07 | 深入浅出HashMap的设计与优化

2019-06-04 刘超

你好，我是刘超。

在上一讲中我提到过Collection接口，那么在Java容器类中，除了这个接口之外，还定义了一个很重要的Map接口，主要用来存储键值对数据。

HashMap作为我们日常使用最频繁的容器之一，相信你一定不陌生了。今天我们就从HashMap的底层实现讲起，深度了解下它的设计与优化。

常用的数据结构

我在05讲分享List集合类的时候，讲过ArrayList是基于数组的数据结构实现的，LinkedList是基于链表的数据结构实现的，而我今天要讲的HashMap是基于哈希表的数据结构实现的。我们不妨一起来温习下常用的数据结构，这样也有助于你更好地理解后面地内容。

数组：采用一段连续的存储单元来存储数据。对于指定下标的查找，时间复杂度为O(1)，但在数组中间以及头部插入数据时，需要复制移动后面的元素。

链表：一种在物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。

链表由一系列结点（链表中每一个元素）组成，结点可以在运行时动态生成。每个结点都包含“存储数据单元的数据域”和“存储下一个结点地址的指针域”这两个部分。

由于链表不用必须按顺序存储，所以链表在插入的时候可以达到O(1)的复杂度，但查找一个结点或者访问特定编号的结点需要O(n)的时间。

哈希表：根据关键码值（Key value）直接进行访问的数据结构。通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做哈希函数，存放记录的数组就叫做哈希表。

树：由n（n≥1）个有限结点组成的一个具有层次关系的集合，就像是一棵倒挂的树。

HashMap的实现结构

了解完数据结构后，我们再来看下HashMap的实现结构。作为最常用的Map类，它是基于哈希表实现的，继承了AbstractMap并且实现了Map接口。

哈希表将键的Hash值映射到内存地址，即根据键获取对应的值，并将其存储到内存地址。也就是说HashMap是根据键的Hash值来决定对应值的存储位置。通过这种索引方式，HashMap获取数据的速度会非常快。

例如，存储键值对（x，“aa”）时，哈希表会通过哈希函数f(x)得到"aa"的实现存储位置。

但也会有新的问题。如果再来一个(y，“bb”)，哈希函数f(y)的哈希值跟之前f(x)是一样的，这样两个对象的存储地址就冲突了，这种现象就被称为哈希冲突。那么哈希表是怎么解决的呢？方式有很多，比如，开放定址法、再哈希函数法和链地址法。

开放定址法很简单，当发生哈希冲突时，如果哈希表未被装满，说明在哈希表中必然还有空位置，那么可以把key存放到冲突位置的空位置上去。这种方法存在着很多缺点，例如，查找、扩容等，所以我不建议你作为解决哈希冲突的首选。

再哈希法顾名思义就是在同义词产生地址冲突时再计算另一个哈希函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但却增加了计算时间。如果我们不考虑添加元素的时间成本，且对查询元素的要求极高，就可以考虑使用这种算法设计。

HashMap则是综合考虑了所有因素，采用链地址法解决哈希冲突问题。这种方法是采用了数组（哈希表）+链表的数据结构，当发生哈希冲突时，就用一个链表结构存储相同Hash值的数据。

HashMap的重要属性

从HashMap的源码中，我们可以发现，HashMap是由一个Node数组构成，每个Node包含了一个key-value键值对。

```
transient Node<K,V>[] table;
```

Node类作为HashMap中的一个内部类，除了key、value两个属性外，还定义了一个next指针。当有哈希冲突时，HashMap会用之前数组当中相同哈希值对应存储的Node对象，通过指针指向新增的相同哈希值的Node对象的引用。

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
}
```

HashMap还有两个重要的属性：加载因子（loadFactor）和边界值（threshold）。在初始化HashMap时，就会涉及到这两个关键初始化参数。

```
int threshold;

    final float loadFactor;
```

LoadFactor属性是用来间接设置Entry数组（哈希表）的内存空间大小，在初始HashMap不设置参数的情况下，默认LoadFactor值为0.75。**为什么是0.75这个值呢？**

这是因为对于使用链表法的哈希表来说，查找一个元素的平均时间是O(1+n)，这里的n指的是遍历链表的长度，因此加载因子越大，对空间的利用就越充分，这就意味着链表的长度越长，查找效率也就越低。如果设置的加载因子太小，那么哈希表的数据将过于稀疏，对空间造成严重浪费。

那有没有什么办法来解决这个因链表过长而导致的查询时间复杂度高的问题呢？你可以先想想，我将在后面的内容中讲到。

Entry数组的Threshold是通过初始容量和LoadFactor计算所得，在初始HashMap不设置参数的情况下，默认边界值为12。如果我们在初始化时，设置的初始化容量较小，HashMap中Node的

数量超过边界值，HashMap就会调用resize()方法重新分配table数组。这将会导致HashMap的数组复制，迁移到另一块内存中去，从而影响HashMap的效率。

HashMap添加元素优化

初始化完成后，HashMap就可以使用put()方法添加键值对了。从下面源码可以看出，当程序将一个key-value对添加到HashMap中，程序首先会根据该key的hashCode()返回值，再通过hash()方法计算出hash值，再通过putVal方法中的(n - 1) & hash决定该Node的存储位置。

```
public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }
```

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

```
 if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        //通过putVal方法中的(n - 1) & hash决定该Node的存储位置
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
```

如果你不太清楚hash()以及(n-1)&hash的算法，就请你看下面的详述。

我们先来了解下hash()方法中的算法。如果我们没有使用hash()方法计算hashCode，而是直接使用对象的hashCode值，会出现什么问题呢？

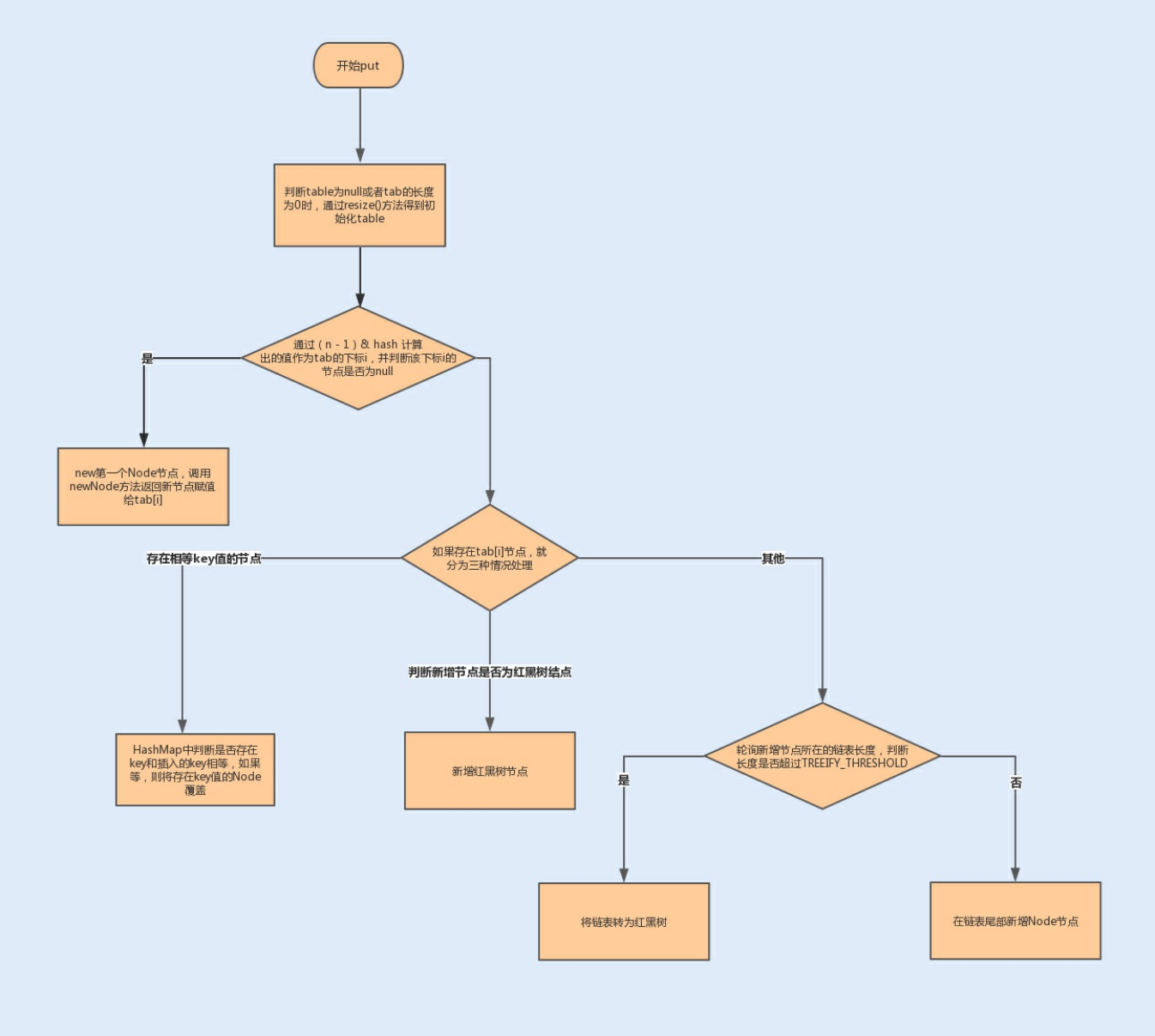
假设要添加两个对象a和b，如果数组长度是16，这时对象a和b通过公式(n - 1) & hash运算，也就是(16-1)&a.hashCode和(16-1)&b.hashCode，15的二进制为00000000000000000000000000001111，假设对象 A 的 hashCode 为10000100011100010000011110000000，对象 B 的 hashCode 为01110111001110001010000010100000，你会发现上述与运算结果都是0。这样的哈希结果就太让人失望了，很明显不是一个好的哈希算法。

但如果我们将 hashCode 值右移 16 位（h >>> 16代表无符号右移16位），也就是取 int 类型的

一半，刚好可以将该二进制数对半切开，并且使用位异或运算（如果两个数对应的位置相反，则结果为1，反之为0），这样的话，就能避免上面的情况发生。这就是hash()方法的具体实现方式。**简而言之，就是尽量打乱hashCode真正参与运算的低16位。**

我再来解释下(n - 1) & hash是怎么设计的，这里的n代表哈希表的长度，哈希表习惯将长度设置为2的n次方，这样恰好可以保证(n - 1) & hash的计算得到的索引值总是位于table数组的索引之内。例如：hash=15，n=16时，结果为15；hash=17，n=16时，结果为1。

在获得Node的存储位置后，如果判断Node不在哈希表中，就新增一个Node，并添加到哈希表中，整个流程我将用一张图来说明：



从图中我们可以看出：在JDK1.8中，HashMap引入了红黑树数据结构来提升链表的查询效率。

这是因为链表的长度超过8后，红黑树的查询效率要比链表高，所以当链表超过8时，HashMap就会将链表转换为红黑树，这里值得注意的一点是，这时的新增由于存在左旋、右旋效率会降

低。讲到这里，我前面我提到的“因链表过长而导致的查询时间复杂度高”的问题，也就迎刃而解了。

以下就是put的实现源码:

```
 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        if ((tab = table) == null || (n = tab.length) == 0)
//1、判断当table为null或者tab的长度为0时，即table尚未初始化，此时通过resize()方法得到初始化的table
            n = (tab = resize()).length;
        if ((p = tab[i = (n - 1) & hash]) == null)
//1.1、此处通过（n - 1） & hash 计算出的值作为tab的下标i，并另p表示tab[i]，也就是该链表第一个节点的位置。
            tab[i] = newNode(hash, key, value, null);
//1.1.1、当p为null时，表明tab[i]上没有任何元素，那么接下来就new第一个Node节点，调用newNode方法返回新节
        else {
//2.1下面进入p不为null的情况，有三种情况：p为链表节点；p为红黑树节点；p是链表节点但长度为临界长度TREE
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
//2.1.1HashMap中判断key相同的条件是key的hash相同，并且符合equals方法。这里判断了p.key是否和插入的key


                e = p;
            else if (p instanceof TreeNode)
//2.1.2现在开始了第一种情况，p是红黑树节点，那么肯定插入后仍然是红黑树节点，所以我们直接强制转型p后调用
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
//2.1.3接下里就是p为链表节点的情形，也就是上述说的另外两类情况：插入后还是链表/插入后转红黑树。另外，上
                for (int binCount = 0; ; ++binCount) {
//我们需要一个计数器来计算当前链表的元素个数，并遍历链表，binCount就是这个计数器


                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1)
// 插入成功后，要判断是否需要转换为红黑树，因为插入后链表长度加1，而binCount并不包含新节点，所以判断时
                            treeifyBin(tab, hash);
//当新长度满足转换条件时，调用treeifyBin方法，将该链表转换为红黑树
                        break;
```

```
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

HashMap获取元素优化

当HashMap中只存在数组，而数组中没有Node链表时，是HashMap查询数据性能最好的时候。一旦发生大量的哈希冲突，就会产生Node链表，这个时候每次查询元素都可能遍历Node链表，从而降低查询数据的性能。

特别是在链表长度过长的情况下，性能将明显降低，红黑树的使用很好地解决了这个问题，使得查询的平均复杂度降低到了O(log(n))，链表越长，使用黑红树替换后的查询效率提升就越明显。

我们在编码中也可以优化HashMap的性能，例如，重新key值的hashCode()方法，降低哈希冲突，从而减少链表的产生，高效利用哈希表，达到提高性能的效果。

HashMap扩容优化

HashMap也是数组类型的数据结构，所以一样存在扩容的情况。

在JDK1.7 中，HashMap整个扩容过程就是分别取出数组元素，一般该元素是最后一个放入链表中的元素，然后遍历以该元素为头的单向链表元素，依据每个被遍历元素的 hash 值计算其在新数组中的下标，然后进行交换。这样的扩容方式会将原来哈希冲突的单向链表尾部变成扩容后单向链表的头部。

而在 JDK 1.8 中，HashMap对扩容操作做了优化。由于扩容数组的长度是 2 倍关系，所以对于假设初始 tableSize = 4 要扩容到 8 来说就是 0100 到 1000 的变化（左移一位就是 2 倍），在扩容中只用判断原来的 hash 值和左移动的一位（newtable 的值）按位与操作是 0 或 1 就行，0 的话索引不变，1 的话索引变成原索引加上扩容前数组。

之所以能通过这种“与运算“来重新分配索引，是因为 hash 值本来就是随机的，而hash 按位与上 newTable 得到的 0（扩容前的索引位置）和 1（扩容前索引位置加上扩容前数组长度的数值索引处）就是随机的，所以扩容的过程就能把之前哈希冲突的元素再随机分布到不同的索引中去。

总结

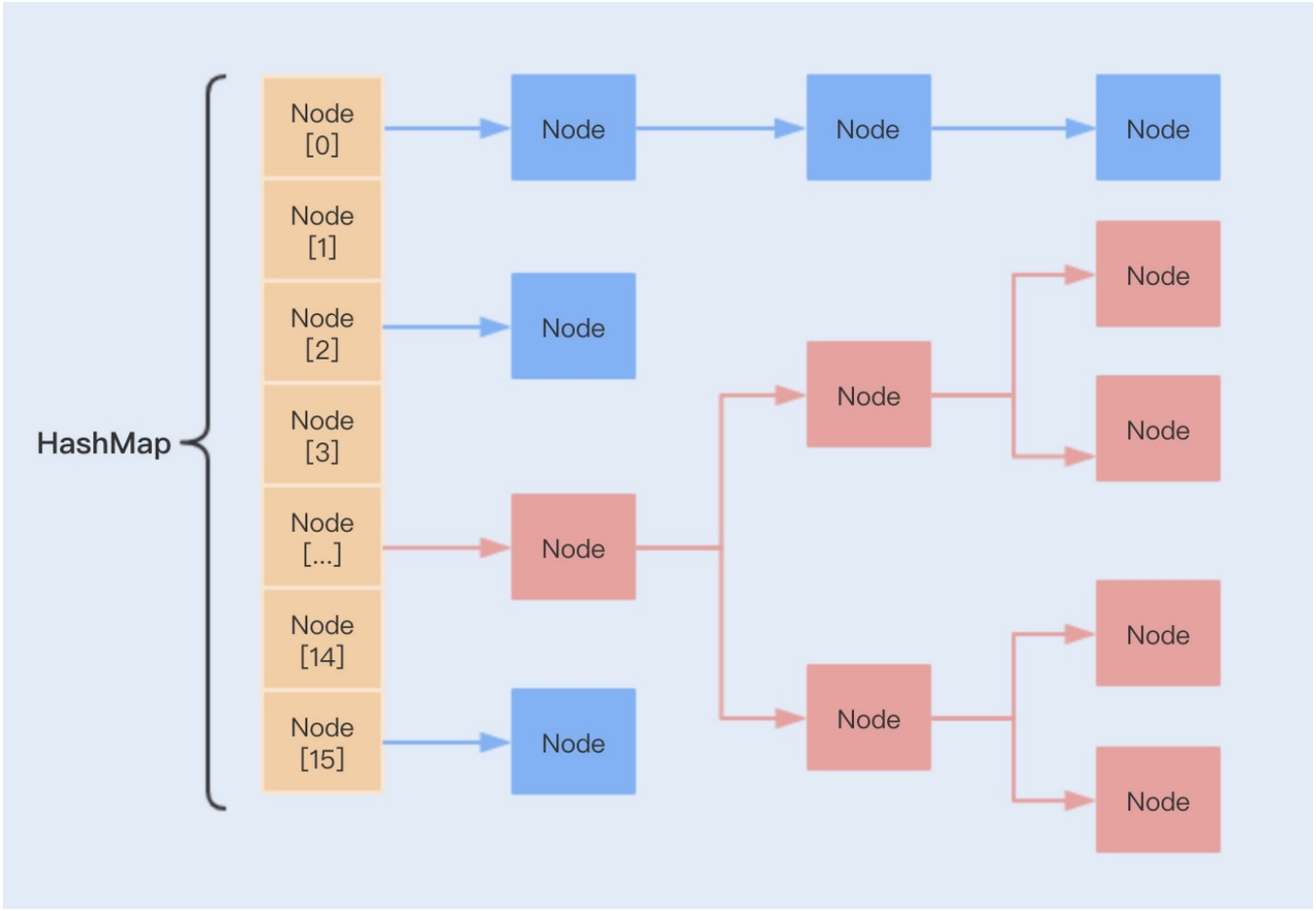
HashMap通过哈希表数据结构的形式来存储键值对，这种设计的好处就是查询键值对的效率高。

我们在使用HashMap时，可以结合自己的场景来设置初始容量和加载因子两个参数。当查询操作较为频繁时，我们可以适当地减少加载因子；如果对内存利用率要求比较高，我可以适当的增加加载因子。

我们还可以在预知存储数据量的情况下，提前设置初始容量（初始容量=预知数据量/加载因子）。这样做的好处是可以减少resize()操作，提高HashMap的效率。

HashMap还使用了数组+链表这两种数据结构相结合的方式实现了链地址法，当有哈希值冲突时，就可以将冲突的键值对链成一个链表。

但这种方式又存在一个性能问题，如果链表过长，查询数据的时间复杂度就会增加。HashMap就在Java8中使用了红黑树来解决链表过长导致的查询性能下降问题。以下是HashMap的数据结构图：



思考题

实际应用中，我们设置初始容量，一般得是2的整数次幂。你知道原因吗？

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起学习。

精选留言

陆离 16

2的幂次方减1后每一位都是1，让数组每一个位置都能添加到元素。
例如十进制8，对应二进制1000，减1是0111，这样在&hash值使数组每个位置都是可以添加到元素的，如果有一个位置为0，那么无论hash值是多少那一位总是0，例如0101，&hash后第二位总是0，也就是说数组中下标为2的位置总是空的。
如果初始化大小设置的不是2的幂次方，hashmap也会调整到比初始化值大且最近的一个2的幂作为capacity。

2019-06-04

作者回复

回答正确，就是减少哈希冲突，均匀分布元素。

2019-06-04

AiSmart4J 7

1）通过将 Key 的 hash 值与 length-1 进行 & 运算，实现了当前 Key 的定位，2 的幂次方可以减少冲突（碰撞）的次数，提高 HashMap 查询效率；
2）如果 length 为 2 的次幂，则 length-1 转化为二进制必定是 11111…… 的形式，在于 h 的二进制与操作效率会非常的快，而且空间不浪费；如果 length 不是 2 的次幂，比如 length 为 15，则 length-1 为 14，对应的二进制为 1110，在于 h 与操作，最后一位都为 0，而 0001，0011，0101，1001，1011，0111，1101 这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！这样就会造成空间的浪费。

2019-06-04

作者回复

回答非常到位

2019-06-04

tyul 3

老师，您好，请教一个问题，为什么 HashMap 的容量等于数组长度？但是扩容的时候却是根据 Map 里的所有元素总数去扩容，这样会不会导致数组中的某一个 node 有很长的链表或红黑树，数组中的其他位置都没有元素？谢谢

2019-06-05

大虫子 3

老师您好，能解答下，为什么JDK1.8之前，链表元素增加采用的是头插法，1.8之后改成尾插法了。1.8之前采用头插法是基于什么设计思路呢？

2019-06-04

作者回复

你好，JDK1.7是考虑新增数据大多数是热点数据，所以考虑放在链表头位置，也就是数组中，

这样可以提高查询效率，但这种方式会出现插入数据是逆序的。在JDK1.8开始hashmap链表在节点长度达到8之后会变成红黑树，这样一来在数组后节点长度不断增加时，遍历一次的次数就会少很多，相比头插法而言，尾插法操作额外的遍历消耗已经小很多了。

也有很多人说避免多线程情况下hashmap扩容时的死循环问题，我个人觉得避免死循环的关键不在尾插法的改变，而是扩容时，用了首尾两个指针来避免死循环。这个我会在后面的多线程中讲到hashmap扩容导致死循环的问题。

2019-06-05

晓杰

2

初始容量2的n次方是偶数，在计算key的索引位置时，是通过(n-1)&hash计算的，这样n-1得到的奇数，那么通过在进行与操作时，如果hash的第一位是0，那么(n-1)&hash得到的是偶数，如果hash的第一位是1，那么(n-1)&hash得到的是奇数，因此可以让数据分布更加均匀，减少hash冲突，相反如果n-1是偶数，那无论hash的第一位是偶数还是奇数，(n-1)&hash得到的都是偶数，不利于数据的分布

2019-06-05

小小征

2

0 的话索引不变，1 的话索引变成原索引加上扩容前数组。 这句有点不理解 老师

2019-06-05

作者回复

以下是resize中判断是否位移的部分代码，我们可以看到元素的hash值与原数组容量运算，如果运算结果为0，保持原位，如果运算结果为1，则意向扩容的高位。

```
if ((e.hash & oldCap) == 0) {
if (loTail == null)
loHead = e;
else
loTail.next = e;
loTail = e;
}
else {
if (hiTail == null)
hiHead = e;
else
hiTail.next = e;
hiTail = e;
}
```

假设链表中有4、8、12，他们的二进制位00000100、00001000、00001100，而原来数组容量

为4，则是 00000100，以下与运算：

00000100 & 00000100 = 0 保持原位
00001000 & 00000100 = 1 移动到高位
00001100 & 00000100 = 1 移动到高位

2019-06-06

嘉嘉 1

加载因子那块儿，感觉有点跳跃，为什么加载因子越大，对空间利用越充分呢？

2019-06-13

作者回复

加载因子是扩容的参考标准，如果加载因子越大，例如默认数组初始化大小为16，加载因子由0.75改成1.0，原来数组长度到了12就扩容，变成数组大小为16，也就是说被占满了，才会进行扩容，这也可能意味着已经发生了很多哈希冲突，这样导致某些数组中的链表长度增加，影响查询效率。

2019-06-14

孙志强 1

以前看源码，我记得好像链表转换红黑树不光链表元素大于8个，好像还有一个表的大小大于64

2019-06-05

作者回复

对的，有这样一个条件。

2019-06-06

Liam 1

Hash字典发生扩容时，需要复制元素，请问这个过程是一次完成的吗？redis里的字典是准备了两个字典，一个原始字典，一个rehash字典，扩容后，不是一次完成数据迁移的，每次操作字典都考虑两个数组并复制数据，扩容完毕后交换两个数组指针

2019-06-05

作者回复

HashMap的扩容是一次性完成的。

2019-06-05

-W.LI- 1

老师好。hashmap的put和get的时间复杂度算多少啊?最好O(1)。最坏复杂度是O(log(n))平均是O(1)么?。。。treeMap的,treeMap，putO(n)，getO(1)?之前面试被问了，不晓得哪错了

2019-06-04

作者回复

面试的时候，问到时间复杂度，大部分是考察你对数据结构的了解程度。建议可以多复习下数

据结构的知识。

hashmap的最优时间复杂度是O（1），而最坏时间复杂度是O(n)。

在没有产生hash冲突的情况下，查询和插入的时间复杂度是O(1)；

而产生hash冲突的情况下，如果是最终插入到链表，链表的插入时间复杂度为O(1)，而查询的时候，时间复杂度为O(n)；

在产生hash冲突的情况下，如果最终插入的是红黑树，插入和查询的平均时间复杂度是O（log n）。

而TreeMap是基于红黑树实现的，所以时间复杂度你也清楚了吧。

2019-06-05

强哥 1

最主要的原因是位运算替代%取模操作，提高运算性能，说什么降低冲突的，是否比较过质数和偶数冲突概率呢？

2019-06-04

作者回复

用与运算是提高了运算性能，而容量大小为2的幂次方是为了降低哈希冲突。

2019-06-05

WolvesLeader 1

hashmap在多线程情况下数据丢失，大师，能不能分析分析原因

2019-06-04

作者回复

你好，这个在后面多线程中会讲到，可以留意关注下。

2019-06-05

Darren 1

因为选择下标和扩容的时候都依赖和(n-1）按位与，2的幂次方满足n-1都是1的情况，因此必须选择2的幂次方，且如果使用时传入的参数不是2的幂次方，HashMap会自动转成大于传入参数的最小的2的幂次方来确保下标和扩容机制的正常运行

2019-06-04

Fever 0

Hash冲突，那么可以把 key 存放到冲突位置的空位置上去，这里是不是写错了，应该是冲突位置后面的空位置吧？

2019-06-12

小辉辉 0

HashMap源码看过几遍，跟着老师的思路来又学到了新东西

2019-06-11

jimmy

0

请问下为什么链表转换成红黑树的阈值是8?

2019-06-10

作者回复

这是兼容链表查询和转红黑树两者性能一起考虑的，这个值经过不是绝对的最佳值，是基于大数据求得比较合适的阈值。

2019-06-10

bro.

0

编程中有遇到这种情况,当判断一个数n 是否为偶数使用n%2 == 0 ,计算机语言已经做了优化为n&(2-1) = n & 1 == 0,对于二进制来说与操作只需要一个电路即可完成比取余速度快多了,相同的对于hash扩容,只需要判断前一位hash值是0还是1,如果是0保持数组位置不变,如果为1增加原来扩容前数组长度即可,而且由于hash值计算每一位都是平均分配0或者1,所以保持均匀分布

2019-06-10

趙衍

0

对于老师提到的hash函数的目的是为了尽量打乱hashcode真正参与运算的低16位。我想问为什么只考虑了低16位，假设出现一种情况，hashcode的低位全部相等，高位不相等，那岂不是会大大加大哈希冲突吗？

2019-06-07

作者回复

一个开源框架更多是考虑一个大环境下的优化场景，所以对于某些特殊场景，比如你说的这种低位全相等，高位不相等的情况，自己需要考虑如何去优化key值。

2019-06-10

迎风劲草

0

主要是为了减少hash冲突，hash & (n-1) n为2的n次幂，n-1换成比特位都是1

2019-06-05

晓杰

0

用与运算代替取模运算，可以提高运算的效率，而且当n是2的幂次方时：hash & (n- 1) == hash % n

2019-06-05