

讲堂 > 数据结构与算法之美 > 文章详情

19 | 散列表（中）：如何打造一个工业级水平的散列表？

2018-11-02 王争



19 | 散列表（中）：如何打造一个工业级水平的散列表？

朗读人：修阳 17'57" | 8.23M

通过上一节的学习，我们知道，散列表的查询效率并不能笼统地说成是 $O(1)$ 。它跟散列函数、装载因子、散列冲突等都有关系。如果散列函数设计得不好，或者装载因子过高，都可能导致散列冲突发生的概率升高，查询效率下降。

在极端情况下，有些恶意的攻击者，还有可能通过精心构造的数据，使得所有的数据经过散列函数之后，都散列到同一个槽里。如果我们使用的是基于链表的冲突解决方法，那这个时候，散列表就会退化为链表，查询的时间复杂度就从 $O(1)$ 急剧退化为 $O(n)$ 。

如果散列表中有 10 万个数据，退化后的散列表查询的效率就下降了 10 万倍。更直接点说，如果之前运行 100 次查询只需要 0.1 秒，那现在就需要 1 万秒。这样就有可能因为查询操作消耗大量 CPU 或者线程资源，导致系统无法响应其他请求，从而达到拒绝服务攻击（DoS）的目的。这也就是散列表碰撞攻击的基本原理。

今天，我们就来学习一下，**如何设计一个可以应对各种异常情况的工业级散列表，来避免在散列冲突的情况下，散列表性能的急剧下降，并且能抵抗散列碰撞攻击？**

如何设计散列函数？

散列函数设计的好坏，决定了散列表冲突的概率大小，也直接决定了散列表的性能。那什么才是好的散列函数呢？


首先，散列函数的设计不能太复杂。过于复杂的散列函数，势必会消耗很多计算时间，也就间接的影响到散列表的性能。其次，散列函数生成的值要尽可能随机并且均匀分布，这样才能避免或者最小化散列冲突，而且即便出现冲突，散列到每个槽里的数据也会比较平均，不会出现某个槽内数据特别多的情况。

实际工作中，我们还需要综合考虑各种因素。这些因素有关键字的长度、特点、分布、还有散列表的大小等。散列函数各式各样，我举几个常用的、简单的散列函数的设计方法，让你有个直观的感受。

第一个例子就是我们上一节的学生运动会的例子，我们通过分析参赛编号的特征，把编号中的后两位作为散列值。我们还可以用类似的散列函数处理手机号码，因为手机号码前几位重复的可能性很大，但是后面几位就比较随机，我们可以取手机号的后四位作为散列值。这种散列函数的设计方法，我们一般叫作“数据分析法”。

第二个例子就是上一节的开篇思考题，如何实现 Word 拼写检查功能。这里面的散列函数，我们就可以这样设计：将单词中每个字母的[ASCII 码值](#)“进位”相加，然后再跟散列表的大小求余、取模，作为散列值。比如，英文单词 nice，我们转化出来的散列值就是下面这样：

```
1 hash("nice")=((("n" - "a") * 26*26*26 + ("i" - "a")*26*26 + ("c" - "a")*26 + ("e" - "a")) / 78978)
```



实际上，散列函数的设计方法还有很多，比如直接寻址法、平方取中法、折叠法、随机数法等，这些你只要了解就行了，不需要全都掌握。

装载因子过大了怎么办？

我们上一节讲到散列表的装载因子的时候说过，装载因子越大，说明散列表中的元素越多，空闲位置越少，散列冲突的概率就越大。不仅插入数据的过程要多次寻址或者拉很长的链，查找的过程也会因此变得很慢。

对于没有频繁插入和删除的静态数据集合来说，我们很容易根据数据的特点、分布等，设计出完美的、极少冲突的散列函数，因为毕竟之前数据都是已知的。

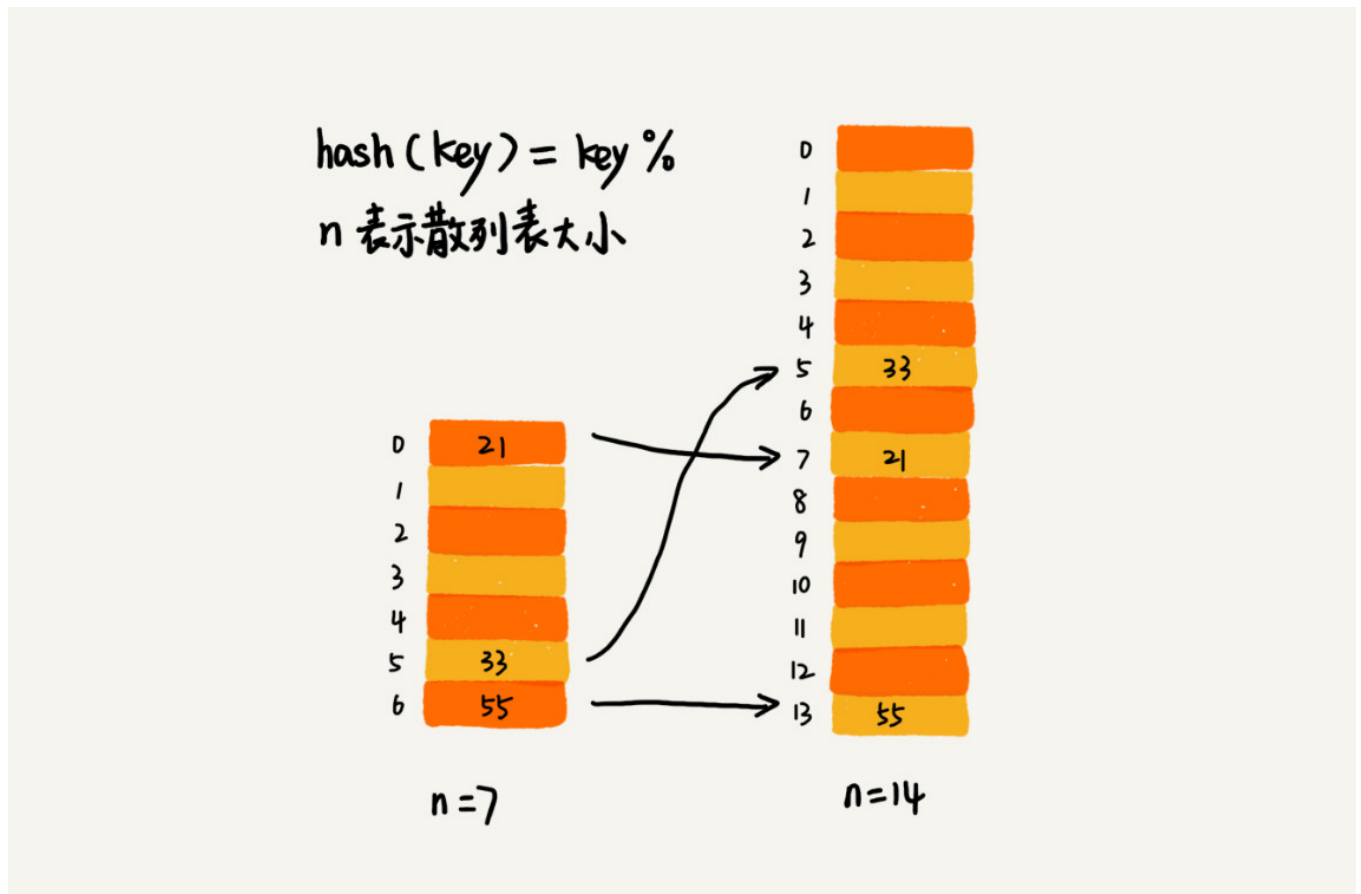
对于动态散列表来说，数据集合是频繁变动的，我们事先无法预估将要加入的数据个数，所以我们也无法事先申请一个足够大的散列表。随着数据慢慢加入，装载因子就会慢慢变大。当装载因子大到一定程度之后，散列冲突就会变得不可接受。这个时候，我们该如何处理呢？

还记得我们前面多次讲的“动态扩容”吗？你可以回想一下，我们是如何做数组、栈、队列的动态扩容的。

针对散列表，当装载因子过大时，我们也可以进行动态扩容，重新申请一个更大的散列表，将数据搬移到这个新散列表中。假设每次扩容我们都申请一个原来散列表大小两倍的空间。如果原来散列表的装载因子是 0.8，那经过扩容之后，新散列表的装载因子就下降为原来的一半，变成了 0.4。

针对数组的扩容，数据搬移操作比较简单。但是，针对散列表的扩容，数据搬移操作要复杂很多。因为散列表的大小变了，数据的存储位置也变了，所以我们需要通过散列函数重新计算每个数据的存储位置。

你可以看我图里这个例子。在原来的散列表中，21 这个元素原来存储在下标为 0 的位置，搬移到新的散列表中，存储在下标为 7 的位置。



对于支持动态扩容的散列表，插入操作的时间复杂度是多少呢？前面章节我已经多次分析过支持动态扩容的数组、栈等数据结构的时间复杂度了。所以，这里我就不啰嗦了，你要是还不清楚的话，可以回去复习一下。

插入一个数据，最好情况下，不需要扩容，最好时间复杂度是 $O(1)$ 。最坏情况下，散列表装载因子过高，启动扩容，我们需要重新申请内存空间，重新计算哈希位置，并且搬移数据，所以时间复杂度是 $O(n)$ 。用摊还分析法，均摊情况下，时间复杂度接近最好情况，就是 $O(1)$ 。

实际上，对于动态散列表，随着数据的删除，散列表中的数据会越来越少，空闲空间会越来越多。如果我们对空间消耗非常敏感，我们可以在装载因子小于某个值之后，启动动态扩容。当然，如果我们更加在意执行效率，能够容忍多消耗一点内存空间，那就可以不用费劲来扩容了。

我们前面讲到，当散列表的装载因子超过某个阈值时，就需要进行扩容。装载因子阈值需要选择得当。如果太大，会导致冲突过多；如果太小，会导致内存浪费严重。

装载因子阈值的设置要权衡时间、空间复杂度。如果内存空间不紧张，对执行效率要求很高，可以降低负载因子的阈值；相反，如果内存空间紧张，对执行效率要求又不高，可以增加负载因子的值，甚至可以大于 1。

如何避免低效地扩容？

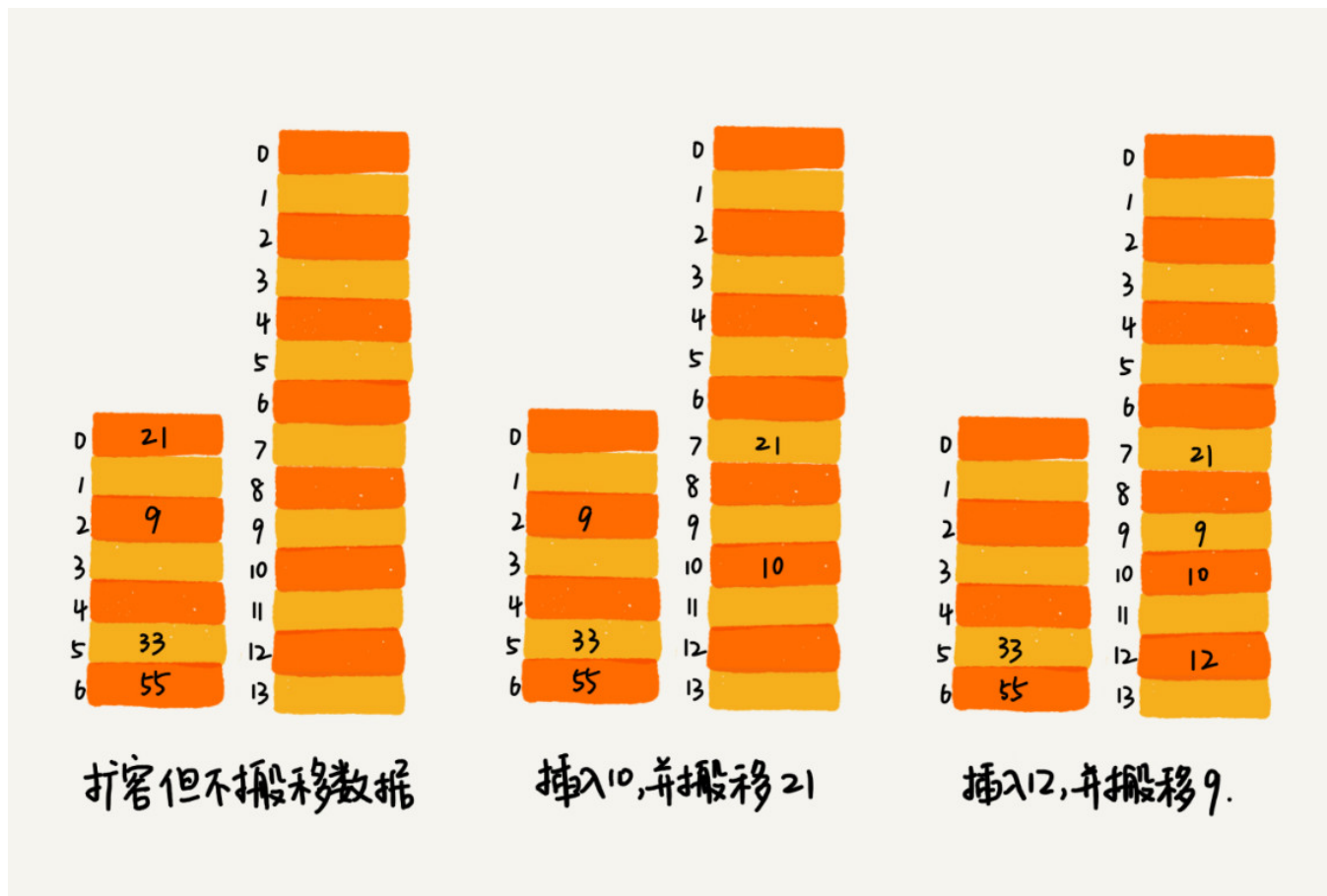
我们刚刚分析得到，大部分情况下，动态扩容的散列表插入一个数据都很快，但是在特殊情况下，当装载因子已经到达阈值，需要先进行扩容，再插入数据。这个时候，插入数据就会变得很慢，甚至会无法接受。

我举一个极端的例子，如果散列表当前大小为 1GB，要想扩容为原来的两倍大小，那就需要对 1GB 的数据重新计算哈希值，并且从原来的散列表搬移到新的散列表，听起来就很耗时，是不是？

如果我们的业务代码直接服务于用户，尽管大部分情况下，插入一个数据的操作都很快，但是，极个别非常慢的插入操作，也会让用户崩溃。这个时候，“一次性”扩容的机制就不合适了。

为了解决一次性扩容耗时过多的情况，我们可以将扩容操作穿插在插入操作的过程中，分批完成。当装载因子触达阈值之后，我们只申请新空间，但并不将老的数据搬移到新散列表中。

当有新数据要插入时，我们将新数据插入新散列表中，并且从老的散列表中拿出一个数据放入到新散列表。每次插入一个数据到散列表，我们都重复上面的过程。经过多次插入操作之后，老的散列表中的数据就一点一点全部搬移到新散列表中。这样没有了集中的一次性数据搬移，插入操作就都变得很快了。



这期间的查询操作怎么来做呢？对于查询操作，为了兼容了新、老散列表中的数据，我们先从新散列表中查找，如果没有找到，再去老的散列表中查找。

通过这样均摊的方法，将一次性扩容的代价，均摊到多次插入操作中，就避免了一次性扩容耗时过多的情况。这种实现方式，任何情况下，插入一个数据的时间复杂度都是 $O(1)$ 。

如何选择冲突解决方法？

上一节我们讲了两种主要的散列冲突的解决办法，开放寻址法和链表法。这两种冲突解决方法在实际的软件开发中都非常常用。比如，Java 中 `LinkedHashMap` 就采用了链表法解决冲突，`ThreadLocalMap` 是通过线性探测的开放寻址法来解决冲突。那你知道，这两种冲突解决方法各有什么优势和劣势，又各自适用哪些场景吗？

1. 开放寻址法

我们先来看看，开放寻址法的优点有哪些。

开放寻址法不像链表法，需要拉很多链表。散列表中的数据都存储在数组中，可以有效地利用 CPU 缓存加快查询速度。而且，这种方法实现的散列表，序列化起来比较简单。链表法包含指针，序列化起来就没那么容易。你可不要小看序列化，很多场合都会用到的。我们后面就有一节会讲什么是数据结构序列化、如何序列化，以及为什么要序列化。

我们再来看下，开放寻址法有哪些缺点。

上一节我们讲到，用开放寻址法解决冲突的散列表，删除数据的时候比较麻烦，需要特殊标记已经删除掉的数据。而且，在开放寻址法中，所有的数据都存储在一个数组中，比起链表法来说，冲突的代价更高。所以，使用开放寻址法解决冲突的散列表，装载因子的上限不能太大。这也导致这种方法比链表法更浪费内存空间。

所以，我总结一下，当数据量比较小、装载因子小的时候，适合采用开放寻址法。这也是 Java 中的 `ThreadLocalMap` 使用开放寻址法解决散列冲突的原因。

2. 链表法

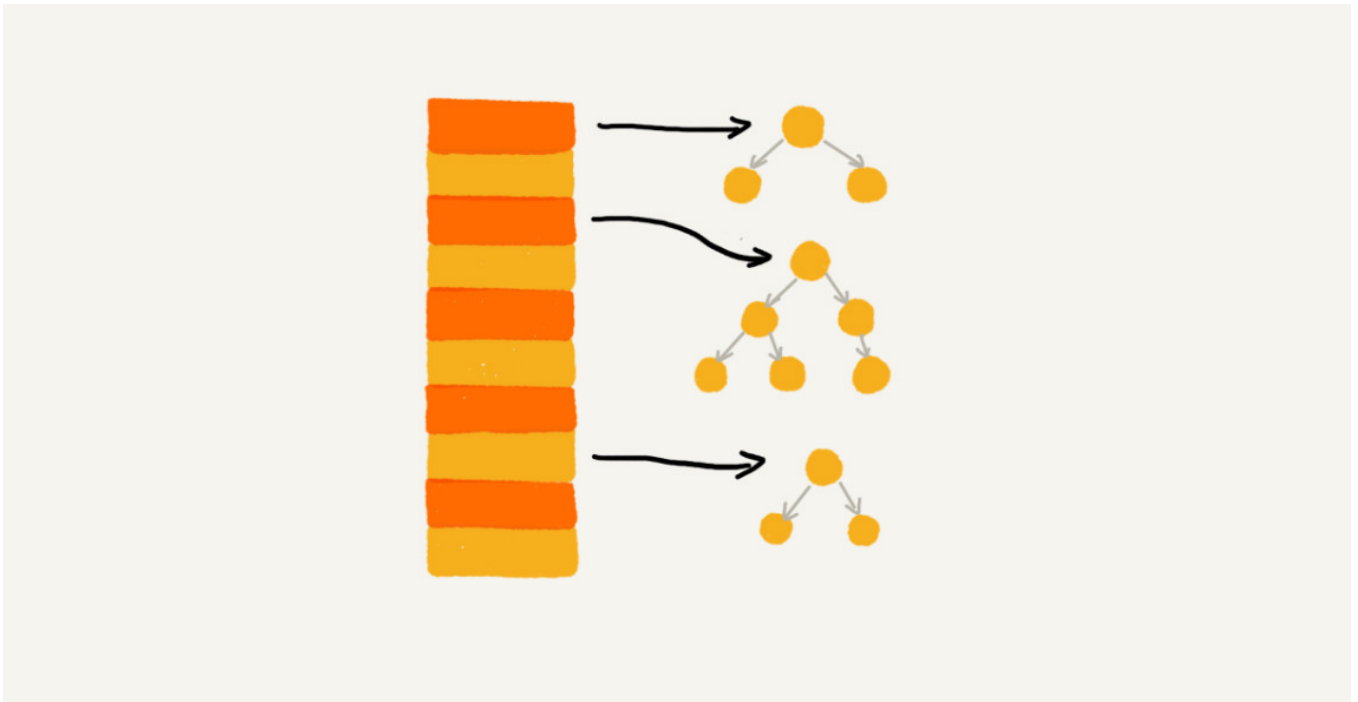
首先，链表法对内存的利用率比开放寻址法要高。因为链表结点可以在需要的时候再创建，并不需要像开放寻址法那样事先申请好。实际上，这一点也是我们前面讲过的链表优于数组的地方。

链表法比起开放寻址法，对大装载因子的容忍度更高。开放寻址法只能适用装载因子小于 1 的情况。接近 1 时，就可能会有大量的散列冲突，导致大量的探测、再散列等，性能会下降很多。但是对于链表法来说，只要散列函数的值随机均匀，即便装载因子变成 10，也就是链表的长度变长了而已，虽然查找效率有所下降，但是比起顺序查找还是快很多。

还记得我们之前在链表那一节讲的吗？链表因为要存储指针，所以对于比较小的对象的存储，是比较消耗内存的，还有可能会让内存的消耗翻倍。而且，因为链表中的结点是零散分布在内存中的，不是连续的，所以对 CPU 缓存是不友好的，这方面对于执行效率也有一定的影响。

当然，如果我们存储的是大对象，也就是说要存储的对象的大小远远大于一个指针的大小（4 个字节或者 8 个字节），那链表中指针的内存消耗在大对象面前就可以忽略了。

实际上，我们对链表法稍加改造，可以实现一个更加高效的散列表。那就是，我们将链表法中的链表改造为其他高效的动态数据结构，比如跳表、红黑树。这样，即便出现散列冲突，极端情况下，所有的数据都散列到同一个桶内，那最终退化成的散列表的查找时间也只不过是 $O(\log n)$ 。这样也就有效避免了前面讲到的散列碰撞攻击。



所以，我总结一下，基于链表的散列冲突处理方法比较适合存储大对象、大数据量的散列表，而且，比起开放寻址法，它更加灵活，支持更多的优化策略，比如用红黑树代替链表。

工业级散列表举例分析

刚刚我讲了实现一个工业级散列表需要涉及的一些关键技术，现在，我就拿一个具体的例子，Java 中的 HashMap 这样一个工业级的散列表，来具体看下，这些技术是怎么应用的。

1. 初始大小

HashMap 默认的初始大小是 16，当然这个默认值是可以设置的，如果事先知道大概的数据量有多大，可以通过修改默认初始大小，减少动态扩容的次数，这样会大大提高 HashMap 的性能。

2. 装载因子和动态扩容

最大装载因子默认是 0.75，当 HashMap 中元素个数超过 $0.75 * \text{capacity}$ （capacity 表示散列表的容量）的时候，就会启动扩容，每次扩容都会扩容为原来的两倍大小。

3. 散列冲突解决方法

HashMap 底层采用链表法来解决冲突。即使负载因子和散列函数设计得再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响 HashMap 的性能。

于是，在 JDK1.8 版本中，为了对 HashMap 做进一步优化，我们引入了红黑树。而当链表长度太长（默认超过 8）时，链表就转换为红黑树。我们可以利用红黑树快速增删改查的特点，提高 HashMap 的性能。当红黑树结点个数少于 8 个的时候，又会将红黑树转化为链表。因为在数据量较小的情况下，红黑树要维护平衡，比起链表来，性能上的优势并不明显。

4. 散列函数

散列函数的设计并不复杂，追求的是简单高效、分布均匀。我把它摘抄出来，你可以看看。

```
1 int hash(Object key) {  
2     int h = key.hashCode();  
3     return (h ^ (h >>> 16)) & (capicity - 1); //capicity 表示散列表的大小  
4 }
```

[复制代码](#)

其中，hashCode() 返回的是 Java 对象的 hash code。比如 String 类型的对象的 hashCode() 就是下面这样：

```
1 public int hashCode() {  
2     int var1 = this.hash;  
3     if(var1 == 0 && this.value.length > 0) {  
4         char[] var2 = this.value;  
5         for(int var3 = 0; var3 < this.value.length; ++var3) {  
6             var1 = 31 * var1 + var2[var3];  
7         }  
8         this.hash = var1;  
9     }  
10    return var1;  
11 }
```

[复制代码](#)

解答开篇

今天的内容就讲完了，我现在来分析一下开篇的问题：如何设计的一个工业级的散列函数？如果这是一道面试题或者是摆在你面前的实际开发问题，你会从哪几个方面思考呢？

首先，我会思考，何为工业级的散列表？工业级的散列表应该具有哪些特性？

结合已经学习过的散列知识，我觉得应该有这样几点要求：

- 支持快速的查询、插入、删除操作；
- 内存占用合理，不能浪费过多的内存空间；
- 性能稳定，极端情况下，散列表的性能也不会退化到无法接受的情况。

如何实现这样一个散列表呢？根据前面讲到的知识，我会从这三个方面来考虑设计思路：

- 设计一个合适的散列函数；
- 定义装载因子阈值，并且设计动态扩容策略；
- 选择合适的散列冲突解决方法。

关于散列函数、装载因子、动态扩容策略，还有散列冲突的解决办法，我们前面都讲过了，具体如何选择，还要结合具体的业务场景、具体的业务数据来具体分析。不过只要我们朝这三个方向努力，就离设计出工业级的散列表不远了。

内容小结

上一节的内容比较偏理论，今天的内容侧重实战。我主要讲了如何设计一个工业级的散列表，以及如何应对各种异常情况，防止在极端情况下，散列表的性能退化过于严重。我分了三部分来讲解这些内容，分别是：如何设计散列函数，如何根据装载因子动态扩容，以及如何选择散列冲突解决方法。

关于散列函数的设计，我们要尽可能让散列后的值随机且均匀分布，这样会尽可能地减少散列冲突，即便冲突之后，分配到每个槽内的数据也比较均匀。除此之外，散列函数的设计也不能太复杂，太复杂就会太耗时间，也会影响散列表的性能。

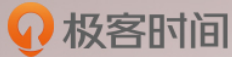
关于散列冲突解决方法的选择，我对比了开放寻址法和链表法两种方法的优劣和适应的场景。大部分情况下，链表法更加普适。而且，我们还可以通过将链表法中的链表改造成其他动态查找数据结构，比如红黑树，来避免散列表时间复杂度退化成 $O(n)$ ，抵御散列碰撞攻击。但是，对于小规模数据、装载因子不高的散列表，比较适合用开放寻址法。

对于动态散列表来说，不管我们如何设计散列函数，选择什么样的散列冲突解决方法。随着数据的不断增加，散列表总会出现装载因子过高的情况。这个时候，我们就需要启动动态扩容。

课后思考

在你熟悉的编程语言中，哪些数据类型底层是基于散列表实现的？散列函数是如何设计的？散列冲突是通过哪种方法解决的？是否支持动态扩容呢？

欢迎留言和我分享，我会第一时间给你反馈。



数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



©版权归极客邦科技所有，未经许可不得转载

上一篇 18 | 散列表（上）：Word文档中的单词拼写检查功能是如何实现的？

下一篇 20 | 散列表（下）：为什么散列表和链表经常会一起使用？

写留言

精选留言



Jerry银银

19

```
int hash(Object key) {  
    int h = key.hashCode();  
    return (h ^ (h >>> 16)) & (capacity - 1); //capacity 表示散列表的大小  
}
```

先补充下老师使用的这段代码的一些问题：在JDK HashMap源码中，是分两步走的：

1. hash值的计算，源码如下：

```
static final int hash(Object key) {  
    int hash;  
    return key == null ? 0 : (hash = key.hashCode()) ^ hash >>> 16;  
}
```

2. 在插入或查找的时候，计算Key被映射到桶的位置：

```
int index = hash(key) & (capacity - 1)
```

JDK HashMap中hash函数的设计，确实很巧妙：

首先hashcode本身是个32位整型值，在系统中，这个值对于不同的对象必须保证唯一（JAV A规范），这也是大家常说的，重写equals必须重写hashcode的重要原因。

获取对象的hashcode以后，先进行移位运算，然后再和自己做异或运算，即： $\text{hashcode} \wedge (\text{hashcode} \ggg 16)$ ，这一步甚是巧妙，是将高16位移到低16位，这样计算出来的整型值将“具有”高位和低位的性质

最后，用hash表当前的容量减去一，再和刚刚计算出来的整型值做位与运算。进行位与运算，很好理解，是为了计算出数组中的位置。但这里有个问题：

为什么要用容量减去一？

因为 $A \% B = A \& (B - 1)$ ，所以， $(h \wedge (h \ggg 16)) \& (\text{capacity} - 1) = (h \wedge (h \ggg 16)) \% \text{capacity}$ ，可以看出这里本质上是使用了「除留余数法」

综上，可以看出，hashcode的随机性，加上移位异或算法，得到一个非常随机的hash值，再通过「除留余数法」，得到index，整体的设计过程与老师所说的“散列函数”设计原则非常吻合！

有分析不准确的地方，请指正！

2018-11-04

| 作者回复



2018-11-05



天王

能否每节讲完都有个代码的demo?

2018-11-02

| 作者回复

是个好建议 我考虑下

2018-11-02



拉欧

比如Redis中的hash,set,hset,都是散列表实现，他们的动态扩容策略是同时维护两个散列表，然后一点点搬移数据

2018-11-02



喜欢你的笑

能分析一下HashMap的散列函数吗？

2018-11-02

| 作者回复

不建议搞得这么详细：) 你就看一眼 有个印象就好了

👍 19

👍 12

👍 4

2018-11-02



Infinite_gao

👍 4

老师可以分享一下，你对hashmap的默认负载因子是0.75的理解吗？是与泊松分布有关吗？

2018-11-02

| 作者回复

大牛 能否详细说说

2018-11-02



□

👍 4

老师能不能就具体的题，讲讲数据结构呀。这种高大上的，对我来说有点难😞

2018-11-02

| 作者回复

我后面还打算把所有的课后题集中写一写答案 那个时候会具体分析题目对应的就解决思路

2018-11-02



w1sl1y

👍 3

我怎么hashmap记得红黑树树化的阈值是8，退化的阈值是6，回头看看源码确认下

2018-11-03

| 作者回复

确认好留言给我啊

2018-11-05



Allen Zou

👍 3

老师，开放寻址法如果冲突了，占用其它hash code对应的位置，那该位置真正的数据来的时候怎么办，接着往后放么？删除的时候是否要搬回来？

2018-11-03

| 作者回复

不存在真正的数据的说法 都是先来先占坑

2018-11-05



Zhangwh

👍 3

链表和哈希表结合成lru 缓存，老师能讲讲不，记得老师在链表那块说过

2018-11-02

| 作者回复

下一节课就要讲了

2018-11-02



猫头鹰爱拿铁

👍 2

集合类的带hash的，例如hashmap、hashset、hashtable等。hashmap中散列函数是key的hashcode与key的hashcode右移16位异或，这是为了把key的高位考虑进去，如果key是0，hash值为0。在put的时候，如果表没有初始化，需要初始化下，在计算key的位置的时候很巧妙，使用表的length-1和key的hash值与计算的，实际上就是对key的hash值对表长取

模，基于hashmap是2的幂次方特性，这种位运算速度更快。如果put后hashmap的数据容量超过了表的容量*负载因子，就会自动扩容，默认是两倍，自动扩容方法是将key的hash与表长直接与判断是否有高位，有高位就把这个node放到新表里旧表对应位置加旧表长的地方。没有高位就直接是新表旧位置。这是hashmap1.8的处理方法。hashmap1.7还是对key的hash取模。如果是个非常大的数，赋值为integer.max。hashmap采用的是链地址法结合红黑树解决hash冲突，当桶中链表长度大于8就会将桶中数据结构转化为红黑树。hashtable默认的初使容量11，负载因子也是0.75，如果要指定初始化hashtable容量最好是给一个素数。这是因为放入table的时候需要对表长取模，尽量分散地映射。hashtable通过链地址法解决hash冲突，当数据容量大于数据容量*负载因子自动扩容，扩容原表长两倍+1。

2018-11-02



失火的夏天

👍 2

JAVA中hash化的好像都是基于散列表的，比如hashmap,linkedhashmap,hashset,linkedhashset，还有hashtable，concurrenthashmap之类的。感觉hashCode的设计真的很巧妙，包括treemap都应用到hashCode方法。一个hashCode里面包含了很多的数学思想

2018-11-02

| 作者回复

是的 哈希算法是个很大的话题

2018-11-02



陈华应

👍 2

对hashmap的认识有个新的高度，还是还是有很多细节需要研究，以前对数据结构的认知浮于表面，要认真去对接每个细节

2018-11-02

| 作者回复

深挖无止境 加油💪

2018-11-02



Infinite_gao

👍 2

老师可以分享一下你对hashmap的默认负载因子是0.75的理解吗？

2018-11-02

| 作者回复

😊 我也不是很清楚 不过应该是基于一些实验数据得来的吧

2018-11-02



辰陌

👍 1

python的字典就是封装好的散列吧

2018-11-05

| 作者回复

嗯嗯

2018-11-06



w1sl1y

👍 1

看了下，的确是TREEFY_THRESHOLD等于8
UNTREEFY_THRESHOLD等于6

2018-11-05



Jerry银银

👍 1

每次都会看留言，从留言中能学到不少东西。比如之前还真不能理解为什么Java HashMap的负载因子要用0.75

2018-11-04

作者回复

我也看你经常留言 你能不能研究这个问题 回复到留言区呢

2018-11-05



煦暖

👍 1

老师，HashMap 的散列函数看不懂，可以讲解一下吗？

2018-11-03

作者回复

我也没仔细研究 不建议研究这么深

2018-11-05



姜威

👍 1

总结：散列表（中）

面试题目：如何设计一个工业级的散列函数？

思路：

何为一个工业级的散列表？工业级的散列表应该具有哪些特性？结合学过的知识，我觉的应该有这样的要求：

- 1.支持快速的查询、插入、删除操作；
- 2.内存占用合理，不能浪费过多空间；
- 3.性能稳定，在极端情况下，散列表的性能也不会退化到无法接受的情况。

方案：

如何设计这样一个散列表呢？根据前面讲到的知识，我会从3个方面来考虑设计思路：

- 1.设计一个合适的散列函数；
- 2.定义装载因子阈值，并且设计动态扩容策略；
- 3.选择合适的散列冲突解决方法。

知识总结：

一、如何设计散列函数？

- 1.要尽可能让散列后的值随机且均匀分布，这样会尽可能减少散列冲突，即便冲突之后，分配到每个槽内的数据也比较均匀。
- 2.除此之外，散列函数的设计也不能太复杂，太复杂就会太耗时间，也会影响到散列表的性能。
- 3.常见的散列函数设计方法：直接寻址法、平方取中法、折叠法、随机数法等。

二、如何根据装载因子动态扩容？

1.如何设置装载因子阈值？

①可以通过设置装载因子的阈值来控制是扩容还是缩容，支持动态扩容的散列表，插入数据

的时间复杂度使用摊还分析法。

②装载因子的阈值设置需要权衡时间复杂度和空间复杂度。如何权衡？如果内存空间不紧张，对执行效率要求很高，可以降低装载因子的阈值；相反，如果内存空间紧张，对执行效率要求又不高，可以增加装载因子的阈值。

2.如何避免低效扩容？分批扩容

①分批扩容的插入操作：当有新数据要插入时，我们将数据插入新的散列表，并且从老的散列表中拿出一个数据放入新散列表。每次插入都重复上面的过程。这样插入操作就变得很快了。

②分批扩容的查询操作：先查新散列表，再查老散列表。

③通过分批扩容的方式，任何情况下，插入一个数据的时间复杂度都是 $O(1)$ 。

三、如何选择散列冲突解决方法？

①常见的2中方法：开放寻址法和链表法。

②大部分情况下，链表法更加普适。而且，我们还可以通过将链表法中的链表改造成其他动态查找数据结构，比如红黑树、跳表，来避免散列表时间复杂度退化成 $O(n)$ ，抵御散列冲突攻击。

③但是，对于小规模数据、装载因子不高的散列表，比较适合用开放寻址法。

2018-11-03



Kudo

👍 1

以前只是用，现在终于了解其背后的设计思想了，讲得真不赖。

2018-11-02



Demter

👍 1

分析的很透彻。

2018-11-02



凡

👍 1

看最近几篇文章多次提到红黑树！不知道什么时候会讲一下红黑树！然后这节课看的有点吃力呀，唯一一节看完没啥印象的

2018-11-02

作者回复

快讲了

2018-11-02



朱月俊

👍 1

C++中的`std::ordered_map`就是基于散列表实现的，支持动态扩容，链表法。还不清楚(1)动态扩容是否也是新插入一个元素，迁移一个元素；还是一次性全部迁移完。(2)链表数量过大是否改变数据结构也不清楚。

2018-11-02



cricket1981

👍 0

散列表在并发环境下扩容导致环链死锁的情况能否细讲一下？谢谢！

2018-11-13

作者回复

这个课程不涉及多线程 不好意思啊

2018-11-13



yoooh

0

hashmap退化链的数是6,升级红黑树8

2018-11-10



Light Lin

0

最近有点掉队，很多不懂的，接下来要努力克服。

2018-11-06



Ricky

0

散列表是通过key查找数据的，其存储方式是采用散列函数规则散列到每个槽，在每个槽建立跳表的时候，应该是按照key大小进行插入的吧，因为要实现O1的查找操作。跳表实现了元素有序，但是实际上散列表元素应该无序才对吧

2018-11-05



在路上

0

老师，请问下，如何用代码去模拟这种hash冲突呢？我试过，但还是不会。

2018-11-04

作者回复

这个跟数据和哈希函数还有散列表大小有关。比如散列函数是 $key \% n$ 。n是散列表大小7。那key是5和12的时候就冲突了

2018-11-05



ALAN

0

老师，有个问题请教下。开放寻址法查询的时候，碰到散列表为空的位置后，就不继续往后找了吗？这样设计不合理吧，因为存储的时候，存数据的散列表的位置是随机的，空的位置后面也许存了数据呢？如果是继续找的话，那为什么删除数据后，要进行特殊标记，这样标记也没意义啊，反正碰到空的位置，还是会继续找，这样标不标记都无所谓啊？

2018-11-04



Jerry银银

0

老师，有一个疑问点：“工业级水平”，这个词该怎么定义和衡量，像Python的Dictionary，JAVA的hashmap，以及c++中的unordered_map，这些散列表应该就是属于工业级水平了吧。那我们自己在设计数据结构和算法时，如何评估自己做的东西离工业级水平差多远呢？

2018-11-04

作者回复

工业级应该也没有一个标准吧 更多的是指代码健壮 在任何情况下都能正确 高效运行

2018-11-05



Jerry银银

0

分享自己猜测的一个有意思的点：专栏留言的排序规则应该是按照点赞数排序，再按照留言时间排序的！？

可以引申一个思考题，在实际应用中，你会怎么排序，又会使用什么排序算法呢？

2018-11-04

作者回复

👍 这个问题很好

2018-11-05



sarahsnow

👍 0

看到排序，插入、查找、删除这些算法，其实都和不同数据的特点、规模息息相关。

作为笨鸟一只，勉强跟上课程的进度和思想，实战还是懵圈。

自己练习这些算法时，需要先准备数据，和预处理数据，把数据存储装载到数据结构中是个难点。

请问老师能否提供一些现成的典型数据(不同规模、特点的)，和数据预处理的代码呢？

2018-11-04

作者回复

可以是可以 但实话讲 对我来说工作量就太大了

2018-11-05



sarahsnow

👍 0

Java中有用开放寻址法的Hash类吗？Java底层是用C开发的吗？谢谢

2018-11-04

作者回复

文章不是说了嘛 ThreadLocalMap

2018-11-05



許敲敲

👍 0

有没有基于python语言的实例呢

2018-11-03

作者回复

你去看看dict

2018-11-05



asdf100

👍 0

散列表存储方式分两种：

一种是以类似数组一样的存储方式，存储的数据量小，维护的话使用开放寻址方式。

另一种是单向链表的存储形式，适合数据量大的情况。

两者都可能根据情况进行扩容和缩容。

这样理解对么？

2018-11-03



吴彪

0

我看Fluent Python这本书p89, dict解决散列冲突的方法比较好玩。取部分哈希值来查找bucket、如果冲突了, 再另取部分哈希值查找。按书上的说法, 碰撞概率极其低。推测不是基于链表法实现, 可能是优化过的开放寻址法, 具体要看c的源码。。

2018-11-03



Allen Zou

0

老师, 开放寻址法如果冲突后会顺序占用后面的空间, 比如值x 的hash code 为 n, 冲突后占用 $n + 1, n + 2, \dots, n + k$, 那这时候如果来了一个数 y 的 hash code 为 $n + 1$, 那它怎么存呢? 如果放在 $n + k + 1$ 位, 那删除的时候如何判断呢? 再计算一遍 hash code 么?

2018-11-02

| 作者回复

对比原始数据 看是不是要删除的那个

2018-11-05



Ricky

0

我想问一下老师, hashmap 用跳表实现的时候, 数值应该按顺序存储, 方便查找, 但是字典型数据难道不应该是无序的吗

2018-11-02

| 作者回复

没太看懂你说的呢

2018-11-05



有铭

0

之前看专栏的Java高级特性的时候就提到Java的HashMap设计的非常巧妙, 如果能把Java的那堆容器设计的算法都搞的很明白的话绝对是高手

2018-11-02



柠檬C

0

这节干活满满, 2节下来对hashmap底层更熟悉了

2018-11-02



王小李

0

请问老师这边讲数据结构, 会涉及线程是否安全的例子吗? 比如上面的哈希表, 怎么样改进才能达到线程安全提高效率。还有之前看了跳跃表的思想, 我觉得是否对于大的哈希表的每个 bucket 里再嵌套一个哈希表。这样做是否也可以达到优化的效果?

2018-11-02



中午要吃鱼摆摆

0

老师讲的太好啦!

2018-11-02



白了少年头

0



老师的讲解十分透彻，听了这节课再去看相应散列表源码的时候，就会有一种恍然大悟的感觉，非常棒！

2018-11-02



NeverMore

👍 0

学习啦。由浅入深。

2018-11-02



alex44jzy

👍 0

醍醐灌顶 很详细了

2018-11-02



醉比

👍 0

HashMap从入门到精通

2018-11-02



侯金彪

👍 0

之前看java8 hashmap扩容的时候没看明白，今天老师的讲解真是让我茅塞顿开！谢谢老师！

2018-11-02