08 HashMap 源码解析

更新时间：2019-09-27 12:52:38

> 自信和希望是青年的特权。
>
> ——大仲马

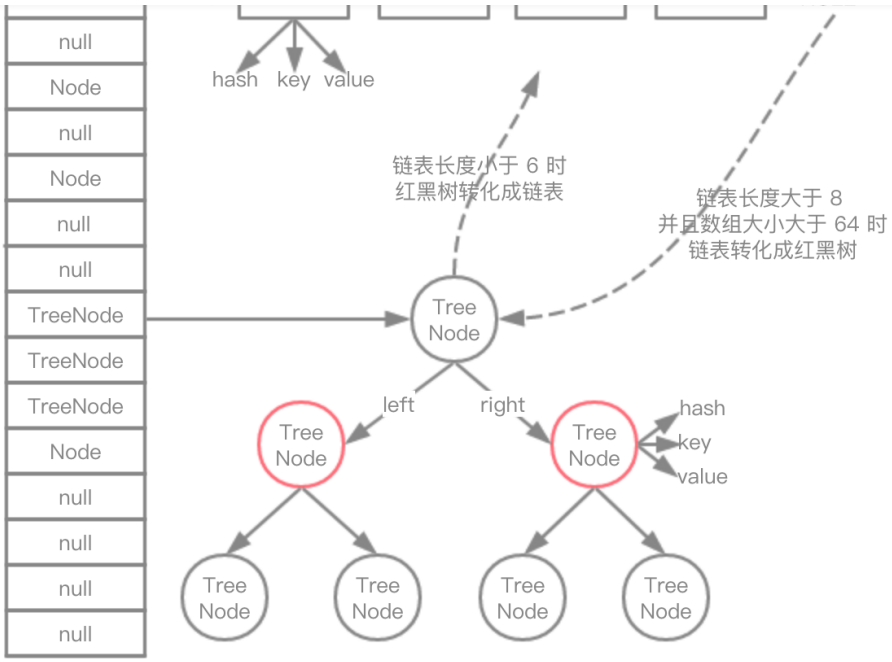
引导语

HashMap 源码很长，面试的问题也非常多，但这些面试问题，基本都是从源码中衍生出来的，所以我们只需要弄清楚其底层实现原理，回答这些问题就会游刃有余。

1 整体架构

HashMap 底层的数据结构主要是：数组 + 链表 + 红黑树。其中当链表的长度大于等于 8 时，链表会转化成红黑树，当红黑树的大小小于等于 6 时，红黑树会转化成链表，整体的数据结构如下：

目录



图中左边竖着的是 HashMap 的数组结构，数组的元素可能是单个 Node，也可能是个链表，也可能是个红黑树，比如数组下标索引为 2 的位置就是一个链表，下标索引为 9 的位置对应的就是红黑树，具体细节我们下文再说。

1.1 类注释

从 HashMap 的类注释中，我们可以得到如下信息：

- 允许 null 值，不同于 HashTable ，是线程不安全的；
- load factor（影响因子） 默认值是 0.75， 是均衡了时间和空间损耗算出来的值，较高的值会减少空间开销（扩容减少，数组大小增长速度变慢），但增加了查找成本（hash 冲突增加，链表长度变长），不扩容的条件：数组容量 > 需要的数组大小 /load factor；
- 如果有很多数据需要储存到 HashMap 中，建议 HashMap 的容量一开始就设置成足够的大小，这样可以防止在其过程中不断的扩容，影响性能；
- HashMap 是非线程安全的，我们可以自己在外部加锁，或者通过 Collections#synchronizedMap 来实现线程安全，Collections#synchronizedMap 的实现是在每个方法上加上了 synchronized 锁；
- 在迭代过程中，如果 HashMap 的结构被修改，会快速失败。

1.2 常见属性

```
//初始容量为 16
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;

//最大容量
static final int MAXIMUM_CAPACITY = 1 << 30;

//负载因子默认值
static final float DEFAULT_LOAD_FACTOR = 0.75f;

//桶上的链表长度大于等于8时，链表转化成红黑树
static final int TREEIFY_THRESHOLD = 8;

//桶上的红黑树大小小于等于6时，红黑树转化成链表
static final int UNTREEIFY_THRESHOLD = 6;
```

目录

```
//记录迭代过程中 HashMap 结构是否发生变化，如果有变化，迭代时会 fail-fast
transient int modCount;

//HashMap 的实际大小，可能不准(因为当你拿到这个值的时候，可能又发生了变化)
transient int size;

//存放数据的数组
transient Node<K,V>[] table;

// 扩容的门槛，有两种情况
// 如果初始化时，给定数组大小的话，通过 tableSizeFor 方法计算，数组大小永远接近于 2 的幂次方
// 如果是通过 resize 方法进行扩容，大小 = 数组容量 * 0.75
int threshold;

//链表的节点
static class Node<K,V> implements Map.Entry<K,V> {

//红黑树的节点
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
```

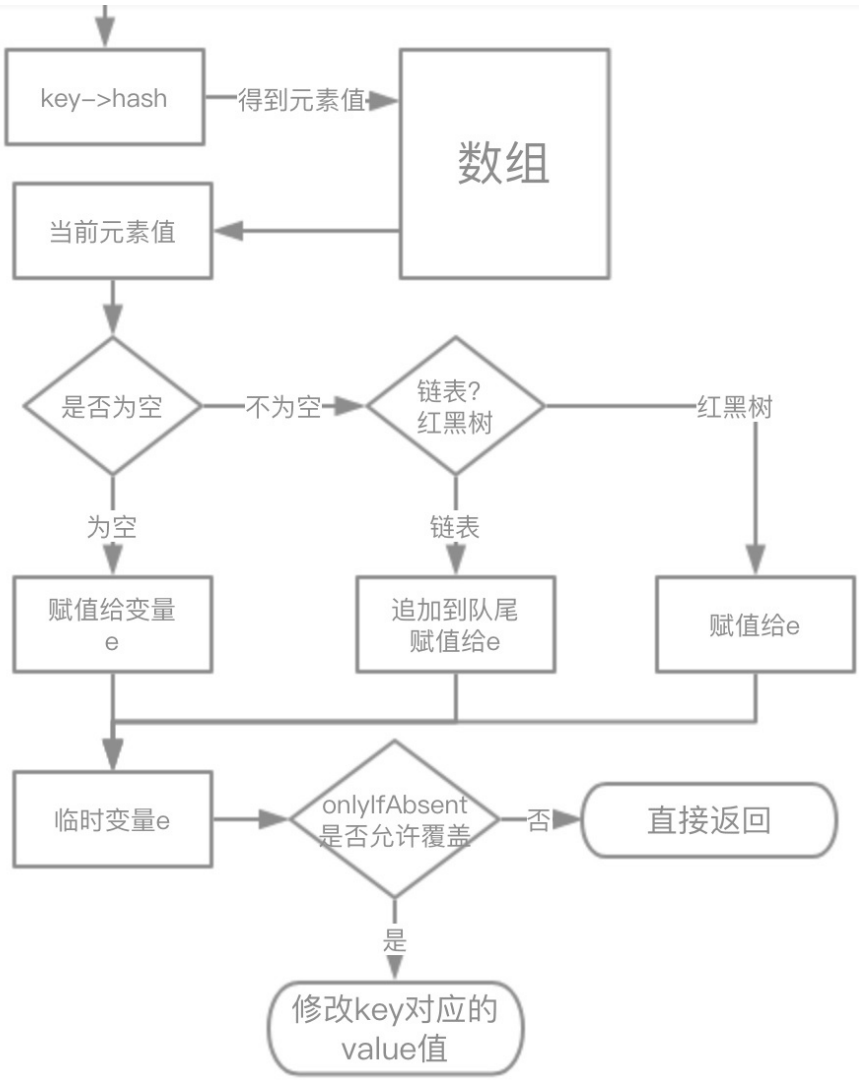
2 新增

新增 key，value 大概的步骤如下：

1. 空数组有无初始化，没有的话初始化；
2. 如果通过 key 的 hash 能够直接找到值，跳转到 6，否则到 3；
3. 如果 hash 冲突，两种解决方案：链表 or 红黑树；
4. 如果是链表，递归循环，把新元素追加到队尾；
5. 如果是红黑树，调用红黑树新增的方法；
6. 通过 2、4、5 将新元素追加成功，再根据 onlyIfAbsent 判断是否需要覆盖；
7. 判断是否需要扩容，需要扩容进行扩容，结束。

我们来画一张示意图来描述下：

目录



代码细节如下：

```
// 入参 hash：通过 hash 算法计算出来的值。
// 入参 onlyIfAbsent：false 表示即使 key 已经存在了，仍然会用新值覆盖原来的值，默认为 false
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    // n 表示数组的长度，i 为数组索引下标，p 为 i 下标位置的 Node 值
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    //如果数组为空，使用 resize 方法初始化
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 如果当前索引位置是空的，直接生成新的节点在当前索引位置上
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    // 如果当前索引位置有值的处理方法，即我们常说的如何解决 hash 冲突
    else {
        // e 当前节点的临时变量
        Node<K,V> e; K k;
        // 如果 key 的 hash 和值都相等，直接把当前下标位置的 Node 值赋值给临时变量
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        // 如果是红黑树，使用红黑树的方式新增
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 是个链表，把新节点放到链表的尾端
        else {
```

目录

```
                // p.next == null 表明 p 是链表的尾节点
                if ((e = p.next) == null) {
                    // 把新节点放到链表的尾部
                    p.next = newNode(hash, key, value, null);
                    // 当链表的长度大于等于 8 时，链表转红黑树
                    if (binCount >= TREEIFY_THRESHOLD - 1)
                        treeifyBin(tab, hash);
                    break;
                }
                // 链表遍历过程中，发现有元素和新增的元素相等，结束循环
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                //更改循环的当前元素，使 p 在遍历过程中，一直往后移动。
                p = e;
            }
        }
        // 说明新节点的新增位置已经找到了
        if (e != null) {
            V oldValue = e.value;
            // 当 onlyIfAbsent 为 false 时，才会覆盖值
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            // 返回老值
            return oldValue;
        }
    }
    // 记录 HashMap 的数据结构发生了变化
    ++modCount;
    //如果 HashMap 的实际大小大于扩容的门槛，开始扩容
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

新增的流程上面应该已经表示很清楚了，接下来我们来看看链表和红黑树新增的细节：

2.1 链表的新增

链表的新增比较简单，就是把当前节点追加到链表的尾部，和 LinkedList 的追加实现一样的。

当链表长度大于等于 8 时，此时的链表就会转化成红黑树，转化的方法是：treeifyBin，此方法有一个判断，当链表长度大于等于 8，并且整个数组大小大于 64 时，才会转成红黑树，当数组大小小于 64 时，只会触发扩容，不会转化成红黑树，转化成红黑树的过程也比较简单，具体转化的过程源码可以去 github：https://github.com/luanqiu/java8 上面去查看。

可能面试的时候，有人问你为什么是 8，这个答案在源码中注释有说，中文翻译过来大概的意思是：

链表查询的时间复杂度是 O (n)，红黑树的查询复杂度是 O (log (n))。在链表数据不多的时候，使用链表进行遍历也比较快，只有当链表数据比较多的时候，才会转化成红黑树，但红黑树需要的占用空间是链表的 2 倍，考虑到转化时间和空间损耗，所以我们需要定义出转化的边界值。

在考虑设计 8 这个值的时候，我们参考了泊松分布概率函数，由泊松分布中得出结论，链表各个长度的命中概率为：

```
* 2:    0.07581633
* 3:    0.01263606
* 4:    0.00157952
* 5:    0.00015795
* 6:    0.00001316
* 7:    0.00000094
* 8:    0.00000006
```

意思是，当链表的长度是 8 的时候，出现的概率是 0.00000006，不到千万分之一，所以说正常情况下，链表的长度不可能到达 8 ，而一旦到达 8 时，肯定是 hash 算法出了问题，所以在这种情况下，为了让 HashMap 仍然有较高的查询性能，所以让链表转化成红黑树，我们正常写代码，使用 HashMap 时，几乎不会碰到链表转化成红黑树的情况，毕竟概念只有千万分之一。

2.2 红黑树新增节点过程

1. 首先判断新增的节点在红黑树上是不是已经存在，判断手段有如下两种：

   1.1. 如果节点没有实现 Comparable 接口，使用 equals 进行判断；

   1.2. 如果节点自己实现了 Comparable 接口，使用 compareTo 进行判断。

2. 新增的节点如果已经在红黑树上，直接返回；不在的话，判断新增节点是在当前节点的左边还是右边，左边值小，右边值大；

3. 自旋递归 1 和 2 步，直到当前节点的左边或者右边的节点为空时，停止自旋，当前节点即为我们新增节点的父节点；

4. 把新增节点放到当前节点的左边或右边为空的地方，并于当前节点建立父子节点关系；

5. 进行着色和旋转，结束。

具体源码如下：

```
//入参 h：key 的hash值
final TreeNode<K,V> putTreeVal(HashMap<K,V> map, Node<K,V>[] tab,
                               int h, K k, V v) {
    Class<?> kc = null;
    boolean searched = false;
    //找到根节点
    TreeNode<K,V> root = (parent != null) ? root() : this;
    //自旋
    for (TreeNode<K,V> p = root;;) {
        int dir, ph; K pk;
        // p hash 值大于 h，说明 p 在 h 的右边
        if ((ph = p.hash) > h)
            dir = -1;
        // p hash 值小于 h，说明 p 在 h 的左边
        else if (ph < h)
            dir = 1;
        //要放进去key在当前树中已经存在了(equals来判断)
        else if ((pk = p.key) == k || (k != null && k.equals(pk)))
            return p;
        //自己实现的Comparable的话，不能用hashcode比较了，需要用compareTo
        else if ((kc == null &&
                  //得到key的Class类型，如果key没有实现Comparable就是null
```

目录

```
            if (!searched) {
                TreeNode<K,V> q, ch;
                searched = true;
                if (((ch = p.left) != null &&
                     (q = ch.find(h, k, kc)) != null) ||
                    ((ch = p.right) != null &&
                     (q = ch.find(h, k, kc)) != null))
                    return q;
            }
            dir = tieBreakOrder(k, pk);
        }

        TreeNode<K,V> xp = p;
        //找到和当前hashcode值相近的节点(当前节点的左右子节点其中一个为空即可)
        if ((p = (dir <= 0) ? p.left : p.right) == null) {
            Node<K,V> xpn = xp.next;
            //生成新的节点
            TreeNode<K,V> x = map.newTreeNode(h, k, v, xpn);
            //把新节点放在当前子节点为空的位置上
            if (dir <= 0)
                xp.left = x;
            else
                xp.right = x;
            //当前节点和新节点建立父子，前后关系
            xp.next = x;
            x.parent = x.prev = xp;
            if (xpn != null)
                ((TreeNode<K,V>)xpn).prev = x;
            //balanceInsertion 对红黑树进行着色或旋转，以达到更多的查找效率，着色或旋转的几种场
            //着色：新节点总是为红色；如果新节点的父亲是黑色，则不需要重新着色；如果父亲是红色
            //旋转： 父亲是红色，叔叔是黑色时，进行旋转
            //如果当前节点是父亲的右节点，则进行左旋
            //如果当前节点是父亲的左节点，则进行右旋

            //moveRootToFront 方法是把算出来的root放到根节点上
            moveRootToFront(tab, balanceInsertion(root, x));
            return null;
        }
    }
}
```

红黑树的新增，要求大家对红黑树的数据结构有一定的了解。面试的时候，一般只会问到新增节点到红黑树上大概是什么样的一个过程，着色和旋转的细节不会问，因为很难说清楚，但我们要清楚着色指的是给红黑树的节点着上红色或黑色，旋转是为了让红黑树更加平衡，提高查询的效率，总的来说都是为了满足红黑树的 5 个原则：

1. 节点是红色或黑色
2. 根是黑色
3. 所有叶子都是黑色
4. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点
5. 从每个叶子到根的所有路径上不能有两个连续的红色节点

3 查找

HashMap 的查找主要分为以下三步：

- 判断当前节点有无 next 节点，有的话判断是链表类型，还是红黑树类型。
- 分别走链表和红黑树不同类型的查找方法。

链表查找的关键代码是：

```
// 采用自旋方式从链表中查找 key，e 初始为为链表的头节点
do {
    // 如果当前节点 hash 等于 key 的 hash，并且 equals 相等，当前节点就是我们要找的节点
    // 当 hash 冲突时，同一个 hash 值上是一个链表的时候，我们是通过 equals 方法来比较 key 是否
    if (e.hash == hash &&
        ((k = e.key) == key || (key != null && key.equals(k))))
        return e;
    // 否则，把当前节点的下一个节点拿出来继续寻找
} while ((e = e.next) != null);
```

红黑树查找的代码很多，我们大概说下思路，实际步骤比较复杂，可以去 github 上面去查看源码：

1. 从根节点递归查找；
2. 根据 hashcode，比较查找节点，左边节点，右边节点之间的大小，根本红黑树左小右大的特性进行判断；
3. 判断查找节点在第 2 步有无定位节点位置，有的话返回，没有的话重复 2，3 两步；
4. 一直自旋到定位到节点位置为止。

如果红黑树比较平衡的话，每次查找的次数就是树的深度。

总结

HashMap 的内容虽然较多，但大多数 api 都只是对数组 + 链表 + 红黑树这种数据结构进行封装而已，本小节我们从新增和查找两个角度进行了源码的深入分析，分析了是如何对数组、链表和红黑树进行操作的。想了解更多，可以前往 github 上查看更多源码。

← 07 List 源码会问哪些面试题

09 TreeMap 和 LinkedHashMap 核心源码解析 →

精选留言 29

欢迎在这里发表留言，作者筛选后可公开显示

qq_起风了_90

老师，怎么debug扩容过程啊，我在hashmap的构造函数里面都给定了负载因子和容量，但程序运行的过程，没有体现出扩容啊。

0 回复 2019-12-07

allen平凡之路

目录

文贺 回复 **allen平凡之路**

同学你好，具体是哪里有误呢？数组下标从0开始是常识哈，HashMap 计算数组下标主要是通过 Hash 算法计算出的哈，不一定从 0 开始有值。

回复 7天前

qq_起风了_90

老师，resize的过程中，这段代码： // 原索引 if ((e.hash & oldCap) == 0) { if (loTail == null) loHead = e; else loTail.next = e; loTail = e; } 中的e.hash & oldCap==0，就判断是原索引，不是应是新索引吗，

0 回复 2019-11-30

阿岑XD

老师 链表中的值应该如何去取

0 回复 2019-11-27

qq_起风了_90 回复 **阿岑XD**

同问

回复 2019-11-29 08:08:43

文贺 回复 **阿岑XD**

递归遍历寻找，可以参考 getTreeNode 方法

回复 2019-11-30 13:23:42

AntChenxi

老师，文档里面怎么都不讲tableSizeFor、resize等等这些方法呢?

0 回复 2019-11-25

文贺 回复 **AntChenxi**

源码注释很早就提交了，github 上有的，文章的篇幅有限。

回复 2019-11-30 13:20:41

凉凉那个凉凉 回复 **文贺**

github 地址哪里有?

回复 9天前

慕先生9458666

老师，能不能讲讲hashMap 的扩容 resize

0 回复 2019-11-25

文贺 回复 **慕先生9458666**

源码注释很早就提交了，github 上有的，文章的篇幅有限。

回复 2019-11-30 13:20:38

目录

看起来有些难懂，尤其是源码

0 回复 2019-11-23

文贺 回复 **慕村6418685**

同学你说的很有道理，如果仅仅是看的话，估计永远都看不懂，动起手吧，debug！

回复 2019-11-30 13:24:19

大LOVE辉

不覆盖，就不插入了吗？？e就没用了呢？

0 回复 2019-11-22

大LOVE辉

老师，在插入的时候，是不是没有对应哈西情况下，就插入到数组下面呢？？

0 回复 2019-11-22

文贺 回复 **大LOVE辉**

没有太看明白问题哈，这个可以看一下 put 方法的过程，可以下载源码看下注释，很详细的

回复 2019-11-23 16:35:22

大LOVE辉 回复 **文贺**

老师这个问题我看明白了，还有第二个问题：是不是数据的数量是一直在记录，比如有30个值插入进来，其中有九个哈希冲突，这个咋办呢，是扩容吗，扩容后重新排列？不可以用转红黑树的情况，因为他64还没到。

回复 2019-11-23 16:42:16

文贺 回复 **大LOVE辉**

如果你使用 HashMap 的API 的话，这种场景几乎不可能出现，因为 HashMap 自带的算法的 Hash 冲突概念很小哈。如果是自己写的 Hash 算法，建议修改下算法。 假设你这种场景出现在 JDK 的 HashMap 中，按照源码来说，小于 64 时，链表会扩容，而不转化成红黑树。

回复 2019-11-23 16:47:50

点击展开后面 1 条

qq_呀个呸的_0

老师您好，看第一张图片中写道 链表长度小于6时，红黑树转换成链表，这就不大理解？ 如果从红黑树转换成链表，是在哪个方法中调用的呢？举个例子：比如移除元素时，在remove()方法里有对应的检测，然后进行一个转换，但是我也没找到呢？请老师讲解一下吧，这是面试中问到的

0 回复 2019-11-16

payzul 回复 **qq_呀个呸的_0**

在split方法中，这个方法只有在resize方法里被调用的。

回复 2019-11-18 14:16:11

文贺 回复 **qq_呀个呸的_0**

remove 方法里面是没有这个的，是在 resize 方法里面，看源码的时候，可以根据UNTREEIFY_THRESHOLD快速定位到源码位置。

回复 2019-11-23 16:33:56

目录

qq_呀个呸的_0

请问老师，往haspmap放数据时，取的hash值对数组长度取模为零的，是放到table[0]么？concurrenthashmap中是如何实现size的，是否安全呢？求解答

0 回复 2019-11-13

文贺 回复 **qq_呀个呸的_0**

是的，通过 hash 算法计算出数组的索引下标，如果是0的话，就表示在 table 的0 的索引下标

点击展开剩余评论

千学不如一看，千看不如一练