03 | 哈希检索：如何根据用户ID快速查询用...

在实际应用中，我们经常会面临需要根据键（Key）来查询数据的问题。比如说，给你一个用户 ID，要求你查出该用户的具体信息。这样的需求我们应该如何实现呢？你可能会想到，使用有序数组和二叉检索树都可以来实现。具体来说，我们可以将用户 ID 和用户信息作为一个整体的元素，然后以用户 ID 作为 Key 来排序，存入有序数组或者二叉检索树中，这样我们就能通过二分查找算法快速查询到用户信息了。

但是，**不管是有序数组、二叉检索树还是跳表，它们的检索效率都是 O(log n)**。那有没有更高效的检索方案呢？也就是说，**有没有能实现 O(1) 级别的查询方案呢？**今天，我们就一起来探讨一下这个问题。

使用 Hash 函数将 Key 转换为数组下标

数组具有随机访问的特性。那给定一个用户 ID，想要查询对应的用户信息，我们能否利用数组的随机访问特性来实现呢？

我们先来看一个例子。假设系统中的用户 ID 是从 1 开始的整数，并且随着注册数的增加而增加。如果系统中的用户数是有限的，不会大于 10 万。那么用户的 ID 范围就会被固定在 1 到 10 万之间。在数字范围有限的情况下，我们完全可以申请一个长度为 10 万的数组，然后将用户 ID 作为数组下标，从而实现 O(1) 级别的查询能力。

注意，刚才我们举的这个例子中有一个假设：用户的 ID 是一个数字，并且范围有限。符合这种假设的用户 ID 才能作为数组下标，使用数组的随机访问特性，达到 O(1) 时间代价的高效检索能力。那如果用户的 ID 数字范围很大，数组无法申请这么大的空间该怎么办呢？或者，用户的 ID 不是数字而是字符串，还能作为数组下标吗？

我们假设有一个系统使用字符串作为用户 ID。如果有一个用户的 ID 是“tom”，我们该怎么处理呢？我们能否将它转换为一个数字来表示呢？你可以先想一想解决方案，再和我继续往下分析。我们来考虑这样一种方案：字母表是有限的，只有 26 个，我们可以用字母在字母表中的位置顺序作为数值。于是，就有：“t” = 20，“o” = 15，“m” = 13。我们可以把这个 ID 看作是 26 进制的数字，那么对于“tom”这个字符串，把它转为对应的数值就是 20 * 26^2 + 15*26 + 13 =149123，这是一个小于 26^4 = 456976的数。

如果所有用户的 ID 都不超过 3 个字符，使用这个方法，我们用一个可以存储 456976个元素的数组就可以存储所有用户的信息了。**实际上，工业界有许多更复杂的将字符串转为整数的哈**

希算法，但核心思想都是利用每个字符的编码和位置信息进行计算，这里我就不展开了。

那如果内存空间有限，我们只能开辟一个存储 10000 个元素的数组该怎么办呢？这个时候，我们可以使用“tom”对应的数值 149123 除以数组长度 10000，得到余数 9123，用这个余数作为数组下标。

这种将对象转为有限范围的正整数的表示方法，就叫作 Hash，翻译过来叫散列，也可以直接音译为哈希。而我们常说的 Hash 函数就是指具体转换方法的函数。我们将对象进行 Hash，用得到的 Hash 值作为数组下标，将对应元素存在数组中。**这个数组就叫作哈希表。这样我们就可以利用数组的随机访问特性，达到 O(1) 级别的查询性能。**

说到这里，你可能会有疑问了，Hash 函数真的这么神奇吗？如果两个对象的哈希值是相同的怎么办？事实上，任何 Hash 函数都有可能造成对象不同，但 Hash 值相同的冲突。而且，数组空间是有限的，只要被 Hash 的元素个数大于数组上限，就一定会产生冲突。

对于哈希冲突这个问题，**我们有两类解决方案:** 一类是构造尽可能理想的 Hash 函数，使得 Hash 以后得到的数值尽可能平均分布，从而减少冲突发生的概率；另一类是在冲突发生以后，通过“提供冲突解决方案”来完成存储和查找。最常用的两种冲突解决方案是“开放寻址法”和“链表法”。下面，我就来介绍一下这两种方法，并且重点看看它们对检索效率的影响。

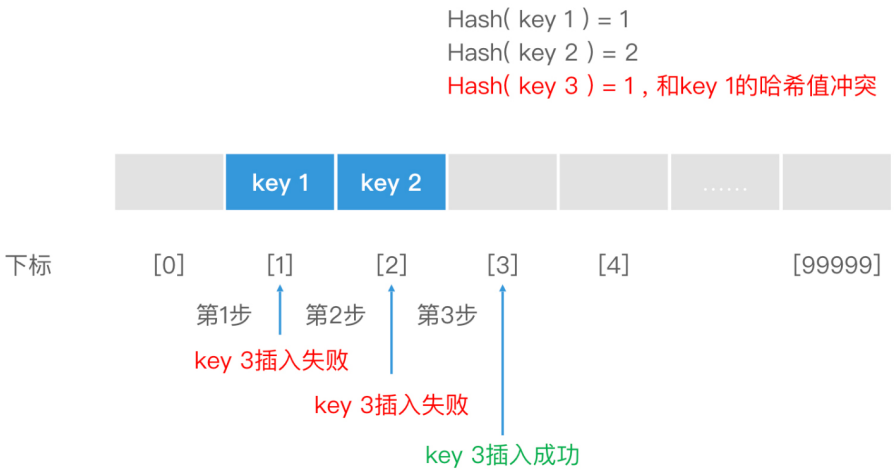
如何利用开放寻址法解决 Hash 冲突?

所谓“开放寻址法”，就是在冲突发生以后，最新的元素需要寻找新空闲的数组位置完成插入。那我们该如何寻找新空闲的位置呢？我们可以使用一种叫作“线性探查”（Linear Probing）的方案来进行查找。

“线性探查”的插入逻辑很简单：在当前位置发现有冲突以后，就顺序去查看数组的下一个位置，看看是否空闲。如果有空闲，就插入；如果不是空闲，再顺序去看下一个位置，直到找到空闲位置插入为止。

查询逻辑也和插入逻辑相似。我们先根据 Hash 值去查找相应数组下标的元素，如果该位置不为空，但是存储元素的 Key 和查询的 Key 不一致，那就顺序到数组下一个位置去检索，就这样依次比较 Key。**如果访问元素的 Key 和查询 Key 相等，我们就在哈希表中找到了对应元素**；如果遍历到

为了帮助你更好地理解，我们来看一个例子。假设一个哈希表中已经插入了两个 Key，key1 和 key2。其中 Hash(key1) = 1, Hash(key2) = 2。这时，如果我们要插入一个 Hash 值为 1 的 key3。根据线性探查的插入逻辑，通过 3 步，我们就可以将 key3 插入到哈希表下标为 3 的位置中。插入的过程如下：



在查找 key3 的时候，因为 Hash（key3）= 1，我们会从哈希表下标为 1 的位置开始顺序查找，经过 3 步找到 key3，查询结束。讲到这里，你可能已经发现了一个问题：当我们插入一个 Key 时，如果哈希表已经比较满了，这个 Key 就会沿着数组一直顺序遍历，直到遇到空位置才会成功插入。查询的时候也一样。

但是，顺序遍历的代价是 O(n)，这样的检索性能很差。更糟糕的是，如果我们在插入 key1 后，先插入 key3 再插入 key2，那 key3 就会抢占 key2 的位置，影响 key2 的插入和查询效率。**因此，“线性探查”会影响哈希表的整体性能，而不只是 Hash 值冲突的 Key。**

为了解决这个问题，**我们可以使用“二次探查”（Quadratic Probing）和“双散列”（Double Hash）这两个方法进行优化。**

下面，我来分别解释一下这两个方法的优化原理。二次探查就是将线性探查的步长从 i 改为 i^2：第一次探查，位置为 Hash(key) + 1^2；第二次探查，位置为 Hash(key) +2^2；第三次探查，位置为 Hash(key) + 3^2，依此类推。

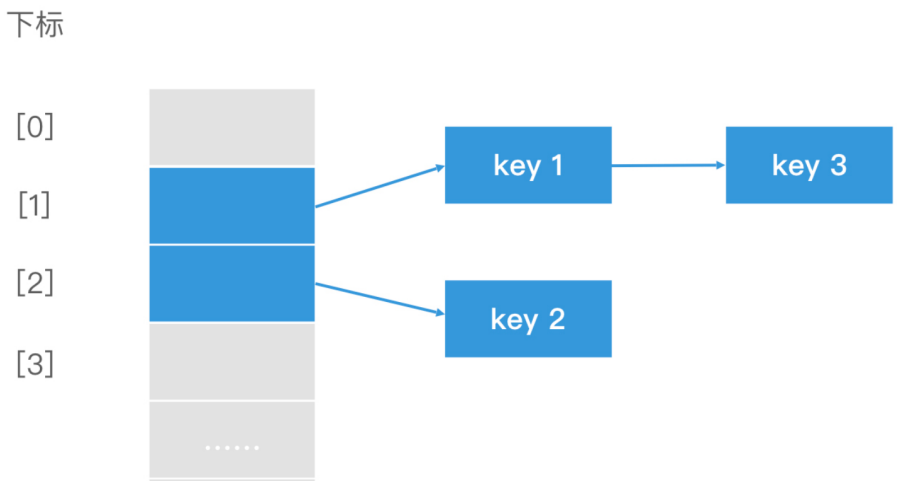
双散列就是使用多个 Hash 函数来求下标位置，当第一个 Hash 函数求出来的位置冲突时，启用第二个 Hash 函数，算出第二次探查的位置；如果还冲突，则启用第三个 Hash 函数，算出第三次探查的位置，依此类推。

无论是二次探查还是双散列，核心思路其实都是在发生冲突的情况下，将下个位置尽可能地岔开，让数据尽可能地随机分散存储，来降低对不相干 Key 的干扰，从而提高整体的检索效率。但是，对于开放寻址法来说，无论使用什么优化方案，随着插入的元素越多、哈希表越满，插入和检索的性能也就下降得越厉害。在极端情况下，当哈希表被写满的时候，为了保证能插入新元素，我们只能重新生成一个更大的哈希表，**将旧哈希表中的所有数据重新 Hash 一次写入新的哈希表，也就是 Re-Hash，这样会造成非常大的额外开销。因此，在数据动态变化的场景下，使用开放寻址法并不是最合适的方案。**

如何利用链表法解决 Hash 冲突?

相比开放寻址法，还有一种更常见的冲突解决方案，链表法。所谓“链表法”，就是在数组中不存储一个具体元素，而是存储一个链表头。

如果一个 Key 经过 Hash 函数计算，得到了对应的数组下标，那么我们就将它加入该位置所存的链表的尾部。这样做的好处是，如果 key3 和 key1 发生了冲突，那么 key3 会通过链表的方式链接在 key1 的后面，而不是去占据 key2 的位置。这样当 key2 插入时，就不会有冲突了。最终效果如下图。



讲到这里，你可能已经发现了，**其实链表法就是将我们前面讲过的数组和链表进行结合，既利用了数组的随机访问特性，又利用了链表的动态修改特性，同时提供了快速查询和动态修改的能力**。

想要查询时，我们会先根据查询 Key 的 Hash 值，去查找相应数组下标的链表。如果链表为空，则表示不存在该元素；如果链表不为空，则遍历链表，直到找到 Key 相等的对应元素为止。

但是，如果链表很长，遍历代价还是会很高。那我们有没有更好的检索方案呢？你可以回想一下，在上一讲中我们就是用**二叉检索树或跳表代替链表，来提高检索效率的**。

实际上，**在 JDK1.8 之后，Java 中 HashMap 的实现就是在链表到了一定的长度时，将它转为红黑树**；而当红黑树中的节点低于一定阈值时，就将它退化为链表。

第一个阶段，通过 Hash 函数将要查询的 Key 转为数组下标，去查询对应的位置。这个阶段的查询代价是 O(1) 级别。第二阶段，将数组下标所存的链表头或树根取出。如果是链表，就使用遍历的方式查找，这部分查询的时间代价是 O(n)。由于链表长度是有上限的，因此实际开销并不会很大，可以视为常数级别时间。如果是红黑树，则用二分查找的方式去查询，时间代价是 O(log n)。如果哈希表中冲突的元素不多，那么落入红黑树的数据规模也不会太大，红黑树中的检索代价也可以视为常数级别时间。

哈希表有什么缺点?

哈希表既有接近 O(1) 的检索效率，又能支持动态数据的场景，看起来非常好，那是不是在任何场景下，我们都可以用它来代替有序数组和二叉检索树呢？**答案是否定的。**

前面我们说了这么多哈希表的优点，下面我们就来讲讲它的缺点。首先，哈希表接近 O(1) 的检索效率是有前提条件的，就是哈希表要足够大和有足够的空闲位置，否则就会

非常容易发生冲突。我们一般用装载因子（load factor）来表示哈希表的填充率。装载因子 = 哈希表中元素个数 / 哈希表的长度。

如果频繁发生冲突，大部分的数据会被持续地添加到链表或二叉检索树中，检索也会发生在链表或者二叉检索树中，这样检索效率就会退化。因此，为了保证哈希表的检索效率，我们需要预估哈希表中的数据量，提前生成足够大的哈希表。按经验来说，我们一般要预留一半以上的空闲位置，哈希表才会有足够优秀的检索效率。

这就让哈希表和有序数组、二叉检索树相比，需要的存储空间更多了。另一方面，尽管哈希表使用 Hash 值直接进行下标访问，带来了 O(1) 级别的查询能力，**但是也失去了“有序存储”这个特点。**因此，**如果我们的查询场景需要遍历数据，或者需要进行范围查询，那么哈希表本身是没有什么加速办法的。**比如说，如果我们在一个很大的哈希表中存储了少数的几个元素，为了知道存储了哪些元素，我们只能从哈希表的第一个位置开始遍历，直到结尾，这样性能并不好。

回顾

你会看到，**哈希表的本质是一个数组**，它通过 Hash 函数将查询的 Key 转为数组下标，利用数组的随机访问特性，使得我们能在 O(1) 的时间代价内完成检索。

尽管哈希检索没有使用二分查找，但无论是设计理想的哈希函数，还是保证哈希表有足够的空闲位置，包括解决冲突的“二次探查”和“双散列”方案，本质上都是希望数据插入哈希表的时候，分布能均衡，这样检索才能更高效。从这个角度来看，其实哈希检索提高检索效率的原理，和二叉检索树需要平衡左右子树深度的原理是一样的，也就是说，**高效的检索需要均匀划分检索空间**。

另一方面，你会看到，复杂的数据结构和检索算法其实都是由最基础的数据结构和算法组成的。比如**说 JDK1.8 中哈希表的实现，就是使用了数组、链表、红黑树这三种数据结构和相应的检索算法。因此，对于这些基础的数据结构，我们需要深刻地理解它们的检索原理和适用场景，这也为我们未来学习更复杂的系统打下了扎实的基础。**

讨论

假设一个哈希表是使用开放寻址法实现的，如果我们需要删除其中一个元素，可以直接删除吗？为什么呢？如果这个哈希表是使用链表法实现的会有不同吗？

链表法可以直接删除，开放寻址法不行。

开放寻址法在 hash 冲突后会继续往后面看，如果为空，就放到后面，这样会存在连续的几个值的 hash 值都相同的情况，但如果想删除的数据在中间的话，就会影响对后面数据的查询了

可以增加一个删除标识，这种添加删除标识的在数据库中也常用