<https://mp.weixin.qq.com/s/u0iUL-d-qH-r1vjDEX9yzA>

# Redis(5)——亿级数据过滤和布隆过滤器

# **一、布隆过滤器简介**

上一次 我们学会了使用 **HyperLogLog** 来对大数据进行一个估算，它非常有价值，可以解决很多精确度要求不高的统计需求。但是如果我们想知道某一个值是不是已经在 **HyperLogLog** 结构里面了，它就无能为力了，它只提供了 pfadd 和 pfcount 方法，没有提供类似于 contains 的这种方法。

就举一个场景吧，比如你 **刷抖音**：



你有 **刷到过重复的推荐内容** 吗？这么多的推荐内容要推荐给这么多的用户，它是怎么保证每个用户在看推荐内容时，保证不会出现之前已经看过的推荐视频呢？也就是说，抖音是如何实现 **推送去重** 的呢？

你会想到服务器 **记录** 了用户看过的 **所有历史记录**，当推荐系统推荐短视频时会从每个用户的历史记录里进行 **筛选**，过滤掉那些已经存在的记录。问题是当 **用户量很大**，每个用户看过的短视频又很多的情况下，这种方式，推荐系统的去重工作 **在性能上跟的上么？**

实际上，如果历史记录存储在关系数据库里，去重就需要频繁地对数据库进行 exists 查询，当系统并发量很高时，数据库是很难抗住压力的。

你可能又想到了 **缓存**，但是这么多用户这么多的历史记录，如果全部缓存起来，那得需要 **浪费多大的空间** 啊.. *(可能老板看一眼账单，看一眼你..)* 并且这个存储空间会随着时间呈线性增长，就算你用缓存撑得住一个月，但是又能继续撑多久呢？不缓存性能又跟不上，咋办呢？



如上图所示，**布隆过滤器(Bloom Filter)** 就是这样一种专门用来解决去重问题的高级数据结构。但是跟 **HyperLogLog** 一样，它也一样有那么一点点不精确，也存在一定的误判概率，但它能在解决去重的同时，在 **空间上能节省 90%** 以上，也是非常值得的。

## **布隆过滤器是什么**

**布隆过滤器（Bloom Filter）** 是 1970 年由布隆提出的。它 **实际上** 是一个很长的二进制向量和一系列随机映射函数 *(下面详细说)*，实际上你也可以把它 **简单理解** 为一个不怎么精确的 **set** 结构，当你使用它的 contains 方法判断某个对象是否存在时，它可能会误判。但是布隆过滤器也不是特别不精确，只要参数设置的合理，它的精确度可以控制的相对足够精确，只会有小小的误判概率。

当布隆过滤器说某个值存在时，这个值 **可能不存在**；当它说不存在时，那么 **一定不存在**。打个比方，当它说不认识你时，那就是真的不认识，但是当它说认识你的时候，可能是因为你长得像它认识的另外一个朋友 *(脸长得有些相似)*，所以误判认识你。

## **布隆过滤器的使用场景**

基于上述的功能，我们大致可以把布隆过滤器用于以下的场景之中：

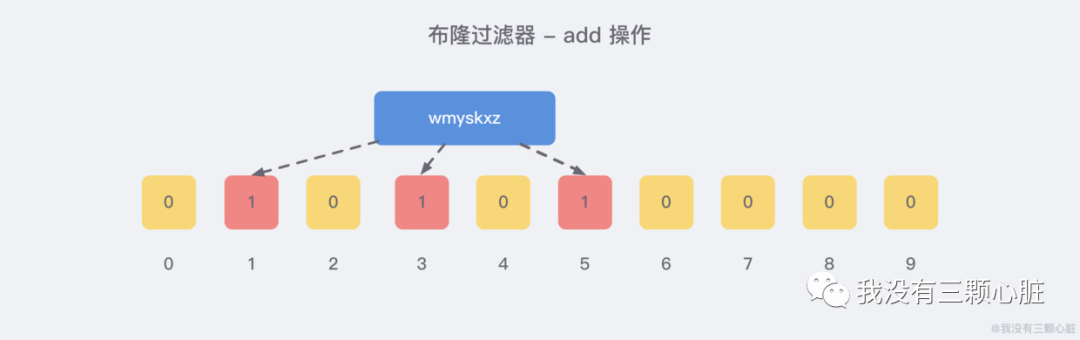
* **大数据判断是否存在**：这就可以实现出上述的去重功能，如果你的服务器内存足够大的话，那么使用 HashMap 可能是一个不错的解决方案，理论上时间复杂度可以达到 O(1 的级别，但是当数据量起来之后，还是只能考虑布隆过滤器。
* **解决缓存穿透**：我们经常会把一些热点数据放在 Redis 中当作缓存，例如产品详情。通常一个请求过来之后我们会先查询缓存，而不用直接读取数据库，这是提升性能最简单也是最普遍的做法，但是 **如果一直请求一个不存在的缓存**，那么此时一定不存在缓存，那就会有 **大量请求直接打到数据库** 上，造成 **缓存穿透**，布隆过滤器也可以用来解决此类问题。
* **爬虫/ 邮箱等系统的过滤**：平时不知道你有没有注意到有一些正常的邮件也会被放进垃圾邮件目录中，这就是使用布隆过滤器 **误判** 导致的。

# **二、布隆过滤器原理解析**

布隆过滤器 **本质上** 是由长度为 m 的位向量或位列表（仅包含 0 或 1 位值的列表）组成，最初所有的值均设置为 0，所以我们先来创建一个稍微长一些的位向量用作展示：

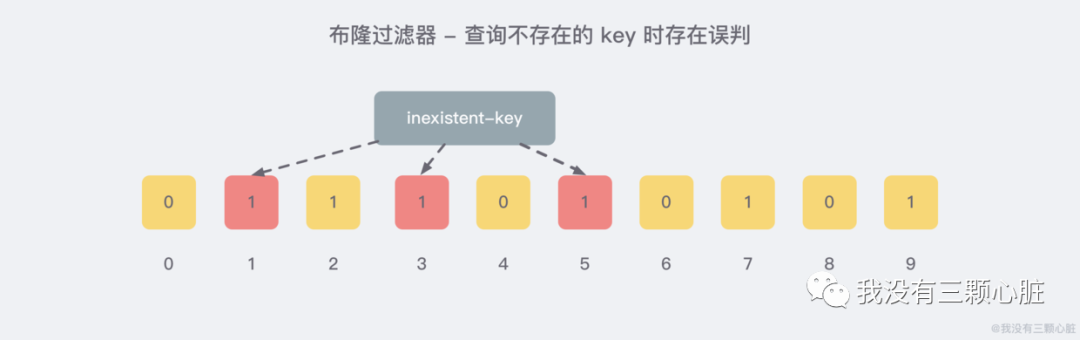


当我们向布隆过滤器中添加数据时，会使用 **多个** hash 函数对 key 进行运算，算得一个证书索引值，然后对位数组长度进行取模运算得到一个位置，每个 hash 函数都会算得一个不同的位置。再把位数组的这几个位置都置为 1 就完成了 add 操作，例如，我们添加一个 wmyskxz：



向布隆过滤器查查询 key 是否存在时，跟 add 操作一样，会把这个 key 通过相同的多个 hash 函数进行运算，查看 **对应的位置** 是否 **都** 为 1，**只要有一个位为**0，那么说明布隆过滤器中这个 key 不存在。如果这几个位置都是 1，并不能说明这个 key 一定存在，只能说极有可能存在，因为这些位置的 1 可能是因为其他的 key 存在导致的。

就比如我们在 add 了一定的数据之后，查询一个 **不存在** 的 key：

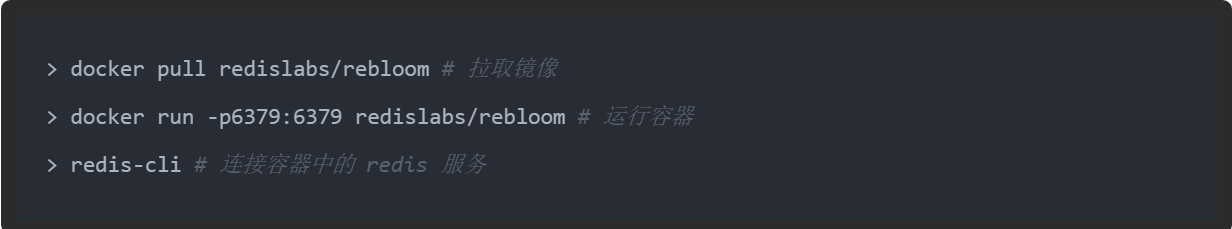


很明显，1/3/5 这几个位置的 1 是因为上面第一次添加的 wmyskxz 而导致的，所以这里就存在 **误判**。幸运的是，布隆过滤器有一个可以预判误判率的公式，比较复杂，感兴趣的朋友可以自行去阅读，比较烧脑.. 只需要记住以下几点就好了：

* 使用时 **不要让实际元素数量远大于初始化数量**；
* 当实际元素数量超过初始化数量时，应该对布隆过滤器进行 **重建**，重新分配一个 size 更大的过滤器，再将所有的历史元素批量 add 进行；

# **三、布隆过滤器的使用**

**Redis 官方** 提供的布隆过滤器到了 **Redis 4.0** 提供了插件功能之后才正式登场。布隆过滤器作为一个插件加载到 Redis Server 中，给 Redis 提供了强大的布隆去重功能。下面我们来体验一下 Redis 4.0 的布隆过滤器，为了省去繁琐安装过程，我们直接用 Docker 吧。

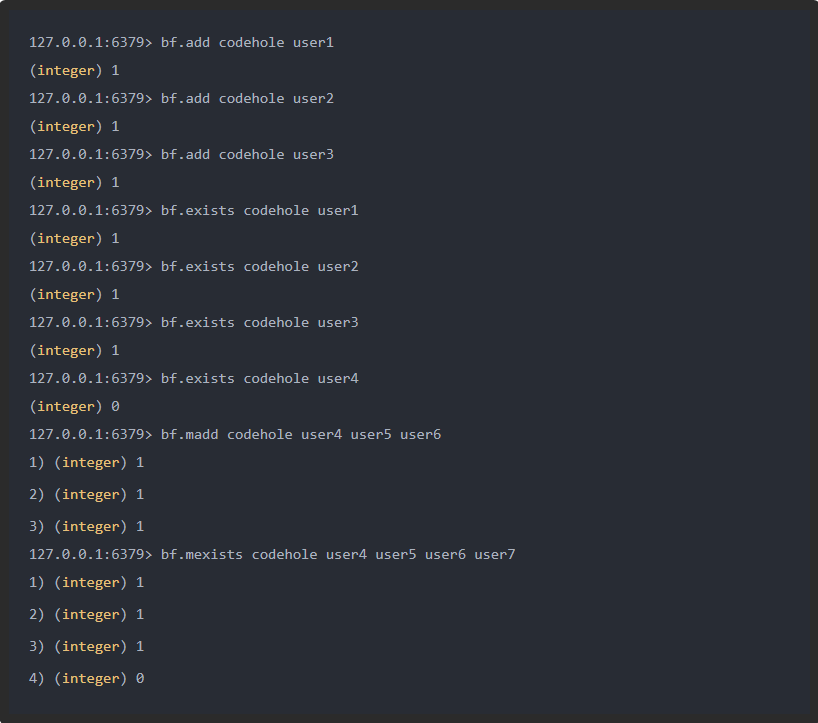


如果上面三条指令执行没有问题，下面就可以体验布隆过滤器了。

* 当然，如果你不想使用 Docker，也可以在检查本机 Redis 版本合格之后自行安装插件，可以参考这里: https://blog.csdn.net/u013030276/article/details/88350641

## **布隆过滤器的基本用法**

布隆过滤器有两个基本指令，bf.add 添加元素，bf.exists 查询元素是否存在，它的用法和 set 集合的 sadd 和 sismember 差不多。注意 bf.add 只能一次添加一个元素，如果想要一次添加多个，就需要用到 bf.madd 指令。同样如果需要一次查询多个元素是否存在，就需要用到 bf.mexists 指令。



上面使用的布隆过过滤器只是默认参数的布隆过滤器，它在我们第一次 add 的时候自动创建。Redis 也提供了可以自定义参数的布隆过滤器，只需要在 add 之前使用 bf.reserve 指令显式创建就好了。如果对应的 key 已经存在，bf.reserve 会报错。

bf.reserve 有三个参数，分别是 key、error\_rate *(错误率)* 和 initial\_size：

* error\_rate**越低，需要的空间越大**，对于不需要过于精确的场合，设置稍大一些也没有关系，比如上面说的推送系统，只会让一小部分的内容被过滤掉，整体的观看体验还是不会受到很大影响的；
* initial\_size**表示预计放入的元素数量**，当实际数量超过这个值时，误判率就会提升，所以需要提前设置一个较大的数值避免超出导致误判率升高；

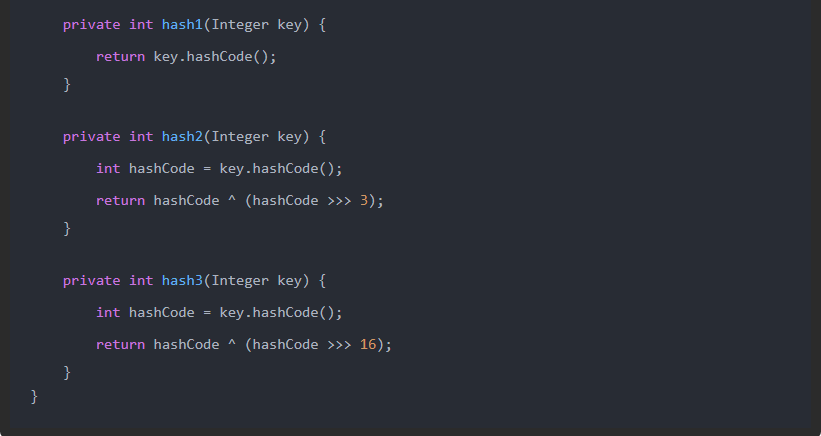
如果不适用 bf.reserve，默认的 error\_rate 是 0.01，默认的 initial\_size 是 100。

# **四、布隆过滤器代码实现**

## **自己简单模拟实现**

根据上面的基础理论，我们很容易就可以自己实现一个用于 简单模拟 的布隆过滤器数据结构：





这里很简单，内部仅维护了一个 byte 类型的 data 数组，实际上 byte 仍然占有一个字节之多，可以优化成 bit 来代替，这里也仅仅是用于方便模拟。另外我也创建了三个不同的 hash 函数，其实也就是借鉴 HashMap 哈希抖动的办法，分别使用自身的 hash 和右移不同位数相异或的结果。并且提供了基础的 add 和 contains 方法。

下面我们来简单测试一下这个布隆过滤器的效果如何：

这就是前面说到的，当布隆过滤器说某个值 **存在时**，这个值 **可能不存在**，当它说某个值 **不存在时**，那就 **肯定不存在**，并且还有一定的误判率...

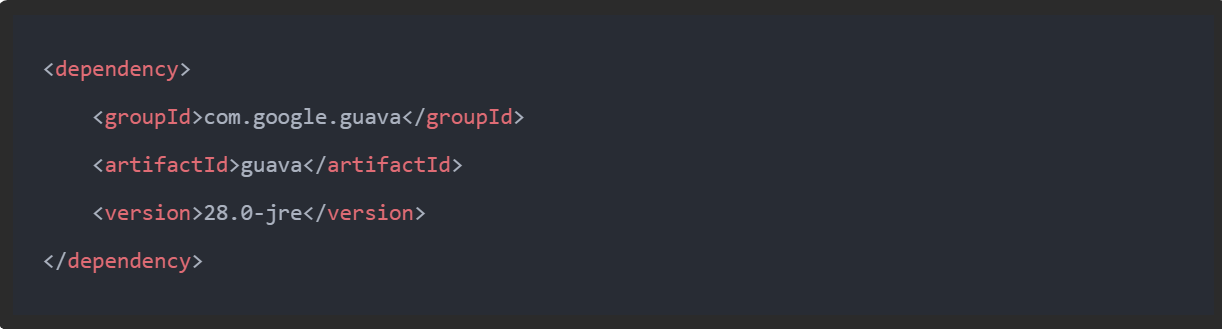
## **手动实现参考**

当然上面的版本特别 low，不过主体思想是不差的，这里也给出一个好一些的版本用作自己实现测试的参考：

## **使用 Google 开源的 Guava 中自带的布隆过滤器**

自己实现的目的主要是为了让自己搞懂布隆过滤器的原理，Guava 中布隆过滤器的实现算是比较权威的，所以实际项目中我们不需要手动实现一个布隆过滤器。

首先我们需要在项目中引入 Guava 的依赖：



实际使用如下：

我们创建了一个最多存放 最多 1500 个整数的布隆过滤器，并且我们可以容忍误判的概率为百分之（0.01）



在我们的示例中，当 mightContain() 方法返回 true 时，我们可以 **99％** 确定该元素在过滤器中，当过滤器返回 false 时，我们可以 **100％** 确定该元素不存在于过滤器中。

Guava 提供的布隆过滤器的实现还是很不错的 *（想要详细了解的可以看一下它的源码实现）*，但是它有一个重大的缺陷就是只能单机使用 *（另外，容量扩展也不容易）*，而现在互联网一般都是分布式的场景。为了解决这个问题，我们就需要用到 **Redis** 中的布隆过滤器了。