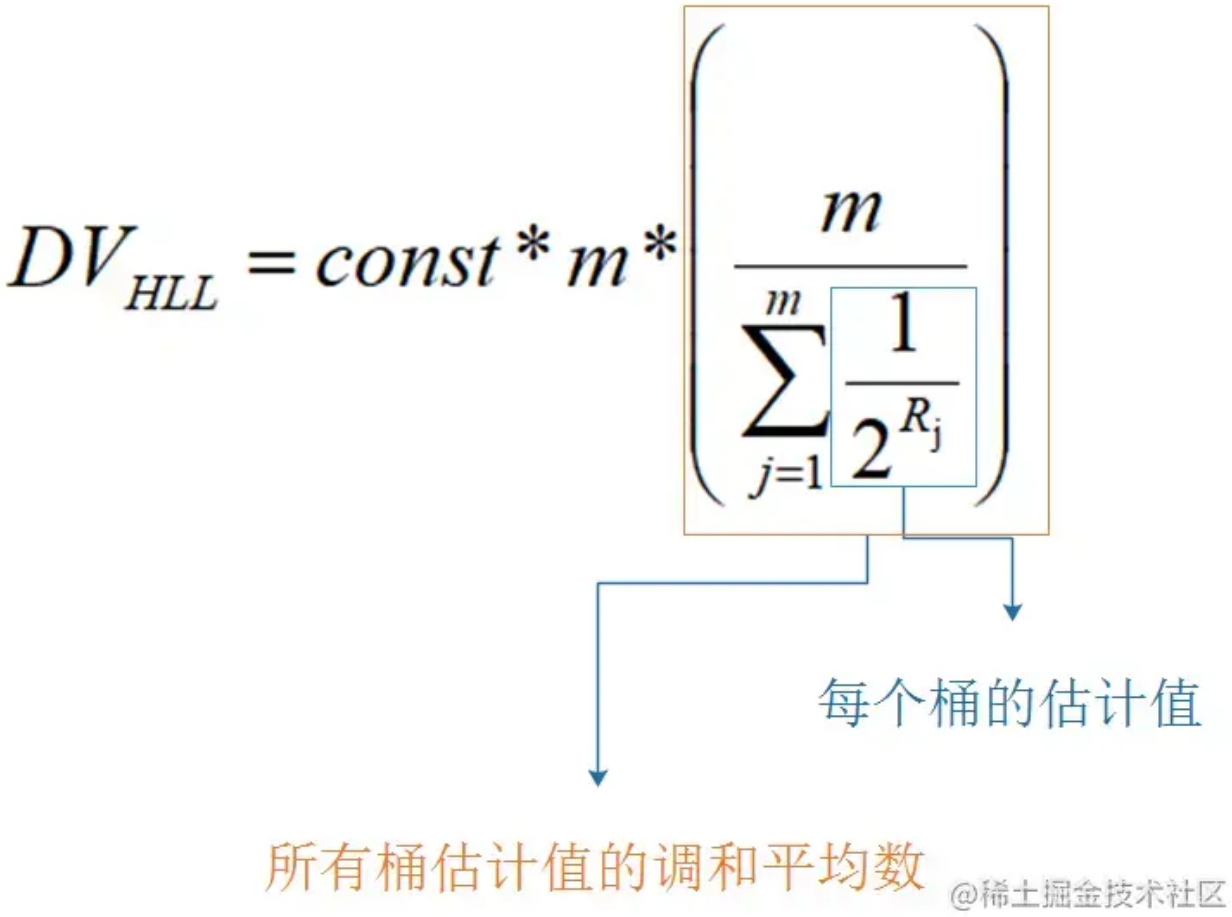
不同的用户 id，必然拥有不同的 `比特串` 。每一个 `比特串` ，也必然会至少出现一次 1 的位置。我们类比每一个 `比特串` 为一次 `伯努利试验` 。

现在要 `分轮` ，也就是 `分桶` 。所以我们可以设定，每个 `比特串` 的前多少位转为10进制后，其值就对应于所在桶的标号。假设 `比特串` 的低两位用来计算桶下标志，此时有一个用户的id的 `比特串` 是：1001011000011。它的所在桶下标为： `11(2) = 1*2^1 + 1*2^0 = 3` ，处于第3个桶，即第3轮中。

上面例子中，计算出桶号后，剩下的 `比特串` 是：10010110000，从低位到高位看，第一次出现 1 的位置是 5 。也就是说，此时第3个桶，第3轮的试验中， `k_max = 5` 。5 对应的二进制是：101，又因为每个桶有 p 个比特位。当 p>=3 时，便可以将 101 存进去。

模仿上面的流程，多个不同的用户 id，就被分散到不同的桶中去了，且每个桶有其 k_max。然后当要统计出 `mian` 页面有多少用户点击量的时候，就是一次估算。最终结合所有桶中的 k_max，代入估算公式，便能得出估算值。

下面是 `HyperLogLog` 的结合了调和平均数的估算公式，变量释意和 `LogLog` 的一样:



Redis 中对 HyperLogLog 的应用

首先，在 Redis 中，HyperLogLog 是它的一种高级数据结构。提供有包含但不限于下面两条命令：

- pfadd key value，将 key 对应的一个 value 存入
- pfcount key，统计 key 的 value 有多少个

回想一下，原始APP页面统计用户的问题。如果 key 对应页面名称，value 对应用户id。那么问题就刚刚好对应上了。

Redis 中的 HyperLogLog 原理

前面我们已经认识到，它的实现中，设有 16384 个桶，即：2^14 = 16384，每个桶有 6 位，每个桶可以表达的最大数字是：2^5+2^4+...+1 = 63 ，二进制为： `111 111` 。

对于命令：`pfadd key value`

在存入时，value 会被 hash 成 64 位，即 64 bit 的比特字符串，前 14 位用来分桶，前 14 位的二进制转为 10 进制就是桶标号。

之所以选 `14位` 来表达桶编号是因为，分了 16384 个桶，而 2^14 = 16384，刚好地，最大的时候可以把桶利用完，不造成浪费。假设一个字符串的前 14 位是：00 0000 0000 0010，其十进制值为 2。那么 index 将会被转化后放到编号为 2 的桶。

index 的转化规则：

首先因为完整的 value 比特字符串是 64 位形式，减去 14 后，剩下 50 位，那么极端情况，出现 1 的位置，是在第 50 位，即位置是 50。此时 index = 50。此时先将 index 转为 2 进制，它是：110010 。

因为16384 个桶中，每个桶是 6 bit 组成的。刚好 110010 就被设置到了第 2 号桶中去了。请注意，50 已经是最坏的情况，且它都被容纳进去了。那么其他的不用想也肯定能被容纳进去。

因为 fpadd 的 key 可以设置多个 value。例如下面的例子：

复制代码

```
pfadd lgh golang
pfadd lgh python
pfadd lgh java
```

根据上面的做法，不同的 value，会被设置到不同桶中去，如果出现了在同一个桶的，即前 14 位值是一样的，但是后面出现 1 的位置不一样。那么比较原来的 index 是否比新 index 大。是，则替换。否，则不变。

最终地，一个 key 所对应的 16384 个桶都设置了很多的 value 了，每个桶有一个 `k_max` 。此时调用 pfcount 时，按照前面介绍的估算方式，便可以计算出 key 的设置了多少次 value，也就是统计值。

value 被转为 64 位的比特串，最终被按照上面的做法记录到每个桶中去。64 位转为十进制就是：2^64，`HyperLogLog` 仅用了： `16384 * 6 /8 / 1024 K` 存储空间就能统计多达 2^64 个数。

偏差修正

在估算的计算公式中， `constant` 变量不是一个定值，它会根据实际情况而被分支设置，例如下面的样子。

假设：m为分桶数，p是m的以2为底的对数。

$$p = \log_2 m$$

复制代码

```
// m 为桶数
switch (p) {
   case 4:
       constant = 0.673 * m * m;
   case 5:
       constant = 0.697 * m * m;
   case 6:
       constant = 0.709 * m * m;
   default:
       constant = (0.7213 / (1 + 1.079 / m)) * m * m;
}
```

巨人的肩膀

由简单的抛硬币试验可以引导出如此的震撼的算法，数学之强大。

感谢下面两遍博文的指引：

本文所有图片来源于：

www.jianshu.com/p/55defda6d...

本文内容参考于：

www.rainybowe.com/blog/2017/0...

手动直观观察 `LogLog` 和 `HyperLogLog` 变化的网站：

content.research.neustar.biz/blog/hll.ht...