04 | 状态检索：如何快速判断一个用户是否存在？

2020-03-30 陈东

讲述：陈东
时长 15:56 大小 14.61M

你好，我是陈东。

在实际工作中，我们经常需要判断一个对象是否存在。比如说，在注册新用户时，我们需要先快速判断这个用户 ID 是否被注册过；再比如说，在爬虫系统抓取网页之前，我们要判断一个 URL 是否已经被抓取过，从而避免无谓的、重复的抓取工作。

那么，对于这一类是否存在的状态检索需求，如果直接使用我们之前学习过的检索技术，有序数组、二叉检索树以及哈希表来实现的话，它们的检索性能如何呢？是否还有优化[illegible]呢？今天，我们就一起来讨论一下这些问题。

如何使用数组的随机访问特性提高查询效率？

以注册新用户时查询用户 ID 是否存在为例，我们可以直接使用有序数组、二叉检索树或者哈希表来存储所有的用户 ID。

我们知道，无论是有序数组还是二叉检索树，它们都是使用二分查找的思想从中间元素开始查起的。所以，在查询用户 ID 是否存在时，它们的平均检索时间代价都是 O(log n)，而哈希表的平均检索时间代价是 O(1)。因此，如果我们希望能快速查询出元素是否存在，那哈希表无疑是最合适的选择。不过，如果从工程实现的角度来看的话，哈希表的查询过程还是可以优化的。

比如说，如果我们要查询的对象 ID 本身是正整数类型，而且 ID 范围有上限的话。我们就可以申请一个足够大的数组，让数组的长度超过 ID 的上限。然后，把数组中所有位置的值都初始化为 0。对于存在的用户，我们**直接将用户 ID 的值作为数组下标**，将该位置的值从 0 设为 1 就可以了。

这种情况下，当我们查询一个用户 ID 是否存在时，会直接以该 ID 为数组下标去访问数组，如果该位置为 1，说明该 ID 存在；如果为 0，就说明该 ID 不存在。和哈希表的查找流程相比，这个流程就节省了计算哈希值得到数组下标的环节，并且直接利用数组随机访问的特性，在 O(1) 的时间内就能判断出元素是否存在，查询效率是最高的。

但是，直接使用 ID 作为数组下标会有一个问题：如果 ID 的范围比较广，比如说在 10 万之内，那我们就需要保证数组的长度大于 10 万。所以，这种方案的占用空间会很大。

而且，如果这个数组是一个 int 32 类型的整型数组，那么每个元素就会占据 4 个字节，用 4 个字节来存储 0 和 1 会是一个巨大的空间浪费。那我们该如何优化呢？你可以先想一想，然后我们一起来讨论。

如何使用位图来减少存储空间？

最直观的一个想法就是，使用最少字节的类型来定义数组。比如说，使用 1 个字节的 char 类型数组，或者使用 bool 类型的数组（在许多系统中，一个 bool 类型的元素也是 1 个字节）。它们和 4 个字节的 int 32 数组相比，空间使用效率提升了 4 倍，这已经算是不错的改善了。

但是，使用 char 类型的数组，依然是一个非常“浪费空间”的方案。因为表示 0 或者 1，理论上只需要一个 bit。所以，如果我们能以 bit 为单位来构建这个数组，那使用空间就是

int 32 数组的 1/32，从而大幅减少了存储使用的内存空间。这种以 bit 为单位构建数组的方案，就叫作 **Bitmap**，翻译为**位图**。

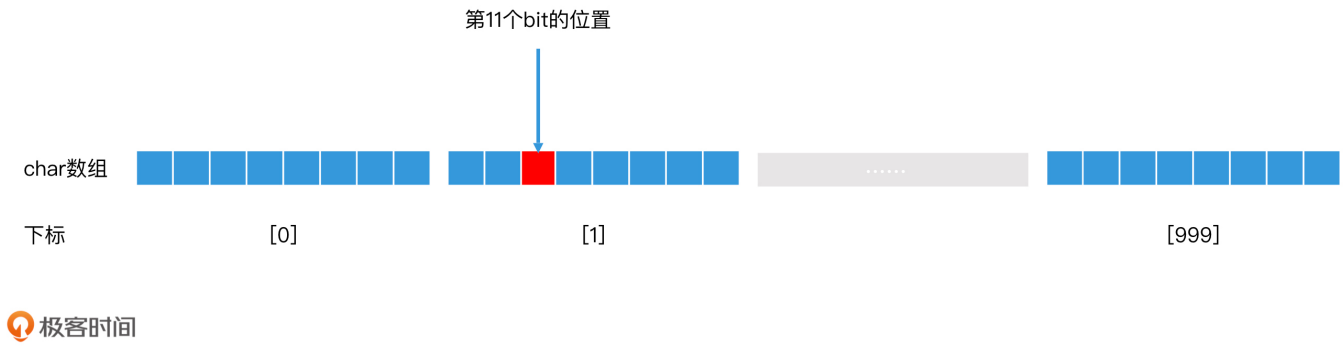
位图的优势非常明显，但许多系统中并没有以 bit 为单位的数据类型。因此，我们往往需要对其他类型的数组进行一些转换设计，使其能对相应的 bit 位的位置进行访问，从而实现位图。

我们以 char 类型的数组为例子。假设我们申请了一个 1000 个元素的 char 类型数组，每个 char 元素有 8 个 bit，如果一个 bit 表示一个用户，那么 1000 个元素的 char 类型数组就能表示 8*1000 = 8000 个用户。如果一个用户的 ID 是 11，那么位图中的第 11 个 bit 就表示这个用户是否存在的信息。

这种情况下，我们怎么才能快速访问到第 11 个 bit 呢?

首先，数组是以 char 类型的元素为一个单位的，因此，我们的第一步，就是要找到第 11 个 bit 在数组的第几个元素里。具体的计算过程：一个元素占 8 个 bit，我们用 11 除以 8，得到的结果是 1，余数是 3。这就代表着，第 11 个 bit 存在于第 2 个元素里，并且在第 2 个元素里的位置是第 3 个。

对于第 2 个元素的访问，我们直接使用数组下标[1]就可以在 O(1) 的时间内访问到。对于第 2 个元素中的第 3 个 bit 的访问，我们可以通过位运算，先构造一个二进制为 00100000 的字节（字节的第 3 位为 1），然后和第 2 个元素做 and 运算，就能得知该元素的第 3 位是 1 还是 0。这也是一个时间代价为 O(1) 的操作。这样一来，通过两次 O(1) 时间代价的查找，我们就可以知道第 11 个 bit 的值是 0 还是 1 了。



用户 ID 为 11 的位图定位

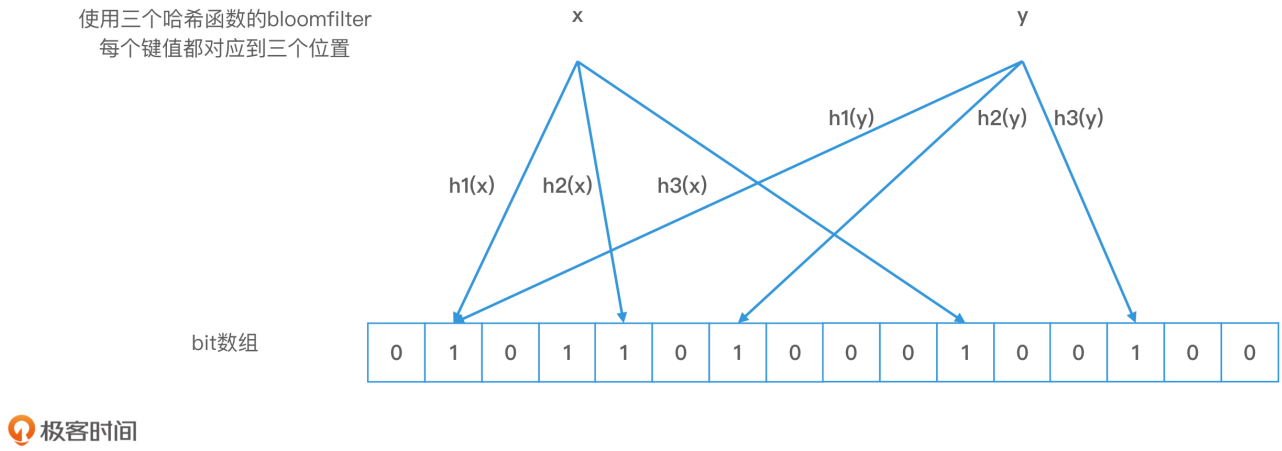
尽管位图相对于原始数组来说，在元素存储上已经有了很大的优化，但如果我们还想进一步优化存储空间，是否还有其他的优化方案呢？我们知道，**一个数组所占的空间其实就是“数组元素个数 * 每个元素大小”**。我们已经将每个元素大小压缩到了最小单位 1 个 bit，如果还要进行优化，那么自然会想到优化“数组元素个数”。

没错，限制数组的长度是一个可行的方案。不过前面我们也说了，数组长度必须大于 ID 的上限。因此，如果我们希望将数组长度压缩到一个足够小的值之内，我们就需要使用哈希函数将大于数组长度的用户 ID，转换为一个小于数组长度的数值作为下标。除此以外，使用哈希函数也带来了另一个优点，那就是我们不需要把用户 ID 限制为正整数了，它也可以是字符串。这样一来，压缩数组长度，并使用哈希函数，就是一个更加通用的解决方案。

但是我们也知道，数组压缩得越小，发生哈希冲突的可能性就会越大，如果两个元素 A 和 B 的哈希值冲突了，映射到了同一个位置。那么，如果我们查询 A 时，该位置的结果为 1，其实并不能说明元素 A 一定存在。因此，如何在数组压缩的情况下缓解哈希冲突，保证一定的查询正确率，是我们面临的主要问题。

在第 3 讲中，我们讲了哈希表解决哈希冲突的两种常用方法：开放寻址法和链表法。开放寻址法中有一个优化方案叫“双散列”，它的原理是使用多个哈希函数来解决冲突问题。我们能否借鉴这个思想，在位图的场景下使用多个哈希函数来降低冲突概率呢？没错，这其实就是布隆过滤器（Bloom Filter）的设计思想。

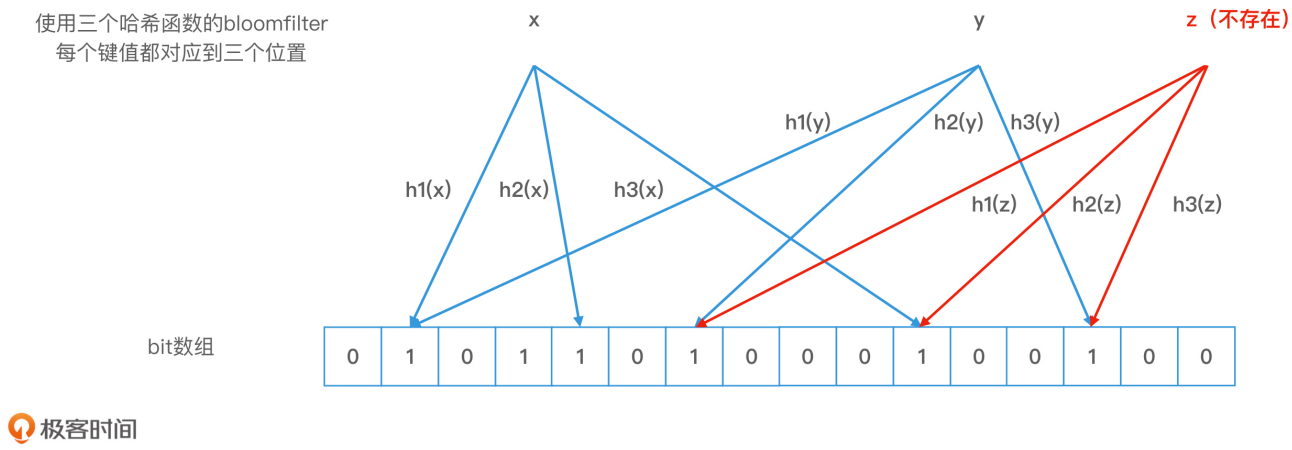
布隆过滤器最大的特点，就是对一个对象使用多个哈希函数。如果我们使用了 k 个哈希函数，就会得到 k 个哈希值，也就是 k 个下标，我们会把数组中对应下标位置的值都置为 1。布隆过滤器和位图最大的区别就在于，我们不再使用一位来表示一个对象，而是使用 k 位来表示一个对象。这样两个对象的 k 位都相同的概率就会大大降低，从而能够解决哈希冲突的问题了。



Bloom filter 示例

但是，布隆过滤器的查询有一个特点，就是即使任何两个元素的哈希值不冲突，而且我们查询对象的 k 个位置的值都是 1，查询结果为存在，这个结果也可能是错误的。这就叫作**布隆过滤器的错误率**。

我在下图给出了一个例子。我们可以看到，布隆过滤器中存储了 x 和 y 两个对象，它们对应的 bit 位被置为 1。当我们查询一个不存在的对象 z 时，如果 z 的 k 个哈希值的对应位置的值正好都是 1，z 就会被错误地认定为存在。而且，这个时候，z 和 x，以及 z 和 y，两两之间也并没有发生哈希冲突。



Bloom filter 错误率示例

那遇到“可能存在”这样的情况，我们该怎么办呢？不要忘了我们的使用场景：我们希望用更小的代价快速判断 ID 是否已经被注册了。在这个使用场景中，就算我们无法确认 ID 是否已经被注册了，让用户再换一个 ID 注册，这也不会损害新用户的体验。在系统不要求结

果 100% 准确的情况下，我们可以直接当作这个用户 ID 已经被注册了就可以了。这样，我们使用布隆过滤器就可以快速完成“是否存在”的检索。

除此之外，对于布隆过滤器而言，如果哈希函数的个数不合理，比如哈希函数特别多，布隆过滤器的错误率就会变大。因此，除了使用多个哈希函数避免哈希冲突以外，我们还要控制布隆过滤器中哈希函数的个数。有这样一个**计算最优哈希函数个数的数学公式: 哈希函数个数 k = (m/n) * ln(2)**。其中 m 为 bit 数组长度，n 为要存入的对象的个数。实际上，如果哈希函数个数为 1，且数组长度足够，布隆过滤器就可以退化成一个位图。所以，我们可以认为“**位图是只有一个特殊的哈希函数，且没有被压缩长度的布隆过滤器**”。

重点回顾

好了，状态检索的内容我们就讲到这里。我们一起来总结一下，这一讲你要掌握的重点内容。

今天，我们主要解决了快速判断一个对象是否存在的问题。相比于有序数组、二叉检索树和哈希表这三种方案，位图和布隆过滤器其实更适合解决这类状态检索的问题。这是因为，在不要求 100% 判断正确的情况下，使用位图和布隆过滤器可以达到 O(1) 时间代价的检索效率，同时空间使用率也非常高效。

虽然位图和布隆过滤器的原理和实现都非常简单，但是在许多复杂的大型系统中都可以见到它们的身影。

比如，存储系统中的数据是存储在磁盘中的，而磁盘中的检索效率非常低，因此，我们往往会先使用内存中的布隆过滤器来快速判断数据是否存在，不存在就直接返回，只有可能存在才会去磁盘检索，这样就避免了为无效数据读取磁盘的额外开销。

再比如，在搜索引擎中，我们也需要使用布隆过滤器快速判断网站是否已经被抓取过，如果一定不存在，我们就直接去抓取；如果可能存在，那我们可以根据需要，直接放弃抓取或者再次确认是否需要抓取。**你会发现，这种快速预判断的思想，也是提高应用整体检索性能的一种常见设计思路。**

课堂讨论

这节课的内容，你可以结合这道讨论题进一步加深理解：

如果位图中一个元素被删除了，我们可以将对应 bit 位置为 0。但如果布隆过滤器中一个元素被删除了，我们直接将对应的 k 个 bit 位置为 0，会产生什么样的问题呢？为什么？

欢迎在留言区畅所欲言，说出你的思考过程和最终答案。如果有收获，也欢迎把这篇文章分享给你的朋友。

上一篇　03 | 哈希检索：如何根据用户ID快速查询用户信息?

下一篇　05 | 倒排索引：如何从海量数据中查询同时带有“极”和“客”的唐诗?

精选留言 (7)

写留言

徐洲更
2020-03-30

因为同一个ID经过哈希函数会得到多个位置，不同的ID可能会有一些位置overlap。如果ID A和B刚好有一个位置重合，那么删除A的时候，如果直接将它对应的位置清零，就导致B也被认为是不存在。因此bloom filter删除操作很麻烦

展开

作者回复: 是的。bloom filter的删除会很麻烦。我们一般用在数据不删除的场景中（比如文中举的注册ID的场景）。
如果真要删除，可以使用上一课提到的re - hash的思路重新生成。（因为bloom filter本来就允许错误率，因此可以周期性重新生成）。
此外，还可以将bloomfilter改造成带引用计数的。

1 2

峰
2020-03-30

bitmap 是一个集合，每个元素在集合中有一个唯一不冲突的编号(用户自己保证，在数据库中这个编号可以是行号)，是双射关系。而布隆过滤器是一个不准确的集合，而且是一对多的关系，会发生冲突，也就是说布隆过滤器的为1的位可能代表多个元素，自然不能因为一个元素删除就把它干掉？，或者说他就不支持删除操作，感觉它要支持了，反而把它本身的优势给丢了。...
展开

作者回复: 你的思考很深入！
1.对于布隆过滤器的删除问题，的确无法直接删除。但也有带引用计数的布隆过滤器，存的不是0，1，而是一个计数。其实所有的设计都是trade off。应该视具体使用场景而定。比如一个带4个bit位计数器的布隆过滤器，相比于哈希表依然有优势。
2.布隆过滤器是否省空间，要看怎么比较。
布隆过滤器 vs 原始位图:
原始位图要存一个int 32的数，就要先准备好512m的空间的长数组。布隆过滤器不用这么长的数组，因此比原始位图省空间。
布隆过滤器 vs 哈希表:
假设布隆过滤器数组长度和哈希表一样。但是哈希表存的是一个int 32，而布隆过滤器存的是一个bit，因此比同样长度的哈希表省空间。
当然，如果哈希表也改为只存一个bit的数组，那么他们的大小是一样的。这时候就是你说的多个哈希函数的作用场景了。
其实，你会发现，只存一个bit的哈希表，其实也可以看做是只有一个哈希函数的布隆过滤器。很多时候，布隆过滤器，哈希表，还有位图，它们的边界是模糊的。我们最重要的是了解清楚他们的特点，知道在什么场景用哪种结构就好了。
3.roaring bitmap是一个优秀的设计。我在基础篇的加餐中会和大家分享。在这里，我也说一下它和布隆过滤器的差异:
布隆过滤器 vs roaring bitmap:
所有的设计都是trade off。roaring bitmap尽管压缩率很高，还支持精准查找，但是它放弃的是速度。高16位是采用二分查找，array container也是二分查找。因此，在这一点上布隆过滤器是有优势的。此外，它还不能保证压缩空间，它的空间会随着元素增多而变大，极端情况下恢复回bitmap。

而布隆过滤器保持了高效的查找能力和空间控制能力，但是放弃了精准查找能力，精准度会随着元素增多而下降。
因此，尽管都是对bitmap进行压缩，但是两者的设计思路不一样，使用场景也不同。在不要求精准，但是要求快速和省空间的场景下，布隆过滤器是不错的选择。

2

刘凯
2020-03-30

增加可以容忍误判，错误的判断用户存在，换个账号注册就行了，那么删除也会存在误判，可能将真正的用户没有删除掉，这可就不可取了，老师，我蒙对没，算法好头疼

作者回复: 你可以看我文中的例子想一想，x和y有共同的位，因此，如果删除x时，把x对应的3个bit位都改为0，就会影响y的查询。因此，对于布隆过滤器，不能直接删除。
一般来说，我们可以周期性重建布隆过滤器解决这个问题。
算法是不容易。但是不着急，慢慢来，一步一步扎扎实实学，你会收获得更多。

一步
2020-03-30

位图 一个位置就只有一个元素使用，布隆过滤器一个位置可能多个元素都会使用

作者回复: 是的。所以布隆过滤器不能直接删除。如果真的发生了删除，可以用类似re - hash的机制重新生成。
此外，一些场景会将布隆过滤器改造为带引用计数的结构。通过一个小数值的count进行计数。

千里之行
2020-03-30

会造成其他元素存在状态的错误判断，因为多个对象可能共用一个元素。但是极端情况下，甚至有可能一个对象对应的K个元素都与其他对象共用，这种情况下不知道该怎么办了，请老师帮忙解答一下，谢谢

作者回复: 的确，一般来说布隆过滤器是不能直接删除的。它适用于数据不删除的场景（比如文中举的注册id的场景）。如果真有删除需求，可以像前一课学过的re - hash一样，重新生成。
此外，删除频繁的场景下，还可以将布隆过滤器带上计数器。就是将一个bit改为4个bit，可以存

一个数。
尽管空间变大了，但是依然比哈希表存一个int 32的元素更省空间。


努力努力再努力Xmn
2020-03-30

对于布隆过滤器，删除元素时如果将对应的k个元素全部设置为0的话，会影响其他元素的判断，我想到一个方法，就是对于每一个数组中每一位，再设置一个标志count，用于记录出现1得次数，删除元素时将count减1，如果count为0的话，再将1设置为0。但是这样做的话，存储count不是又需要花费存储空间，这与布隆过滤器的设计目的不就冲突了吗？想知道布隆过滤器对于删除元素时如何实现的？希望老师解答。

展开

作者回复: 布隆过滤器的确是无法直接删除的。要删除的话，有两种思路，一种就是重新生成（和re - hash一个思路）。另一种就是你说的引用计数。
其实引用计数是可行的。它的确性能会比原始的布隆过滤器差，但依然好于哈希表。因为我们对于引用计数，完全可以用少数几个bit位来记录，比如说4个比特位就能记录到16。
这样的存一个4bit计数值的布隆过滤器，依然会比存int 32的哈希表更省空间。
所有的设计都是要根据具体场景灵活变通。因此，如果应用场景真的有频繁删除的需求，那么这样一种结构也是可以考虑的。


1


范闲
2020-03-30

1.bitmap和bloomfilter都是为了判断状态存在的。
2.bitmap只有一个位置用来判断状态
3.bloomfilter有多个位置用来判断状态
4.针对bloomfilter来说若果不所在一定不存在，存在不一定存在(因为hash冲突，可能是另外的元素状态)...

展开

作者回复: 总结得很好!
对于第五个问题，如何确定大小:
如果是原始位图，假设id是int 32，如果你不清楚数值分布范围，那么只能覆盖所有int 32的取值区间。这时候的位图大小是512m。
如果是布隆过滤器，你需要预估你的用户数量，
此外，还要设置一个你能接受的错误率p，使用这个公式:m = - n ln p / （ln 2）^2 ，可以算出来bit 位数组m的大小。