HashMap 源码分析

HashMap 是 Java 面试必考的知识点，面试官从这个小知识点就可以了解我们对 Java 基础的掌握程度。网上的源码分析总结太多太多了，现饭炒了三遍也还是要吃的，所以我在这里做一些整理和总结分享给大家，也便于自己为明年实习生面试做准备。

HashMap 简介

一句话概括之：HashMap 是一个散列表，它存储的内容是键值对 (key-value) 映射。

它根据键的 hashCode 值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。

HashMap 最多只允许一条记录的键为 null，允许多条记录的值为 null。

HashMap 使用 hash 算法进行数据的存储和查询。
内部使用一个 Entry 表示键值对 key-value。
用 Entry 的数组保存所有键值对, Entry 通过链表的方式链接后续的节点 (1.8 后会根据链表长度决定是否转换成一棵树类似 TreeMap 来节省查询时间)
Entry 通过计算 key 的 hash 值来决定映射到具体的~~哪个数组~~数组位置（Bucket）（也叫 Bucket）中。

HashMap 非线程安全，即任一时刻可以有多个线程同时写 HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections 的 synchronizedMap 方法使 HashMap 具有线程安全的能力，或者使用 ConcurrentHashMap。

HashMap 特性

- Hash 相关的数据结构本质上都是 key value pair
- Hash 中不能存在 duplicate key
- HashMap 提供非常快速查找时间复杂度
- HashMap 具体实现中，null 可以作为 key 或者 value 存在
- HashMap 不是线程安全

HashMap 实现

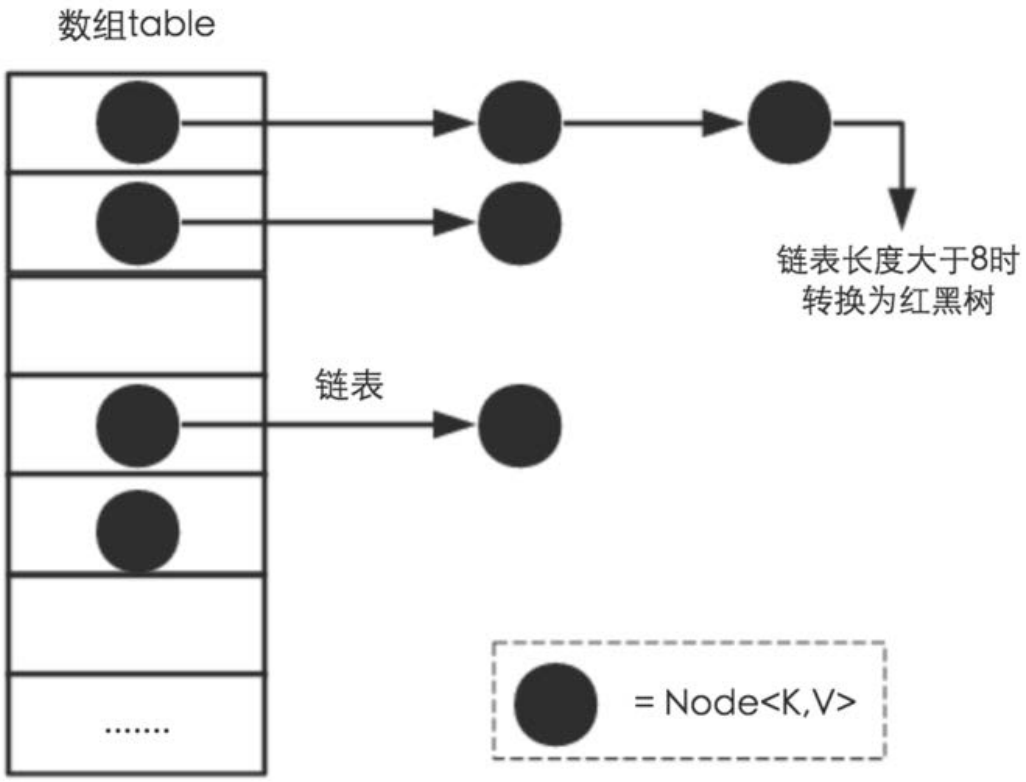
HashMap 1.7 版本

java 集合框架 08——HashMap 和源码分析

HashMap 1.8 版本

存储结构

从结构实现来讲，HashMap 是数组 + 链表 + 红黑树（JDK1.8 增加了红黑树部分）实现的。



hashMap 内存结构图.png

HashMap 常量定义：

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable,
Serializable {

    private static final long serialVersionUID = 362498820763181265L;

    /**

     * HashMap 的默认初始容量为 16，必须为 2 的 n 次方 (一定是合数)

     */

    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;



    /**

     * HashMap 的最大容量为 2 的 30 次幂

     */

    static final int MAXIMUM_CAPACITY = 1 << 30;



    /**

     *  HashMap 的默认负载因子

     */

    static final float DEFAULT_LOAD_FACTOR = 0.75f;



    /**

     * 链表转成红黑树的阈值。即在哈希表扩容时，当链表的长度(桶中元素个数)超过这个值的时候，
进行链表到红黑树的转变

     */
```

```
    static final int TREEIFY_THRESHOLD = 8;



    /**

     * 红黑树转为链表的阈值。即在哈希表扩容时，如果发现链表长度(桶中元素个数)小于 6，则会由红黑树重新退化为链表

     */

    static final int UNTREEIFY_THRESHOLD = 6;



    /**

     * HashMap 的最小树形化容量。这个值的意义是：位桶（bin）处的数据要采用红黑树结构进行存储时，整个 Table 的最小容量（存储方式由链表转成红黑树的容量的最小阈值）

     * 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

     */

    static final int MIN_TREEIFY_CAPACITY = 64;



    /**

     * Node 是 HashMap 的一个内部类，实现了 Map.Entry 接口，本质是就是一个映射 (键值对)

     * Basic hash bin node, used for most entries.

     */

    static class Node<K,V> implements Map.Entry<K,V> {

        final int hash; // 用来定位数组索引位置

        final K key;
```

```
        V value;
        Node<K,V> next; // 链表的下一个 node

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

        public final K getKey()        { ... }
        public final V getValue()      { ... }
        public final String toString() { ... }
        public final int hashCode() { ... }
        public final V setValue(V newValue) { ... }
        public final boolean equals(Object o) { ... }
    }

    /**
     * 哈希桶数组，分配的时候，table 的长度总是 2 的幂
     */
    transient Node<K,V>[] table;

    /**
     * Holds cached entrySet(). Note that AbstractMap fields are used
     * for keySet() and values().
     */
```

```
    transient Set<Map.Entry<K,V>> entrySet;



    /**
     * HashMap 中实际存储的 key-value 键值对数量
     */
    transient int size;



    /**
     * 用来记录 HashMap 内部结构发生变化的次数，主要用于迭代的快速失败机制
     */
    transient int modCount;



    /**
     * HashMap 的门限阀值/扩容阈值，所能容纳的 key-value 键值对极限，当 size>=threshold
时，就会扩容
     * 计算方法：容量 capacity * 负载因子 load factor
     */
    int threshold;
    /**
     * HashMap 的负载因子
     */
    final float loadFactor;
}
```

Node[] table 的初始化长度 length(默认值是 16)，loadFactor 为负载因子 (默认值 DEFAULT_LOAD_FACTOR 是 0.75)，threshold 是 HashMap 所能容纳的最大数据量的 Node(键值对) 个数。

threshold = length * loadFactor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。
这里我们需要加载因子 (load_factor)，加载因子默认为 0.75，当 HashMap 中存储的元素的数量大于 (容量 × 加载因子)，也就是默认大于 16*0.75=12 时，HashMap 会进行扩容的操作。

size 这个字段其实很好理解，就是 HashMap 中**实际存在的键值对数量**。注意和 table 的长度 length、容纳最大键值对数量 threshold 的区别。而 modCount 字段主要**用来记录 HashMap 内部结构发生变化的次数**，主要**用于迭代的快速失败**。强调一点，内部结构发生变化指的是**结构发生变化**，例如 put 新键值对，但是某个 key 对应的 value 值被覆盖不属于结构变化。

在 HashMap 中，哈希桶数组 table 的长度 length 大小**必须为 2 的 n 次方** (一定是合数)，这是一种**非常规的设计**，常规的设计是把桶的大小设计为素数。相对来说素数导致冲突的概率要小于合数，具体证明可以参考 http://blog.csdn.net/liuqiyao_01/article/details/14475159，Hashtable 初始化桶大小为 11，就是桶大小设计为素数的应用（Hashtable 扩容后不能保证还是素数）。HashMap 采用这种非常规设计，主要是**为了在取模和扩容时做优化，同时为了减少冲突**，HashMap 定位哈希桶索引位置时，也加入了高位参与运算的过程。

这里存在一个问题，即使负载因子和 Hash 算法设计的再合理，也免不了会出现拉链过长的情况，**一旦出现拉链过长，则会严重影响 HashMap 的性能**。于是，在 JDK1.8 版本中，对数据结构做了进一步的优化，引入了红黑树。而**当链表长度太长（默认超过 8）时，链表就转换为红黑树**，利用**红黑树快速增删改查的特点**提高 HashMap 的性能，其中会用到红黑树的插入、删除、查找等算法。本文不再对红黑树展开讨论，想了解更多红黑树数据结构的工作原理可以参考：http://blog.csdn.net/v_july_v/article/details/6105630。

功能实现

HashMap 的内部功能实现很多，本文主要从根据 key 获取哈希桶数组索引位置、put 方法的详细执行、扩容过程等具有代表性的点深入展开讲解。

解决 Hash 的的冲突的 `hash()`方法

HashMap 的 hash 计算时先计算 hashCode(), 然后进行二次 hash。

```
// 计算二次 Hash

int hash = hash(key.hashCode());



// 通过 Hash 找数组索引

int i = hash & (tab.length-1);



static final int hash(Object key) {

    int h;

    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}
```

这个方法非常巧妙，它总是通过 `h &(table.length -1)` 来得到该对象的保存位置，而 HashMap 底层数组的长度总是 2 的 n 次方。

当 length 总是 2 的倍数时，`h & (length-1)` 将是一个非常巧妙的设计：
假设 h=5,length=16, 那么 h & length - 1 将得到 5；
如果 h=6,length=16, 那么 h & length - 1 将得到 6
如果 h=15,length=16, 那么 h & length - 1 将得到 15；
但是当 h=16 时 , length=16 时，那么 h & length - 1 将得到 0 了；
当 h=17 时 , length=16 时，那么 h & length - 1 将得到 1 了。

这样保证计算得到的索引值总是位于 table 数组的索引之内。

`put()`方法

视频讲解：[How HashMap works in Java? With Animation!! whats new in java8 tutorial]

`put()`方法大致的思路为：

1. 对 key 的 hashCode() 做 hash，然后再计算 index;
2. 如果没碰撞直接放到 bucket 里；

3. 如果碰撞了，以链表的形式存在 buckets 后；
4. 如果碰撞导致链表过长 (大于等于 TREEIFY_THRESHOLD=8)，就把链表转换成红黑树；
5. 如果节点已经存在就替换 old value(保证 key 的唯一性)
6. 如果 bucket 满了 (超过 load factor*current capacity)，就要 resize。

具体步骤为：

1. 如果 table 没有使用过的情况(tab=table)==null || (n=tab.length) == 0，则以默认大小进行一次 resize
2. 计算 key 的 hash 值，然后获取底层 table 数组的第 (n-1)&hash 的位置的数组索引 tab[i] 处的数据，即 hash 对 n 取模的位置，依赖的是 n 为 2 的次方这一条件
3. 先检查该 bucket 第一个元素是否是和插入的 key 相等 (如果是同一个对象则肯定 equals)
4. 如果不相等并且是 TreeNode 的情况，调用 TreeNode 的 put 方法
5. 否则循环遍历链表，如果找到相等的 key 跳出循环否则达到最后一个节点时将新的节点添加到链表最后，当前面找到了相同的 key 的情况下替换这个节点的 value 为新的 value。
6. 最后如果新增了 key-value 对，则增加 size 并且判断是否超过了 threshold，如果超过则需要进行 resize 扩容

```
put(K key, V val);

index = hash(key) & (n-1)



V val = get(Object key);

index = hash(key) & (n-1)

比较 hashCode...

public V put(K key, V value) {
```

```
    // 对 key 的 hashCode()做 hash
    return putVal(hash(key), key, value, false, true);
}


/**
 * Implements Map.put and related methods
 *
 * @param hash hash for key
 * @param key the key
 * @param value the value to put
 * @param onlyIfAbsent if true, don't change existing value
 * @param evict if false, the table is in creation mode.
 * @return previous value, or null if none
 */
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // table 为空或者 length=0 时，以默认大小扩容，n 为 table 的长度
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 计算 index，并对 null 做处理，table[i]==null
    if ((p = tab[i = (n - 1) & hash]) == null)
```

```
        // (n-1)&hash 与 Java7 中 indexFor 方法的实现相同，若 i 位置上的值为空，则新建一个 Node，table[i]指向该 Node。

        // 直接插入

        tab[i] = newNode(hash, key, value, null);

    else {

        // 若 i 位置上的值不为空，判断当前位置上的 Node p 是否与要插入的 key 的 hash 和 key 相同

        Node<K,V> e; K k;

        // 若节点 key 存在，直接覆盖 value

        if (p.hash == hash &&

            ((k = p.key) == key || (key != null && key.equals(k))))

            e = p;

        // 判断 table[i]该链是否是红黑树，如果是红黑树，则直接在树中插入键值对

        else if (p instanceof TreeNode)

            // 不同，且当前位置上的的 node p 已经是 TreeNode 的实例，则再该树上插入新的 node

            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

        // table[i]该链是普通链表，进行链表的插入操作

        else {

            // 在 i 位置上的链表中找到 p.next 为 null 的位置，binCount 计算出当前链表的长度，如果继续将冲突的节点插入到该链表中，会使链表的长度大于 tree 化的阈值，则将链表转换成 tree。

            for (int binCount = 0; ; ++binCount) {

                // 如果遍历到了最后一个节点，说明没有匹配的 key，则创建一个新的节点并添加到最后

                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;
                }
                // 遍历过程中若发现 key 已经存在直接覆盖 value 并跳出循环即可
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        // 已经存在该 key 的情况时，将对应的节点的 value 设置为新的 value
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
```

```
    ++modCount;
    // 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
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

红黑树结构的 putVal 方法：

```
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;
    TreeNode<K,V> root = (parent != null) ? root() : this;
    for (TreeNode<K,V> p = root;;) {
        int dir, ph; K pk;
        if ((ph = p.hash) > h)
            dir = -1;
        else if (ph < h)
            dir = 1;
        else if ((pk = p.key) == k || (k != null && k.equals(pk)))
            return p;
```

```
        else if ((kc == null &&
                  (kc = comparableClassFor(k)) == null) ||
                  (dir = compareComparables(kc, k, pk)) == 0) {
            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;
        if ((p = (dir <= 0) ? p.left : p.right) == null) {
            Node<K,V> xpn = xp.next;
            TreeNode<K,V> x = map.newTreeNode(h, k, v, xpn);
            if (dir <= 0)
                xp.left = x;
            else
```

```
                xp.right = x;
            xp.next = x;
            x.parent = x.prev = xp;
            if (xpn != null)
                ((TreeNode<K,V>)xpn).prev = x;
            moveRootToFront(tab, balanceInsertion(root, x));
            return null;
        }
    }
}
```

`get()`方法

`get(key)` 方法时获取 key 的 hash 值，计算 `hash&(n-1)` 得到在链表数组中的位置 `first=tab[hash&(n-1)]`，先判断 first 的 key 是否与参数 key 相等，不等就遍历后面的链表找到相同的 key 值返回对应的 Value 值即可。

```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

// 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
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
```

```
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

`resize()`方法

扩容 (resize) 就是重新计算容量，向 HashMap 对象里不停的添加元素，而 HashMap 对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然 Java 里的数组是无法自动扩容的，方法是**使用一个新的数组代替已有的容量小的数组**，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

由于需要考虑 hash 冲突解决时采用的可能是链表也可能是红黑树的方式，因此 resize 方法相比 JDK7 中复杂了一些。

rehashing 触发的条件：1、超过默认容量 * 加载因子；2、加载因子不靠谱，比如远大于 1。

在 HashMap 进行扩容时，会进行 2 倍扩容，而且会将哈希碰撞处的数据再次分散开来，一部分依照新的 hash 索引值呆在 “原处”，一部分加上偏移量移动到新的地方。

具体步骤为：

1. 首先计算 resize() 后的新的 capacity 和 threshold 值。如果原有的 capacity 大于零则将 capacity 增加一倍，否则设置成默认的 capacity。
2. 创建新的数组，大小是新的 capacity
3. 将旧数组的元素放置到新数组中

```
final Node<K,V>[] resize() {

    // 将字段引用 copy 到局部变量表，这样在之后的使用时可以减少 getField 指令的调用

    Node<K,V>[] oldTab = table;

    // oldCap 为原数组的大小或当空时为 0

    int oldCap = (oldTab == null) ? 0 : oldTab.length;

    int oldThr = threshold;

    int newCap, newThr = 0;

    if (oldCap > 0) {

        if (oldCap >= MAXIMUM_CAPACITY) {

            // 如果超过最大容量 1>>30，无法再扩充 table，只能改变阈值

            threshold = Integer.MAX_VALUE;

            return oldTab;

        }
```

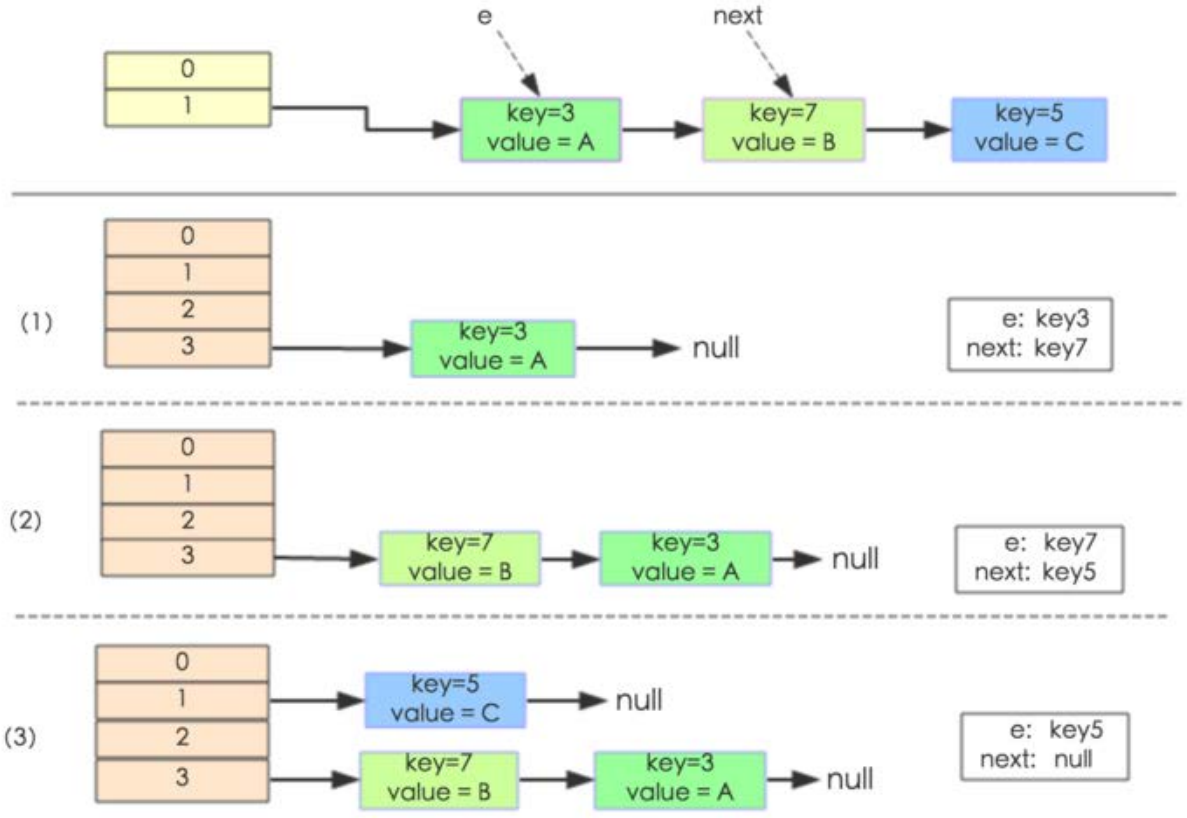
```
        // 新的数组的大小是旧数组的两倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            // 当旧的的数组大小大于等于默认大小时，threshold 也扩大一倍
            newThr = oldThr << 1;
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        // 初始化操作
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
    // 创建容量为 newCap 的 newTab，并将 oldTab 中的 Node 迁移过来，这里需要考虑链表和 tree
两种情况。
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
```

```
    table = newTab;
    // 将原数组中的数组复制到新数组中
    if (oldTab != null) {
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    // 如果 e 是该 bucket 唯一的一个元素，则直接赋值到新数组中
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    // split 方法会将树分割为 lower 和 upper tree 两个树，如果子树的节点数小于了 UNTREEIFY_THRESHOLD 阈值，则将树 untreeify，将节点都存放在 newTab 中。
                    // TreeNode 的情况则使用 TreeNode 中的 split 方法将这个树分成两个小树
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order 保持顺序
                    // 否则则创建两个链表用来存放要放的数据，hash 值&oldCap 为 0 的(即 oldCap 的 1 的位置的和 hash 值的同样的位置都是 1，同样是基于 capacity 是 2 的次方这一前提)为 low 链表，反之为 high 链表，通过这种方式将旧的数据分到两个链表中再放到各自对应余数的位置
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
```

```
                // 按照 e.hash 值区分放在 loTail 后还是 hiTail 后
                if ((e.hash & oldCap) == 0) {
                    // 运算结果为 0 的元素，用 lo 记录并连接成新的链表
                    if (loTail == null)
                        loHead = e;
                    else
                        loTail.next = e;
                    loTail = e;
                }
                else {
                    // 运算结果不为 0 的数据，用 li 记录
                    if (hiTail == null)
                        hiHead = e;
                    else
                        hiTail.next = e;
                    hiTail = e;
                }
            } while ((e = next) != null);
            // 处理完之后放到新数组中
            if (loTail != null) {
                loTail.next = null;
                // lo 仍然放在“原处”，这个“原处”是根据新的 hash 值算出来的
```

```
                    newTab[j] = loHead;
                }
                if (hiTail != null) {
                    hiTail.next = null;
                    // li 放在 j+oldCap 位置
                    newTab[j + oldCap] = hiHead;
                }
            }
        }
    }
    return newTab;
}
```

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。



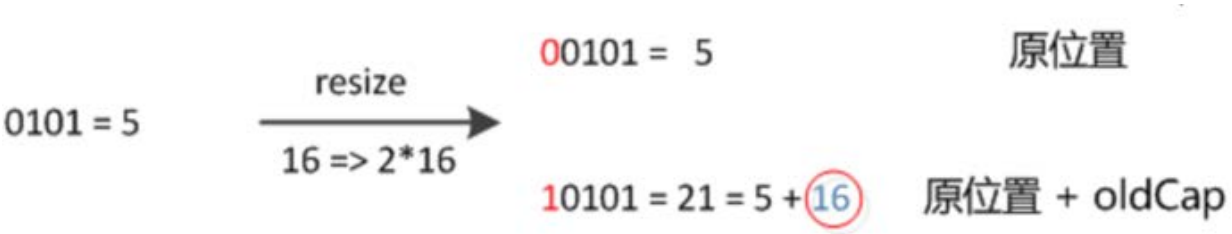
jdk1.7 扩容例图.png

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。

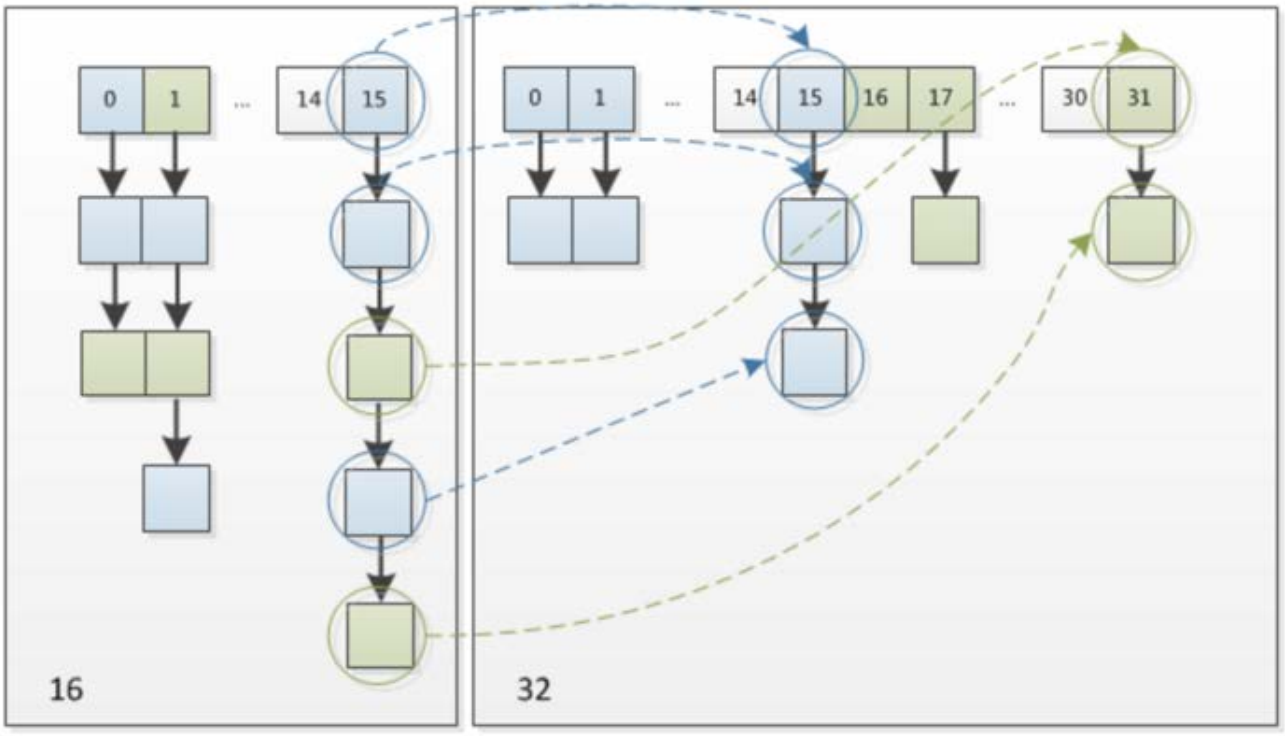
hashMap 1.8 哈希算法例图 1.png

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：



hashMap 1.8 哈希算法例图 2.png

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：



jdk1.8 hashMap 扩容例图.png

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。

红黑树结构

```
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {

    TreeNode<K,V> parent;  // red-black tree links 父节点

    TreeNode<K,V> left;    // 左子树

    TreeNode<K,V> right;   // 右子树

    TreeNode<K,V> prev;    // needed to unlink next upon deletion

    boolean red;           // 颜色属性

    TreeNode(int hash, K key, V val, Node<K,V> next) {

        super(hash, key, val, next);

    }

}
```

树形化操作

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

```
// MIN_TREEIFY_CAPACITY 的值为 64，若当前 table 的 length 不够，则 resize()

// 将桶内所有的 链表节点 替换成 红黑树节点

final void treeifyBin(Node<K,V>[] tab, int hash) {

    int n, index; Node<K,V> e;

    // 如果当前哈希表为空，或者哈希表中元素的个数小于树形化阈值(默认为 64)，就去新建(扩容)
```

```
if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
    resize();
// 如果哈希表中的元素个数超过了树形化阈值，则进行树形化
// e 是哈希表中指定位置桶里的链表节点，从第一个开始
else if ((e = tab[index = (n - 1) & hash]) != null) {
    // 红黑树的头、尾节点
    TreeNode<K,V> hd = null, tl = null;
    do {
        // 新建一个树形节点，内容和当前链表节点 e 一致
        TreeNode<K,V> p = replacementTreeNode(e, null);
        // 确定树头节点
        if (tl == null)
            hd = p;
        else {
            p.prev = tl;
            tl.next = p;
        }
        tl = p;
    } while ((e = e.next) != null);
    // 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
    if ((tab[index] = hd) != null)
        hd.treeify(tab);
```

```
    }
}


TreeNode<K,V> replacementTreeNode(Node<K,V> p, Node<K,V> next) {
    return new TreeNode<>(p.hash, p.key, p.value, next);
}
```

`size()`方法

HashMap 的大小很简单，不是实时计算的，而是每次新增加 Entry 的时候，size 就递增。删除的时候就递减。空间换时间的做法。因为它不是线程安全的。完全可以这么做，效率高。

面试问题

HashMap 的实现原理

HashMap 是典型的空间换时间的一种技术手段。

- 如何解决 hash 冲突
- loadFactor 等核心概念
- 扩容机制

构造函数中 initialCapacity 与 loadFactor 两个参数

`HashMap(int initialCapacity, float loadFactor)`：构造一个指定容量和加载因子的空 HashMap

在这里提到了两个参数：初始容量，加载因子。这两个参数是影响 HashMap 性能的重要参数，其中容量表示哈希表中桶的数量，初始容量是创建哈希表时的容量，加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，它衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间

是 O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。系统默认负载因子为 0.75，一般情况下我们是无需修改的。

initialCapacity 和 loadFactor 参数设什么样的值好呢？

initialCapacity 的默认值是 16，有些人可能会想如果内存足够，是不是可以将 initialCapacity 设大一些，即使用不了这么大，就可避免扩容导致的效率的下降，反正无论 initialCapacity 大小，我们使用的 get 和 put 方法都是常数复杂度的。这么说没什么不对，但是可能会忽略一点，实际的程序可能不仅仅使用 get 和 put 方法，也有可能使用迭代器，如 initialCapacity 容量较大，那么会使迭代器效率降低。所以理想的情况还是在使用 HashMap 前估计一下数据量。

加载因子默认值是 0.75，是 JDK 权衡时间和空间效率之后得到的一个相对优良的数值。如果这个值过大，虽然空间利用率是高了，但是对于 HashMap 中的一些方法的效率就下降了，包括 get 和 put 方法，会导致每个 hash 桶所附加的链表增长，影响存取效率。如果比较小，除了导致空间利用率较低外没有什么坏处，只要有的是内存，毕竟现在大多数人把时间看的比空间重要。但是实际中还是很少有人会将这个值设置的低于 0.5。

size 为什么必须是 2 的整数次幂

Java 编程：浅析 HashMap 中数组的 size 为什么必须是 2 的整数次幂

这是为了服务 key 映射到 index 的 Hash 算法的，公式 `index=hashcode(key)&(length-1)`，初始长度(16-1)，二进制为 1111&hashcode 结果为 hashcode 最后四位，能最大程度保持平均，二的幂数保证二进制为 1，保持 hashcode 最后四位。这种算法在保持分布均匀之外，效率也非常高。

HashMap 的 key 为什么一般用字符串比较多，能用其他对象，或者自定义的对象吗？为什么？

能用其他对象，必须是 immutable 的，但是自实现的类必须 Override 两个方法：equals()和 hashCode()。否则会调用默认的 Object 类的对应方法。

为什么需要使用加载因子，为什么需要扩容呢？

因为如果填充比很大，说明利用的空间很多，如果一直不进行扩容的话，链表就会越来越长，这样查找的效率很低，因为链表的长度很大（当然最新版本使用了红黑树后会改进很多），扩容之后，将原来链表数组的每一个链表分成奇偶两个子链表分别挂在新链表数组的散列位置，这样就减少了每个链表的长度，增加查找效率。

HashMap 本来是以空间换时间，所以填充比没必要太大。但是填充比太小又会导致空间浪费。如果关注内存，填充比可以稍大，如果主要关注查找性能，填充比可以稍小。

使用红黑树的改进

在 java jdk8 中对 HashMap 的源码进行了优化，在 jdk7 中，HashMap 处理 “碰撞” 的时候，都是采用链表来存储，当碰撞的结点很多时，查询时间是 O(n)。
在 jdk8 中，HashMap 处理 “碰撞” 增加了红黑树这种数据结构，当碰撞结点较少时，采用链表存储，当较大时（>8 个），采用红黑树（特点是查询时间是 O(logn)）存储（有一个阀值控制，大于阀值 (8 个)，将链表存储转换成红黑树存储）

问题分析：
哈希碰撞会对 hashMap 的性能带来灾难性的影响。如果多个 hashCode() 的值落到同一个桶内的时候，这些值是存储到一个链表中的。最坏的情况下，所有的 key 都映射到同一个桶中，这样 hashmap 就退化成了一个链表——查找时间从 O(1) 到 O(n)。

随着 HashMap 的大小的增长，get() 方法的开销也越来越大。由于所有的记录都在同一个桶里的超长链表内，平均查询一条记录就需要遍历一半的列表。
JDK1.8HashMap 的红黑树是这样解决的：
如果某个桶中的记录过大的话（当前是 TREEIFY_THRESHOLD = 8），HashMap 会动态的使用一个专门的 treemap 实现来替换掉它。这样做的结果会更好，是 O(logn)，而不是糟糕的 O(n)。

它是如何工作的？前面产生冲突的那些 KEY 对应的记录只是简单的追加到一个链表后面，这些记录只能通过遍历来进行查找。但是超过这个阈值后 HashMap 开始将列表升级成一个二叉树，使用哈希值作为树的分支变量，如果两个哈希值不等，但指向同一个桶的话，较大的那个会插入到右子树里。如果哈希值相等，HashMap 希望 key 值最好是实现了 Comparable 接口的，这样它可以按照顺序来进行插入。这对 HashMap 的 key 来说并不是必须的，不过如果实现了当然最好。如果没有实现这个接口，在出现严重的哈希碰撞的时候，你就并别指望能获得性能提升了。

HashMap 的 key 和 value 都能为 null 么？如果 key 能为 null，那么它是怎么样查找值的？

如果 key 为 null，则直接从哈希表的第一个位置 table[0] 对应的链表上查找，由 putForNullKey（）实现。记住，key 为 null 的键值对永远都放在以 table[0] 为头结点的链表中。

平时在使用 HashMap 时一般使用什么类型的元素作为 Key？

面试者通常会回答，使用 String 或者 Integer 这样的类。这个时候可以继续追问为什么使用 String、Integer 呢？这些类有什么特点？如果面试者有很好的思考，可以回答出这些类是 Immutable 的，并且这些类已经很规范的覆写了 hashCode() 以及 equals() 方法。作为不可变类天生是线程安全的，而且可以很好的优化比如可以缓存 hash 值，避免重复计算等等，那么基本上这道题算是过关了。

在 HashMap 中使用可变对象作为 Key 带来的问题：如果 HashMap Key 的哈希值在存储键值对后发生改变，Map 可能再也查找不到这个 Entry 了。详见：一道面试题看 HashMap 的存储方式

Key 对应的 hashCode()方法

对于非 String 类型的 key，hash() 使用 key 的 hashCode() 计算出该 key-value pair 对应在数组的索引 (String 类型的 key 有另一套计算 hash() 的方法，再次不做赘述），在 get() 方法中会使用 key 的 equal() 方法判断数组中的中的 key 是否与传入的 key 相等。由此可见，在 HashMap 的使用中，key 类型中定义的 hashCode() 和 equal() 方法都是非常重要的。假设 key 内部的值发生变化，导致 hashCode()/equal() 的结果改变，那么该 key 在 HashMap 中的存取则会产生问题。

如何创建不可变类(Immutable)/如果让你实现一个自定义的 class 作为 HashMap 的 key 该如何实现？

如何创建不可变（Immutable）的 Java 类或对象
HashMap 的 key 可以是可变的对象吗
危险！在 HashMap 中将可变对象用作 Key
如何使用建造者模式 (Builder Pattern) 创建不可变类

Immutable Objects 就是那些一旦被创建，它们的状态就不能被改变的 Objects，每次对他们的改变都是产生了新的 immutable 的对象，而 mutable Objects 就是那些创建后，状态可以被改变的 Objects。
举个例子：String 是 immutable 的，每次对于 String 对象的修改都将产生一个新的 String 对象，而原来的对象保持不变。而 StringBuilder 是 mutable，因为每次对于它的对象的修改都作用于该对象本身，并没有产生新的对象。
但有的时候 String 的 immutable 特性也会引起安全问题，这就是密码应该存放在字符数组中而不是 String 中的原因！

要写出这样的类，需要遵循以下几个原则：

1. immutable 对象的状态在创建之后就不能发生改变，任何对它的改变都应该产生一个新的对象。
2. Immutable 类的所有的属性都应该是 final 的。
3. 对象必须被正确的创建，比如：对象引用在对象创建过程中不能泄露(leak)。
4. 对象应该是 final 的，以此来限制子类继承父类，以避免子类改变了父类的 immutable 特性。
5. 如果类中包含 mutable 类对象，那么返回给客户端的时候，返回该对象的一个拷贝，而不是该对象本身（该条可以归为第一条中的一个特例）

```
public class MutableKey {

    private int i;

    private int j;



    public MutableKey(int i, int j) {

        this.i = i;

        this.j = j;

    }



    public final int getI() {
```

```
        return i;
    }

    public final void setI(int i) {
        this.i = i;
    }

    public final int getJ() {
        return j;
    }

    public final void setJ(int j) {
        this.j = j;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + i;
        result = prime * result + j;
        return result;
```

```
    }



    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof MutableKey)) {
            return false;
        }
        MutableKey other = (MutableKey) obj;
        if (i != other.i) {
            return false;
        }
        if (j != other.j) {
            return false;
        }
        return true;
    }
```

```
    public static void main(String[] args) {
        // Object created
        MutableKey key = new MutableKey(10, 20);
        System.out.println("Hash code: " + key.hashCode());

        // Object State is changed after object creation.
        key.setI(30);
        key.setJ(40);
        System.out.println("Hash code: " + key.hashCode());
    }
}
public class MutableSafeKey {
    // Cannot be changed once object is created. No setter for this field.
    private final int id;
    private String name;

    public MutableSafeKey(final int id) {
        this.id = id;
    }

    public final String getName() {
```

```
        return name;
    }

    public final void setName(final String name) {
        this.name = name;
    }

    public int getId() {
        return id;
    }

    // Hash code depends only on 'id' which cannot be
    // changed once object is created. So hash code will not change
    // on object's state change
    @Override
    public int hashCode() {
        int result = 17;
        result = 31 * result + id;
        return result;
    }

    @Override
```

```
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        MutableSafeKey other = (MutableSafeKey) obj;
        if (id != other.id)
            return false;
        return true;
    }
}
```

你能设计一个算法（输入是 java 源文件），判断一个类是否是 Immutable 的吗？

如何衡量一个 hash 算法的好坏

hashCode 不要求唯一但是要尽可能的均匀分布，而且算法效率要尽可能的快。

HashMap 中 hash 函数怎么是是实现的？

高 16bit 不变，低 16bit 和高 16bit 做了一个异或：（n - 1） & hash --> 得到下标

拓展：为什么 `h = 31 * h + val[off++];` 这一行使用 31 ，而不是别的数字，这是一个魔术吗？

HashMap 怎样解决哈希冲突，讲一下扩容过程。

JDK 使用了链地址法，hash 表的每个元素又分别链接着一个单链表，元素为头结点，如果不同的 key 映射到了相同的下标，那么就使用头插法，插入到该元素对应的链表。

扩容过程：

- 将新节点加到链表后
- 容量扩充为原来的两倍，然后对每个节点重新计算哈希值。
- 这个值只可能在两个地方，一个是原下标的位置，另一种是在下标为 <原下标 + 原容量> 的位置。

哈希冲突的常见解决方法：

- 开放定址法（线性探测再散列，二次探测再散列，伪随机探测再散列）
- 再哈希法，就是在原 hash 函数的基础，再次执行 hash 算法
- 链地址法，各种处理哈希碰撞的方法中，这种最简单，也是 HashMap 中使用的方法
- 建立一个公共溢出区

failure case/resize()

Load factor(default to 75%) 和 Initial capacity(default to 16) 是 HashMap 表的两个重要属性，如果 HashMap 中 entry 数量超过了 threshold(loadfactor *capacity) ，那么 HashMap 就不得不扩充 capacity（否则 hash collison 发生的概率就会大大增加，导致整个 HashMap 性能下降），扩充 capacity 是一件比较麻烦的事情，因为数组的连续性，HashMap 不得不开辟一块更大数组，还要把原来的 entries 全部 transfer 到新的数组中，在某些情况下还需要重新计算 key 的 hash() 结果。另一方面，HashMap 的 capacity 也不是越大越好，事实上 HashMap 的遍历本质上是基于内部数组的遍历，如果内部数组是无意义的大，那么遍历 HashMap 相对来说不是特别高效。

为什么 HashMap 是线程不安全的，实际会如何体现？

第一，如果多个线程同时使用 put 方法添加元素
假设正好存在两个 put 的 key 发生了碰撞 (hash 值一样)，那么根据 HashMap 的实现，这两个 key 会添加到数组的同一个位置，这样最终就会发生其中一个线程的 put 的数据被覆盖。

第二，如果多个线程同时检测到元素个数超过数组大小 * loadFactor
这样会发生多个线程同时对 hash 数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给 table，也就是说其他线程的都会丢失，并且各自线程 put 的数据也丢失。且会引起死循环的错误。

具体细节上的原因，可以参考：不正当使用 HashMap 导致 cpu 100% 的问题追究

HashMap 不是线程安全的，你怎么理解线程安全。原理是什么？几种方式避免线程安全的问题

如何线程安全的使用 HashMap

1. 直接使用 Hashtable，但是当一个线程访问 HashTable 的同步方法时，其他线程如果也要访问同步方法，会被阻塞住。举个例子，当一个线程使用 put 方法时，另一个线程不但不可以使用 put 方法，连 get 方法都不可以，效率很低，现在基本不会选择它了。
2. HashMap 可以通过下面的语句进行同步：
   Collections.synchronizeMap(hashMap);
3. 直接使用 JDK 5 之后的 ConcurrentHashMap。

HashTable 和 HashMap 的区别有哪些？

HashMap 和 Hashtable 都实现了 Map 接口，但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步 (synchronization)，以及速度。

理解 HashMap 是 Hashtable 的轻量级实现（非线程安全的实现，hashtable 是非轻量级，线程安全的），都实现 Map 接口，主要区别在于：

1. 由于 HashMap 非线程安全，在只有一个线程访问的情况下，效率要高于 HashTable
2. HashMap 允许将 null 作为一个 entry 的 key 或者 value，而 Hashtable 不允许。
3. HashMap 把 Hashtable 的 contains 方法去掉了，改成 containsValue 和 containsKey。因为 contains 方法容易让人引起误解。
4. Hashtable 继承自陈旧的 Dictionary 类，而 HashMap 是 Java1.2 引进的 Map 的一个实现。
5. Hashtable 和 HashMap 扩容的方法不一样，HashTable 中 hash 数组默认大小 11，扩容方式是 old*2+1。HashMap 中 hash 数组的默认大小是 16，而且一定是 2 的指数，增加为原来的 2 倍，没有加 1。
6. 两者通过 hash 值散列到 hash 表的算法不一样，HashTbale 是古老的除留余数法，直接使用 hashcode，而后者是强制容量为 2 的幂，重新根据 hashcode 计算 hash 值，在使用 hash 位与 （hash 表长度 – 1），也等价取膜，但更加高效，取得的位置更加分散，偶数，奇数保证了都会分散到。前者就不能保证。
7. 另一个区别是 HashMap 的迭代器 (Iterator) 是 fail-fast 迭代器，而 Hashtable 的 enumerator 迭代器不是 fail-fast 的。所以当有其它线程改变了 HashMap 的结构（增加或者移除元素），将会抛出 ConcurrentModificationException，但迭代器本身的 remove() 方法移除元素则不会抛出 ConcurrentModificationException 异常。但这并不是一个一定发生的行为，要看 JVM。这条同样也是 Enumeration 和 Iterator 的区别。

   fail-fast 和 iterator 迭代器相关。如果某个集合对象创建了 Iterator 或者 ListIterator，然后其它的线程试图 “结构上” 更改集合对象，将会抛出 ConcurrentModificationException 异常。但其它线程可以通过 set() 方法更改集合对象是允许的，因为这并没有从 “结构上” 更改集合。但是假如已经从结构上进行了更改，再调用 set() 方法，将会抛出

IllegalArgumentException 异常。

结构上的更改指的是删除或者插入一个元素，这样会影响到 map 的结构。该条说白了就是在使用迭代器的过程中有其他线程在结构上修改了 map，那么将抛出 ConcurrentModificationException，这就是所谓 fail-fast 策略。

引申扩展：建议用 ConcurrentHashMap 代替 Hashtable。

为什么 HashTable 的默认大小和 HashMap 不一样？

前面分析了，Hashtable 的扩容方法是乘 2 再 + 1，不是简单的乘 2，故 hashtable 保证了容量永远是奇数，结合之前分析 hashmap 的重算 hash 值的逻辑，就明白了，因为在数据分布在等差数据集合 (如偶数) 上时，如果公差与桶容量有公约数 n，则至少有 (n-1)/n 数量的桶是利用不到的，故之前的 hashmap 会在取模（使用位与运算代替）哈希前先做一次哈希运算，调整 hash 值。这里 hashtable 比较古老，直接使用了除留余数法，那么就需要设置容量起码不是偶数（除（近似）质数求余的分散效果好）。而 JDK 开发者选了 11。