

腾讯太狠：40亿QQ号，给1G内存，怎么去重？

原创 鬼仔 码农鬼仔 2023-07-29 18:30 发表于广东

收录于合集

#特殊题型面经

2个

在鬼仔粉丝中，最近有小伙伴拿到了一线互联网企业如腾讯、美团、阿里、字节的面试资格，遇到几个很重要的面试题：

- 40亿QQ号如何设计算法去重，相同的QQ号码仅保留一个，内存限制为1个G？
- 40亿个QQ号，限制1G内存，如何去重？

类似的问题还包括以下几种情况：

- 如何在1G内存限制下对60亿个URL实现去重处理？
- 在1G内存限制下，文件中包含40亿个QQ号码，请设计一个算法进行去重，使得相同的QQ号码只保留一个。
- 等等等等.....

在这里，鬼仔将为大家提供一个**系统化、体系化**的整理方法，以便大家在面试过程中充分展示自己强大的“技术实力”，让面试官毫不犹豫地当场发offer！

本文目录

- 一、问题场景分析
- 二、方法1：利用Bitmap实现海量数据去重
 - Bitmap是什么？它的作用是什么？
 - 如何借助Bitmap实现40亿个QQ号的去重？
 - Bitmap位图的优点和局限性
- 三、方法2：采用布隆过滤器实现海量数据去重
 - 布隆过滤器是什么？它的实现原理是什么？
 - 哈希冲突的概念
 - 布隆过滤器的运行过程
 - 布隆过滤器示例
 - 布隆过滤器的应用场景
 - 如何实现布隆过滤器

- 基于Guava的布隆过滤器：
- 基于Redisson的布隆过滤器
- 基于Jedis的布隆过滤器

四、面临海量数据去重场景：如何在布隆过滤器和位图之间做出选择

五、总结

六、参考文献

问题场景分析

分析QQ号码的数量：

腾讯的QQ号采用4字节正整数表示，共32个bit位，因此QQ号码的数量约为43亿，理论值是 $2^{32}-1$ 个。

由于是无符号整数，所以其范围翻倍，因此总数约为43亿。

回顾问题：如何设计算法对40亿个QQ号码去重，使得相同的QQ号码仅保留一个，同时内存限制为1G？

问题本质：这是一个在内存受限条件下，处理海量数据去重的问题。

解决方案有很多种，但主流的方法主要有两种：

- **方法1：利用BitMap实现海量数据去重**
- **方法2：采用布隆过滤器实现海量数据去重**

方法1：利用BitMap实现海量数据去重

BitMap是什么？它的作用是什么？

位图（BitMap）的本质是一个bit数组，即每个位置都是一个bit，其取值可以是0或1。

位图 (BitMap) 的核心思想：利用一个bit来标记元素的存在与否。bit是计算机中最基本的单位，通常表示为0和1。

上图表示一个BitMap，我们可以利用它来记录三个数字：1，4，6，因为第1位、第4位、第6位这三个位置的值为1。

如果不使用位图，我们想要记录1，4，6这三个整数的话，该如何处理呢？

我们需要使用三个unsigned int来表示这三个整数。已知每个unsigned int占用4个字节，那么总共需要 $3 * 4 = 12$ 个字节。一个字节包含8比特，因此总共需要 $12 * 8 = 96$ 个比特。

结论：位图的最大优势在于节省空间。在这个例子中，位图比直接使用整数表示节省了12倍的空间。

如何借助BitMap实现40亿个QQ号的去重？

回到问题：如何在1G内存限制下对40亿个QQ号实现去重？

前面我们分析过：一个QQ号码可以用一个unsigned int表示。

40亿个QQ号码，就相当于40亿个unsigned int，而一个unsigned int占用4个字节。

假设我们直接用内存来存储这40亿个unsigned int，那么需要多少内存呢？

简单计算一下：

因此，如果直接将40亿个QQ号存储在内存中，需要15个G的空间，而我们只有1G的空间，显然无法直接存储。

那么，我们应该如何解决这个问题呢？

由于QQ号是数字，我们可以使用位图（BitMap）进行处理。

例如，要将一个QQ号"12345678"放入位图中，我们只需找到第12345678个位置，并将其设置为1即可。这样，我们就可以在有限的内存空间中实现去重。

这样一来，将40亿个数字都放入位图后，值为1的位置表示该数字存在，值不为1的位置表示不存在。

对于相同的QQ号，我们只需将其对应位置设置为1一次即可。因此，最后只需遍历所有值为1的数字，便可得到去重后的结果。

使用位图时，一个数字仅需占用1个比特。那么40亿个数字所需的空间为：

与之前需要的14.9G相比，位图方法大大节省了空间。大约节省了30倍的空间。**这使得我们能够在1G内存限制下实现40亿个QQ号码的去重。**

BitMap位图的优点和局限性

位图 (BitMap) 的基本思想是用一个比特位来标记元素，这是计算机中最小的单位，通常表示为0和1。

BitMap位图的主要优势在于其高效地进行集合运算。具体而言，当我们需要对一个集合进行多次交集、并集、差集等操作时，使用BitMap可以将这些操作的时间复杂度降低到 $O(1)$ 级别。相比之下，传统的集合实现需要 $O(n)$ 的时间复杂度，其中 n 是集合的大小。

此外，BitMap还可以节省存储空间。对于一个仅包含0和1的集合，我们可以使用一个比特位来表示一个元素是否在集合中，从而将集合的存储空间降低到原来的1/8左右。

因此，位图的最大优点在于节省空间。

位图适用于多种场景，尤其适合去重、排序等应用。著名的布隆过滤器便是基于位图实现的。

然而，位图也存在一定的局限性。首先，它只能表示0和1，无法存储其他数字。因此，BitMap适用于表示ture or false的场景。

其次，位图仅适用于值域较小的集合。如果值域过大，BitMap的存储空间也会过大，此时使用布隆过滤器可能更为合适。

最后，位图不支持删除操作，因为删除一个元素需要将对应的比特位设置为0，这可能会影响到其他元素的状态。

方法2：采用布隆过滤器实现海量数据去重

当值域过大时，BitMap的存储空间也会过大，这个时候可以考虑使用布隆过滤器来进一步压缩空间。

布隆过滤器是什么？它的实现原理是什么？

布隆过滤器是一种概率型数据结构，它能在内存占用较低的情况下有效地检测一个元素是否存在于集合中。不过，布隆过滤器可能会出现一定程度的误报率，即误判一个不存在的元素为存在。

布隆过滤器通过使用多个哈希函数将输入元素映射到位数组中。当查询一个元素是否存在时，布隆过滤器会检查该元素经过多个哈希函数映射后对应的位是否都为1。如果所有位都为1，则认为元素可能存在于集合中；如果有任何一个位为0，则元素肯定不存在于集合中。

布隆过滤器在某些情况下比位图更适合处理大值域的数据去重问题，因为它可以在保持较低内存占用的同时，接受一定程度的误报率。根据需要，可以调整布隆过滤器的参数以平衡内存占用和误报率。

本质上：布隆过滤器内部包含一个比特数组和多个哈希函数，每个哈希函数都会生成一个索引值。

布隆过滤器由两个部分组成：

- **一个比特数组，用于存储数据。**
- **多个哈希函数，用于计算键值的索引。**

如下图所示，其中有三个键值：“鬼仔”、“史上最棒”和“八股文”。这些键值经过多个哈希函数映射后，在比特数组中对应的位置被设置为1。通过这种方式，布隆过滤器可以在查询时快速判断一个元素是否可能存在于集合中。

问题：如何进行exist（key）这种存在性判断？

答案：在查询一个元素时，需要检查该元素经过多个哈希函数映射后对应的位是否都为1。如果所有位都为1，则认为元素可能存在于集合中；如果有任何一个位为0，则元素肯定不存在于集合中。

布隆过滤器可以准确的判断一个元素是否一定不存在。请注意，这里的判断是确定元素不存在。

为什么呢？这是因为哈希冲突的存在。

由于多个元素可能经过哈希函数映射到相同的位，因此当所有对应位都为1时，我们只能说元素可能存在于集合中，而不能确定它一定存在。然而，如果有任何一个位为0，那么我们可以确定该元素一定不存在于集合中，因为如果它存在，那么所有对应的位都应该为1。这就是布隆过滤器在存在性判断上的特点。

哈希冲突的概念

哈希冲突是指两个或多个不同的键值被映射到了相同的哈希值。

以下是一个例子：

例如，现在有一个新的键值"码农鬼仔"，我们需要判断它是否存在？

结果显示它存在。为什么呢？

- $\text{hash1}(\text{"码农鬼仔"})=1$ ，因为之前 $\text{hash1}(\text{"史上最棒"})$ 和 $\text{hash2}(\text{"鬼仔"})$ 都设置过这个位置为1，共设置了两次。
- $\text{hash2}(\text{"码农鬼仔"})=1$ ，因为之前 $\text{hash2}(\text{"史上最棒"})$ 已经设置过这个位置为1。
- $\text{hash3}(\text{"码农鬼仔"})=1$ ，因为之前 $\text{hash1}(\text{"鬼仔"})$ 已经设置过这个位置为1。

由于 $\text{hash1}(\text{"码农鬼仔"})=1$ 、 $\text{hash2}(\text{"码农鬼仔"})=1$ 、 $\text{hash3}(\text{"码农鬼仔"})=1$ ，所以 $\text{exist}(\text{"码农鬼仔"})$ 的结果为true。

然而，实际上键值"码农鬼仔"之前并未设置过，它是不存在的。

结论：由于哈希冲突，布隆过滤器无法判断一个元素一定存在。它只能判断元素可能存在或者不存在。

如何降低存在性误判的概率？

降低存在性误判的概率的主要方法是**降低哈希冲突的概率，以及引入更多的哈希算法**。通过这些措施，我们可以在一定程度上减少布隆过滤器的误判率。

布隆过滤器的运行过程

1. 初始化布隆过滤器

在初始化布隆过滤器时，需要指定集合的大小和误判率。

2. 将元素添加到布隆过滤器

要将一个元素添加到布隆过滤器中，首先需要通过多个哈希函数为该元素生成多个索引值，然后将这些索引值对应的位设置为1。如果这些索引值已经被设置为1，则无需再次设置。

3. 查询元素是否存在于布隆过滤器中

要查询一个元素是否存在于布隆过滤器中，需要将该元素通过多个哈希函数生成多个索引值，并判断这些索引值对应的位是否都被设置为1。如果这些位都被设置为1，则认为元素可能存在于集合中；否则，元素肯定不存在。

布隆过滤器的主要优点是可以快速判断一个元素是否属于某个集合，并且在空间和时间上实现较高效率。

然而，它也存在一些缺点，例如：

- 布隆过滤器在判断元素是否存在时具有一定的误判率。

- 布隆过滤器删除元素较为困难，因为删除一个元素需要将其对应的多个位设置为0，但这些位可能被其他元素共享。

布隆过滤器示例

1. 布隆过滤器初始状态

布隆过滤器使用一个二进制数组进行数据存储。一开始，二进制数组里没有值。

2. 存储操作

假设要存储一个数据"hello"。首先，对数据"hello"进行三次哈希运算，分别得到三个值（假设为1, 3, 5）。然后，在对应的二进制数组里，将下标为1, 3, 5的值置为1。

3. 查询操作

对于数据"hello"，需要对其进行三次哈希运算，分别得到三个值（假设为1, 3, 5）。在二进制数组里，将下标为1, 3, 5的值取出来，如果都为1，则表示该数据已经存在。

4. 删除操作

在使用布隆过滤器时，不建议进行删除操作。布隆过滤器里的部分比特位可能被复用。假设有两个键："hello"和"world"，如果`hash2("hello")`的结果为3，`hash2("world")`的结果也为3，那么如果删除了"hello"的`hash2("hello")`值，就意味着"world"的`hash2("world")`值也会被删除，从而造成数据的误删。

5. 误判率

假设保存了两个值："hello"和"world"。"hello"对应的三个哈希计算后的索引为1，3，5，"world"三个哈希对应的索引（也就是哈希计算后的值）也为1，3，5。那么`exist("world")=true`，这就是一种误判。因为实际上，"world"并没有被存储进过滤器中，但由于哈希冲突，查询结果却显示它存在。这就是布隆过滤器的误判现象。

布隆过滤器的应用场景

布隆过滤器因为其高效性，得到了广泛应用。以下是一些典型的应用场景：

- 1. 网页爬虫：**爬虫程序可以使用布隆过滤器来过滤已经爬取过的网页，避免重复爬取和资源浪费。
- 2. 缓存系统：**缓存系统可以使用布隆过滤器来判断一个查询是否可能存在于缓存中，从而减少查询缓存的次数，提高查询效率。布隆过滤器也经常用来解决缓存穿透问题。
- 3. 分布式系统：**在分布式系统中，可以使用布隆过滤器来判断一个元素是否存在于分布式缓存中，避免在所有节点上进行查询，降低网络负载。
- 4. 垃圾邮件过滤：**布隆过滤器可以用于判断一个邮件地址是否在垃圾邮件列表中，从而有效过滤垃圾邮件。
- 5. 黑名单过滤：**布隆过滤器可以用于判断一个IP地址或手机号码是否在黑名单中，从而阻止恶意请求。

如何实现布隆过滤器

在Java中，可以使用第三方库来实现布隆过滤器，常见的有Google Guava库、Apache Commons库以及Redis。

基于Guava的布隆过滤器：

Guava版本的布隆过滤器：值得一提的是，这个版本的布隆过滤器已在指导简历时被小伙伴们使用过。


Guava 20.0版本已经引入了布隆过滤器(BloomFilter)的实现。你可以按照以下步骤来使用Guava的布隆过滤器：

1. 引入Guava依赖：

2. 创建布隆过滤器：

```
int expectedInsertions = 1000000;  
double fpp = 0.01;  
BloomFilter<String> bloomFilter = BloomFilter.create(Funnels.stringFunnel(  

```



在这里，**expectedInsertions**表示预期插入的元素数量，**fpp**表示误判率(false positive probability)，

Funnels.stringFunnel(Charset.defaultCharset()) 表示元素类型为String。

3. 添加元素：

4. 判断元素是否存在：

需要注意的是，布隆过滤器在判断元素是否存在时具有一定的误判率。如果 **mightContain** 返回false，则可以确定该元素一定不存在；如果 **mightContain** 返回true，则该元素可能存在，需要进一步验证。

5. 序列化和反序列化：

在序列化和反序列化的过程中，需要让 **BloomFilter** 类实现 **Serializable** 接口。

基于Redisson的布隆过滤器

Redisson是一个基于Redis的Java客户端，提供了丰富的分布式对象和服务，其中包括布隆过滤器。Redisson的布隆过滤器实现了标准的布隆过滤器算法，并提供了一些额外的功能，如自动扩容和持久化等。

使用Redisson的布隆过滤器非常简单，只需创建一个 **RedissonClient** 对象，然后通过该对象获取一个 **RBloomFilter** 对象即可。

RBloomFilter提供了一系列的方法，包括添加元素、判断元素是否存在、清空过滤器等。

以下是一个简单的使用Redisson布隆过滤器的示例代码：

需要注意的是，Redisson的布隆过滤器不支持动态修改预计元素数量和误判率，因此在初始化时需要仔细考虑这两个参数的取值。

基于Jedis的布隆过滤器

如果没有使用Redisson，可以使用Jedis来实现布隆过滤器。参考代码如下：

由于布隆过滤器存在一定的误判率，因此不能完全替代传统的数据结构，应该根据具体应用场景进行选择。

面临海量数据去重场景：如何在布隆过滤器和位图之间做出选择

布隆过滤器和位图都属于常用的数据结构，然而，它们在应用领域和实现方法上有所不同。

布隆过滤器是一种基于概率的数据结构，主要用于判断某个元素是否属于一个集合。但它具有一定的误报率。因此，布隆过滤器并不适用于那些要求“绝对精确”的场景。在可以接受较低错误率的情况下，布隆过滤器能够以很小的错误率换取大量的存储空间节省。

位图则是一种简洁的数据结构，用于展示一个二进制序列。它通过一个位数组来表示一个集合，每个位表示一个元素是否属于该集合。要判断一个元素是否属于集合，只需查看相应的位是否为1或0即可。

相较而言，布隆过滤器在空间效率上更具优势，但会有一定的误报率；而位图的空间效率相对较低，但不存在误报。因此，在实际应用过程中，需要根据具体场景来选择适当的数据结构。

总结

处理海量数据去重问题，是一个非常常见的面试题目。

掌握这两大方案，如果能够熟练地回答并展示自己的专业素养，面试官大概率会被你的表现所震撼和吸引。

最终，让面试官爱到“**不能自己、口水直流**”。offer，也就来了！

在学习过程中，如果遇到任何问题，欢迎来联系大厂一线搬砖工程师鬼仔进行交流。

参考文献

1. <https://www.infoq.cn/article/1afyz3b6hnhprg12833>

- 2. <https://www.iamle.com/archives/2900.html>
- 3. <https://blog.51cto.com/lianghecai/4755693>
- 4. <https://qinyuanpei.github.io/posts/1333693167/>
- 5. <https://github.com/alibaba/canal/wiki/ClientAdapter>

收录于合集 #特殊题型面经 2

上一篇 · 面试真题：28道经典智力题最详汇总

喜欢此内容的人还喜欢

38岁国企员工，非常稳定的饭碗，突然通知部门解散
码农鬼仔



网友爆料：北京某公司裁员，从2000人裁到600
码农鬼仔



24届计算机听劝，银行是个大坑？
码农鬼仔

