二

07 深入浅出HashMap的设计与优化

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。

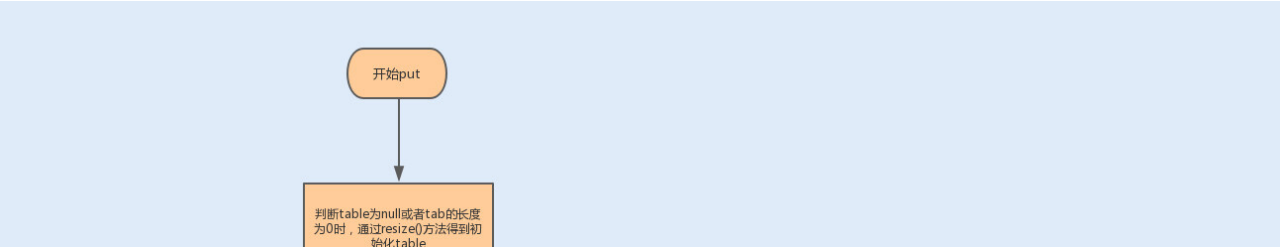
我们先来了解下 hash() 方法中的算法。如果我们没有使用 hash() 方法计算 hashCode，而是直接使用对象的 hashCode 值，会出现什么问题呢?

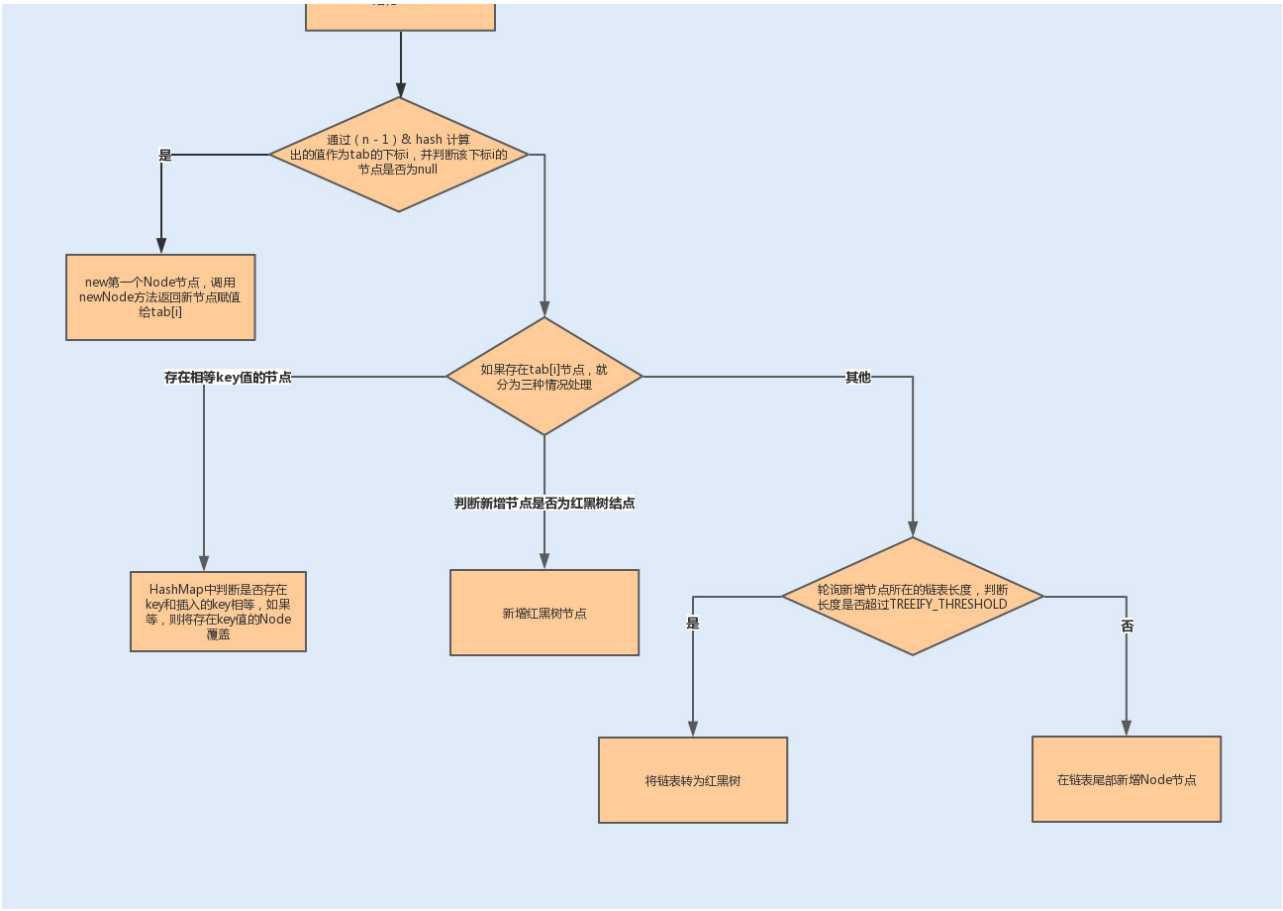
假设要添加两个对象 a 和 b，如果数组长度是 16，这时对象 a 和 b 通过公式 (n - 1) & hash 运算，也就是 (16-1)&a.hashCode 和 (16-1)&b.hashCode，15 的二进制为 00000000000000000000000000001111，假设对象 A 的 hashCode 为 10000100011100010000011111000000，对象 B 的 hashCode 为 01110111001110001010000010100000，你会发现上述与运算结果都是 0。这样的哈希结果就太让人失望了，很明显不是一个好的哈希算法。

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(
            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i
            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，
        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
            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

                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
                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
                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
                            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

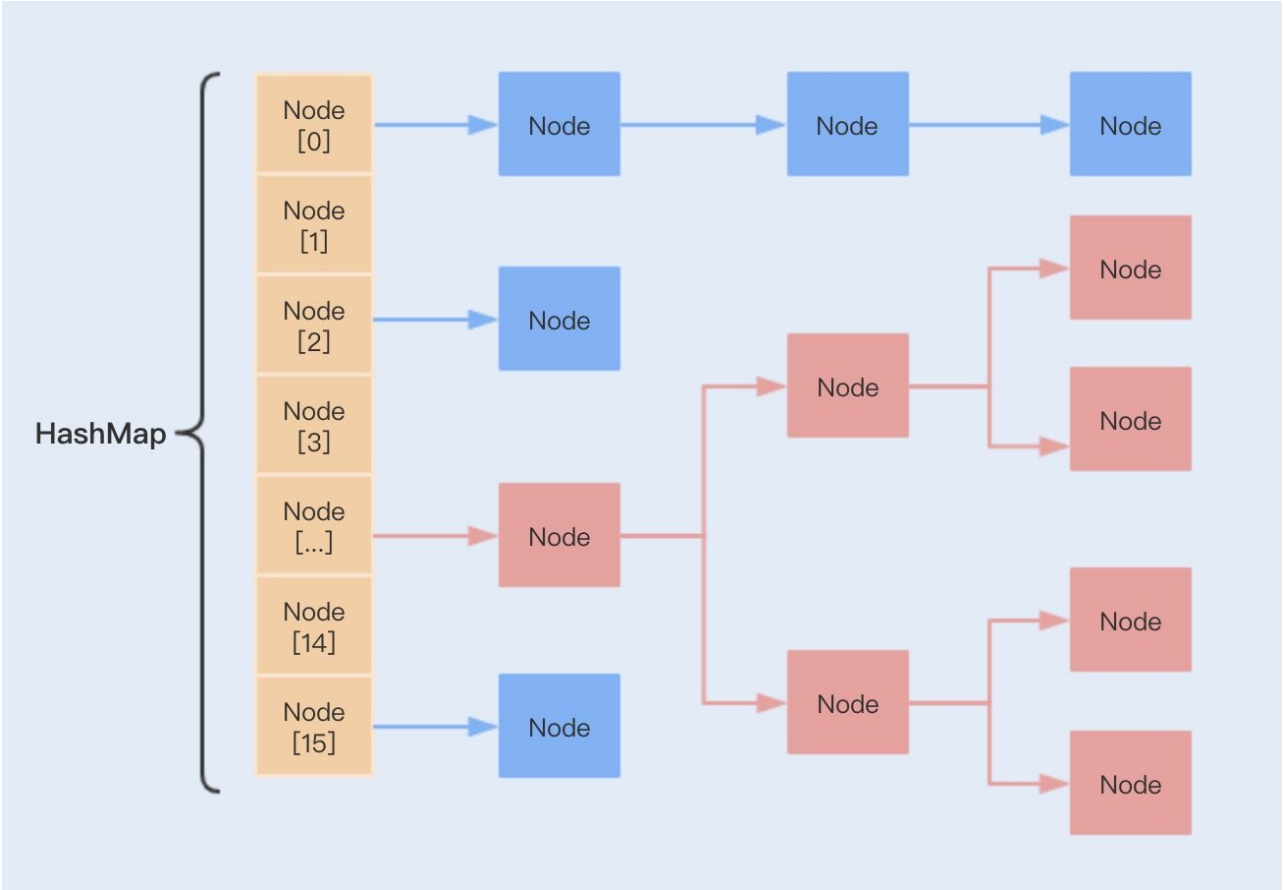
HashMap 通过哈希表数据结构的形式来存储键值对，这种设计的好处就是查询键值对的效率高。

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？

上一页 下一页