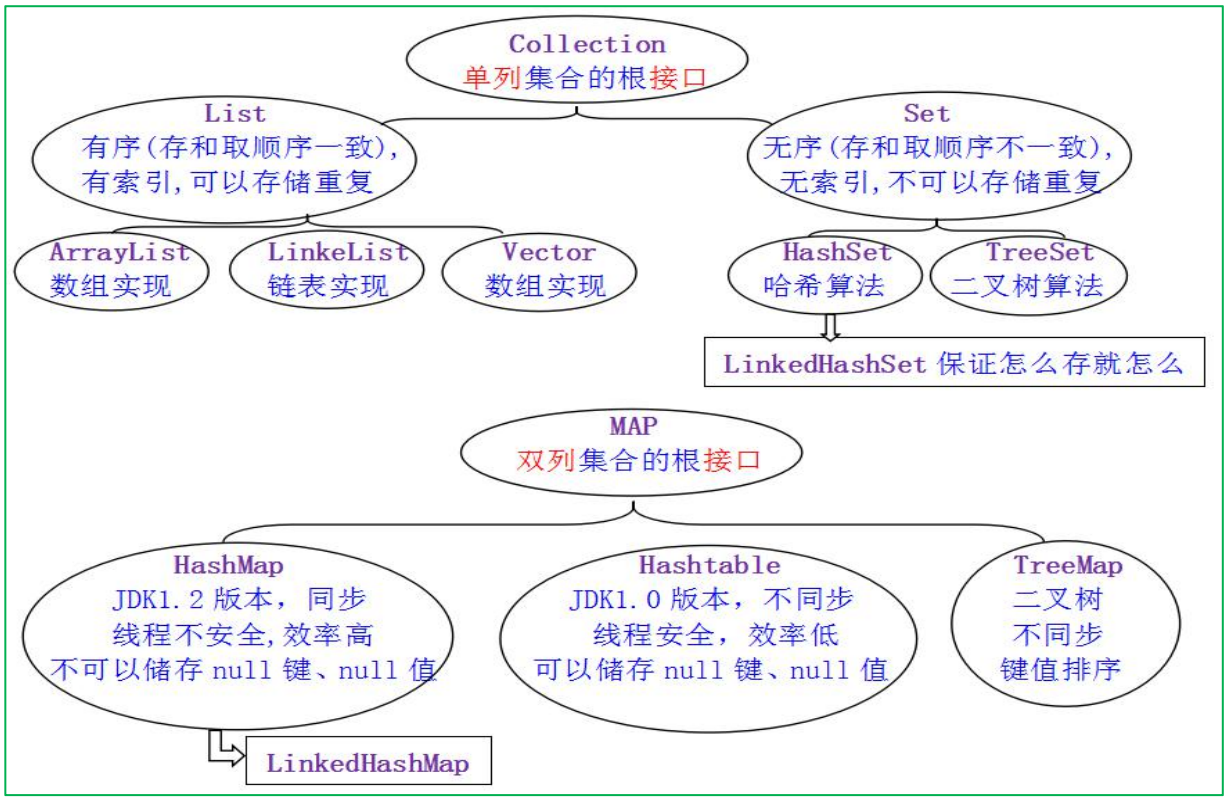
秒杀 Java 面试官——集合篇

一、集合的大体架构图

希望大家能牢牢记住下面这张框架图，一旦面试官让你“说说集合吧”，希望大家能立马给他画出来，边画边逐一介绍每个集合的特点，以及彼此的差异。重点是要从底层源代码的角度来给面试官分析。

一说到底层代码，可能很多人就头疼了，总认为知道和不知道对开发根本没多大实用价值，会应用就行了。这个观点，我暂不做评论。但是大家很庆幸的是，看到了本篇博客，博主将会带大家，来分析一些在面试中一定会用到的集合底层实现原理。



二、List

1.ArrayList 和 Vector 的区别

第一句话：ArrayList 和 Vector 底层都是数组实现的，初始容量都为 10；在 ArrayList 的底层，是通过定义一个 ***DEFAULT_CAPACITY*** 的常量来指定的，而 Vector 的底层，是直接在空参构造中，通过写死了一个 `this(10)` 来指定的；

（PS:标**黑**色的部分，估计学过 Java 的人都会说，人云亦云，但是你要是在面试中，说出了后面标红的部分，虽然意思一样，但是，你这样一讲，绝对瞬间提升了一个档次，立马就能够脱颖而出。后面的解析同理：**黑**色是众人皆知的常识，红色字体才是精髓）

ArrayList 源码片段：

```
/**
 * Default initial capacity.默认初始容量
 */
private static final int DEFAULT_CAPACITY = 10;
```

Vector 源码片段：

```
/**
 * Constructs an empty vector so that its internal data array
 * has size {@code 10} and its standard capacity increment is
 * zero.空参构造,其内部数据数组的大小为 10，并且它的标准容量增量为零
 */
public Vector() {
    this(10);
}
```

第二句话:Vector 大部分方法的底层实现,都加了 synchronized 关键字,所以 Vector 是线程同步的，而 ArrayList 不是;

Vector 源码片段：

```
/**
 * Returns the number of components in this vector.
 */
public synchronized int size() {
    return elementCount;
}
/**
 * Tests if this vector has no components.
 */
public synchronized boolean isEmpty() {
    return elementCount == 0;
}
```

★第三句话：在查看 API 时，发现 Vector 有 4 个构造方法，比 ArrayList 多了一个。而多的这个构造方法，是跟扩容有关的。 ArrayList 默认的扩容，在 JDK1.6 时，是按照新容量 =（原容量*3）/2+1 来计算的，大约 50%左右；而在 JDK1.7 以后，是按照新容量 = 原容量 +（原容量 >> 1）来计算的，大约也在 50%左右，所以都不是很多资料上说的就是 50%,同时由于位运算的速度比快，所以 ArrayList 在 JDK1.7 之后效率更高，也可以看出来，；而在 Vector 中，默认情况下，是 100%增长的，但是我们可以通过比 ArrayList 多的那个构造方法，来指定它增容的大小。

构造方法摘要
ArrayList() 构造一个初始容量为 10 的空列表。
ArrayList(Collection<? extends E> c) 构造一个包含指定 collection 的元素的列表，这些元素是按照该 collection 的迭代器返回它们的顺序排列的。
ArrayList(int initialCapacity) 构造一个具有指定初始容量的空列表。

构造方法摘要
Vector() 构造一个空向量，使其内部数据数组的大小为 10，其标准容量增量为零。
Vector(Collection<? extends E> c) 构造一个包含指定 collection 中的元素的向量，这些元素按其 collection 的迭代器返回元素的顺序排列。
Vector(int initialCapacity) 使用指定的初始容量和等于零的容量增量构造一个空向量。
Vector(int initialCapacity, int capacityIncrement) 使用指定的初始容量和容量增量构造一个空的向量。

ArrayList 源码片段：

```
/**
 * jdk1.6 中: int newCapacity = (oldCapacity * 3)/2 + 1;
 */
public void ensureCapacity(int minCapacity) {
    modCount++;
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity * 3)/2 + 1;
          if (newCapacity < minCapacity)
       newCapacity = minCapacity;
          // minCapacity is usually close to size, so this is a win:
           elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
/**
 * jdk1.7 之后: int newCapacity = oldCapacity + (oldCapacity >> 1);
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

Vector 源码片段：

```
/**
 * Vector中 int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                     capacityIncrement : oldCapacity);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

2. ArrayList 与 LinkedList

第一句话：ArrayList 是实现了基于动态数组的数据结构，LinkedList 基于链表的数据结构，它继承于 AbstractSequentialList 的双向链表，由于 AbstractSequentialList 实现了 get(i)、set()、add() 和 remove()这些骨干性函数，这也降低了 List 接口的复杂程度。

LinkedList 源码片段：

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

第二句话：ArrayList 与 LinkedList 都是不是同步的。如果多个线程同时访问一个链接列表，而其中至少一个线程从结构上修改了该列表，则它必须保持外部同步。同步的方法就是使用 Collections.synchronizedList(Collection<T> c)来“包装”该列表。

static <T> Collection<T>	synchronizedCollection(Collection<T> c) 返回指定 collection 支持的同步（线程安全的）

第三句话：对于随机访问 get 和 set，ArrayList 绝对优于 LinkedList，因为从源码可以看出，ArrayList 想要 get(int index)元素时，直接返回 index 位置上的元素；而 LinkedList 需要通过 for 循环进行查找，虽然 LinkedList 已经在查找方法上做了优化，比如 index < size / 2，则从左边开始查找，反之从右边开始查找，但是还是比 ArrayList（随机查找）要慢。

ArrayList 源码片段：

```
public E get(int index) {
    rangeCheck(index);
    checkForComodification();
    return ArrayList.this.elementData(offset + index); // 直接返回索引
}
```

LinkedList 源码片段：

```
public E get(int index) {
    checkElementIndex(index);
    return node(index).item; // get()方法会调用 node()方法，
}
//node()方法需要循环遍历查找索引；
Node<E> node(int index) {
    // assert isElementIndex(index);
        if (index < (size >> 1)) {  // 加速机制
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

第四句话：从源码来看，ArrayList 想要在指定位置插入或删除元素时，主要耗时的是 System.arraycopy 动作，会移动 index 后面所有的元素；LinkedList 主耗时的是要先通过 for 循环找到 index，然后直接插入或删除。这就导致了两者并非一定谁快谁慢！

具体情况，可参看如下测试数据（数据来源于网络）：

```
#在 index=1000 出插入结果：
    array time:4
    linked time:240
    array insert time:20
    linked insert time:18
#在 index=5000 处插入结果：
    array time:4
    linked time:229
    array insert time:13
    linked insert time:90
#在 index=9000 处插入结果：
    array time:4
    linked time:237
    array insert time:7
    linked insert time:92
```

所以，不是很多资料中说得那样简单：以为 ArrayList 查询快，增删慢，因为它是集合实现的，要改角标；而 LinkedList 是链表实现的，所以查询慢，增删快！！

结论：当插入的数据量很小时，两者区别不太大；当插入的数据量大时，大约在容量的

1/10 之前，LinkedList 会优于 ArrayList；在其后就完全劣于 ArrayList，且越靠近后面越差。所以个人觉得，一般首选用 ArrayList，因为 LinkedList 还可以实现栈、队列以及双端队列等数据结构。

三、HashMap 底层实现原理（基于 JDK1.8）

面试中，你是否也曾被问过以下问题呢：

你知道 HashMap 的数据结构吗？HashMap 是如何实现存储的？底层采用了什么算法？为什么采用这种算法？如何对 HashMap 进行优化？如果 HashMap 的大小超过了负载因子定义的容量，怎么办？等等。

有觉得很难吗？别怕！下面博主就带着大家深度剖析，以源代码为依据，逐一分析，看看 HashMap 到底是怎么玩的：

① HashMap 源码片段 —— 总体介绍：

```
/* Hash table based implementation of the <tt>Map</tt> interface（HashMap 实现了 Map 接口）. This implementation provides all of the optional map operations, and permits <tt>null</tt> values and the <tt>null</tt> key（允许储存 null 值和 null 键）. (The <tt>HashMap</tt> class is roughly equivalent to <tt>Hashtable</tt>, except that it is unsynchronized and permits nulls.(HashTable 和 HashMap 很相似，除了 HashTable 的方法是同步的，并且不允许储存 null 值和 null 键)) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time(HashMap 不保证映射的顺序，特别是它不保证该顺序不随时间变化).*/
```

② HashMap 源码片段 —— 六大初始化参数：

```
    /**
     * 初始容量 1 << 4 = 16
     */
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
    /**
     * 最大容量 1 << 30 = 1073741824
     */
    static final int MAXIMUM_CAPACITY = 1 << 30;
    /**
     * 默认负载因子 0.75f
     */
    static final float DEFAULT_LOAD_FACTOR = 0.75f;
    /**
     * 由链表转换成树的阈值：即当 bucket(桶)中 bin（箱子）的数量超过
     * TREEIFY_THRESHOLD 时使用树来代替链表。默认值是 8
     */
    static final int TREEIFY_THRESHOLD = 8;
    /**
     * 由树转换成链表的阈值：当执行 resize 操作时，当 bucket 中 bin 的数量少于此值，
```

```
     * 时使用链表来代替树。默认值是 6
     */
    static final int UNTREEIFY_THRESHOLD = 6;
    /**
     * 树的最小容量
     */
    static final int MIN_TREEIFY_CAPACITY = 64;
```

③ HashMap 源码片段 —— 内部结构：

```
    /**
     * Basic hash bin node, used for most entries.
     */
    // Node 是单向链表，它实现了 Map.Entry 接口
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;   // 键对应的 Hash 值
        final K key;      // 键
        V value;          // 值
        Node<K,V> next;   // 下一个节点
        // 构造函数
        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
    // 存储（位桶）的数组</k,v>
    transient Node<K,V>[] table;
    // 红黑树
    static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
        TreeNode<K,V> parent; // 父节点
        TreeNode<K,V> left;   // 左节点
        TreeNode<K,V> right;  // 右节点
        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);
        }
```

简单看：在 JDK1.8 中，HashMap 采用位桶+链表+红黑树实现。具体实现原理，我们继续看源码。关于红黑树，我将在后期《算法篇》详细介绍。

④ HashMap 源码片段 —— 数组 Node[]位置：

```
    // 第一步：先计算 key 对应的 Hash 值
    static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }
```

```
// 第二步：保证哈希表散列均匀
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;
}
```

★对第二步的作用，进行简要说明（很高级！）：

{ 可以从源码看出，在 HashMap 的构造函数中，都直接或间接的调用了 **tableSizeFor** 函数。下面分析原因：length 为 2 的整数幂保证了 length-1 最后一位（当然是二进制表示）为 1，从而保证了取索引操作 h&（length-1）的最后一位同时有为 0 和为 1 的可能性，保证了散列的均匀性。反过来讲，当 length 为奇数时，length-1 最后一位为 0，这样与 h 按位与的最后一位肯定为 0，即索引位置肯定是偶数，这样数组的奇数位置全部没有放置元素，浪费了大量空间。简而言之：length 为 2 的幂保证了按位与最后一位的有效性，使哈希表散列更均匀。}

```
// 第三步：计算索引：index = (tab.length - 1) & hash
if (tab == null || (n = tab.length) == 0) return;
int index = (n - 1) & hash;
```

（区别于 HashTable ：index = (hash & 0x7FFFFFFF) % tab.length;

取模中的除法运算效率很低，但是 HashMap 的位运算效率很高）

⑤ HashMap 源码片段 —— 常用 get（）/put（）操作：

```
/**
 * Implements Map.get and related methods
 *
 * @param hash hash for key
 * @param key the key
 * @return the node, or null if none
 */
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 &&
        //tab[(n - 1) & hash]得到对象的保存位
        (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) {
```

```
                //判断：如果第一个节点是 TreeNode,则采用红黑树处理冲突
                if (first instanceof TreeNode)
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do {
                    //反之，采用链表处理冲突
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
        }
```

```
    /**
     * 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;
        if ((tab = table) == null || (n = tab.length) == 0)
            //如果 tab 为空或长度为 0，则分配内存 resize()
            n = (tab = resize()).length;
        if ((p = tab[i = (n - 1) & hash]) == null)
            //tab[i = (n - 1) & hash]找到 put 位置，如果为空,则直接 put
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
             //先判断 key 的 hash()方法判断，再调用 equals()方法判断
            if (p.hash == hash &&  ((k = p.key) == key || (key != null &&
key.equals(k))))
                e = p;
            else if (p instanceof TreeNode)
              //属于红黑树处理冲突
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
            else {
```

```
                //链表处理冲突
                for (int binCount = 0; ; ++binCount) {
                    //p 第一次指向表头,之后依次后移
                    if ((e = p.next) == null) {
                        //e 为空，表示已到表尾也没有找到 key 值相同节点，则新建节点
                        p.next = newNode(hash, key, value, null);
                        //新增节点后如果节点个数到达阈值，则将链表转换为红黑树
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    //允许存储 null 键 null 值
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    //指针下移一位
                    p = e;
                }
            }
            //更新 hash 值和 key 值均相同的节点 Value 值
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
        }
```

⑥ HashMap 源码片段 —— 扩容 resize（）：

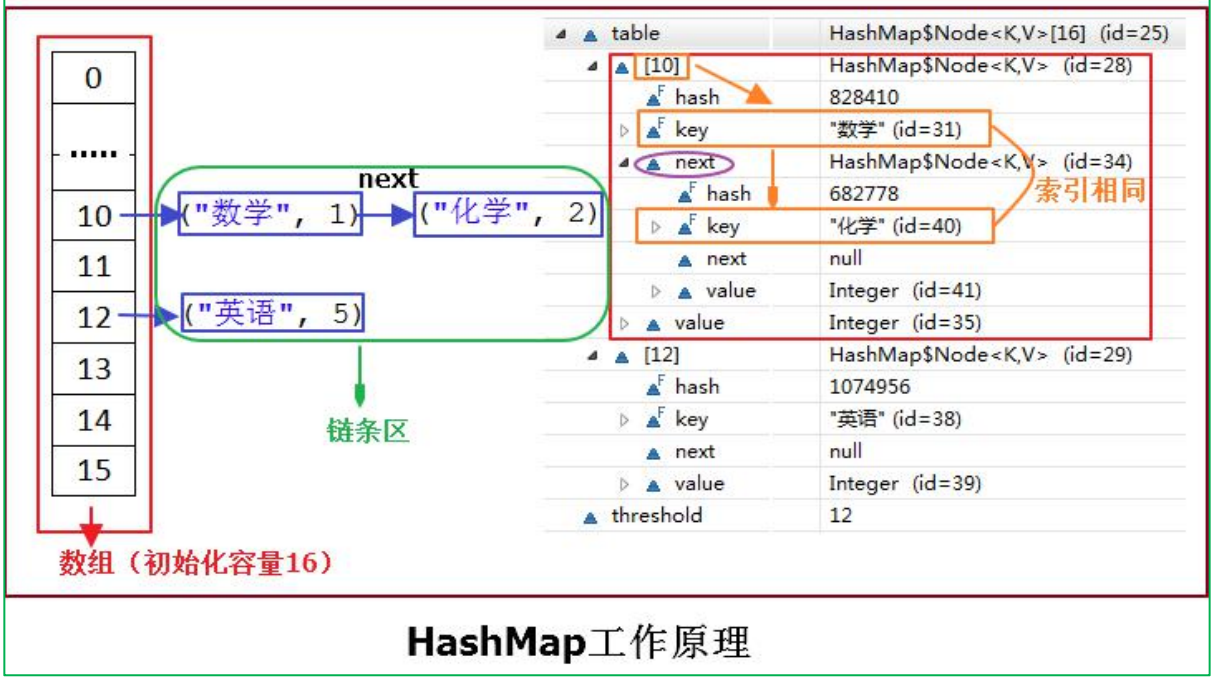
```
    //可用来初始化 HashMap 大小 或重新调整 HashMap 大小 变为原来 2 倍大小
    final Node<K,V>[] resize() {
        Node<K,V>[] oldTab = table;
        int oldCap = (oldTab == null) ? 0 : oldTab.length;
        int oldThr = threshold;
        int newCap, newThr = 0;
        if (oldCap > 0) {
```

```
            if (oldCap >= MAXIMUM_CAPACITY) {
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY)
                newThr = oldThr << 1; // 扩容阈值加倍
        }
        else if (oldThr > 0)  // oldCap=0 ,oldThr>0 此时 newThr=0
            newCap = oldThr;
        else {   // oldCap=0,oldThr=0 相当于使用默认填充比和初始容量 初始化
            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"})
            Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        // 数组辅助到新的数组中，分红黑树和链表讨论
        table = newTab;
        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)
                        newTab[e.hash & (newCap - 1)] = e;
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    else { // preserve order
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
                        do {
                            next = e.next;
                            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);
                if (loTail != null) {
                    loTail.next = null;
                    newTab[j] = loHead;
                }
                if (hiTail != null) {
                    hiTail.next = null;
                    newTab[j + oldCap] = hiHead;
                }
            }
        }
    }
    return newTab;
    }
```

看完以上源码，是否感觉身体被掏空了？别慌，博主现在以一个简单的小例子为主导，带领大家重新梳理一下。

```
Map<String,Integer> map = new HashMap<>();
map.put("数学", 1);
map.put("化学", 2);
map.put("英语", 5);
```



HashMap工作原理

简析底层实现过程：

①创建 HashMap，初始容量为 16，实际容量 = 初始容量*负载因子（默认 0.75）= 12；

②调用 put 方法，会先计算 key 的 hash 值：hash = key.hashCode()。

③调用 tableSizeFor()方法，保证哈希表散列均匀。

④计算 Nodes[index]的索引：先进行 index = (tab.length - 1) & hash。

⑤如果索引位为 null，直接创建新节点，如果不为 null，再判断所因为上是否有元素

⑥如果有：则先调用 hash()方法判断，再调用 equals()方法进行判断，如果都相同则直接用新的 Value 覆盖旧的；

⑦如果不同，再判断第一个节点类型是否为树节点（涉及到：链表转换成树的阈值，默认 8），如果是，则按照红黑树的算法进行存储；如果不是，则按照链表存储；

⑧当存储元素过多时，需要进行扩容：

默认的负载因子是 0.75，如果实际元素所占容量占分配容量的 75%时就要扩容了。大约变为原来的 2 倍（newThr = oldThr << 1）；