

1 位运算

运算符	运算	位运算符的细节
<<	左移	空位补0，被移除的高位丢失
>>	右移	被移位的二进制最高位是0，右移后，空缺位补0，最高位是1，最高位补1
>>>	无符号右移	被移位二进制无论是0或者1，空缺位都是用0补
&	与运算	任何二进制位和0进行&运算，结果都是0，和1进行&运算，都是原值
	或运算	任何二进制位和0进行 运算，结果都是原值，和1进行 运算，都是1
^	异或运算	任何相同二进制位进行^运算，结果都是0，不同二进制位进行^运算，结果都是1
~	反码	https://blog.csdn.net/sky1988818

补码

在计算机系统中，数值一律用补码来表示和存储，其中最高位表示符号位，1表示负数，0表示正数。

- 正数的补码是原码自身。
- 负数补码是通过原码计算得到，计算过程为：符号位不变，其余位按照原码取反加1
-100的原码：10000000 00000000 00000000 01100100 符号位保持不变，取反：11111111 11111111 11111111 10011100 加1后，-100补码为：11111111 11111111 11111111 10011100

• 正数右移(100右移4位为例)

操作	二进制	对应十进制
补码	00000000 00000000 00000000 01100100	100
右移4位	00000000 00000000 00000000 00000110	6
源码	00000000 00000000 00000000 00000110	6

负数右移(-100右移4位为例)

操作	二进制	对应十进制
原码	10000000 00000000 00000000 01100100	-100
转换为补码	11111111 11111111 11111111 10011100	-100
右移4位，高位补1	11111111 11111111 11111111 11111001	
保留符号位，按位取反	10000000 00000000 00000000 00000110	
加1后转为源码	10000000 00000000 00000000 00000111	-7

无符号右移(-100右移4位为例)

操作	二进制	对应十进制
原码	10000000 00000000 00000000 01100100	-100
转换为补码	11111111 11111111 11111111 10011100	-100
右移4位, 高位补0	00001111 11111111 11111111 11111001	
转为原码	00001111 11111111 11111111 11111001	268435449

常用操作:

判定奇偶: $a \& 1 = 0$ 偶数 $a \& 1 = 1$ 奇数

交换两个变量: $a \wedge b$ $b \wedge a$ $a \wedge b$

取模运算: $a \% 2^n$ 等价于 $a \& (2^n - 1)$

乘法运算: $a * (2^n)$ 等价于 $a \ll n$

除法运算: $a / (2^n)$ 等价于 $a \gg n$

对称加密: 采用 \wedge 操作, 原理 \wedge 同一个值就会得到原值 例如 $a = a \wedge b \wedge b$

2 基本数据类型&包装类

bit(1位) byte(字节-8位) char(2个字节-16位) short(2个字节-16位) int(4个字节-32位) long(8个字节-64位)
float(4个字节-32位) double(8个字节-64位)

- * 构造方法有如下几种
- * `Integer a = 1;` 字节码底层是 `Integer.valueOf(1)` 实现的
- * `Integer b = new Integer(1);`
- * `Integer.valueOf(1)` -127到128 会先查看缓存中是否有值, 如果没有值 `new Integer(i)`
- *
- * `Integer.equals` 比较的是两个值是否相等
- * `==` 比较的是内存地址
- *
- * 如果一个 `Integer` 对象和 `int` 值做对比, 则先调用 `Integer.intValue` 获取 `Integer` 的值, 然后两个 `int` 值做对比

3 Thread、ThreadLocal

```
/**
 *
 * Thread类继承了Runnable接口
 * 构造器中主要有两个参数一个是Runnable接口, 一个是名称 new Thread(Runnable runnable,
String name)
 *
 * 线程状态
 * Thread.State.NEW
 * Thread.State.RUNNABLE
 * Thread.State.BLOCKED(waiting for a monitor lock, enter a synchronized
block/method)
 * Thread.State.WAITING(Object.wait, Thread.join, LockSupport.park)
 * Thread.State.TIMED_WAITING(Thread.sleep, Object.wait with timeout,
Thread.join with timeout, LockSupport.parkNanos, LockSupport.parkUntil)
 * Thread.State.TERMINATED
 *
 * java中线程中断的两种方法:
```

```

* 1 设置或者清除中断标志（interrupt status）
* Thread.currentThread().isInterrupted();实例方法    返回线程是否中断的标识,不会重置中断标志
* Thread.interrupted();静态方法    返回线程是否中断的标识&重置中断标志
* 2 抛出中断异常(interrupted Exception)
* Thread.currentThread().interrupt();实例方法--    中断线程
* 中断线程的意义是：给等待或者执行中的线程一个机会，可以终止等待或者中断正在执行的任务。但是仅仅是一个机会，并不会真的终止线程。
*
* Thread.sleep, Object.wait, Thread.join等方法在被打断之后，会获取到打断异常，在抛出异常之后（即捕获到异常之后）会清除当前线程的中断标志
*
* 所谓中断一个线程，并不是让线程停止运行。仅仅是将线程的中断标志设置为true，或者在某些情况下抛出异常。在被中断的线程的角度看，仅仅是自己的中断标志变成true，或者代码中抛出了异常而已。（至于用不用这个标志来做业务处理，或者处理不处理异常，全靠自己的业务实现。）
*
* 若线程被中断前，如果该线程处于非阻塞状态(未调用过wait,sleep,join方法)，那么该线程的中断状态将被设为true，除此之外，不会发生任何事。
* 若线程被中断前，该线程处于阻塞状态(调用了wait,sleep,join方法)，那么该线程将会立即从阻塞状态中退出，并抛出一个InterruptedException异常，同时，该线程的中断状态被设为false，除此之外，不会发生任何事。
*
* */

```

```

/**
* ThreadLocal<T>
*
* 典型的以空间换时间的多线程并发解决方案
*
* 原理：
* 每个Thread中都有一个ThreadLocalMap属性threadLocals,该对象是一个KV结构对象
* 其中key=ThreadLocal对象本身    value=T对象，也就是你真正需要的对象
* 注意，key是WeakReference
* 此处可能会产生内存泄漏问题，原因
* 如果线程是一个线程池中的核心线程（长时间存在），那么value这个值会长时间存在内存中，
* 虽然这个值已经没有用了（因为KEY是弱引用，GC的时候已经回收，但是VALUE的指针会一直存在线程中）
*
* 最佳实践就是 调用了ThreadLocal#set()方法之后，需要调用ThreadLocal#remove()方法
* */

```

Unsafe类

```

/**
* Unsafe类
* 单例模式(饿汉模式)
* 通过静态getUnsafe方法获取单例对象，而且调用类必须是引导类加载器加载的，否则会报错
java.lang.SecurityException
*
* 如果想要使用，1 要么写的class被bootstrap-classloader加载 2 要么通过反射

```

```

*
* 相关操作：
*   内存操作：allocateMemory、freeMemory（堆外内存回收，是通过虚引用实现的）
*   cas操作：compareAndSwapInt、compareAndSwapObject
*   线程调度：park、unPark(LockSupport)
*   对象属性的操作：objectFieldOffset、getObject、putObject
*
*
*
*/

```

4 TreeMap--基于红黑树实现的

```
private final Comparator<? super K> comparator;
```

```
private transient Entry<K,V> root;
```

```

static final class Entry<K,V> implements Map.Entry<K,V> {
    K key;
    V value;
    Entry<K,V> left;
    Entry<K,V> right;
    Entry<K,V> parent;
    boolean color = BLACK;
}

```

CopyOnWriteArrayList

```

/**
 * CopyOnWriteArrayList解析
 * final transient ReentrantLock lock = new ReentrantLock();
 * private transient volatile Object[] array;
 *
 * add代码
 * 流程梳理：首先获取锁，然后数组copy,最后添加元素,释放锁
 * public boolean add(E e) {
 *     final ReentrantLock lock = this.lock;
 *     lock.lock();
 *     try {
 *         Object[] elements = getArray();
 *         int len = elements.length;
 *         Object[] newElements = Arrays.copyOf(elements, len + 1);
 *         newElements[len] = e;
 *         setArray(newElements);
 *         return true;
 *     } finally {
 *         lock.unlock();
 *     }
 * }
 *
 * get代码流程
 * 获取数组(此时的数组可能是原始数据，也可能是锁定之后扩容之后的数组)，获取数组元素
 * 注意，此时读取数据和添加数据可以同时进行，读取并没有加锁，只是不能保证读取的一定是最新的数据
 * public E get(int index) {
 *     return get(getArray(), index);
 * }
 */

```

```

*    }
*
*    final Object[] getArray() {
*        return array;
*    }
*
*
* */

```

ConcurrentHashMap

```

/**
 * 1.7中的ConcurrentHashMap
 * 1.7中采用 Segment数组+HashEntry数组+链表实现(Segment 数组长度为 16, 不可以扩容,
 * Segment[i] 的默认大小为 2, 负载因子是 0.75, 得出初始阈值为 1.5, 也就是以后插入第一个元素
 * 不会触发扩容, 插入第二个会进行第一次扩容
 * 只初始化了 segment[0], 其他位置仍然是 null)
 * Segment类继承了ReentrantLock, 采用分段锁的形式
 * 构造函数中初始化Segment数组和Segment[0]位置的HashEntry数组
 * 查询或者插入数据时, 要经过两次hash定位
 * put操作, 先通过hash(key)定位出Segment位置, 如果该位置Segment为空, 使用Segment[0]处的
 * 数组长度和负载因子初始化, 采用cas的方式设置Segment[i]的对象
 * 获取锁, 然后再hash定位出HashEntry的位置, 剩余操作同HashMap操作。
 * size方法
 * 先采用不加锁的方式, 连续计算元素的个数, 最多计算3次:
 * 1、如果前后两次计算结果相同, 则说明计算出来的元素个数是准确的;
 * 2、如果前后两次计算结果都不同, 则给每个Segment进行加锁, 再计算一次元素的个数;
 *
 * 1.8中的ConcurrentHashMap
 * 1.8中采用 Node数组+链表+红黑树实现
 * 第一次put操作才进行数组的初始化操作(懒加载模式)
 * put操作:1 计算hash(key) 2 for(;;)死循环
 * 死循环内容如下:
 * 如果Node数组为空, 则初始化Node数组
 * 如果table[i]为空, 则创建一个Node节点, 采用cas操作赋值
 * 如果table[i]不为空, 则synchronized(table[i]节点), 循环遍历赋值或者替换(考虑
 * 红黑树等情况)。
 * size方法: for(CountCell[]数组)相加, 采用CountCell[]数组实现, 底层是cas实现, put方
 * 法也会给CountCell[i]+1;
 *
 *
 *
 *
 * */

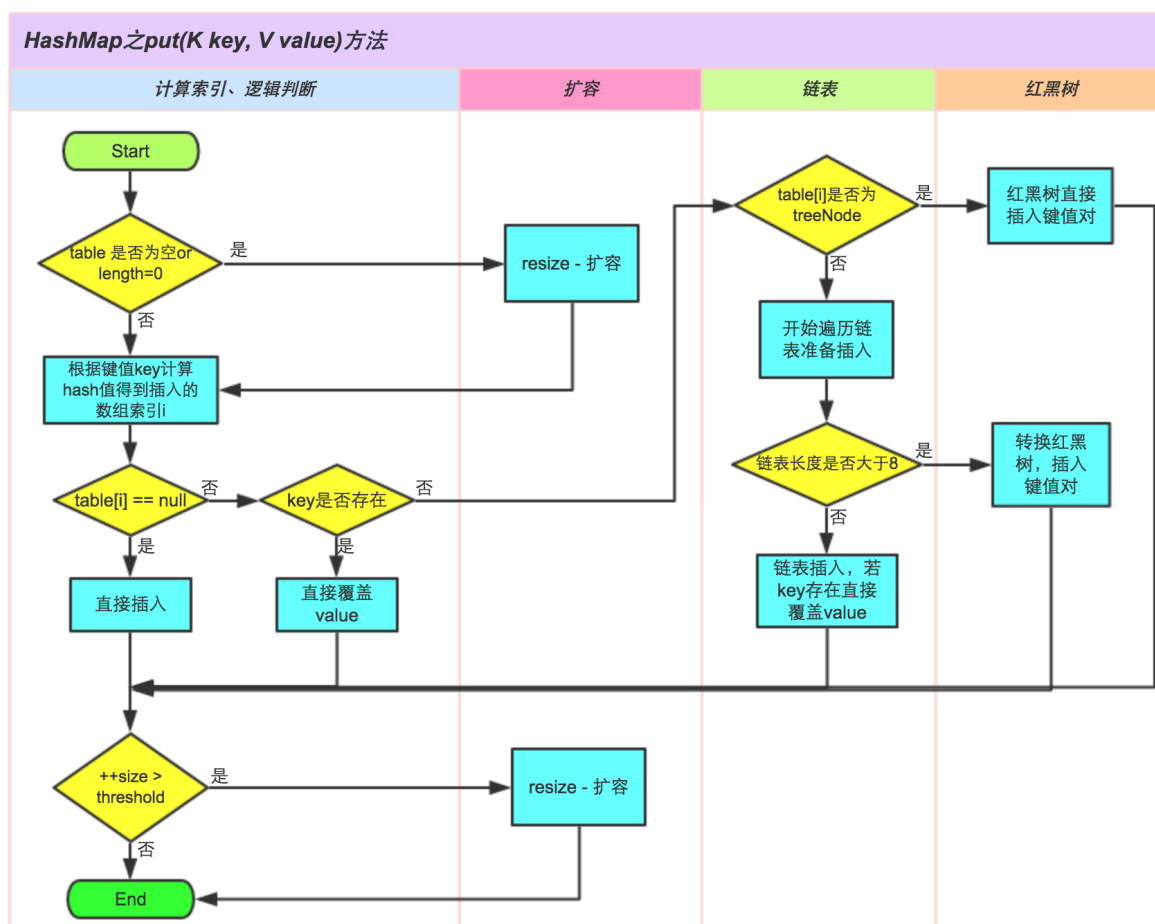
```

主要看put方法、扩容、树化、参数值等方面入手、每个参数的含义是什么、为什么说hashMap是线程不安全的

1 $(h = \text{key.hashCode>()) \wedge (h \ggg 16)$ h是int类型, 总共32位, 意思是, 高位也参与运算, 减少碰撞次数
 1 HashMap的容量为什么是2的次幂 put数据时, 需要根据key来决定存放位置, 一般都采用%取模运算来定位位置, 即 $\text{hash}(\text{key})\%n$, 当n为2的n次幂时, $(n - 1) \& \text{hash} = \text{hash}(\text{key})\%n$ 还有是扩容的时候, 只需要移动部分数据就可以 比如0101和0001在0011的HashMap中的位置都是0001, 当0011扩容到0111之

后，0101的位置是0101，就是原来的位置+扩容的大小 也能说明为什么，N必须是偶数，如果是奇数，N-1最后一位是0，&运算后，最后一位就是0，会浪费空间。

1.奇数不行的解释很能被接受，在计算hash的时候，确定落在数组的位置的时候，计算方法是 $(n - 1) \& \text{hash}$ ，奇数n-1为偶数，偶数2进制的结尾都是0，经过&运算末尾都是0，会增加hash冲突。2.为啥要是2的幂，不能是2的倍数么，比如6，10？2.1 hashmap 结构是数组，每个数组里面的结构是node（链表或红黑树），正常情况下，如果你想放数据到不同的位置，肯定会想到取余数确定放在那个数据里，计算公式： $\text{hash} \% n$ ，这个是十进制计算。在计算机中， $(n - 1) \& \text{hash}$ ，当n为2次幂时，会满足一个公式： $(n - 1) \& \text{hash} = \text{hash} \% n$ ，计算更加高效。2.2 只有是2的幂的数字经过n-1之后，二进制肯定是...11111111这样的格式，这种格式计算的位置的时候（&），完全是由产生的hash值类决定，而不受n-1(组数长度的二进制)影响。你可能会想，受影响不是更好么，又计算了一下，类似于扰动函数，hash冲突可能更低了，这里要考虑到扩容了，2的幂次方*2，在二进制中比如4和8，代表2的2次方和3次方，他们的2进制结构相似，比如4和8 00000100 0000 1000 只是高位向前移了一位，这样扩容的时候，只需要判断高位hash,移动到之前位置的倍数就可以了，免去了重新计算位置的运算，重新计算采用的方法是 $\text{key.hash} \& \text{oldCap} == 0$ 在原位置，否则则位移 oldCap 。1.7中，移动链表，是会倒序排列到新链表，所以会有循环指针问题，也就是死锁问题，1.8中是正序排列，没有死锁问题。



```
/**
 *
 * AVL树 红黑树 B+树
 * HashMap线程不安全的表现&线程不安全的原因&改进方法
 * 线程安全：共享资源，在多线程竞争情况下，每次执行均可预测出固定结果。
 * 表现：1 多个线程读取，多个线程写入会报错ConcurrentModificationException（每次修改modCount++）
 *       2 多线程写入，会出现值覆盖的情况（一个线程执行到链表头赋值时，挂起，另一个线程执行完，此时上个线程继续执行，则出现值覆盖的情况）
 * 改进方法：
 *       1 HashTable 方法加了synchronized关键字,锁定的是整个table对象
```

```

*      2 Collections.synchronizedMap 方法会生成一个Collections内部类
SynchronizedMap,
*      该内部类有两个属性Map<K,V> m, final Object      mutex;其中mutex就是锁对象,
方法执行前需要锁定mutex.
*      3 ConcurrentHashMap 分段锁
*
*
* 1.7中死循环的过程分析
* 头插法的关键代码
* //从旧数组中遍历, 先遍历数组, 然后遍历数组中的某个位置上的链表
* //先找出节点在数组中的位置, 然后把新数组中i位置上的值赋值给变量, 把变量赋值给数组中i的位置,
最后把变量指向原来的下一个变量
* void transfer(Entry[] newTable, boolean rehash) {
*     int newCapacity = newTable.length;
*     for (Entry<K,V> e : table) {
*         while(null != e) {
*             Entry<K,V> next = e.next;
*             if (rehash) {
*                 e.hash = null == e.key ? 0 : hash(e.key);
*             }
*             int i = indexFor(e.hash, newCapacity);
*             e.next = newTable[i];
*             newTable[i] = e;
*             e = next;
*         }
*     }
* }
*
* 1.8中HashMap
* 有四个容量需要注意: 1数组的大小 2数组被占用的大小 3链表的大小 4node节点的总个数
* 常量:
*     DEFAULT_INITIAL_CAPACITY = 1 << 4
*     默认初始化数组大小--此处请注意是数组的大小
*     DEFAULT_LOAD_FACTOR = 0.75f
*     默认的加载因子--通过加载因子和数组容量, 可以得出一个扩容的阈值。例如数组大小是16,
*     加载因子是0.75, 则扩容的阈值是16*0.75=12, 意思是如果node的数量大于12则需要扩容。
*     此处注意, hashmap中已经存储的node数量, 并不是数组中占有的数量, 如果所放位置为
空, 不会触发扩容, 非空才会触发扩容
*     TREEIFY_THRESHOLD = 8
*     MIN_TREEIFY_CAPACITY = 64
*     如果一个链表中的node个数超过8&数组的长度大于等于64, 则把该链表变成红黑树(注意其
他链表不会变成红黑树)
*     UNTREEIFY_THRESHOLD = 6
*     如果一个链表中的node个数小于6个, 则把红黑树变成链表(不需要考虑数组长度, 因为数组
长度不会再变小了)
*
* 变量:
* Node<K,V>[] table--HashMap底层是数组+链表的结构
* int size--HashMap中Node节点的个数
* int modCount--HashMap被修改的次数, 用户快速失败
* int threshold--阈值, 超过此值, 就会引发resize.
*     如果在构造函数中, 表示数组大小, 构造函数中如果指定了数组大小, 最终会找到一个比指定值大
的最小的2的N次幂的数值作为数组的最终大小,
*     但是在第一次put是, 此threshold会作为新的table的大小, 然后重新计算阈值。
* float loadFactor--加载因子
*
* 构造方法-无参数
* public HashMap() {

```

```

*     this.loadFactor = DEFAULT_LOAD_FACTOR;
* }
*
* 构造方法-有参数
* public HashMap(int initialCapacity) {
*     this(initialCapacity, DEFAULT_LOAD_FACTOR);
* }
*
* public HashMap(int initialCapacity, float loadFactor) {
*     this.loadFactor = loadFactor;
*     this.threshold = tableSizeFor(initialCapacity);
* }
*
* //使用无符号右移和或运算，找出比当前值大的，最小的2的N次幂
* static final int tableSizeFor(int cap) {
*     int n = cap - 1;
*     n |= n >> 1;
*     n |= n >> 2;
*     n |= n >> 4;
*     n |= n >> 8;
*     n |= n >> 16;
*     return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n +
1;
* }
*
* hashMap.put方法流程
*
* 1 使用hash方法获取key值的hash值
* public V put(K key, V value) {
*     return putVal(hash(key), key, value, false, true);
* }
*
* 2 首先通过key的hashCode方法获取到hash值，该值是int类型，注意key=null的hash值是0，即会
放在数组中的第一个位置
*     再通过扰动函数，获取到最终的hash值，扰动函数是高16位与低16位做异或运算，降低冲突的概率
*     int类型的hashCode返回的是int本身
*     String类型的hashCode返回的是s[0]*31^[n-1]
* static final int hash(Object key) {
*     int h;
*     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >> 16);
* }
*
* 3 put方法源码解析
* hash--通过hash方法返回的key的hash值
* onlyIfAbsent--是否使用新值覆盖旧址
*
* 梳理两个流程
*
* 3.1 空的HashMap第一次put值
*     发现数组为空->扩容->计算该节点在数组中的位置->放在此位置上结束
*
* 3.2 HashMap-put值之后扩容
*     发现数组不为空->
* final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
*                 boolean evict) {
*     Node<K,V>[] tab; Node<K,V> p; int n, i;
*     //如果tab为空，表明是第一次PUT值，则进行扩容--懒加载的模式
*     if ((tab = table) == null || (n = tab.length) == 0) {
*         n = (tab = resize()).length; //返回的是新数组的大小
*     }

```



```

*          //采用(n - 1) & hash来确定该Node在数组中的位置
*          if ((p = tab[i = (n - 1) & hash]) == null) {
*              //如果该位置上没有值，则创建一个Node节点放入此位置
*              tab[i] = newNode(hash, key, value, null);
*          } else {
*              //该位置上有值的逻辑--参考hasNode伪方法
*          }
*          ++modCount;
*          //如果默认情况，第12次put对象并不会扩容，第13次才会扩容
*          if (++size > threshold)
*              resize();
*          afterNodeInsertion(evict);
*          return null;
*      }
*
*      //进入此方法，意味着该位置上的链表不为空，至少有一个node
*      hasNode()伪方法 {
*          //p是table[i]上的node
*          Node<K,V> e; K k;
*          //如果p节点的hash值与新节点相同&&key相同（包含==与equals两种情况）
*          if (p.hash == hash &&
*              ((k = p.key) == key || (key != null && key.equals(k)))) {
*              e = p;
*              //如果节点属于TreeNode节点，即已经变成红黑树节点
*          } else if (p instanceof TreeNode) {
*              e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
*
*              //链表的插入形式--1 循环遍历列表，如果key相同，则进行替换
*              //如果key不相同，则看是否需要树化，如果不需要树化，则创建新节点，并放在队列的
尾部(1.8是尾插法、1.7是头查法)
*              } else {
*                  for (int binCount = 0; ; ++binCount) {
*                      if ((e = p.next) == null) {
*                          p.next = newNode(hash, key, value, null);
*                          //如果是链表中第9个元素，则开启树化操作--如果数组大小
<MIN_TREEIFY_CAPACITY(64),不树化，而是扩容
*                          if (binCount >= TREEIFY_THRESHOLD - 1)
*                              treeifyBin(tab, hash);
*                          break;
*                      }
*                      if (e.hash == hash &&
*                          ((k = e.key) == key || (key != null &&
key.equals(k))))
*                          break;
*                      p = e;
*                  }
*              }
*          //如果发现了key相同的node,则使用旧值覆盖新值
*          if (e != null) {
*              v oldValue = e.value;
*              if (!onlyIfAbsent || oldValue == null)
*                  e.value = value;
*              afterNodeAccess(e);
*              return oldValue;
*          }
*      }
*
*      //treeifyBin中，如果数组大小<MIN_TREEIFY_CAPACITY(64),不树化，而是扩容

```

```

*         treeifyBin()伪代码 {
*             if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY) {
*                 resize();
*             }
*         }
*
* 4 扩容方法，返回的是新数组
* final Node<K,V>[] resize() {
*     Node<K,V>[] oldTab = table;
*     int oldCap = (oldTab == null) ? 0 : oldTab.length;
*     int oldThr = threshold;
*     int newCap, newThr = 0;
*     //如果hashMap中原来有值，则扩容成原来的两倍，使用左移操作，此处如果就数组的大小大于等于16则阈值也扩容成原来的两倍，否则，阈值是用新容量*加载因子
*     if (oldCap > 0) {
*         if (oldCap >= MAXIMUM_CAPACITY) {
*             threshold = Integer.MAX_VALUE;
*             return oldTab;
*         }
*         else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
*                 oldCap >= DEFAULT_INITIAL_CAPACITY)
*             newThr = oldThr << 1; // double threshold
*     }
*     //此处new HashMap(16)这类的构造函数生成的会走入此方法
*     // initial capacity was placed in threshold
*     else if (oldThr > 0)
*         newCap = oldThr;
*     //此处new HashMap()这类的构造函数生成的会走入此方法
*     // zero initial threshold signifies using defaults
*     else {
*         newCap = DEFAULT_INITIAL_CAPACITY;
*         newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
*     }
*     //如果新阈值是0，则使用新容量*加载因子重新计算阈值
*     if (newThr == 0) {
*         float ft = (float)newCap * loadFactor;
*         newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
*             (int)ft : Integer.MAX_VALUE);
*     }
*     threshold = newThr;
*     //重新生成一个新的Node数组
*     Node<K, V>[] newTab = (Node<K,V>[])new Node[newCap];
*     table = newTab;
*     //如果旧数组不为空，则使用循环遍历的方式，把原来节点上的节点放入新的数组中
*     if (oldTab != null) {
*         for (int j = 0; j < oldCap; ++j) {
*             Node<K,V> e;
*             if ((e = oldTab[j]) != null) {
*                 oldTab[j] = null;
*                 //如果链表上只有一个值，则重新计算在新数组中的位置，然后放入
*                 if (e.next == null)
*                     newTab[e.hash & (newCap - 1)] = e;
*                 else if (e instanceof TreeNode)
*                     //树化的操作
*                     ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
*                 else {
*                     //低位的头节点和尾节点

```

```

*         Node<K,V> loHead = null, loTail = null;
*         //高位的头节点和尾节点
*         Node<K,V> hiHead = null, hiTail = null;
*         Node<K,V> next;
*         //循环遍历
*         do {
*             next = e.next;
*             //采用的是&操作, e.hash & oldCap == 0 则放入低位, 否则
放入高位
*             if ((e.hash & oldCap) == 0) {
*                 if (loTail == null)
*                     loHead = e;
*                 else
*                     loTail.next = e;
*                 loTail = e;
*             }
*             else {
*                 if (hiTail == null)
*                     hiHead = e;
*                 else
*                     hiTail.next = e;
*                 hiTail = e;
*             }
*         } while ((e = next) != null);
*         //低位的放在原来的位置
*         if (loTail != null) {
*             loTail.next = null;
*             newTab[j] = loHead;
*         }
*         //高位的放在原来+oldCap的位置
*         if (hiTail != null) {
*             hiTail.next = null;
*             newTab[j + oldCap] = hiHead;
*         }
*     }
* }
*
*     return newTab;
* }
*/

```