深入分析 HashMap

一、传统 HashMap 的缺点

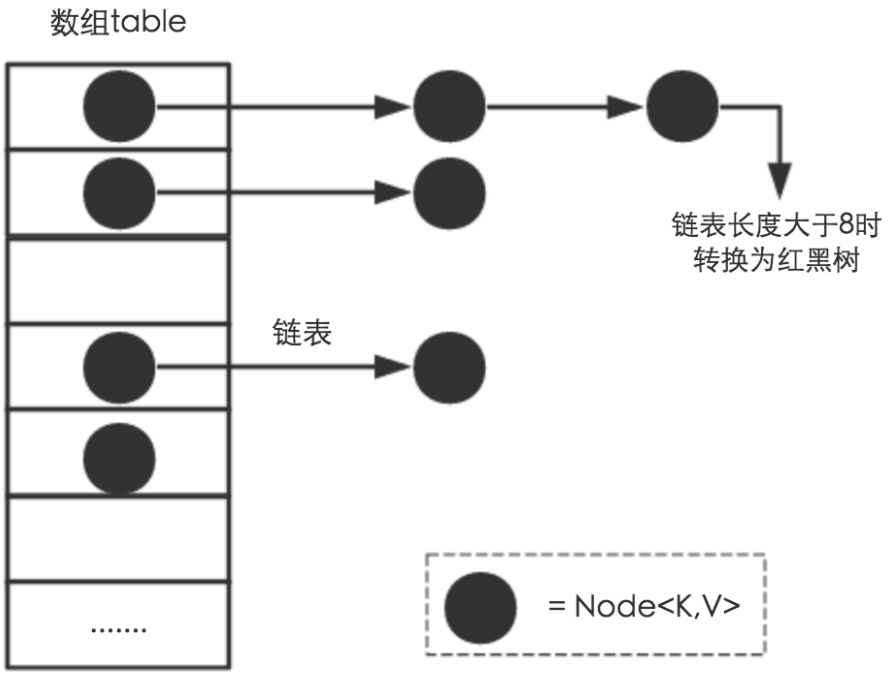
(1)JDK 1.8 以前 HashMap 的实现是 数组+链表，即使哈希函数取得再好，也很难达到元素百分百均匀分布。

(2)当 HashMap 中有大量的元素都存放到同一个桶中时，这个桶下有一条长长的链表，这个时候 HashMap 就相当于一个单链表，假如单链表有 n 个元素，遍历的时间复杂度就是 O(n)，完全失去了它的优势。

(3)针对这种情况，JDK 1.8 中引入了红黑树（查找时间复杂度为 O(logn)）来优化这个问题

二、JDK1.8 中 HashMap 的数据结构

2.1HashMap 是数组+链表+红黑树（JDK1.8 增加了红黑树部分）实现的



新增红黑树

```
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent;  // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;    // needed to unlink next upon deletion
```

```
    boolean red;
}
```

2.2HashMap 中关于红黑树的三个关键参数

TREEIFY_THRESHOLD 一个桶的树化阈值	UNTREEIFY_THRESHOLD 一个树的链表还原阈值
static final int TREEIFY_THRESHOLD = 8	static final int UNTREEIFY_THRESHOLD = 6
当桶中元素个数超过这个值时 需要使用红黑树节点替换链表节点	当扩容时，桶中元素个数小于这个值 就会把树形的桶元素 还原（切分）为链表结构
MIN_TREEIFY_CAPACITY 哈希表的最小树形化容量	
static final int MIN_TREEIFY_CAPACITY = 64	
当哈希表中的容量大于这个值时，表中的桶才能进行树形化 否则桶内元素太多时会扩容，而不是树形化 为了避免进行扩容、树形化选择的冲突，这个值不能小于 4 * TREEIFY_THRESHOLD	

2.3HashMap 在 JDK 1.8 中新增的操作：桶的树形化 treeifyBin()

在 **Java** 8 中，如果一个桶中的元素个数超过 TREEIFY_THRESHOLD(默认是 8)，就使用红黑树来替换链表，从而提高速度。

这个替换的方法叫 treeifyBin() 即树形化。

```
//将桶内所有的 链表节点 替换成 红黑树节点
1 final void treeifyBin(Node<K,V>[] tab, int hash) {
2   int n, index; Node<K,V> e;
3 //如果当前哈希表为空，或者哈希表中元素的个数小于 进行树形化的阈值(默认为 64)，就去新建/扩容
4   if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
5       resize();
6    else if ((e = tab[index = (n - 1) & hash]) != null) {
7       //如果哈希表中的元素个数超过了 树形化阈值，进行树形化
8       // e 是哈希表中指定位置桶里的链表节点，从第一个开始
9       TreeNode<K,V> hd = null, tl = null; //红黑树的头、尾节点
10       do {
11           //新建一个树形节点，内容和当前链表节点 e 一致
12           TreeNode<K,V> p = replacementTreeNode(e, null);
13           if (tl == null) //确定树头节点
14               hd = p;
15          else {
16              p.prev = tl;
17               tl.next = p;
18           }
```

```
19            tl = p;
20        } while ((e = e.next) != null);
21        //让桶的第一个元素指向新建的红黑树头结点，以后这个桶里的元素就是红黑树而
不是链表了
22        if ((tab[index] = hd) != null)
23            hd.treeify(tab);
24    }
25 }
26    TreeNode<K,V> replacementTreeNode(Node<K,V> p, Node<K,V> next) {
27    return new TreeNode<>(p.hash, p.key, p.value, next);
28 }
```

上述操作做了这些事:

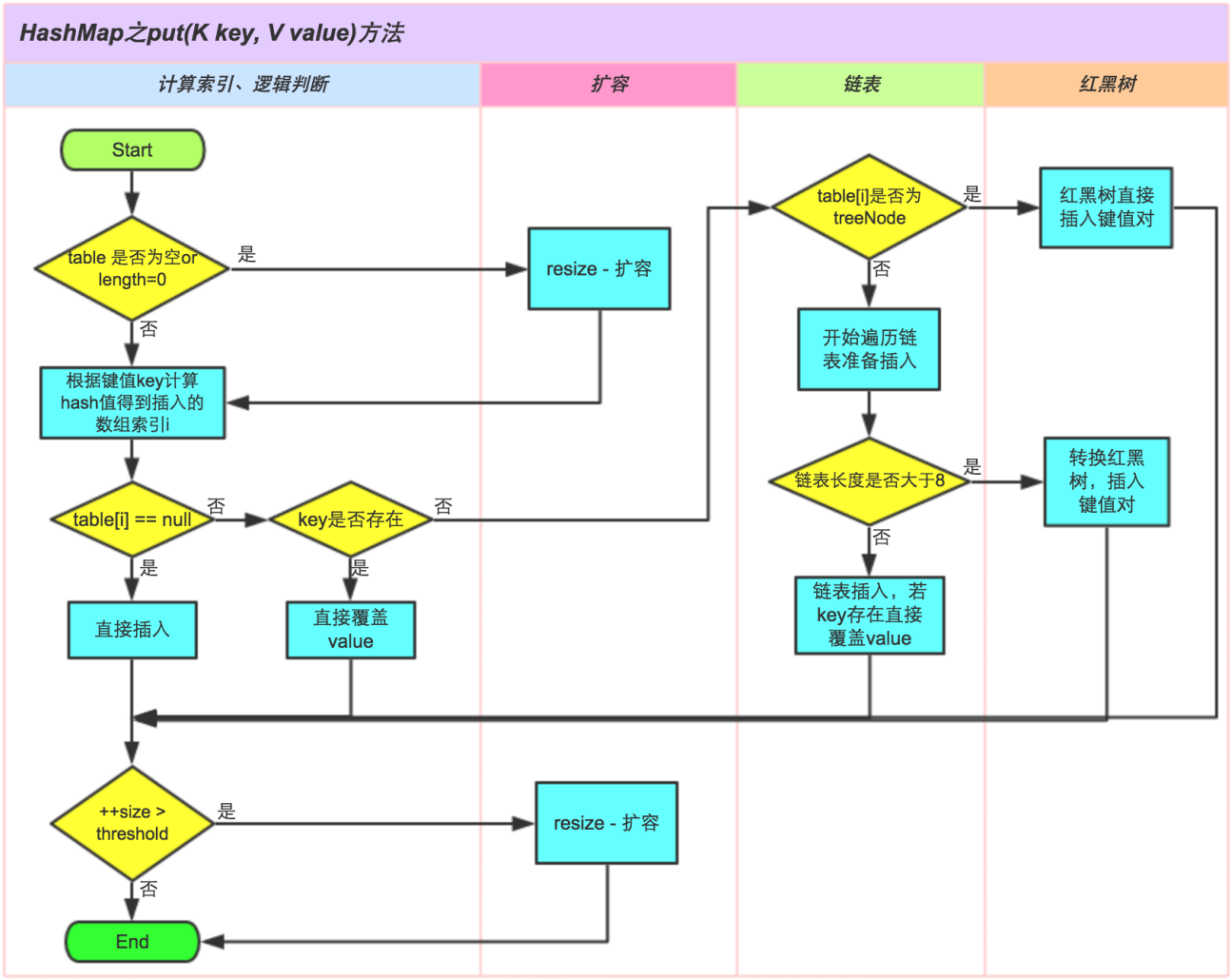
(1)根据哈希表中元素个数确定是扩容还是树形化

(2)如果是树形化遍历桶中的元素，创建相同个数的树形节点，复制内容，建立起联系

(3)然后让桶第一个元素指向新建的树头结点，替换桶的链表内容为树形内容

三、分析 HashMap 的 put 方法

3.1HashMap 的 put 方法执行过程可以通过下图来理解，自己有兴趣可以去对比源码更清楚地研究学习。



①.判断键值对数组 table[i]是否为空或为 null，否则执行 resize()进行扩容；

②.根据键值 key 计算 hash 值得到插入的数组索引 i，如果 table[i]==null，直接新建节点添加，转向⑥，如果 table[i]不为空，转向③；

③.判断 table[i]的首个元素是否和 key 一样，如果相同直接覆盖 value，否则转向④，这里的相同指的是 hashCode 以及 equals；

④.判断 table[i] 是否为 treeNode，即 table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历 table[i]，判断链表长度是否大于 8，大于 8 的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现 key 已经存在直接覆盖 value 即可；

⑥.插入成功后，判断实际存在的键值对数量 size 是否超多了最大容量 threshold，如果超过，进行扩容。

JDK1.8HashMap 的 put 方法源码

```
1 public V put(K key, V value) {
 2     // 对 key 的 hashCode()做 hash
```

```
    return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 步骤①：tab 为空则创建
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 步骤②：计算 index，并对 null 做处理
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        // 步骤③：节点 key 存在，直接覆盖 value
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        // 步骤④：判断该链为红黑树
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 步骤⑤：该链为链表
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key,value,null);
                        //链表长度大于 8 转换为红黑树进行处理
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                    // key 已经存在直接覆盖 value
                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);
```

```
45            return oldValue;
46        }
47    }
48    ++modCount;
49    // 步骤⑥：超过最大容量 就扩容
50    if (++size > threshold)
51        resize();
52    afterNodeInsertion(evict);
53    return null;
54 }
```

JDK1.7HashMap 的 put 方法源码

```
1 public V put(K key, V value) {
2    if (table == EMPTY_TABLE) { //空表 table 的话，根据 size 的阈值填充
3       inflateTable(threshold);
4    }
5    if (key == null)
6        return putForNullKey(value);
7    int hash = hash(key);//成 hash，得到索引 Index 的映射
8    int i = indexFor(hash, table.length);
9    for (Entry<K,V> e = table[i]; e != null; e = e.next) {//遍历当前索引的冲突链，找是否存在对应的 key
10       Object k;
11       if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {//如果存在对应的 key， 则替换 oldValue 并返回 oldValue
12           V oldValue = e.value;
13           e.value = value;
14           e.recordAccess(this);
15           return oldValue;
16       }
17   }
18   //冲突链中不存在新写入的 Entry 的 key
19   modCount++;
20   addEntry(hash, key, value, i);
21   return null;
22 }
```

3.2HashMap 在 JDK 1.8 中新增的操作： 红黑树中查找元素 getTreeNode()

JDK1.8 中 hashMap getNode 操作

```
*/
1 final Node<K,V> getNode(int hash, Object key) {
2    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
3    if ((tab = table) != null && (n = tab.length) > 0 &&
4        (first = tab[(n - 1) & hash]) != null) {
5       if (first.hash == hash && // always check first node
```

```
6           ((k = first.key) == key || (key != null && key.equals(k))))
7           return first;
8       if ((e = first.next) != null) {
9           if (first instanceof TreeNode)
10              return ((TreeNode<K,V>)first).getTreeNode(hash, key);
11          do {
12              if (e.hash == hash &&
13                  ((k = e.key) == key || (key != null && key.equals(k))))
14                  return e;
15          } while ((e = e.next) != null);
16      }
17  }
18  return null;
19}
```

(1)HashMap 的查找方法是 get(),它通过计算指定 key 的哈希值后，调用内部方法 getNode()；

(2)这个 getNode() 方法就是根据哈希表元素个数与哈希值求模（使用的公式是 (n－1) &hash）得到 key 所在的桶的头结点，如果头节点恰好是红黑树节点，

就调用红黑树节点的 getTreeNode() 方法，否则就遍历链表节点。

(3)getTreeNode 方法使通过调用树形节点的 find()方法进行查找：

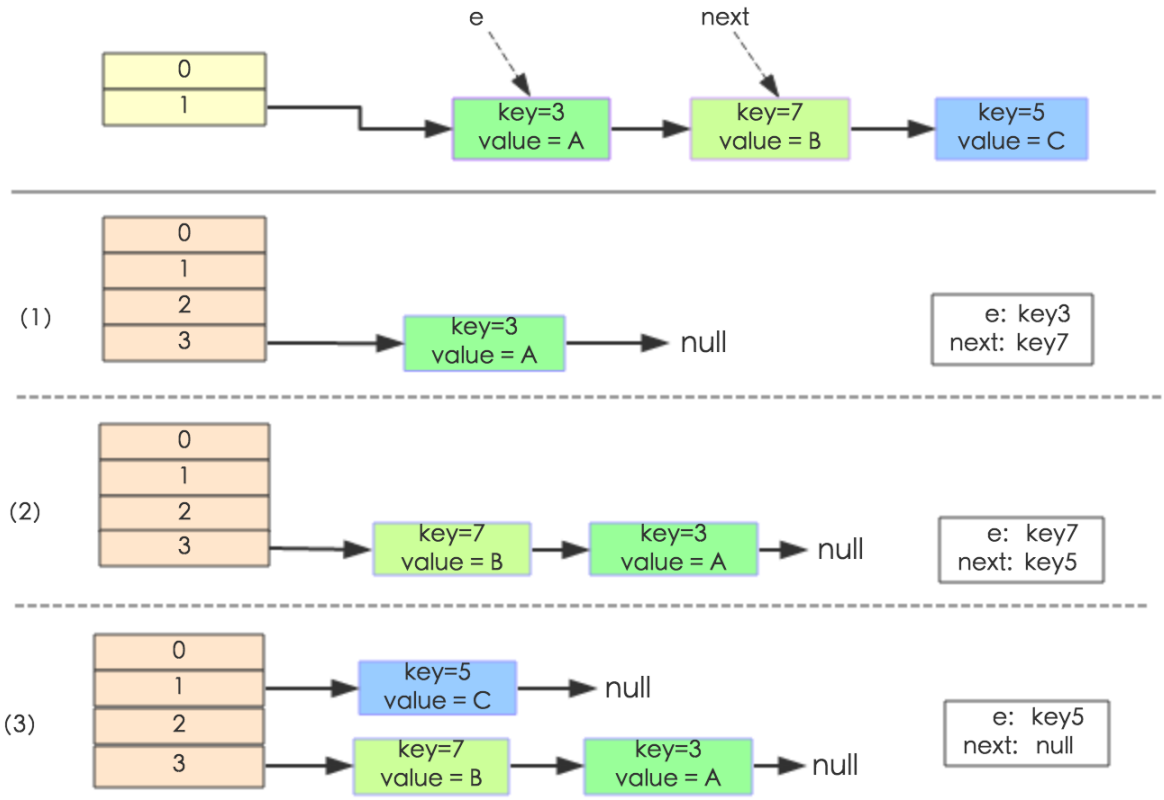
```
1 final TreeNode<K,V> getTreeNode(int h, Object k) {
2    return ((parent != null) ? root() : this).find(h, k, null);
3 }
```

(4)由于之前添加时已经保证这个树是有序的，因此查找时基本就是折半查找，效率很高。

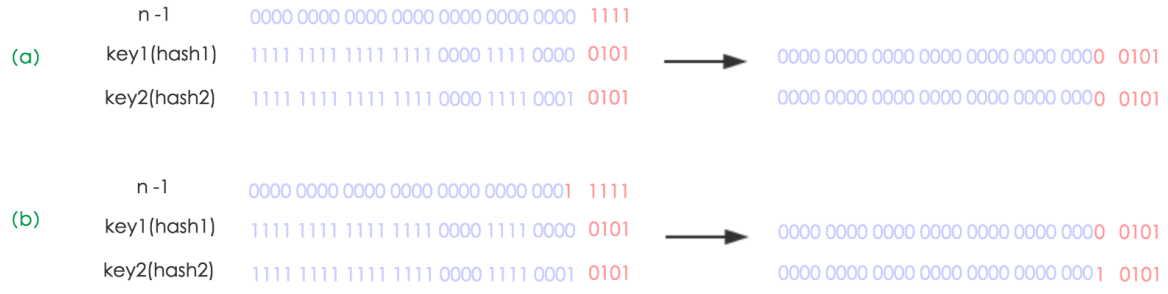
(5)这里和插入时一样，如果对比节点的哈希值和要查找的哈希值相等，就会判断 key 是否相等，相等就直接返回；不相等就从子树中递归查找。

3.3JDK1.8 VS JDK1.7 扩容机制

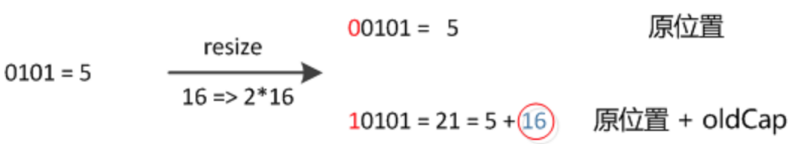
下面举个例子说明下扩容过程。假设了我们的 hash 算法就是简单的用 key mod 一下表的大小（也就是数组的长度）。其中的哈希桶数组 table 的 size=2， 所以 key = 3、7、5，put 顺序依次为 5、7、3。在 mod 2 以后都冲突在 table[1]这里了。这里假设负载因子 loadFactor=1，即当键值对的实际大小 size 大于 table 的实际大小时进行扩容。接下来的三个步骤是哈希桶数组 resize 成 4，然后所有的 Node 重新 rehash 的过程。



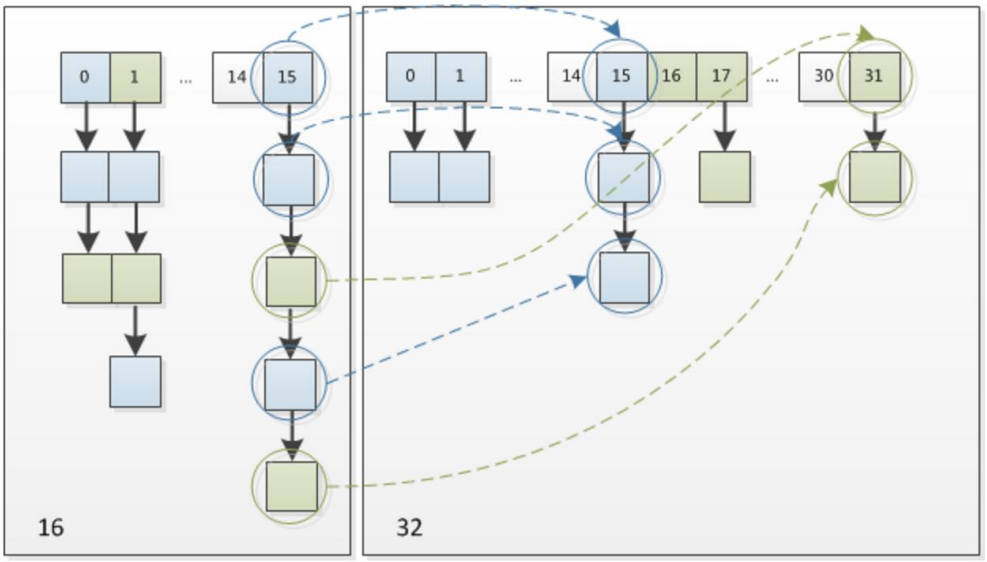
下面我们讲解下 JDK1.8 做了哪些优化。经过观测可以发现，我们使用的是 2 次幂的扩展(指长度扩为原来 2 倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动 2 次幂的位置。看下图可以明白这句话的意思，n 为 table 的长度，图（a）表示扩容前的 key1 和 key2 两种 key 确定索引位置的示例，图（b）表示扩容后 key1 和 key2 两种 key 确定索引位置的示例，其中 hash1 是 key1 对应的哈希与高位运算结果。



元素在重新计算 hash 之后，因为 n 变为 2 倍，那么 n-1 的 mask 范围在高位多 1bit(红色)，因此新的 index 就会发生这样的变化：



因此，我们在扩充 HashMap 的时候，不需要像 JDK1.7 的实现那样重新计算 hash，只需要看看原来的 hash 值新增的那个 bit 是 1 还是 0 就好了，是 0 的话索引没变，是 1 的话索引变成“原索引+oldCap”，可以看看下图为 16 扩充为 32 的 resize 示意图：



这个设计确实非常的巧妙，既省去了重新计算 hash 值的时间，而且同时，由于新增的 1bit 是 0 还是 1 可以认为是随机的，因此 resize 的过程，均匀的把之前的冲突的节点分散到新的 bucket 了。这一块就是 JDK1.8 新增的优化点。

四、JDK1.7 VS JDK1.8 的性能

4.1put 操作

1.hash 比较均匀的时候(负载因子时 0.75 导致的）

次数	10	100	1000	10000	100000
JDK1.7 时间(ns)	1100	720	832	914	912
JDK1.8 时间(ns)	1019	1023	1188	267	115

2.hash 不均匀的时候

次数	10	100	1000	10000	100000
次数	10	100	1000	10000	100000
JDK1.7 时间(ns)	2500	14310	8151	14137	154319
JDK1.8 时间(ns)	3765	38144	60707	1182	373

4.2get 操作

1.hash 比较均匀的时候

次数	10	100	1000	10000	100000
JDK1.7 时间(ns)	900	550	627	302	626
JDK1.8 时间(ns)	2773	1047	318	94	13

2hash 不均匀的时候

次数	10	100	1000	10000	100000

JDK1.7 时间(ns)	2000	14950	4294	2167	16447
JDK1.8 时间(ns)	3430	3932	2028	767	19

参考链接：

（1）红黑树的性质：http://blog.csdn.net/cyp331203/article/details/42677833

（2）JDK1.8 HashMap 性能的提升：http://blog.csdn.net/lc0817/article/details/48213435/