链滴

JCF-HashMap&HashSet

作者：wbq813
原文链接：https://ld246.com/article/1571219108785
来源网站：链滴

Hash Concepts

哈希表的本质是一个数组，数组中每一个元素称为一个箱子(bin)，箱子中存放的是键值对。

- **存储过程**:

  1. 根据Key计算出它的哈希值h；
  2. 假设箱子的个数为n，这个键值应该放在第(h%n)个箱子里；
  3. 如果该箱子中已经有了键值对，使用开放寻址法(Open hash)或者拉链法(closed hash)解决冲

。

- **拉链法**:

每个箱子是一个链表，属于同一个箱子的所有键值对都会排在链表中。

- **负载因子**:

负载因子=总键值对数/箱子个数，负载因子越大，哈希表越满，越容易导致冲突，性能越低。因此当载因子超过某个阈值(eg. 0.75)时就需要对哈希表进行扩容。

- **哈希表扩容&rehash**:

哈希表自动扩容时，一般会创建两倍于原来个数的箱子，因此即使key的哈希值h不变，对箱子个数取(h%n) 也会发生改变，这个过程也成为重哈希（rehash）。

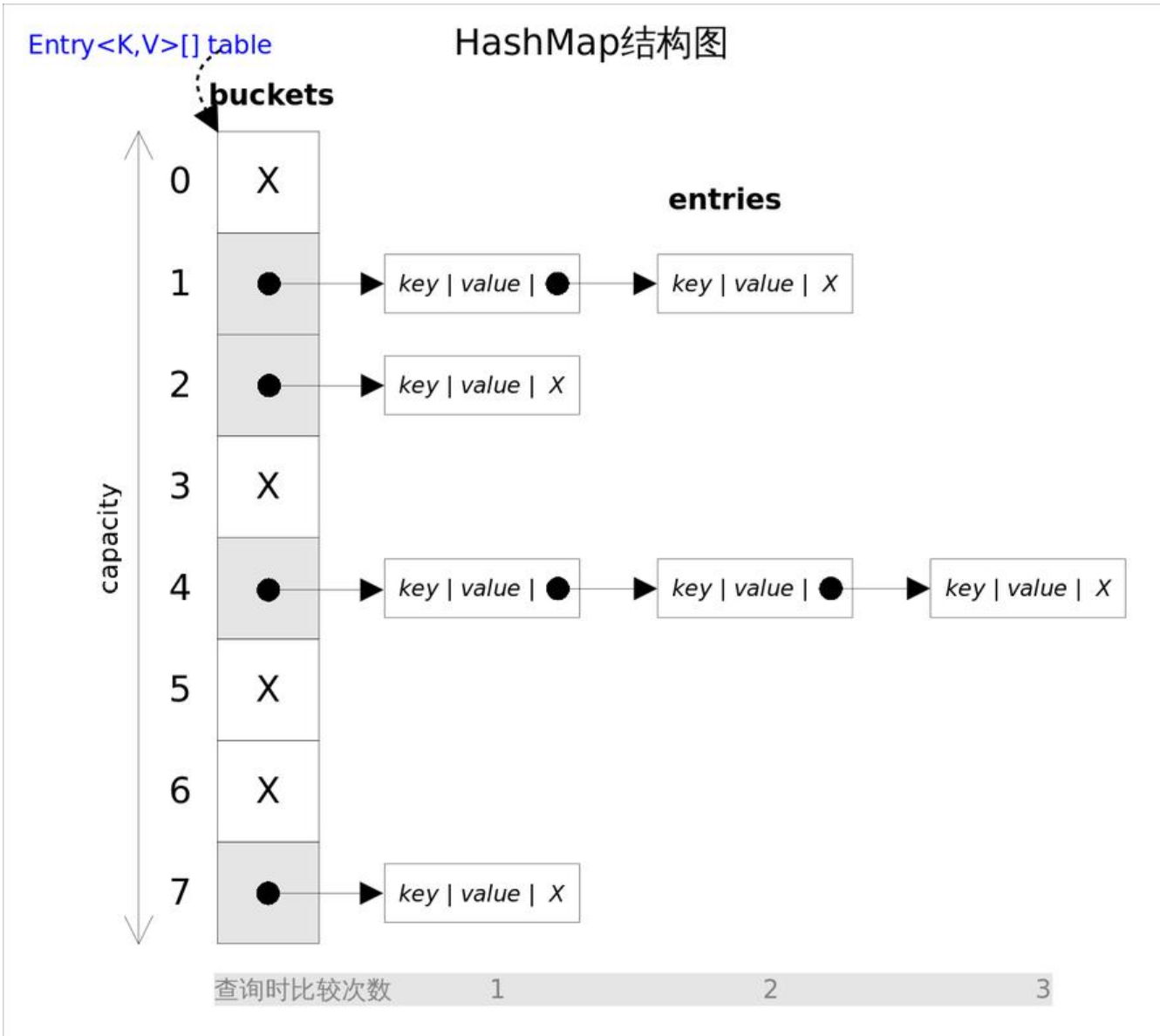
哈希表扩容表面上可以降低负载因子，但可能也无法解决链表过长的问题。假设所有key的哈希值h都样，即使扩容以后它们也分布在同一个bin的链表上，因此也不能提高性能。

- **可能的缺陷**:

  1. 如果hash表中本来箱子比较多，扩容时要重新hash并移动数据，性能影响较大。
  2. 如果hash函数设计不合理，hash表在极端情况下会退化成线性表，性能极低。

HashMap

原文链接：JCF-HashMap&HashSet



- **Important Variables**

```
// 初始容量16个bin，太少，容易触发扩容；太多，遍历hash表会比较慢
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
// hash 表最大容量
static final int MAXIMUM_CAPACITY = 1 << 30;
// 默认负载因子，这种情况键值对数量>16*0.75=12时就会触发扩容。
static final float DEFAULT_LOAD_FACTOR = 0.75f;
// 当某个bin中链表长度大于8时，就转化为树
static final int TREEIFY_THRESHOLD = 8;
// 在hash表扩容时，如果发现链表长度小于6则会将树退化回链表
static final int UNTREEIFY_THRESHOLD = 6;
// 在转变成树之前，还会判断，只有键值对数量>64才会发生转换
static final int MIN_TREEIFY_CAPACITY = 64;
transient Node<K,V>[] table;
transient Set<Map.Entry<K,V>> entrySet;
transient int size;
int threshold;
final float loadFactor;
```

上文说过，如果hash函数不合理，几是扩容也可能无法减少箱子中链表的长度，Java的处理方案是当表太长时，转换成红黑树。转换还有一个条件是：键值对数量大于 MIN_TREEIFY_CAPACITY。这是了避免在hash表建立初期，多个键值对七号被放入了同一个链表中，而导致不必要的转化。

8是如何确定的： （Time-Space Trade-off）TreeNodes占用空间是普通Nodes的两倍，所以只有当in包含足够多的节点时才会转成TreeNodes，而是否足够多就是由TREEIFY_THRESHOLD的值决定的当bin中节点数变少时，又会转成普通的bin。当hashCode离散性很好的时候，数据均匀分布在每个bi中，几乎不会有bin中链表长度会达到阈值，树型bin用到的概率非常小。

理想情况下，随机hashCode算法下所有bin中节点的分布频率会遵循**泊松分布**，P(k)=\frac{\lambda k e^{-\lambda}}{k!} 我们可以看到，一个bin中链表长度达到8个元素的概率为0.00000006，几乎是可能事件。

默认resizing threshold 0.75，得到 \lambda=0.5 (？？？？)，尽管由于调整大小粒度而差异很大。忽略方差，列表大小k的预期出现次数是（exp（-0.5）* pow（0.5，k）/ factorial（k））。

| :--: | :--: | :--: | :--: | :--: | :--: | :--: | :--: | :--: |

| 数量| 0 |1 | 2 |3 |4 |5 | 6 |7 | 8 |

| 概率| 0.60653066 | 0.30326533 | 0.07581633 |0.01263606 | 0.00157952 | 0.00015795 | 0.0000316 | 0.00000094 | 0.00000006 |

然而JDK不能阻止用户实现不好的hash算法，只能使用红黑树做应急处理。

- **Main Methods**

  - **get()**

get() 首先调用hash(key)得到hash值，然后利用getNode方法获取node,然后返回node.value。算思想是首先通过hash()函数得到对应bin的下标，然后依次遍历冲突链表，通过key.equals(k)方法来断是否是要找的那个bin。

hash(key)&(table.length-1) = hash(key)%table.length 因为table.length 始终时2的幂，因此table.ength-1二进制低位全是1，与hash(key)相与就会将hash值的高位抹掉，剩下的就是余数。

```
final Node<K,V> getNode(int hash, Object key) {
   Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
      // 根据hash得到对应的bin
if ((tab = table) != null && (n = tab.length) > 0 && (first = tab[(n - 1) & hash]) != null) {
     // bin的链表的第一个元素是否符合
    if (first.hash == hash && ((k = first.key) == key || (key != null && key.equals(k))))
          return first;
    if ((e = first.next) != null) {
   // 链表是红黑树
      if (first instanceof TreeNode)
          return ((TreeNode<K,V>)first).getTreeNode(hash, key);
   // 遍历链表
      do {
          if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
              return e;
       } while ((e = e.next) != null);
   }
}
return null;
}
```

  - **put()**

将key和value放入到map，通过调用putVal实现。找到对应的链表之后插入，这里jdk1.8是加入到链末尾，之前的版本网上说是在链表头部插入。（尚未验证？？？？）

```
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)
        n = (tab = resize()).length;
    // map中不存在当前元组，可直接插入
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {// 冲突的时候
        Node<K,V> e; K k;
        // bin的第一个节点p完全equal插入的元组
        if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        // 该bin已经是红黑树，只能继续加入
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    // 加入到链表尾部
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // 长度过长时变为红黑树
                        treeifyBin(tab, hash);
                    break;
                }
                // 当前bin的链表中存在与插入元组完全equal的情况
                if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // 存在与插入元组完全equal
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null) // 默认替换原来的value，onlyIfAbsent 是控制参数
                e.value = value;
                // 回调接口
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold) // 重新hash
        resize();
    // 回调接口
    afterNodeInsertion(evict);
    return null;
}
```

- **remove()**

remove(Object key)根据key删除对应的元组，这里调用 removeNode()实现

```
final Node<K,V> removeNode(int hash, Object key, Object value, boolean matchValue, boole
n movable) {
    Node<K,V>[] tab; Node<K,V> p; int n, index;
    if ((tab = table) != null && (n = tab.length) > 0 && (p = tab[index = (n - 1) & hash]) != null)
```

```
{
        Node<K,V> node = null, e; K k; V v;
        // 当前bin的第一个元组p就符合
        if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
            node = p;
        else if ((e = p.next) != null) {
    // 从红黑树中查找
            if (p instanceof TreeNode)
                node = ((TreeNode<K,V>)p).getTreeNode(hash, key);
            else {
                do {// 在链表中查找
                    if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))) {
                        node = e;
                        break;
                    }
                    p = e;
                } while ((e = e.next) != null);
            }
        }
        // 获取的node不为空并且(不用匹配值、值完全相等、值equal)
        if (node != null && (!matchValue || (v = node.value) == value || (value != null && value.e
uals(v)))) {
            if (node instanceof TreeNode) // 从红黑树移除
                ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);
            else if (node == p) // node是当前bin的第一个节点
                tab[index] = node.next;
            else // 是否存在问题？？p和node中间有节点的情况
                p.next = node.next;
            ++modCount;
            --size;
        // 回调函数
            afterNodeRemoval(node);
            return node;
        }
    }
    return null;
}
```

HashSet

HashSet里面有一个HashMap（适配器模式），所有方法都是转发到HashMap实现。

```
// 虚拟值
private static final Object PRESENT = new Object();
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

Summary

- 对于迭代比较频繁的场景，不宜将 HashMap的初始大小设的过大。

如上图所示，选择合适的哈希函数，put()和get()方法可以在常数时间内完成。但在对HashMap进行

代时，需要遍历整个table以及后面跟的冲突链表。

- 对于插入元素较多的场景，将初始容量设大可以减少重新哈希的次数。

有两个参数可以影响HashMap的性能：初始容量（inital capacity）和负载系数（load factor）。初容量指定了初始table的大小，负载系数用来指定自动扩容的临界值。当bin的数量超过capacity*load_actor时，容器将自动扩容并重新哈希。

- 要将自定义的对象放入到 HashMap或HashSet中，需要@Override hashCode()和equals()方法。

将对象放入到HashMap或HashSet中时，有两个方法需要注意：hashCode()和equals()。hashCode()法决定了对象会被放到哪个bin里，当多个对象的哈希值冲突时，equals()方法决定了这些对象是否是同一个对象”。

Reference

- TODO redis https://bestswifter.com/hashtable/