46 | 哈希表与哈希算法：字符串的MD5值是通过哈希算法得到的?

2023-05-29 王健伟 来自北京

《快速上手C++数据结构与算法》

你好，我是王健伟。

上节课我们在讲解哈希函数的设计方法中提到了“哈希冲突”这个词。

通常来说，在传递给哈希函数不同的参数但返回的哈希值是相同的整数时，就产生了哈希冲突。

这节课，我们就谈一谈如何解决哈希冲突。

如何解决哈希冲突?

哈希冲突无法完全避免，尤其是数组大小是有限的，就更容易发生哈希冲突。所以问题的重点在于一旦遇到哈希冲突该如何解决。

哈希冲突解决方法比较多，比如链表法、开放地址法、公共溢出区法等。我们一个一个来说。

链表法 / 拉链法 / 链地址法（Chaining）

这是一种很常用的哈希冲突解决方法，比较简单。在哈希表所代表的数组中，每个数组元素称为“桶”（bucket）或“槽”（slot）或“篮子（basket）”。每个桶 / 槽 / 篮子会对应一个链表，桶代表这个链表的链表头。将这些哈希值相同的元素放到相同桶对应的链表中，如图 1 所示：

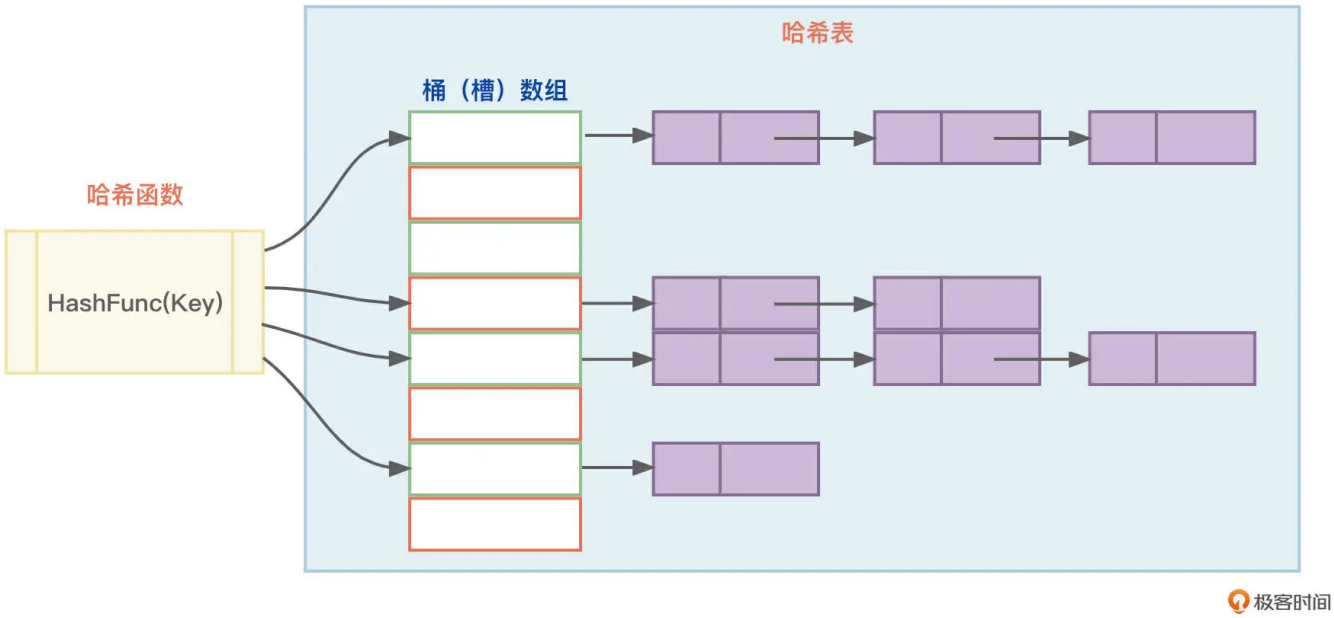

图1 链表法解决哈希冲突（顺序表+链表）

当向哈希表中插入数据的时候，通过哈希函数计算出对应的桶位，将数据插入到桶对应的链表中即可。此时插入的时间复杂度为 O(1)。当在哈希表中查找和删除数据时，可以通过哈希函数计算出桶位，然后就可以通过遍历对应的链表进行查找和删除工作，查找和删除的时间复杂度取决于链表中元素个数，如果链表中元素个数（链表长度）为 k，则时间复杂度为 O(k)。

这里提示一下，C++ 标准库中的 unordered_set 等无序容器可以用诸如 bucket_count() 成员函数获取桶的数量，利用 bucket_size() 成员函数获取每个桶中元素的个数。

话说回来，链表法在遇到哈希冲突时分配堆空间存储数据，对内存的利用率更高。但如果哈希表中的每个对象都很小，那么与对象大小比较而言，每个对象中额外多出的这根指针就显得会额外消耗很多内存，比如对象只占 4 字节，这根指针也占了 4 字节，此时这根指针消耗的内

存就显得特别突出，而如果对象很大占 400 字节，这根指针只占 4 字节，此时这根指针消耗的内存就可以忽略不计了。

当然，也有人对哈希冲突时采用的链表法进行改进。比如不用链表而改用跳表、红黑树，或者刚开始用链表，当数据量达到一定规模时改用跳表或者红黑树等。此时就算是某个桶中哈希冲突的表项非常多，也能有效提升该桶中数据的查找效率。

链表法灵活性比较强，比较适合存储大数据对象、数据量比较大、经常进行插入和删除数据的情形。当然，这种方法也带来了查找数据时需要遍历链表导致的性能损耗。

开放地址法 / 开放寻址法 / 开放定址法（Open Addressing）

我们再说另一种出现了哈希冲突的解决思路：重新寻找一个空闲位置并将数据插入。

那么如何重新寻找一个空闲位置呢？一般有这么几种主要的方法：线性探测法（Linear Probing）、二次探测法（Quadratic probing）；双哈希法（Double hashing）、伪随机序列法（Pseudorandom sequence）等。

线性探测法（Linear Probing）

当向哈希表中插入数据时，若某个数据经过哈希函数计算后得到的存储位置已被占用，则就从当前位置开始依次向后查找空闲位置，一直到找到为止。如果到了哈希表的尾部都没找到空闲位置，则回到哈希表的头部继续寻找空闲位置进行插入。只要哈希表没满，就肯定能找到空闲位置。

显然，**这种方法在查询数据时会带来麻烦，需要检测查询到的数据是否是真正要找的数据。**如果不是，还要依次向后查找，如果查找到了数组中的空闲位置，但还是没找到，就说明要查询的数据不在哈希表中。

这种查询数据时遇到空闲位置就认定要查询的数据不在哈希表中的判断方式，对删除操作会产生影响。当找到要删除的数据时，不能把该数据直接从所在位置删除（否则会导致查询数据的算法出现问题），而是将该位置设置一个“删除”标记，同样，当查询数据时，遇到“删除”标记的位置并不理会而是应该继续向下查找。

线性探测法有一个很大的问题，就是随着哈希表中数据的增多，哈希冲突的机率会越来越大，这样就会导致空闲位置越来越少，插入、查询、删除等所需要耗费的时间就会越来越多，甚至有可能导致遍历整个哈希表，所以最坏情况下插入、查询、删除的时间复杂度会达到 O(n)。

二次 / 平方探测法（Quadratic probing）

二次探测法与线性探测法类似，线性探测法每次向后走一步探测数组中的下个位置（步长为 1），而二次探测法每次走的步长则变成了原来的二次方，这样就防止了哈希冲突的关键字数据聚集在某一块区域。

也就是说，线性探测法的探测的下标序列是 HashFunc(Key)+0、HashFunc(Key)+1、HashFunc(Key)+2……，那么二次探测法探测的下标序列就是 HashFunc(Key)+0、HashFunc(Key)+1^2、HashFunc(Key)-1^2、HashFunc(Key)+2^2、HashFunc(Key)-2^2……，看起来就是每次向后 / 前跳跃的距离会越来越大。

双哈希法 / 再哈希法 / 再散列法（Double hashing）

双哈希法指的是不仅仅使用一个哈希函数，而是使用多个哈希函数 HashFunc1(Key)、HashFunc2(Key)、HashFunc3(Key)……，可以把前面讲过的哈希函数的设计方法都用上来设计哈希函数作为多个备用哈希函数之一。

先用第一个哈希函数，如果计算得到的存储位置已经被占用了，再用第二个哈希函数，以此类推，直到找到空闲的存储位置。当然还穿插着一些冲突计数也会参与到哈希值的计算中，具体不详细探讨。

伪随机序列法（Pseudorandom sequence）

当发生哈希冲突时，哈希值增量为伪随机数序列。有兴趣可以通过搜索引擎了解。

开放地址法的所有数据都保存在数组中，这对于数据查询来讲速度就会非常快。但删除数据时要额外设置删除标记，哈希冲突时解决代价比链表法更高，而且哈希冲突时会降低数据的查询效率。所以**开放地址法比较适合数据量比较小的情形。**

公共溢出区法（Public Overflow Area）

如果冲突的数据并不多，那么公共溢出区这个方法的查找性能还是比较高的。它是怎么实现的呢？设置一个溢出表，凡是冲突的数据全部放入溢出表中。在查找数据时，先到基础的哈希表中寻找，如果没有找到，再到溢出表中去顺序查找。

总结一下，其实，哈希函数对于哈希算法的要求也是灵活的。比如对于哈希冲突，只要不是太过严重，一般都可以通过链表法或者开放寻址法得到较好的解决。另外，对于某些场合，哈希函数中采用的哈希算法得到的哈希值是否是单向的，也许并不是那么重要，最值得关心的其实有两点，一是哈希算法所得到的哈希值是否会均匀分布，以保证数据能均匀地分布在桶中。二是哈希算法要比较简单，这样执行效率才会高。

哈希函数算法效率分析

一般来说，哈希表的查询效率可以达到 O(1)。但这个说法并不准确，因为查询效率跟哈希函数的设计（哈希值分布均匀性，处理冲突方法的效率）、空闲桶位的多少、哈希冲突概率等都有关系。在最坏情况下，如果哈希表扩容、搬移数据时哈希值要重新计算等，时间复杂度就会变为 O(n)。

通常，如果数据不需要频繁地插入和删除，那么根据数据的特点、分布规律等来设计哈希函数是相对比较容易的。但如果数据变动得非常频繁而且数据的个数也不容易预估，那就可能导致哈希表中空闲的位置越来越少。

当哈希表中空闲位置很少的时候，哈希冲突的概率就会越来越大。所以应该尽可能保证哈希表中有一定比例的空闲槽位，避免哈希表性能的下降。比如当哈希表中数据存储量超过 70% 时（也有的是到了 100%），就要考虑扩容——建立一个新的尺寸更大的哈希表，比如是原哈希表尺寸的两倍甚至更多倍。然后把之前的数据通过哈希函数搬到新的哈希表中。这是因为哈希表大小改变了，数据的存储位置也就改变了，所以需要通过哈希函数重新计算数据的存储位置。

这里有一个**装载因子**/ 装填因子（load factor）的概念，装载因子用来表示哈希表中空闲位置的多少。装载因子的计算公式是：

哈希表的装载因子 = 哈希表中元素个数 / 哈希表的长度

装载因子越大，说明空闲位置越少，哈希冲突的机率就会越大，哈希表的性能就会下降。所以哈希表的性能和装载因子紧密相关。

最后来看一看扩容这件事：如果原哈希表本来就非常大，比如已经占了 500M 内存，那么当有数据插入时，一次性执行完扩容动作可能会非常耗时，此时可以考虑将扩容动作穿插在插入操作过程中进行。

比如扩容时先只执行分配扩容后的哈希表空间的动作，然后每次插入新数据时，先将新数据插入到新哈希表中，并且顺便从老哈希表中取出一个数据放入到新哈希表中。这样的话，只要插入数据的动作次数足够多，老的哈希表中的数据最终就会全部搬移到新哈希表中了。当然，如果这中间要查询数据，则可以考虑先从新哈希表中查询，若未查询到再到老的哈希表中查询。这样就做到了将扩容操作耗费的时间均摊到多次插入操作中，避免扩容动作耗费过多时间，此时插入数据的时间复杂度就会稳定在 O(1)。

如果比较频繁地删除哈希表中的数据，哈希表中的数据会变得越来越少，如果对内存消耗非常敏感，也可以通过缩容来节省一定的内存空间。

哈希表实现源码

哈希表的实现代码并不复杂，这里将采用哈希技术实现一个简单的哈希表，其中哈希函数的实现将使用除留余数法，哈希冲突的解决方法将使用链表法，代码中提供的功能包括向哈希表中增加元素、查询某元素是否在哈希表中以及从哈希表中删除元素。实现源码请参考 课件。

程序的执行结果如下：

```
--------begin--------------
[桶下标:0]= 74 37 0
[桶下标:1]= 75 38 1
[桶下标:2]= 76 39 2
[桶下标:3]= 77 40 3
[桶下标:4]= 78 41 4
.....
[桶下标:32]= 69 32
[桶下标:33]= 70 33
[桶下标:34]= 71 34
[桶下标:35]= 72 35
[桶下标:36]= 73 36
--------end--------------
在下标为[24]的桶中找到了元素24
没有找到元素24
--------begin--------------
[桶下标:0]= 0
[桶下标:1]= 1
[桶下标:2]= 2
[桶下标:3]= 3
[桶下标:4]= 4
.....
[桶下标:34]= nullptr
[桶下标:35]= nullptr
[桶下标:36]= nullptr
--------end--------------
```

哈希算法的主要应用领域

哈希算法除了在哈希函数中使用来计算哈希值外，还有很多应用领域。很多知名且成熟的哈希算法已经存在多年，比如 MD5、SHA 等，通常在开发中直接拿来使用即可。

我这里使用一个数据库客户端工具 SQLyog 来针对一些输入数据（字符串）比如“苹果”、“猕猴桃”等做一下 MD5 哈希运算：

```
SELECT MD5('苹果') = e6803e21b9c61f9ab3d04088638cecd2
SELECT MD5('猕猴桃') = 880b9460958a6b0f08564b7d97205d3a
SELECT MD5('猕猴桃1')= a5bbf778aba0cbd5fd549c5f7f8d5d55
SELECT MD5('0')=cfcd208495d565ef66e7dff9f98764da
```

可以看到，因为 MD5 哈希运算得到的值是 128 位的长度，为了方便观察，上面的结果中是把它转换成了 16 进制编码。因为 1 位 16 进制数字代表 4 位二进制数字，所以 128 位二进制数字需要用 128/4=32 位 16 进制数字表示。这个总结一下 MD5 哈希算法的特点：

MD5 哈希运算得到的是一个字符串，无论原始数据多长以及是什么内容，输出的哈希值长度都相同。长度相同代表所能表示的哈希值就是有限的（2^{128}），所以**哈希冲突肯定是存在的（冲突概率：小于 1/(2^{128})），只是算法得到的哈希值越长，哈希值冲突的概率会越低。**

哪怕是对原始数据微小的修改，得到的哈希值也会很不相同。

很难通过哈希值“880b9460958a6b0f08564b7d97205d3a”反推出原始数据“猕猴桃”。

如果你用专门的 MD5 工具计算一个大小达到 1G 的文件的 MD5 值，也可以发现，不到几秒钟就能得到计算结果，所花费的时间还包括读磁盘的时间。这说明 MD5 算法的执行效率也是很高的。

哈希算法的应用范围非常广泛，在许多领域中都能看到它的身影，我举几个例子。

安全加密领域

MD5 算法，全称是 MD5 信息摘要算法（MD5 Message-Digest Algorithm）。SHA 算法，全称是安全散列算法。

比如将用户的密码采用 MD5 加密存储在数据库中，这样即便这个数据库被恶意者拿到，恶意者也无法看到用户密码加密之前（明文）的内容。在需要用户输入密码确认身份的场合，用户输入明文密码，系统会把这个明文密码进行 MD5 加密后和存储在数据库中的 MD5 密文相比较，两者内容相同就表示用户输入的密码是对的。

当然，没有绝对安全的加密算法，加密算法越复杂，计算哈希值需要耗费的时间也会越长。使用这些现成的算法时也要做出权衡。

通常来说，这些加密算法的破解采用的是穷举法（字典攻击），此法需要耗费大量的时间和资源，从投入产出比来讲并不太划算，同时你也要注意不要把自己的密码设置得太过简单比如 000000、123456 等等，否则也是很容易被字典攻击猜中的。安全没有绝对的，只能尽量增大破坏者的破坏成本。

唯一标识

一个图片数据库中记录着海量的图片数据（比如文件名、文件大小、文件存放路径等）。当给定一个图片时，如何确定这个图片是否已经存在于图片数据库中呢?

一个图片文件的大小不一，小则几 K 大小，大则几十上百 M 大小，不可能拿图片文件中的原始二进制信息逐一和图片数据库中所记录的海量图片数据对比，这样做耗费的时间难以估量，必须选择一种更快速的对比方法。

所以，可以为每张图片生成一个摘要信息作为图片的唯一标识保存在图片数据库中。这个摘要信息可以这样生成：比如从图片文件的开头、中间、末尾各获取一定字节数据，将这些字节的数据通过 MD5 加密算法得到 32 位 16 进制数，将这 32 位 16 进制数看成是一个字符串（哈希字符串）作为图片的摘要信息。

当给定一个图片需要确定该图片是否已经存在于图片数据库中时，可以先计算出该图片的摘要信息并到图片数据库中查询看是否存在该摘要信息。如果存在，则从图片数据库中取得文件大小和文件路径信息，将图片数据库中的文件内容与需要比对的文件做内容上的完全对比从而确定某个图片文件是否已经存在于图片数据库中。

数据校验

这里的数据校验的主要目的是防止文件内容被篡改的检验。比如下载一个大小为 2G 的文件，但因为网络传输并不是绝对安全的，所以无法保证下载下来的文件是没有被恶意修改过的。所以可以通过哈希算法对整个下载下来的文件求得 MD5 值，与该文件原本的 MD5 值进行比

较。一般来讲，一个被下载的文件的原本 MD5 值是多少，官方都会提供，这两个值相同，就说明下载下来的文件内容没有被篡改。

利用针对每个文件求得其 MD5 值的方法，也可以确定两个文件的内容是否完全相同。

此外，在分布式系统中，哈希算法也有许多应用，比如负载均衡、数据分片、分布式存储（涉及一致性哈希算法概念）等非常多，有兴趣可以借助搜索引擎详细了解。

小结

这节课，我带你了解了三种解决哈希冲突的方法，分别是链表法、开放地址法、公共溢出区法。

我们还对哈希函数算法的效率进行了分析并提出了哈希冲突概率增大时哈希表的扩容以及为了节省内存进行的哈希表缩容问题。

结合着这两节课的内容，你就可以实现一个简单的哈希表代码了。我给出的示例中，哈希函数采用除留余数法实现，哈希冲突的解决方法采用了链表法。代码中实现了向哈希表中增加元素、查询某元素是否在哈希表中以及从哈希表中删除元素的功能，你都可以参考。

在本节的最后，我们探讨了哈希算法的应用领域问题，哈希算法的应用领域很广，除了在哈希函数中使用来计算哈希值外，还可以应用在安全加密领域、唯一标识领域以及数据校验领域等。

思考题

在这节课的最后，我也给你留了两道复习思考题。

1. 如何设计一个高效的哈希函数？
2. 哈希表的性能问题是如何产生的，如何解决？

欢迎你在留言区和我互动。如果觉得有所收获，也可以把课程分享给更多的朋友一起学习。我们下节课见！

精选留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。