1. JAVA容器有哪些

Java 容器分为 Collection 和 Map 两大类，其下又有很多子类，如下所示是Collection和Map的继承体系：

- Collection
  - List
    - ArrayList
    - LinkedList
    - Vector
    - Stack
  - Set
    - HashSet
    - LinkedHashSet
    - TreeSet
- Map
  - HashMap
    - LinkedHashMap
  - TreeMap
  - ConcurrentHashMap
  - Hashtable

2. HashMap的数据结构是什么样的

HashMap本质是一个一定长度的数组，数组中存放的是链表。它是一个Entry类型的数组，Entry的源码：

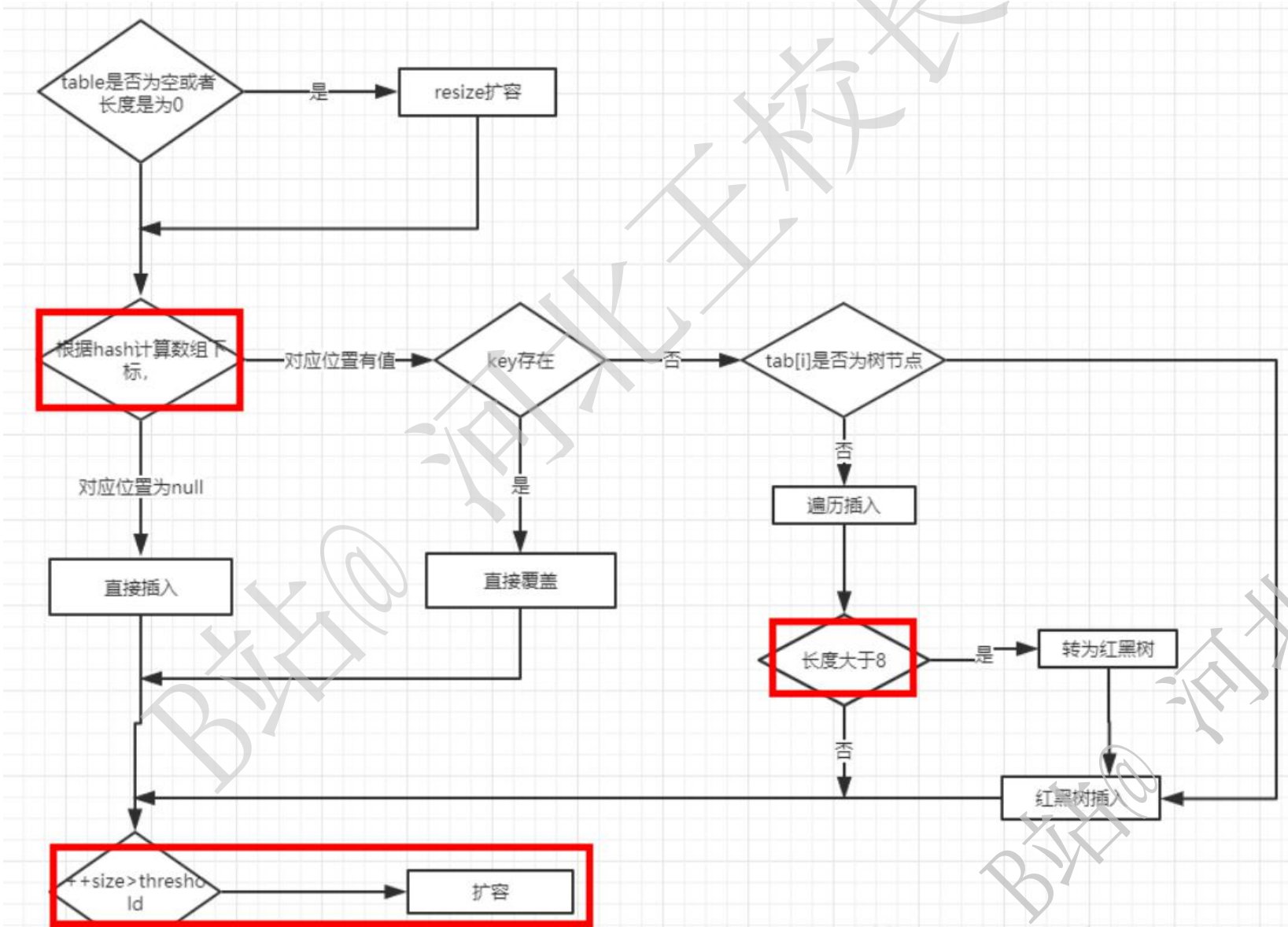
```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    final int hash;
    Entry<K,V> next;
}
```

其中存放了Key，Value，hash值，还有指向下一个元素的引用。

当向HashMap中put(key,value)时，会首先通过hash算法计算出存放到数组中的位置，比如位置索引为i，将其放入到Entry[i]中，如果这个位置上面已经有元素了，那么就将新加入的元素放在链表的头上(JDK1.7 是头插，JDK1.8是尾插)，最先加入的元素在链表尾。比如，第一个键值对A进来，通过计算其key的hash得到的index=0，记做:Entry[0] = A。一会后又进来一个键值对B，通过计算其index也等于0，现在怎么办？HashMap会这样做:B.next = A,Entry[0] = B,如果又进来C,index也等于0,那么C.next = B,Entry[0] = C；这样我们发现index=0的地方其实存取了A,B,C三个键值对,他们通过next这个属性链接在一起,也就是说数组中存储的是最后插入的元素。

3. HashMap的put方法实现原理(源代码讲解)

JDK7中HashMap采用的是位桶+链表的方式，即我们常说的散列链表的方式，而JDK8中采用的是位桶+链表/红黑树



```
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table为空，或者还没有元素时，则扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 如果首结点值为空，则创建一个新的首结点。
    // 注意：(n - 1) & hash才是真正的hash值，也就是存储在table位置的index。在1.6中是封装成indexFor函数。
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {     // 到这儿了，就说明碰撞了，那么就要开始处理碰撞。
        Node<K,V> e; K k;
        // 如果在首结点与我们待插入的元素有相同的hash和key值，则先记录。
        if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        else if (p instanceof TreeNode) // 如果首结点的类型是红黑树类型，则按照红黑树方法添加该元素
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
```

```
else if (p instanceof TreeNode) // 如果首结点的类型是红黑树类型，则按照红黑树方法添加该元素
    e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
else {  // 到这一步，说明首结点类型为链表类型。
    for (int binCount = 0; ; ++binCount) {
        // 如果遍历到末尾时，先在尾部追加该元素结点。
        if ((e = p.next) == null) {
            p.next = newNode(hash, key, value, null);
            // 当遍历的结点数目大于8时，则采取树化结构。
            if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                treeifyBin(tab, hash);
                break;
        }
        // 如果找到与我们待插入的元素具有相同的hash和key值的结点，则停止遍历。此时e已经记录了该结点
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
            break;
        p = e;
    }
}
```

```
                    // 如果找到与我们待插入的元素具有相同的hash和key值的结点，则停止遍历。此时e已经记录了该结点
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
            // 表明，记录到具有相同元素的结点
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);  // 这个是空函数，可以由用户根据需要覆盖
                return oldValue;
            }
        }
        ++modCount;
        // 当结点数+1大于threshold时，则进行扩容
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict); // 这个是空函数，可以由用户根据需要覆盖
        return null;
    }
```

4. Hashmap 的put方法的参数hash是怎么计算的

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

a. 当key = null时，hash值 = 0，所以HashMap的key 可为null

注：对比HashTable，HashTable对key直接hashCode（），若key为null时，会抛出异常，所以HashTable的key不可为null

b. 当key ≠ null时，则通过先计算出 key的 hashCode()（记为h），然后 对哈希码进行 扰动处理： 按位 异或（^）哈希码自身右移16位后的二进制

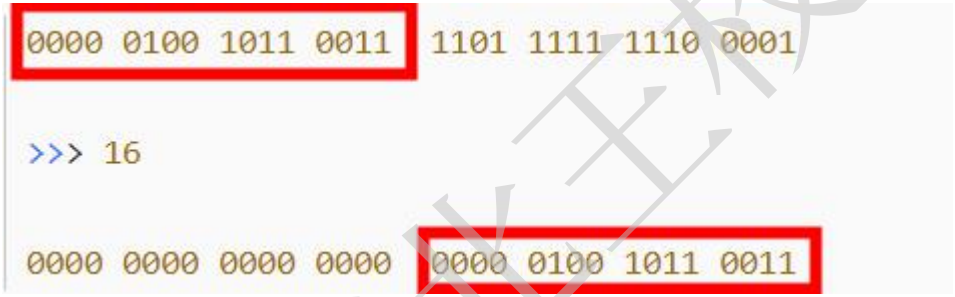
5. Hashmap中插入一条数据，如何计算数据的下标呢

```
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table为空，或者还没有元素时，则扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 如果首结点值为空，则创建一个新的首结点。
    // 注意：(n - 1) & hash才是真正的hash值，也就是存储在table位置的index。在1.6中是封装成indexFor函数。
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {     // 到这儿了，就说明碰撞了，那么就要开始处理碰撞。
            Node<K,V> e; K k;
            // 如果在首结点与我们待插入的元素有相同的hash和key值，则先记录。
            if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            else if (p instanceof TreeNode) // 如果首结点的类型是红黑树类型，则按照红黑树方法添加该元素
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
```

步骤	代码实现	具体计算过程
1. 计算哈希码	h = key.hashCode()	根据 键key，通过 hashCode（）计算 hashCode（）简介 • 定义：Object类的方法，即所有Java对象都可调用 • 作用：根据对象的内存地址 经过特定算法返回一个 哈希码 • 意义：保证每个对象的 哈希码尽可能不同，从而提高在散列结构存储中查找的效率 • 注：可复写，Object 类提供的默认实现确实保证每个对象的 hash 码不同
2. 二次处理哈希码 （最终求得 键对应的hash值）	h ^ (h >>> 16)	• 该处理也称：扰动处理 • 即 哈希码 异或（^） 哈希码自身右移16位后的二进制 • 本质：二次处理低位 = 哈希码的高16位不变、低16位 = 低16位 异或 高16位 （即 高位参与低位的计算）
3. 最终计算存储的数组位置 （根据hash值 & 数组长度）	h & (length-1)	即 二次处理后的哈希码 与运算(&) （数组长度-1）

6. 为什么hash要进行右移16位的异或计算

h >>> 16是用来**取出h的高16**，(>>>是无符号右移) 如下展示：



由于和最终和（length-1）运算，length 绝大多数情况小于2的16次方。所以始终是hashcode 的低16位（甚至更低）参与运算。要是高16位也参与运算，会让得到的下标更加散列。所以这样高16位是用不到的，如何让高16也参与运算呢。所以才有hash(Object key)方法。让他的hashCode()和自己的高16位^运算。所以(h >>> 16)得到他的高16位与hashCode()进行^运算。

例如1：为了方便验证，假设length为8。HashMap的默认初始容量为16。（length-1） = 7；
转换二进制为111；假设一个key的 hashcode = 78897121 转换二进制：
10010110011110111111100001，与（length-1）& 运算如下

```
    0000 0100 1011 0011 1101 1111 1110 0001

&运算

    0000 0000 0000 0000 0000 0000 0000 0111

=   0000 0000 0000 0000 0000 0000 0000 0001  （就是十进制1，所以下标为1）
```

上述运算实质是：001 与 111 & 运算。也就是哈希值的低三位与length与运算。如果让哈希值的低三位更加随机，那么&结果就更加随机，如何让哈希值的低三位更加随机，那么就是让其与高位异或。右位移16位，正好是32bit的一半，自己的高半区和低半区做异或，就是为了混合原始哈希码的高位和低位，以此来加大低位的随机性。而且混合后的低位掺杂了高位的部分特征，这样高位的信息也被变相保留下来。

7.为什么用^而不用&和|

假设均匀随机（1位）输入，AND函数输出概率分布分别为75% 0和25% 1。相反，OR为25% 0和75% 1。
XOR函数为50% 0和50% 1，因此对于合并均匀的概率分布非常有用

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

8.JDK1.8 hashmap为什么引入红黑树

由于在JDK1.7之前，HashMap的数据结构为：数组 + 链表。

链表来存储hash值一样的key-value. 如果按照链表的方式存储，**随着节点的增加数据会越来越多**，这会导致查询节点的时间复杂度会逐渐增加，**平均时间复杂度O(n)**。 为了提高查询效率，故在JDK 1.8中引入了改进方法**红黑树。此数据结构的平均查询效率为O(long n) 。**

9. JDK1.8以后的hashmap为什么在链表长度为8的时候变为红黑树

在JDK1.8以及以后的版本中，hashmap的底层结构，由原来单纯的的数组+链表，更改为链表长度为8时，开始由链表转换为红黑树，我们都知道，链表的时间复杂度是O(n)，红黑树的时间复杂度O(logn)，很显然，红黑树的复杂度是优于链表的。因为树节点所占空间是普通节点的两倍，所以只有当节点足够多的时候，才会使用树节点。也就是说，节点少的时候，尽管时间复杂度上，红黑树比链表好一点，但是红黑树所占空间比较大，综合考虑，认为只能在节点太多的时候，红黑树占空间大这一劣势不太明显的时候，才会舍弃链表，使用红黑树，这也是为什么不直接全部使用红黑树的原因。

那为什么选择8才会选择使用红黑树呢？

在理想状态下，受随机分布的hashCode影响，链表中的节点遵循**泊松分布**，而且根据统计，链表中节点数是8的概率已经接近千分之一，而且此时链表的性能已经很差了。所以在这种比较罕见和极端的情况下，才会把链表转变为红黑树。因为链表转换为红黑树也是需要消耗性能的，特殊情况特殊处理，为了挽回性能，权衡之下，才使用红黑树，提高性能。

10. 泊松分布是什么?

泊松分布的概率函数为:

$$P(X=k)=\frac{\lambda^k}{k!}e^{-\lambda},k=0,1,\cdots$$

泊松分布的参数λ是**单位时间(或单位面积)内随机事件的平均发生次数。 泊松分布适合于描述单位时间内随机事件发生的次数。**

泊松分布的期望和方差均为 λ.

特征函数为 $\psi(t)=exp\{\lambda(e^{it}-1)\}$.

11. 如果链表的节点数大于8，就一定会转换为红黑树吗?

```
else if (p instanceof TreeNode) // 如果首结点的类型是红黑树类型，则按照红黑树方法添加该元素
    e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
else {  // 到这一步，说明首结点类型为链表类型。
        for (int binCount = 0; ; ++binCount) {
            // 如果遍历到末尾时，先在尾部追加该元素结点。
            if ((e = p.next) == null) {
                p.next = newNode(hash, key, value, null);
                // 当遍历的结点数目大于8时，则采取树化结构。
                if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                    treeifyBin(tab, hash);
                    break;
            }
            // 如果找到与我们待插入的元素具有相同的hash和key值的结点，则停止遍历。此时e已经记录了该结点
            if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
                break;
            p = e;
        }
    }
```

我们来看看treeifyBin方法：

```
final void treeifyBin(Node<K,V>[] tab, int hash) {
        int n, index; Node<K,V> e;
    //先判断table的长度是否小于 MIN_TREEIFY_CAPACITY (64)
        if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
    //小于64，则扩容
            resize();
        else if ((e = tab[index = (n - 1) & hash]) != null) {
    //否则才将链表转换为红黑树
            TreeNode<K,V> hd = null, tl = null;
            do {
                TreeNode<K,V> p = replacementTreeNode(e, null);
                if (tl == null)
                    hd = p;
                else {
                    p.prev = tl;
                    tl.next = p;
                }
                tl = p;
            } while ((e = e.next) != null);
            if ((tab[index] = hd) != null)
                hd.treeify(tab);
        }
    }
```

可以看到在treeifyBin中并不是简单地将链表转换为红黑树，而是先判断table的长度是否大于64，如果小于64，就通过扩容的方式来解决，避免红黑树结构化。

12.Hashmap为什么选用红黑树而不用AVL树

AVL树和红黑树有几点比较和区别：
（1）AVL树是更加严格的平衡，因此可以提供更快的查找速度，一般读取查找密集型任务，适用AVL树。
（2）红黑树更适合于插入修改密集型任务。
（3）通常，AVL树的旋转比红黑树的旋转更加难以平衡和调试。

（1）AVL以及红黑树是高度平衡的树数据结构。它们非常相似，真正的区别在于在任何添加/删除操作时完成的旋转操作次数。

（2）两种实现都缩放为a O(lg N)，其中N是叶子的数量，但实际上AVL树在查找密集型任务上更快：利用更好的平衡，树遍历平均更短。另一方面，插入和删除方面，AVL树速度较慢：需要更高的旋转次数才能在修改时正确地重新平衡数据结构。

（3）在AVL树中，从根到任何叶子的最短路径和最长路径之间的差异最多为1。在红黑树中，差异可以是2倍。

（4）两个都给O（log n）查找，但平衡AVL树可能需要O（log n）旋转，而红黑树将需要最多两次旋转使其达到平衡（尽管可能需要检查O（log n）节点以确定旋转的位置）。旋转本身是O（1）操作，因为你只是移动指针。

13. HashMap为什么不直接使用hashCode()处理后的哈希值直接作为table的下标?

hashCode()方法返回的是int整数类型，其范围为-(2 ^ 31)~(2 ^ 31 - 1)，约有40亿个映射空间，而HashMap的容量范围是在16（初始化默认值）~2 ^ 30，HashMap通常情况下是取不到最大值的，并且设备上也难以提供这么多的存储空间，**从而导致通过hashCode()计算出的哈希值可能不在数组大小范围内，进而无法匹配存储位置。**

如何解决呢?

请参看我们的面试题： 为什么进行16位的右移

14.为什么hashmap的数组长度要保证为2的幂次方呢

- 只有当数组长度为2的幂次方时，h&(length-1)才等价于h%length，即实现了key的定位，2的幂次方也可以减少冲突次数，提高HashMap的查询效率；

- 如果 length 为 2 的次幂 则 length-1 转化为二进制必定是 11111……的形式，在与 h 的二进制与操作效率会非常的快，而且空间不浪费；如果 length 不是 2 的次幂，比如 length 为 15，则 length - 1 为 14，对应的二进制为 1110，在与 h 与操作，最后一位都为 0 ，而 0001，0011，0101，1001，1011，0111，1101 这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！这样就会造成空间的浪费。

15. jdk1.7 和 jdk1.8 hashmap有何不同之处

JDK7：

- HashMap底层是数组加链表的形式
- 数组的默认长度为16，加载因子为0.75，也就是16*0.75=12(阈值)当计算出元素的位置在数组中冲突时，那么会以链表的形式存储新的元素，新的元素**插在链表的头部**，然后将链表下移，也就是将数组中的值赋值给新来的元素
- 当数组中12个位置被占据时(也就是达到了阈值)，同时新插入的元素的插入位置不为空，就会进行扩容 2倍扩容
- 并发环境下会产生死锁

JDK8：

- HashMap底层是数组加链表加红黑树
- 数组的默认长度为16，加载因子为0.75，也就是16*0.75=12(阈值)当计算出元素在数组中的位置相同时，则生成链表，并将新的元素**插入到尾部（主要是为了红黑树问题）**，假如链表上元素超过了8个，那么链表将被改为红黑树，同时也提高了增删查效率
- 当数组元素个数达到了阈值，那么此时不需要判断新的元素的位置是否为空，数组都会扩容，2倍扩容
- 并发环境下不会产生死锁

16.Hashmap的主要成员变量（重要参数）

transient Node<K,V>[] table：这是一个Node类型的数组（也有称作Hash桶），可以从下面源码中看到静态内部类Node在这边可以看做就是一个节点，多个Node节点构成链表，当链表长度大于8的时候并且table长度大于64的时候转换为红黑树。

transient int size：表示当前HashMap包含的键值对数量

transient int modCount：表示当前HashMap修改次数

int threshold：表示当前HashMap能够承受的最多的键值对数量，一旦超过这个数量HashMap就会进行扩容

final float loadFactor：负载因子，用于扩容

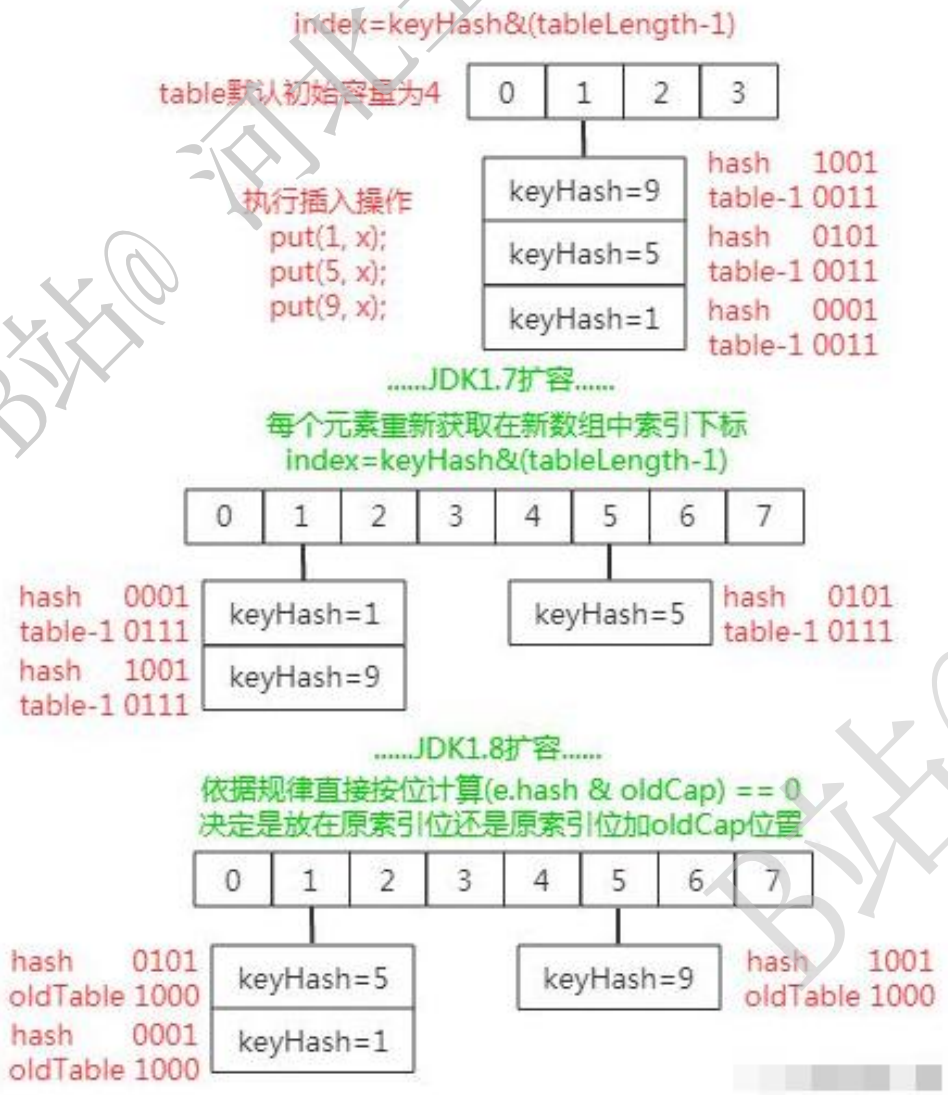
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4：默认的table初始容量

static final float DEFAULT_LOAD_FACTOR = 0.75f：默认的负载因子

static final int TREEIFY_THRESHOLD = 8: 链表长度大于该参数转红黑树

static final int UNTREEIFY_THRESHOLD = 6: 当树的节点数小于该参数转成链表

17.Hashmap JDK1.8 与 1.7 的扩容原理有什么不同



1.7 中整个扩容过程就是一个取出数组元素（实际数组索引位置上的每个元素是每个独立单向链表的头部，也就是发生 Hash 冲突后最后放入的冲突元素）然后遍历以该元素为头的单向链表元素，依据每个被遍历元素的 hash 值计算其在新数组中的下标然后进行交换（**即原来 hash 冲突的单向链表尾部变成了扩容后单向链表的头部）。**

在 JDK 1.8 中 HashMap 的扩容操作就显得更加的骚气了，由于扩容数组的长度是 2 倍关系，所以对于假设初始 tableSize = 4 要扩容到 8 来说就是 0100 到 1000 的变化（左移一位就是 2 倍），**在扩容中只用判断原来的 hash 值与左移动的一位（newtable 的值）按位与操作是 0 或 1 就行，0 的话索引就不变，1 的话索引变成原索引加上扩容前数组**

18.Hashmap JDK1.8 扩容原理源代码分析

```
final Node<K, V>[] resize () {
    Node<K, V>[] oldTab = table;
    //记住扩容前的数组长度和最大容量
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        //超过数组在java中最大容量就无能为力了，冲突就只能冲突
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        //长度和最大容量都扩容为原来的二倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1;
        double threshold
    }
    //..... .....
    //更新新的最大容量为扩容计算后的最大容量
    threshold = newThr;
    //更新扩容后的新数组长度
    Node<K, V>[] newTab = (Node<K, V>[]) new Node[newCap];
    table = newTab;
```

```
        //更新扩容后的新数组长度
        Node<K,V>[] newTab = (Node<K,V>[]) new Node[newCap];
        table = newTab;
        if (oldTab != null) {
            //遍历老数组下标索引
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                //如果老数组对应索引上有元素则取出链表头元素放在e中
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
                    //如果老数组j下标处只有一个元素则直接计算新数组中位置放置
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    else if (e instanceof TreeNode)
                        //如果是树结构进行单独处理
                        ((TreeNode<K,V>) e).split(this, newTab, j, oldCap);
                    else {
                        preserve order
                        //能进来说明数组索引j位置上存在哈希冲突的链表结构
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
                        //循环处理数组索引j位置上哈希冲突的链表中每个元素
                        do {
```

```
Node<K,V> next;
//循环处理数组索引j位置上哈希冲突的链表中每个元素
do {
    next = e.next;
    //判断key的hash值与老数组长度与操作后结果决定元素是放在原索引处还是新索引
    if ((e.hash & oldCap) == 0) {
        //放在原索引处的建立新链表
        if (loTail == null) loHead = e;
        else loTail.next = e;
        loTail = e;
    } else {
        //放在新索引（原索引 + oldCap）处的建立新链表
        if (hiTail == null) hiHead = e;
        else hiTail.next = e;
        hiTail = e;
    }
}
while ((e = next) != null);
if (loTail != null) {
    //放入原索引处 loTail.next = null;
    newTab[j] = loHead;
}
if (hiTail != null) {
    //放入新索引处
    hiTail.next = null;
    newTab[j + oldCap] = hiHead;
}
```

JDK1.7 中扩容操作时，哈希冲突的数组索引处的旧链表元素扩容到新数组时，如果扩容后索引位置在新数组的索引位置与原数组中索引位置相同，则链表元素会发生倒置（即如上面图1，原来链表头扩容后变为尾巴）；而在 JDK1.8 中不会出现链表倒置现象。
其次，由于 JDK1.7 中发生哈希冲突时仅仅采用了链表结构存储冲突元素，所以扩容时仅仅是重新计算其存储位置而已，而 JDK1.8 中为了性能在同一索引处发生哈希冲突到一定程度时链表结构会转换为红黑数结构存储冲突元素，故在扩容时如果当前索引中元素结构是红黑树且元素个数小于链表还原阈值（哈希冲突程度常量）时就会把树形结构缩小或直接还原为链表结构（其实现就是上面代码片段中的 split() 方法）

19.Hashmap JDK1.8红黑树扩容情况的split方法

```
/**
 * 扩容后，红黑树的hash分布，只可能存在于两个位置：原索引位置、原索引位置+oldCap
 */
final void split(HashMap<K,V> map, Node<K,V>[] tab, int index, int bit) {
    TreeNode<K,V> b = this;      // 拿到调用此方法的节点
    TreeNode<K,V> loHead = null, loTail = null; // 存储索引位置为:“原索引位置”的节点
    TreeNode<K,V> hiHead = null, hiTail = null; // 存储索引位置为:“原索引+oldCap”的节点
    int lc = 0, hc = 0;
    // 1.以调用此方法的节点开始，遍历整个红黑树节点
    for (TreeNode<K,V> e = b, next; e != null; e = next) {      // 从b节点开始遍历
        next = (TreeNode<K,V>)e.next;    // next赋值为e的下个节点
        e.next = null;  // 同时将老表的节点设置为空，以便垃圾收集器回收
        // 2.如果e的hash值与老表的容量进行与运算为0,则扩容后的索引位置跟老表的索引位置一样
        if ((e.hash & bit) == 0) {
            if ((e.prev = loTail) == null)  // 如果loTail为空，代表该节点为第一个节点
                loHead = e; // 则将loHead赋值为第一个节点
            else
                loTail.next = e;     // 否则将节点添加在loTail后面
            loTail = e; // 并将loTail赋值为新增的节点
            ++lc;   // 统计原索引位置的节点个数
        }
        // 3.如果e的hash值与老表的容量进行与运算为1,则扩容后的索引位置为:老表的索引位置+oldCap
        else {
            if ((e.prev = hiTail) == null)  // 如果hiHead为空，代表该节点为第一个节点
                hiHead = e; // 则将hiHead赋值为第一个节点
            else
                hiTail.next = e;     // 否则将节点添加在hiTail后面
            hiTail = e; // 并将hiTail赋值为新增的节点
            ++hc;   // 统计索引位置为原索引+oldCap的节点个数
        }
```

```
    // 4.如果原索引位置的节点不为空
    if (loHead != null) {   // 原索引位置的节点不为空
        // 4.1 如果节点个数<=6个则将红黑树转为链表结构
        if (lc <= UNTREEIFY_THRESHOLD)
            tab[index] = loHead.untreeify(map);
        else {
            // 4.2 将原索引位置的节点设置为对应的头节点
            tab[index] = loHead;
            // 4.3 如果hiHead不为空，则代表原来的红黑树(老表的红黑树由于节点被分到两个位置)
            // 已经被改变，需要重新构建新的红黑树
            if (hiHead != null)
                // 4.4 以loHead为根节点，构建新的红黑树
                loHead.treeify(tab);
        }
    }
```

```
// 5.如果索引位置为原索引+oldCap的节点不为空
if (hiHead != null) {    // 索引位置为原索引+oldCap的节点不为空
    // 5.1 如果节点个数<=6个则将红黑树转为链表结构
    if (hc <= UNTREEIFY_THRESHOLD)
        tab[index + bit] = hiHead.untreeify(map);
    else {
        // 5.2 将索引位置为原索引+oldCap的节点设置为对应的头节点
        tab[index + bit] = hiHead;
        // 5.3 loHead不为空则代表原来的红黑树(老表的红黑树由于节点被分到两个位置)
        // 已经被改变，需要重新构建新的红黑树
        if (loHead != null)
            // 5.4 以hiHead为根节点，构建新的红黑树
            hiHead.treeify(tab);
    }
}
```

20.String类适合做hashmap的key的原因是什么

在《Java 编程思想》中有这么一句话：设计 hashCode() 时最重要的因素就是对同一个对象调用 hashCode() 都应该产生相同的值。

String 类型的对象对这个条件有着很好的支持，因为 String 对象的 hashCode() 值是根据 String 对象的内容计算的，并不是根据对象的地址计算。下面是 String 类源码中的 hashCode() 方法：String 对象底层是一个 final 修饰的 char 类型的数组，hashCode() 的计算是根据字符数组的每个元素进行计算的，所以内容相同的 String 对象会产生相同的散列码。

```
public int hashCode() {
        int h = hash;       //private int hash; // Default to 0
        if (h == 0 && value.length > 0) {
            char val[] = value;     //获得 String 对象底层的字符数组

            for (int i = 0; i < value.length; i++) {
                h = 31 * h + val[i];     //在计算的时候加的是 int 类型的 ascii 码
            }
            hash = h;
        }
        return h;
    }
```

第三个原因，看下一道题，
equals方法 string自己就有。

HashMap 内部实现是通过 key 的 hashcode 来确定 value 的存储位置

第一个原因：天生复写了hashCode方法，根据String对象的内容来计算的hashCode。

因为字符串是不可变的，所以当创建字符串时，它的 hashcode 被缓存下来，不需要再次计算，所以相比于其他对象更快（第二个原因）。

21. 如果我想用自定义的对象做hashmap的key，需要进行什么操作

重写hashCode()和equals()方法 看如下代码解析：

当向HashMap中存入k1的时候，首先会调用Key这个类的hashcode方法，计算它的hash值，随后把k1放入hash值所指引的内存位置，在Key这个类中没有定义hashcode方法，就会调用Object类的hashcode方法，而Object类的hashcode方法返回的hash值是对象的地址。这时用k2去拿也会计算k2的hash值到相应的位置去拿，由于k1和k2的内存地址是不一样的，所以用k2拿不到k1的值重写hashcode方法仅仅能够k1和k2计算得到的hash值相同，调用get方法的时候会到正确的位置去找，但当出现散列冲突时，在同一个位置有可能用链表的形式存放冲突元素，这时候就需要用到equals方法去对比了，由于没有重写equals方法，它会调用Object类的equals方法，Object的equals方法判断的是两个对象的内存地址是不是一样，由于k1和k2都是new出来的，k1和k2的内存地址不相同，所以这时候用k2还是达不到k1的值

```
class Key {
    private Integer id;

    public Integer getId() {
        return id;
    }

    public Key(Integer id) {
        this.id = id;
    }
}

public class WithoutHashCode {
    public static void main(String[] args) {
        Key k1 = new Key(1);
        Key k2 = new Key(1);

        HashMap<Key, String> hashMap = new HashMap<>();
        hashMap.put(k1, "Key with id is 1");
        System.out.println(hashMap.get(k2));
    }
}
```

运行结果：
null

22. 什么是哈希，什么是哈希冲突

Hash，一般翻译为“散列”，也有直接音译为“哈希”的，这就是把任意长度的输入通过散列算法，变换成固定长度的输出，该输出就是散列值（哈希值）；这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来唯一的确定输入值。**简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。**

所有散列函数都有如下一个基本特性：**根据同一散列函数计算出的散列值如果不同，那么输入值肯定也不同。但是，根据同一散列函数计算出的散列值如果相同（哈希冲突情况），输入值不一定相同。**

23. 解决哈希冲突的方式

开放定址法就是解决hash冲突的一种方式。它是使用一种 探测方式在整个数组中找到另一个可以存储值的地方。

链地址法（拉链法）HashMap，HashSet其实都是采用的拉链法来解决哈希冲突的，就是在每个位桶实现的时候，我们采用链表（jdk1.8之后采用链表+红黑树）的数据结构来去存取发生哈希冲突的输入域的关键字

再散列法再散列法其实很简单，就是再使用哈希函数去散列一个输入的时候，输出是同一个位置就再次散列，直至不发生冲突位置
缺点：每次冲突都要重新散列，计算时间增加。

24. 开放寻址法的探索方式

（1）线性探测

按顺序决定哈希值时，如果某数据的哈希值已经存在，则在原来哈希值的基础上往后加一个单位，直至不发生哈希冲突。

（2）再平方探测

按顺序决定哈希值时，如果某数据的哈希值已经存在，则在原来哈希值的基础上先加1的平方个单位，若仍然存在则减1的平方个单位。随之是2的平方，3的平方等等。直至不发生哈希冲突。

（3）伪随机探测

按顺序决定哈希值时，如果某数据已经存在，通过随机函数随机生成一个数，在原来哈希值的基础上加上随机数，直至不发生哈希冲突。

25. 开放寻址法和拉链法的优缺点

1.开放定址法：容易产生堆积问题；不适于大规模的数据存储；**散列函数的设计对冲突会有很大的影响**；插入时**可能会出现多次冲突的现象**，删除的元素是多个冲突元素中的一个，需要对后面的元素作处理，实现较复杂；**结点规模很大时会浪费很多空间（**再平方探测**）**；

2.链地址法：处理冲突简单，且无堆积现象，平均查找长度短；链表中的结点是动态申请的，适合构造表不能确定长度的情况；相对而言，拉链法的指针域可以忽略不计，因此较开放地址法更加节省空间。插入结点应该在链尾部（jdk1.8），删除结点比较方便，只需调整指针而不需要对其他冲突元素作调整。

26. Hashmap的查询效率

1. 最理想O(1)（没有冲突）
2. JDK1.7 最坏情况O（N）
3. JDK1.8 最坏是O(logN)

27. Hashmap和Hashtable的区别

1、继承的父类不同

Hashtable继承自Dictionary类，而HashMap继承自AbstractMap类。但二者都实现了Map接口

2、线程安全性不同

javadoc中关于hashmap的一段描述如下：此实现不是同步的。如果多个线程同时访问一个哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。

Hashtable 中的方法是Synchronize的，而HashMap中的方法在缺省情况下是非Synchronize的。在多线程并发的环境下，可以直接使用Hashtable，不需要自己为它的方法实现同步，但使用HashMap时就必须要自己增加同步处理。（结构上的修改是指添加或删除一个或多个映射关系的任何操作；仅改变与实例已经包含的键关联的值不是结构上的修改。）这一般通过对自然封装该映射的对象进行同步操作来完成。如果不存在这样的对象，则应该使用 Collections.synchronizedMap方法来“包装”该映射。最好在创建时完成这一操作，以防止对映射进行意外的非同步访问

3、是否提供contains方法

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。

Hashtable则保留了contains，containsValue和containsKey三个方法，其中contains和containsValue功能相同。

4、key和value是否允许null值

其中key和value都是对象，并且不能包含重复key，但可以包含重复的value。

Hashtable中，key和value都不允许出现null值。但是如果在Hashtable中有类似put(null,null)的操作，编译同样可以通过，因为key和value都是Object类型，但运行时会抛出NullPointerException异常，这是JDK的规范规定的。

HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。当get()方法返回null值时，可能是 HashMap中没有该键，也可能使该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键， 而应该用containsKey()方法来判断。

5、两个遍历方式的内部实现上不同

Hashtable、HashMap都使用了 Iterator。而由于历史原因，Hashtable还使用了Enumeration的方式 。

6、hash值不同

哈希值的使用不同，HashTable直接使用对象的hashCode。而HashMap重新计算hash值。

7、内部实现使用的数组初始化和扩容方式不同

HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。 Hashtable和HashMap它们两个内部实现方式的数组的初始大小和扩容的方式。HashTable中hash数组默认大小是11，增加的方式是 old*2+1。

28. 为什么hashtable扩容方式选为2N+1

首先，Hashtable的初始容量为11。Index的计算方式为：

```
int index = (hash & 0x7FFFFFFF) % tab.length;
```

常用的hash函数是选一个数m取模（余数），这个数在课本中推荐m是素数，但是经常见到选择m=2^n，因为对2^n求余数更快，并认为在key分布均匀的情况下，key%m也是在[0,m-1]区间均匀分布的。但实际上，key%m的分布同m是有关的。

证明如下：key%m = key - x*m，即key减掉m的某个倍数x，剩下比m小的部分就是key除以m的余数。显然，x等于key/m的整数部分，以floor(key/m)表示。假设key和m有公约数g，即key=ag, m=bg, 则key - xm = key - floor(key/m)*m = key - floor(a/b)*m。由于0 <= a/b <= a，所以floor(a/b)只有a+1中取值可能，从而推导出key%m也只有a+1中取值可能。a+1个球放在m个盒子里面，显然不可能做到均匀。

由此可知，一组均匀分布的key，其中同m公约数为1的那部分，余数后在[0,m-1]上还是均匀分布的，但同m公约数不为1的那部分，余数在[0, m-1]上就不是均匀分布的了。**把m选为素数，正是为了让所有key同m的公约数都为1，从而保证余数的均匀分布，降低冲突率。**
鉴于此，在HashTable中，初始化容量是11，是个素数，后面扩容时也是按照2N+1的方式进行扩容，确保扩容之后仍是素数。

29. 简述你所知道的jdk1.7与jdk1.8的hashmap的改动

不同	JDK 1.7	JDK 1.8
存储结构	数组 + 链表	数组 + 链表 + 红黑树
初始化方式	单独函数：`inflateTable()`	直接集成到了扩容函数`resize()`中
hash值计算方式	扰动处理 = 9次扰动 = 4次位运算 + 5次异或运算	扰动处理 = 2次扰动 = 1次位运算 + 1次异或运算
存放数据的规则	无冲突时，存放数组；冲突时，存放链表	无冲突时，存放数组；冲突 & 链表长度 < 8：存放单链表；冲突 & 链表长度 > 8：树化并存放红黑树
插入数据方式	头插法（先讲原位置的数据移到后1位，再插入数据到该位置）	尾插法（直接插入到链表尾部/红黑树）
扩容后存储位置的计算方式	全部按照原来方法进行计算（即hashCode ->> 扰动函数 ->> (h&length-1)）	按照扩容后的规律计算（即扩容后的位置= 原位置 or 原位置 + 旧容量）

30.负载因子为什么会影响HashMap性能

首先回忆HashMap的数据结构，
我们都知道有序数组存储数据，对数据的索引效率都很高，但是插入和删除就会有性能瓶颈，链表存储数据，要一次比较元素来检索出数据，所以索引效率低，但是插入和删除效率高，两者取长补短就产生了哈希散列这种存储方式，也就是HashMap的存储逻辑.

而负载因子表示一个散列表的空间的使用程度，有这样一个公式:
initailCapacity*loadFactor=HashMap的容量。

所以负载因子越大则散列表的装填程度越高，也就是能容纳更多的元素，元素多了，链表大了，所以此时索引效率就会降低。

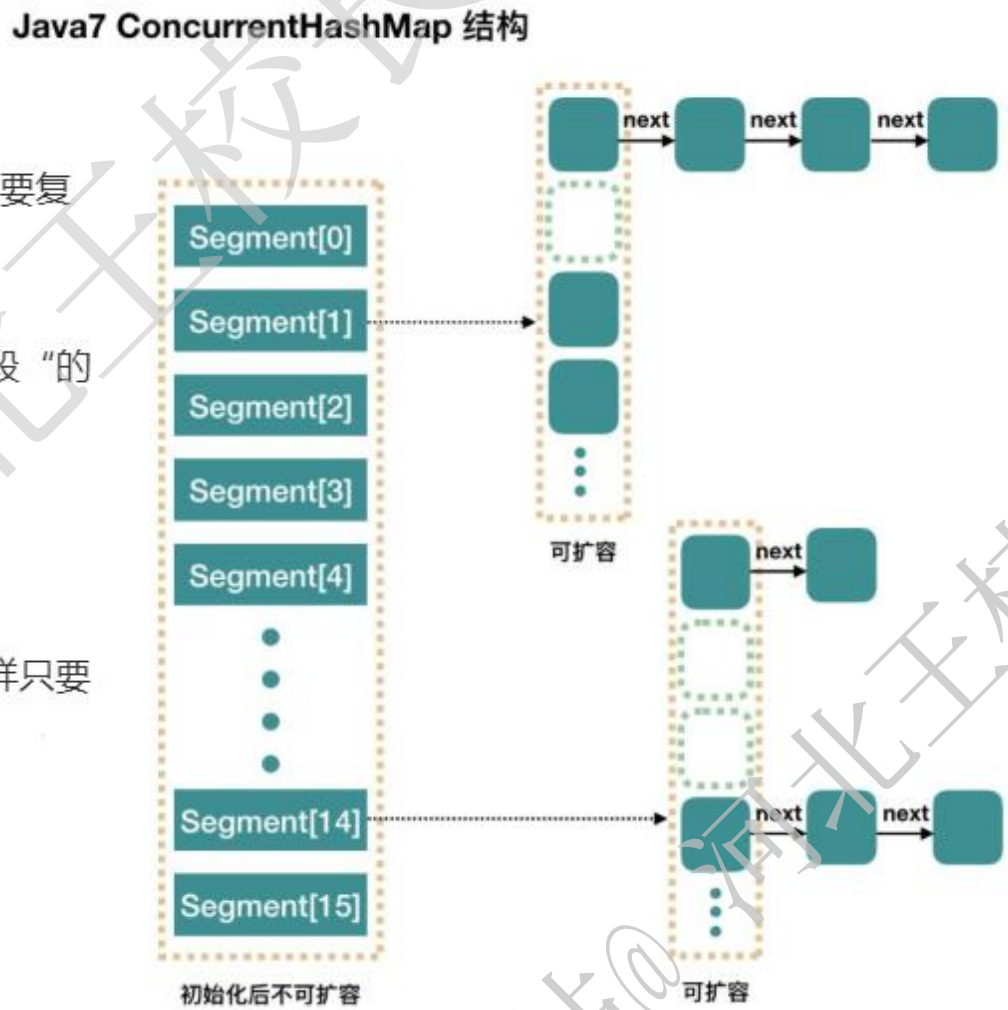
反之，负载因子越小则链表中的数据量就越稀疏，此时会对空间造成烂费，但是此时索引效率高。

31. 介绍下JDK1.7 的ConcurrentHashmap的数据结构

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一些。

整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的意思，所以很多地方都会将其描述为**分段锁**。

简单理解就是，ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。



32. ConcurrentHashmap的构造方法有几个

```java
//无参构造函数
public ConcurrentHashMap() {
}
//可传初始容器大小的构造函数
public ConcurrentHashMap(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap;
}
//可传入map的构造函数
public ConcurrentHashMap(Map<? extends K, ? extends V> m) {
    this.sizeCtl = DEFAULT_CAPACITY;
    putAll(m);
}
//可设置阈值和初始容量
public ConcurrentHashMap(int initialCapacity, float loadFactor) {
    this(initialCapacity, loadFactor, 1);
}
```

```
//可设置初始容量和阈值和并发级别
public ConcurrentHashMap(int initialCapacity,
                         float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    if (initialCapacity < concurrencyLevel)   // Use at least as many bins
        initialCapacity = concurrencyLevel;   // as estimated threads
    long size = (long)(1.0 + (long)initialCapacity / loadFactor);
    int cap = (size >= (long)MAXIMUM_CAPACITY) ?
        MAXIMUM_CAPACITY : tableSizeFor((int)size);
    this.sizeCtl = cap;
}
```

concurrencyLevel： 并行级别、并发数、Segment 数，怎么翻译不重要，理解它。默认是 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。

再具体到每个 Segment 内部，其实每个 Segment 很像之前介绍的 HashMap，不过它要保证线程安全，所以处理起来要麻烦些。

33. 简述下jdk1.7的currenthashmap是如何进行锁操作的

JDK1.7版本的ReentrantLock+Segment+HashEntry。写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作

小伙伴们，ReentrantLock 这部分会在多线程进行面试题分享

```
HashEntry :用来存储元素
static final class HashEntry<K,V> {
        final int hash;
        final K key;
        volatile V value;
        volatile HashEntry<K,V> next;

        HashEntry(int hash, K key, V value, HashEntry<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }

Segment桶  实现线程的关键类（部分属性为列出，太长了。。）
static final class Segment<K,V> extends ReentrantLock implements Serializable {

        //可以看出Segment继承自ReentrantLock，是一个天然的锁


       transient volatile HashEntry<K,V>[] table;

        ...
        transient int count;
```

34. JDK1.8的currenthashmap是如何保证并发的

DK8中ConcurrentHashMap参考了JDK8 HashMap的实现，采用了数组+链表+红黑树的实现方式来设计，内部大量采用CAS操作，这里我简要介绍下CAS。
CAS是compare and swap的缩写，即我们所说的比较交换。cas是一种基于锁的操作，而且是乐观锁。在java中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加version来获取数据，性能较悲观锁有很大的提高。
JDK8中彻底放弃了Segment转而采用的是Node，其设计思想也不再是JDK1.7中的分段锁思想。
Node：保存key，value及key的hash值的数据结构。其中value和next都用volatile修饰，保证并发的可见性。
Java8 ConcurrentHashMap结构基本上和Java8的HashMap一样，不过保证线程安全性。

小伙伴们Volatile，CAS等在多线程篇

存储节点:

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        volatile V val;
        volatile Node<K,V> next;
```

put()方法:

可以看出jdk1.8中取消了桶的设计，使用了Node + CAS + Synchronized来保证并发安全进行实现

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
        if (key == null || value == null) throw new NullPointerException();
        int hash = spread(key.hashCode());
        int binCount = 0;
        for (Node<K,V>[] tab = table;;) {
            Node<K,V> f; int n, i, fh;
            if (tab == null || (n = tab.length) == 0)
                tab = initTable();                    //初始化
            else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {    //如果该位置无元素则
                if (casTabAt(tab, i, null,
                             new Node<K,V>(hash, key, value, null)))
                    break;                   // no lock when adding to empty bin
            }
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);
            else {
                V oldVal = null;
                synchronized (f) {          //如果有元素或cas插入失败 则使用synchronized 锁
                    if (tabAt(tab, i) == f) {
                        if (fh >= 0) {
                            binCount = 1;
                            for (Node<K,V> e = f;; ++binCount) {
                                K ek;
```

35. JDK1.8的currenthashmap和jdk1.7的区别（或者说1.8的改进）

其实可以看出JDK1.8版本的ConcurrentHashMap的数据结构已经接近HashMap，相对而言，ConcurrentHashMap只是增加了同步的操作来控制并发，从JDK1.7版本的ReentrantLock+Segment+HashEntry，到JDK1.8版本中synchronized+CAS+HashEntry+红黑树。

1.数据结构：取消了Segment分段锁的数据结构，取而代之的是数组+链表+红黑树的结构。
2.保证线程安全机制：JDK1.7采用segment的分段锁机制实现线程安全，其中segment继承自ReentrantLock。JDK1.8采用CAS+Synchronized保证线程安全。

3.锁的粒度：原来是对需要进行数据操作的Segment加锁，现调整为对每个数组元素加锁（Node）。

4.链表转化为红黑树:定位结点的hash算法简化会带来弊端,Hash冲突加剧,因此在链表节点数量大于8时，会将链表转化为红黑树进行存储。

5.查询时间复杂度：从原来的遍历链表O(n)，变成遍历红黑树O(logN)。

36. Hashtable多线程下与currentHashMap表现有什么区别吗，为什么

Hashtable
线程安全
concurrentHashMap
线程安全的，**在多线程下效率更高。**

注：hashtable:使用一把锁处理并发问题，当有多个线程访问时，需要多个线程竞争一把锁，导致阻塞。

1.7 concurrentHashMap则使用分段，相当于把一个hashmap分成多个，然后每个部分分配一把锁，这样就可以支持多线程访问。（默认情况下，理论上讲，能同时支持 16条线程并发）

1,8 锁粒度细到了元素本身。理论上讲，是最高级别的并发。

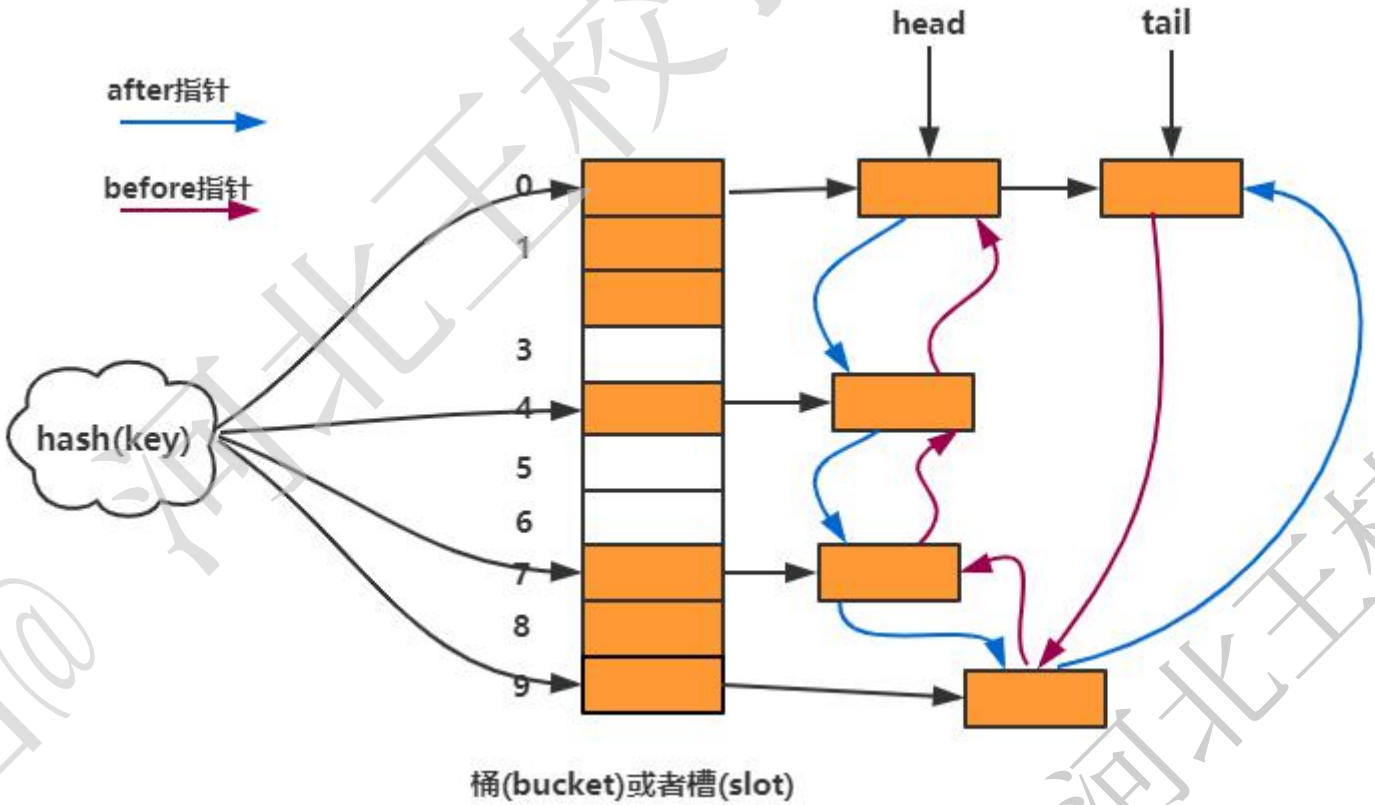
37. LinkedHashMap与hashMap的区别

- LinkedHashMap是继承于HashMap，是基于HashMap和双向链表来实现的。
- HashMap无序；LinkedHashMap有序，可分为插入顺序和访问顺序两种。如果是访问顺序，那put和get操作已存在的Entry时，都会把Entry移动到双向链表的表尾(其实是先删除再插入)。
- LinkedHashMap存取数据，还是跟HashMap一样使用的Entry[]的方式，双向链表只是为了保证顺序。
- LinkedHashMap是线程不安全的。

38. 简要介绍LinkedHashMap 怎样维护双向链表

在LinkedHashMap中，是通过双联表的结构来维护节点的顺序的。实际上在内存中的情况如下图所示，每个节点都进行了双向的连接，维持插入的顺序（默认）。head指向第一个插入的节点，tail指向最后一个节点。

总得来说，**LinkedHashMap底层是数组+单向链表+双向链表**。数组加单向链表就是HashMap的结构，记录数据用；双向链表，存储插入顺序用。



head指向第一个插入的节点

tail指向最后一个节点。

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>
{
  static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }

    // 双向链表头节点
    transient LinkedHashMap.Entry<K,V> head;

    // 双向链表尾节点
    transient LinkedHashMap.Entry<K,V> tail;

    // 指定遍历LinkedHashMap的顺序,true表示按照访问顺序,false表示按照插入顺序，默认为false
    final boolean accessOrder;
  }

}
```

不要搞错了next和before、After，next是用于维护HashMap指定table位置上连接的Entry的顺序的，before、After是用于维护Entry插入的先后顺序的。

39. 使用map结构时，如果要求按put的顺序来进行哈希表的遍历，选择什么map

我们知道HashMap中不存在保存顺序的机制。LinkedHashMap专为此特性而生。在LinkedHashMap中可以保持两种顺序，分别是插入顺序和访问顺序，这个是可以在LinkedHashMap的初始化方法中进行指定的。相对于访问顺序，按照插入顺序进行编排被使用到的场景更多一些，所以默认是按照插入顺序进行编排。

```
public static void main(String[] args)
    {
        Map<String, String> test = new LinkedHashMap<String, String>(9);

        test.put("化学","93");
        test.put("数学","98");
        test.put("生物","92");
        test.put("英语","97");
        test.put("物理","94");
        test.put("历史","96");
        test.put("语文","99");
        test.put("地理","95");

        for (Map.Entry entry : test.entrySet())
        {
            System.out.println(entry.getKey().toString() + ":" + entry.getValue().toSt
        }
}
```

运行结果如下图所示，可以看到，输出的顺序与插入的顺序是一致的。

40. 介绍LinkedHashMap维护的两种顺序，插入顺序和访问顺序

插入顺序：顾名思义，插入的时候能够通过双向链表的关联维护插入的顺序（详见41题的代码举例）。新创建的linkedhashmap存放的元素，在进行遍历输出打印时，按照put的顺序进行的打印，这也是与hashmap不同的地方，因为hashmap中没有维护插入的元素与元素之间的顺序的逻辑。

访问顺序：先不解释，先看图，后边看源码的时候就知道了。

使用默认的构造方法

```
public class DemoTest {
    public static void main(String[] args) {
        Map<String, String> linkedHashMap = new LinkedHashMap<>();
        linkedHashMap.put("First", "1");
        linkedHashMap.put("Second", "2");
        linkedHashMap.put("Third", "3");
        System.out.println("开始时顺序: ");
        for (Map.Entry entry : linkedHashMap.entrySet())
        {
            System.out.println(entry.getKey().toString() + ":" + entry.getValue().toString());
        }
        linkedHashMap.get("First");
        System.out.println("Get后的顺序: ");
        for (Map.Entry entry : linkedHashMap.entrySet())
        {
            System.out.println(entry.getKey().toString() + ":" + entry.getValue().toString());
        }
    }
}
```

打印结果没变

```
开始时顺序:
First:1
Second:2
Third:3
Get后的顺序:
First:1
Second:2
Third:3
```

```java
public class DemoTest {
    public static void main(String[] args) {
        Map<String, String> linkedHashMap = new LinkedHashMap<>( initialCapacity: 16,  loadFactor: 0.75f,  accessOrder: true);
        linkedHashMap.put("First", "1");
        linkedHashMap.put("Second", "2");
        linkedHashMap.put("Third", "3");
        System.out.println("开始时顺序: ");
        for (Map.Entry entry : linkedHashMap.entrySet())
        {
            System.out.println(entry.getKey().toString() + ":" + entry.getValue().toString());
        }
        linkedHashMap.get("First");
        System.out.println("Get后的顺序: ");
        for (Map.Entry entry : linkedHashMap.entrySet())
        {
            System.out.println(entry.getKey().toString() + ":" + entry.getValue().toString());
        }
    }
}
```

使用非默认的构造方法，且accessOrder的值为true

```
开始时顺序:
First:1
Second:2
Third:3
Get后的顺序:
Second:2
Third:3
First:1
```

结果发生改变。

我们发现之前的 123 变成了231， First这个值最后打印了，原因是:

1. accessOrder为true代表开启访问顺序
2. 开启访问顺序后，只要调用get方法，就会把被调用过get方法的元素放到链表的最末端

这就是传说中的访问顺序。

41. LinkedHashMap控制访问顺序的原理，看过源代码吗？在哪里控制

斩钉截铁的说，看过源码。源码要点位置有两个。 第一个，就是上一题所说的构造函数源码：

```
public LinkedHashMap(int initialCapacity,
                     float loadFactor,
                     boolean accessOrder) {
    super(initialCapacity, loadFactor);
    this.accessOrder = accessOrder;
}
```

非常简单质朴的源码，关键点在accessOrder，只要设置成true，就会启动访问权限。

第二个，就是get方法到底干了什么，我们从上题看到，启动了访问顺序后，只要调用get方法，被get的元素就会被放置到最后端，所以这块儿也是大大的要点：

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

非常纯正的if判断，只要accessOrder==true，就会执行afterNodeAccess方法，这个方法，从英文翻译的角度来看，就是 放到node的后边。

思考题：afterNodeAccess我们好像在哪儿见过你记得吗？

42. LinkedHashMap的put方法原理

如果你去看LinkedHashMap.java这个文件的源代码，你会惊奇的发现，这个java类中，没有发现put方法。要了命了。。。那LinkedHashMap的put方法走的是哪儿的方法逻辑呢？

我们知道， LinkedHashMap 的父类是HashMap，如果LinkedHashMap.java没有 put方法，那说明， LinkedHashMap的put方法是直接使用Hashmap的put方法。 发问： 难道LinkedHashMap和Hashmap的put逻辑完全一样吗？

肯定不一样，必须有不同的地方。这道题的关键就在此处了。到底哪里不一样。有两个要点，我们进行源码分析：

1. newNode方法的复写

如果进行put的方法调用，肯定是要进行新节点的添加的，linkedHashMap在 newNode方法中进行了改善，进行了复写，源码如下：

```
Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {
    LinkedHashMap.Entry<K,V> p =
        new LinkedHashMap.Entry<K,V>(hash, key, value, e);
    linkNodeLast(p);
    return p;
}
```

创建一个普通entry，将entry插入到双向链表的末尾，最后返回entry

2. AfterNodeAccess方法的复写，将新增的节点添加到链表的尾部

```
void afterNodeAccess(Node<K,V> e) { // move node to last
    LinkedHashMap.Entry<K,V> last;
    if (accessOrder && (last = tail) != e) {
        LinkedHashMap.Entry<K,V> p =
            (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
        p.after = null;
        if (b == null)
            head = a;
        else
            b.after = a;
        if (a != null)
            a.before = b;
        else
            last = b;
        if (last == null)
            head = p;
        else {
            p.before = last;
            last.after = p;
        }
        tail = p;
        ++modCount;
    }
}
```

思考题：afterNodeAccess我们见过两次了

43. LinkedHashMap的get方法原理

JDK1.8 的HashMap的get方法

（1）计算数据在桶中的位置（tab.length- 1) & hash(key)

（2）通过hash值和key值判断待查找的数据是否在对应桶的首节点，如果在，则返回对应节点数据；否则判断桶首节点的类型。如果节点为红黑树，从红黑树中获取对应数据；如果节点为链表节点，则遍历链表，从中获取对应数据

44. 用LinkedHashMap实现一个简易的缓存清理(LRU算法)

考察点两个：

1. accessOrder 控制的 访问顺序；
2. removeEldestEntry 方法的复写使用

```
void afterNodeInsertion(boolean evict) { // possibly remove eldest
    LinkedHashMap.Entry<K,V> first;
    if (evict && (first = head) != null && removeEldestEntry(first)) {
        K key = first.key;
        removeNode(hash(key), key,  value: null,  matchValue: false,  movable: true);
    }
}
```

源码中， 如果不进行复写操作， removeEldestEntry 默认返回false， if判断中永远不会被执行，所以我们只要复写removeEldestEntry 方法，就会在我们期望的场景下执行if判断中的removeNode方法。

```
import java.util.*;
//扩展一下LinkedHashMap这个类，让他实现LRU算法
class LRULinkedHashMap<K,V> extends LinkedHashMap<K,V>{
    //定义缓存的容量
    private int capacity;
    private static final long serialVersionUID = 1L;
    //带参数的构造器
    LRULinkedHashMap(int capacity){
        //调用LinkedHashMap的构造器，传入以下参数
        super(16,0.75f,true);
        //传入指定的缓存最大容量
        this.capacity=capacity;
    }
    //实现LRU的关键方法，如果map里面的元素个数大于了缓存最大容量，则删除链表的顶端元素
    @Override
    public boolean removeEldestEntry(Map.Entry<K, V> eldest){
        System.out.println(eldest.getKey() + "=" + eldest.getValue());
        return size()>capacity;
    }
}
//测试类
class Test{
public static void main(String[] args) throws Exception{

    //指定缓存最大容量为4
    Map<Integer,Integer> map=new LRULinkedHashMap<>(4);
    map.put(9,3);
    map.put(7,4);
    map.put(5,9);
    map.put(3,4);
    map.put(6,6);
    //总共put了5个元素，超过了指定的缓存最大容量
    //遍历结果
        for(Iterator<Map.Entry<Integer,Integer>> it=map.entrySet().iterator();it.hasNext();){
            System.out.println(it.next().getKey());
        }
    }
}
```

45. hashSet的底层代码是依赖什么数据结构，请介绍下

HashSet 的实现依赖于 HashMap

通过 HashSet 的构造参数我们可以看出每个构造方法，都调用了对应的 HashMap 的构造方法用来初始化成员变量 map ，因此我们可以知道，HashSet 的初始容量也为 1<<4 即16，加载因子默认也是 0.75f。

```
// HashSet 真实的存储元素结构
private transient HashMap<E,Object> map;

// 作为各个存储在 HashMap 元素的键值对中的 Value
private static final Object PRESENT = new Object();

//空参数构造方法 调用 HashMap 的空构造参数
//初始化了 HashMap 中的加载因子 loadFactor = 0.75f
public HashSet() {
        map = new HashMap<>();
}

//指定期望容量的构造方法
public HashSet(int initialCapacity) {
    map = new HashMap<>(initialCapacity);
}
//指定期望容量和加载因子
public HashSet(int initialCapacity, float loadFactor) {
    map = new HashMap<>(initialCapacity, loadFactor);
}
//使用指定的集合填充Set
public HashSet(Collection<? extends E> c) {
        //调用   new HashMap<>(initialCapacity) 其中初始期望容量为 16 和 c 容量 / 默认 load factor 
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
}

// 该方法为 default 访问权限，不允许使用者直接调用，目的是为了初始化 LinkedHashSet 时使用
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
```

46. hashSet是如何判断元素的重复的

•HashSet 内部使用 HashMap 存储元素，对应的键值对的键为 Set 的存储元素，值为一个默认的 Object 对象。

•HashSet 通过存储元素的 hashCode 方法和 equals 方法来确定元素是否重复。

```
/**
 * Adds the specified element to this set if it is not already present.
 * More formally, adds the specified element <tt>e</tt> to this set if
 * this set contains no element <tt>e2</tt> such that
 * <tt>(e==null ? e2==null : e.equals(e2))</tt>.
 * If this set already contains the element, the call leaves the set
 * unchanged and returns <tt>false</tt>.
 *
 * @param e element to be added to this set
 * @return <tt>true</tt> if this set did not already contain the specified
 * element
```

```
public boolean add(E e) { return map.put(e, PRESENT)==null; }
```

我们可以看到，hashset的add方法，其实是直接调用了hashmap的put方法，那么我们看到 第一个参数 e代表了存入的元素，第二个元素PRESENT其实是一个定义且初始化好的 new Object。这样hashset就能在存入一个值的情况下，很好的利用hashmap的Key-value结构。

47. LinkedHashSet 源码解析总结

之前介绍了,HashSet其实就是利用了HashMap进行了元素的存取，但是并非继承关系，因为一方是Set，一方是Map，完全两码事儿。 那么对于LinkedHashSet是否也是通过LinkedHashMap做的双向链表控制呢？先来看LinkedHashSet源代码。

```
public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {

    private static final long serialVersionUID = -2851667679971038690L;

    /**...*/
    public LinkedHashSet(int initialCapacity, float loadFactor) { super(initialCapacity, loadFactor,  dummy: true); }

    /**...*/
    public LinkedHashSet(int initialCapacity) { super(initialCapacity,  loadFactor: .75f,  dummy: true); }

    /**...*/
    public LinkedHashSet() { super( initialCapacity: 16,  loadFactor: .75f,  dummy: true); }

    /**...*/
    public LinkedHashSet( @NotNull  @Flow(sourceIsContainer=true,targetIsContainer=true) Collection<? extends E> c) {
        super(Math.max(2*c.size(), 11),  loadFactor: .75f,  dummy: true);
        addAll(c);
    }

    /**...*/
    @Override
    public Spliterator<E> spliterator() {
        return Spliterators.spliterator( c: this,  characteristics: Spliterator.DISTINCT | Spliterator.ORDERED);
    }
}
```

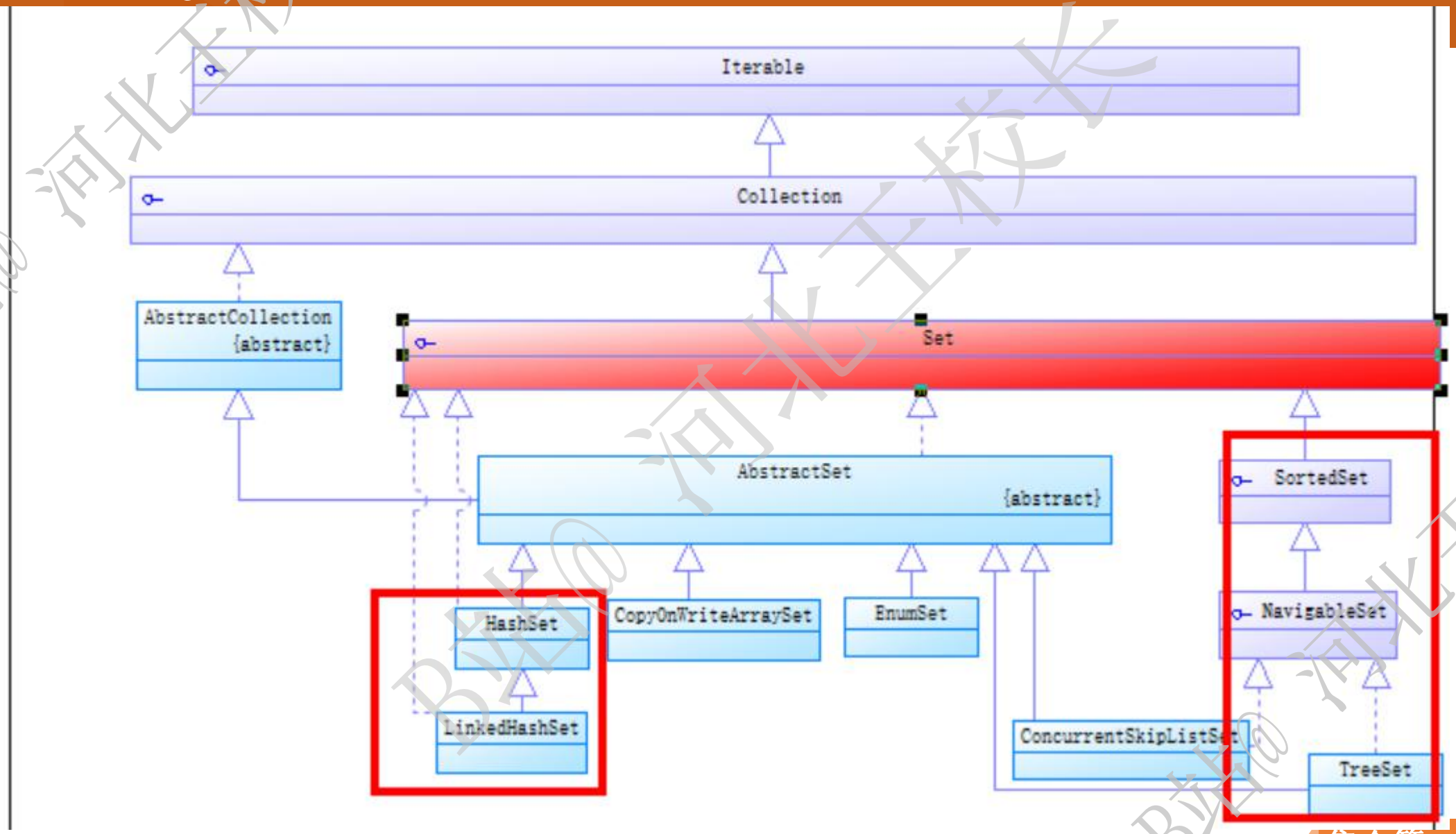
看清楚，一共五个方法，四个构造函数。没天理了，到底哪里控制了双向链表呢？

看构造函数中的代码。。。

你会发现，这个super方法，其实是调用了HashSet的一个default的构造函数（default方法，只有同包才能使用）。我们来仔细看看这个构造方法，首先他是唯一的一个在HashSet中涉及到LinkedHashMap的构造，其次他是default修饰的。总结来说，这个HashSet中的Default的构造函数，就是为他的儿子LinkedHashSet准备的。

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    static final long serialVersionUID = -5024744406713321676L;

    private transient HashMap<E,Object> map;

    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

    /**...*/
    public HashSet() { map = new HashMap<>(); }

    /**...*/
    public HashSet( @NotNull  @Flow(sourceIsContainer=true,targetIsContainer=true) Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
    }

    /**...*/
    public HashSet(int initialCapacity, float loadFactor) { map = new HashMap<>(initialCapacity, loadFactor); }

    /**...*/
    public HashSet(int initialCapacity) { map = new HashMap<>(initialCapacity); }

    /**...*/
    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }

    /**
     * Returns an iterator over the elements in this set.  The elements
     * are returned in no particular order.
     *
     * @return an Iterator over the elements in this set
     * @see ConcurrentModificationException
     */
```

48. jdk1.8 Set的类继承结构



49. TreeSet与HashMap的关系

```
public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable {
    private transient NavigableMap<E, Object> m;
    private static final Object PRESENT = new Object();
    private static final long serialVersionUID = -2479143000061671589L;

    TreeSet(NavigableMap<E, Object> var1) { this.m = var1; }

    public TreeSet() { this((NavigableMap)(new TreeMap())); }

    public TreeSet(Comparator<? super E> var1) { this((NavigableMap)(new TreeMap(var1))); }
```

与HashSet和LinkedHashSet一个套路，TreeSet其实是使用了TreeMap的结构

50. Set集合的特点是什么。见过什么常见的set集合

- Set不允许包含相同的元素，如果试图把两个相同元素加入同一个集合中，add方法返回false。

  Set判断两个对象相同不是使用==运算符，而是根据equals方法。也就是说，只要两个对象用equals方法比较返回true，Set就不会接受这两个对象。

- HashSet与TreeSet都是基于Set接口的实现类。其中TreeSet是Set的子接口SortedSet的实现类。Set接口及其子接口、实现类的结构如下所示：

```
                |——SortedSet接口——TreeSet实现类
Set接口— |——HashSet实现类
                |——LinkedHashSet实现类
```

51. 说说HashSet的特点

HashSet有以下特点

- **不能保证元素的排列顺序，顺序有可能发生变化**
- **不是同步的**
- **集合元素可以是null,但只能放入一个null**

当向HashSet结合中存入一个元素时，HashSet会调用该对象的hashCode()方法来得到该对象的hashCode值，然后根据 对象hashCode值（与hashMap一样，length-1的值 & hashCode的值来确定数组索引位置）来决定该对象在HashSet中存储位置。

说白了，是否记得有一道题，hashMap为什么喜欢使用String作为key？ 还有一道题，如果一个对象作为hashMap的key，我们需要怎么办?
为什么提到这两个题，因为对于hashSet来讲，就是使用对象作为key，new Object()作为value的，
往前看几道题，之前都讲解过。 所以说，hashSet存储对象，对象的判重，就是根据的hashCode和equeals， 这个跟 HashMap的判重一个意思。

52. 说说LinkedHashSet的特点

LinkedHashSet集合同样是根据元素的hashCode(归根结底，还是hashMap)值来决定元素的存储位置，但是它同时使用链表维护元素的次序（次序出自LinkedHashMap）。这样使得元素看起 来像是以插入顺 序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。

53. 讨论下不同Map的性能问题，大量数据下，你觉得哪个表现好，为什么

前提：IdentityHashMap不在讨论范围内。数据100W，机器指标i3处理区，4G内存。单线程环境。

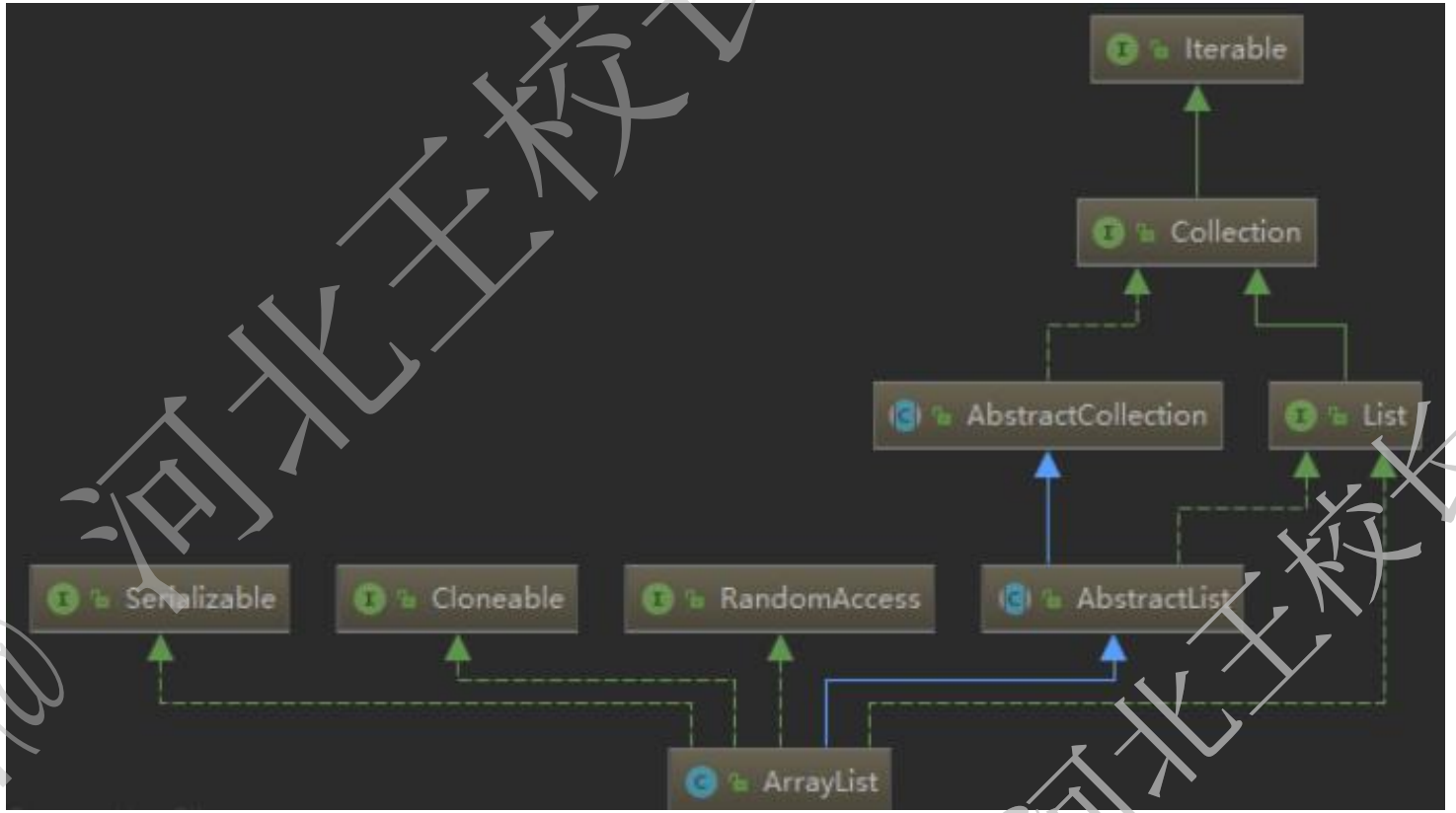
1.看着hashTable和HashMap性能差不多，因为是单线程环境，其实table和map最显著的区别就是同步问题，没有同步问题，两者的性能区别不是很大，就像我们经常所说的， hashTable会慢一点，因为是同步的.
2. LinkedHashSet,不多说，维护双向顺序链表，肯定会累一点，慢一点。其中插入操作慢的更加显著。
3. TreeMap，红黑树结构的有序map。 如果对存入的数据顺序没有要求的话，treeMap的性能存取性能慢的比较显著，因为红黑树需要进行平衡的旋转变色。况且，多数情况下，hashMap的存取时间复杂度都是O（1），红黑树是O(logn)，所以treemap慢一点，正常。

Java中各个Map的性能测试（台式机，i3/4G）数据表

Map	Run Time(ms)	Type[order]
HashMap	422, 423, 438	Put[2]
	234, 218, 218	Get[2]
	250, 234, 234	Foreach[1]
Hashtable	407, 422, 406	Put[1]
	249, 250, 250	Get[4]
	250, 250, 249	Foreach[1]
LinkedHashMap	610, 594, 594	Put[4]
	234, 234, 234	Get[3]
	265, 280, 280	Foreach[3]
IdentityHashMap	532, 516, 531	Put[3]
	172, 172, 172	Get[1]
	327, 328, 312	Foreach[4]
TreeMap	1437, 1421, 1431	Put[5]
	1185, 1185, 1186	Get[5]
	327, 328, 343	Foreach[4]

54. 说说jdk1.8的Arraylist的特点

1. ArrayList是一个动态数组,实现了List<E>, RandomAccess, Cloneable, java.io.Serializable, 并允许包括null在内的所有元素。实现了RandomAccess接口标识着其支持随机快速访问,实际上,我们查看RandomAccess源码可以看到,其实里面什么都没有定义.因为ArrayList底层是数组,那么随机快速访问是理所当然的,访问速度O(1).
实现了Cloneable接口,标识着可以它可以被复制.注意,ArrayList里面的clone()复制其实是浅复制。
实现了Serializable， 支持序列化传输



2. 底层使用数组实现,默认初始容量为10. 当超出后， 会自动扩容为原来的1.5倍， 即自动扩容机制。 数组的扩容是新建一个大容量（原始数组大小+扩充容量）的数组， 然后将原始数组数据拷贝到新数组， 然后将新数组作为扩容之后的数组。数组扩容的操作代价很高， 我们应该尽量减少这种操作。

3. 该集合是可变长度数组，数组扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量增长大约是其容量的1.5倍，如果扩容一半不够,就将目标size作为扩容后的容量.这种操作的代价很高。采用的是 Arrays.copyOf浅复制

4. 采用了Fail-Fast机制，面对并发的修改时，迭代器很快就会完全失败，报异常concurrentModificationException(并发修改一次),而不是冒着在将来某个不确定时间发生任意不确定行为的风险。

5. remove方法会让下标到数组末尾的元素向前移动一个单位，并把最后一位的值置空，方便GC

6. 数组扩容代价是很高的，因此在实际使用时，我们应该尽量避免数组容量的扩张。当我们可预知要保存的元素的多少时，要在构造ArrayList实例时，就指定其容量，以避免数组扩容的发生。或者根据实际需求，通过调用ensureCapacity方法来手动增加ArrayList实例的容量。

7. ArrayList不是线程安全的，只能用在单线程环境下，多线程环境下可以考虑用Collections.synchronizedList(List l)函数返回一个线程安全的ArrayList类，也可以使用concurrent并发包下的CopyOnWriteArrayList类。

8 ,如果是删除数组指定位置的元素,那么可能会挪动大量的数组元素;如果是删除末尾元素的话,那么代价是最小的.

55. ArrayList有几个构造方法，简单说下。

三个构造方法

ArrayList()：构造一个初始容量为10的空列表

```
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

ArrayList(Collection<?extend E> c)：构造一个包含指定元素的列表

```
public ArrayList( @NotNull  @Flow(sourceIsContainer=true,targetIsContainer=true) Collection<? extends E> c) {
    elementData = c.toArray();
    if ((size = elementData.length) != 0) {
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    } else {
        // replace with empty array.
        this.elementData = EMPTY_ELEMENTDATA;
    }
}
```

ArrayList(int initialCapcity)：构造一个具有初始容量值得空列表

```
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}
```

我们看到代码逻辑不复杂，从代码逻辑中，依稀可以看到，会有new Object[] 的操作，从这里就能印证，ArrayList就是以数组为底层的。

更直接的，我们通过看源码可以发现：

图中有两个fianl的静态变量引用：

private static final **Object[]** ***EMPTY_ELEMENTDATA*** = {};

private static final **Object[]** ***DEFAULTCAPACITY_EMPTY_ELEMENTDATA*** = {};

Object[]这是什么？数组。

56. 为什么ArrayList中，有两个看似的静态的final的Object数组 *EMPTY_ELEMENTDATA & DEFAULTCAPACITY_EMPTY_ELEMENTDATA*？

首先先得知道，面试关问的啥，问的哪部分的源码：
private static final Object[] ***EMPTY_ELEMENTDATA*** = {};
private static final Object[] ***DEFAULTCAPACITY_EMPTY_ELEMENTDATA*** = {};

这两个值，有着一样的修饰符和初始化值，从上一题的构造代码中可以看到。只有默认构造函数使用了***DEFAULTCAPACITY_EMPTY_ELEMENTDATA***； 其他两个构造函数，在不满足if条件的情况下，使用的是***EMPTY_ELEMENTDATA***。

从构造函数上，看不出来区别，那么为什么要写出两个一样的静态final全局变量呢？
EMPTY_ELEMENTDATA：非常单纯的提供空的数组。
DEFAULTCAPACITY_EMPTY_ELEMENTDATA：用于默认大小的空实例的共享空数组实例。我们将其与空的元素数据区分开来，控制在添加第一个元素的时候，如何对数组容量进行初始化。

上边对***DEFAULTCAPACITY_EMPTY_ELEMENTDATA***的解释有点抽象，在继续深入分析前，先得搞明白一道基础面试题如下

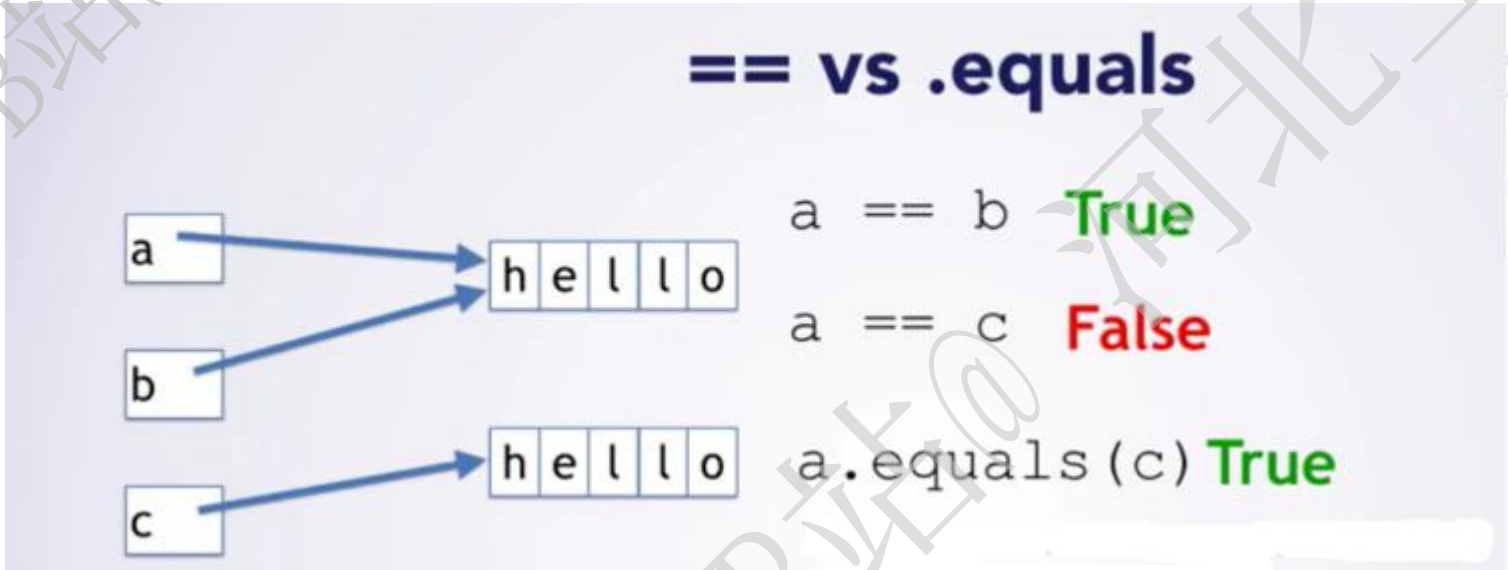
```
public class DemoTest {
    private static final Object[] a = {};
    private static final Object[] b = {};
    private static final String c ="";
    private static final String d="";
    public static void main(String[] args) {
        System.out.println(a==b);
        System.out.println(c==d);
    }
}
```

```java
public class DemoTest {
    private static final Object[] a = {};
    private static final Object[] b = {};
    private static final String c = "";
    private static final String d="";
    public static void main(String[] args) {
        System.out.println(a==b);
        System.out.println(c==d);
    }
}
```

这是啥题： == 和 equesl的区别？ ==比较的是啥？

- ==是判断两个变量或实例是不是指向同一个内存空间，equals是判断两个变量或实例所指向的内存空间的值是不是相同
- ==是指对内存地址进行比较， equals()是对字符串的内容进行比较
- ==指引用是否相同， equals()指的是值是否相同

看图说话：



回到正题， ***DEFAULTCAPACITY_EMPTY_ELEMENTDATA*** 其实是在空list首次调用add方法的时候判断如何初始化数组的容量。

```
public boolean add(E e) {
    ensureCapacityInternal( minCapacity: size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}

/**

private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}

private static int calculateCapacity(Object[] elementData, int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity;
    }
```

注意：用的是＝

从这里就可以看到，如果是通过
DEFAULTCAPACITY_EMPTY_ELEMENTDATA
定义的空list， 首次添加元素的时候，返回默认容量
DEFAULT_CAPACITY=10***；***
如果不是走的***DEFAULTCAPACITY_EMPTY_ELEMENTDATA，***那么第一次会返回1.

```java
public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException, IllegalAccessException {
    ArrayList list = new ArrayList();
    Class<?> clazz = Class.forName("java.util.ArrayList");
    Field ed = clazz.getDeclaredField( name: "elementData");
    ed.setAccessible(true);
    Object[] array = (Object[])ed.get(list);
    System.out.println(array.length);
}
```

默认构造，未添加元素，容量为0

```java
public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException, IllegalAccessException {
    ArrayList list = new ArrayList();
    list.add(1);
    Class<?> clazz = Class.forName("java.util.ArrayList");
    Field ed = clazz.getDeclaredField( name: "elementData");
    ed.setAccessible(true);
    Object[] array = (Object[])ed.get(list);
    System.out.println(array.length);
}
```

默认构造，添加一个元素，容量为10

```java
public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException, IllegalAccessException {
    ArrayList list = new ArrayList( initialCapacity: 0);
    list.add(1);
    Class<?> clazz = Class.forName("java.util.ArrayList");
    Field ed = clazz.getDeclaredField( name: "elementData");
    ed.setAccessible(true);
    Object[] array = (Object[])ed.get(list);
    System.out.println(array.length);
}
```

初始化容量为0，添加第一个元素后，容量为1

57. ArrayList中的 elementData 用 transient 修饰，序列化后数据会丢失吗

说的是源码中的全局变量： transient Object[] **elementData**；在构造函数中，我们就看到过了使用

搞懂这道题之前，先得搞明白其他两个面试题：

1．序列化是什么?

2．Transient关键词是什么意思

1．序列化是什么?

我们知道对象是不能直接进行网络传输的，必须要转化为二进制字节流进行传输。序列化就是将对象转化为字节流的过程。同理，反序列化就是从字节流构建对象的过程。

- 对于 Java 对象来说，如果使用 JDK 的序列化实现。对象只需要实现 java.io.Serializable 接口。
- 可以使用 ObjectOutputStream() 和 ObjectInputStream() 对对象进行手动序列化和反序列化。序列化的时候会调用 writeObject() 方法，把对象转换为字节流。反序列化的时候会调用 readObject() 方法，把字节流转换为对象。
- Java 在反序列化的时候会校验字节流中的 serialVersionUID 与对象的 serialVersionUID 时候一致。如果不一致就会抛出 InvalidClassException 异常。
- 官方强烈推荐为序列化的对象指定一个固定的 serialVersionUID。否则虚拟机会根据类的相关信息通过一个摘要算法生成，所以当我们改变类的参数的时候虚拟机生成的 serialVersionUID 是会变化的。transient 关键字修饰的变量 不会被序列化为字节流

2. Transient关键词是什么意思

被transient` 修饰的变量不会被序列化。这句话说的很笼统，但是理解这道题，够用了，序列化问题，等我们讨论完ArrayList这些东西之后，再说…

进入正题：从源代码中可以看出elementData其实就是代表的ArrayList底层的数组，如果不能被序列化，是不是就丢数据了？

```
transient Object[] elementData; // non-private to simplify nested class access

/**
 * The size of the ArrayList (the number of elements it contains).
 *
 * @serial
 */
private int size;

/**
 * Constructs an empty list with the specified initial capacity.
 *
 * @param  initialCapacity  the initial capacity of the list
 * @throws IllegalArgumentException if the specified initial capacity
 *         is negative
 */
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}
```

不能被序列化，那是不是序列化后，这部分数据就丢了？

构造函数，就是用的 elementData初始化的

我们在进行序列化对象的时候，做的最多的操作就是 implements io的序列化接口java.io.Serializable。 ArrayList同样也实现了这个接口，那说明ArrayList是支持序列化的。 ArrayList，所有的用户数据，都保存在 transient修饰的elementData中，如果序列化后数据全丢了，那ArrayList就是个废品。

那ArrayList用什么巧妙的方式，即防止了elementData的序列化，又保证存入的元素不能丢失？
答案很简答: 不对elementData序列化，对elementData中的元素进行循环，取出来单独进行序列化。

代码在哪里？我们说过，序列化的时候会调用 writeObject() 方法，把对象转换为字节流。反序列化的时候会调用 readObject() 方法，把字节流转换为对象。那我们就直接去看ArrayList中的这两个方法即可。

```java
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    // Write out element count, and any hidden stuff
    int expectedModCount = modCount;
    s.defaultWriteObject();

    // Write out size as capacity for behavioural compatibility with clone()
    s.writeInt(size);

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++) {
        s.writeObject(elementData[i]);
    }

    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}
```

这是问题的关键所在，不序列化elementData这个对象，只序列化其中的元素。

```java
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    elementData = EMPTY_ELEMENTDATA;

    // Read in size, and any hidden stuff
    s.defaultReadObject();

    // Read in capacity
    s.readInt(); // ignored

    if (size > 0) {
        // be like clone(), allocate array based upon size not capacity
        int capacity = calculateCapacity(elementData, size);
        SharedSecrets.getJavaOISAccess().checkArray(s, Object[].class, capacity);
        ensureCapacityInternal(size);

        Object[] a = elementData;
        // Read in all elements in the proper order.
        for (int i=0; i<size; i++) {
            a[i] = s.readObject();
        }
    }
}
```

一个一个进行反序列化，然后存储到初始化完成的新的elementdata中

58. 既然ArrayList会序列化elementData中存储的每一个元素，那为什么不直接序列化elementData呢？ 这样设计有什么好处吗

上一道题，我们说了elementData是一个Object[]对象，不直接序列化这个对象，是因为这个对象绝大多数情况下会有没有存储任何元素的容量空间。这样将会是一个很大的空间浪费。

例子来两个，这也是之前的内容：

1. 浪费九个容量
刚刚添加一个元素，
容量就变成了10.

```
public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException, IllegalAccessException {
    ArrayList list = new ArrayList();
    list.add(1);
    Class<?> clazz = Class.forName("java.util.ArrayList");
    Field ed = clazz.getDeclaredField( name: "elementData");
    ed.setAccessible(true);
    Object[] array = (Object[])ed.get(list);
    System.out.println(array.length);
}
```

默认构造，添加一个元素，容量为10

2. 每次数组容量增长大约是其容量的1.5倍。
试想一下， 10万条数据，扩容编程15万个容量空间，但是只有10万零1条数据，是不是浪费了很多.

59. 说说你对transient的理解，举个使用场景的例子

1）一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。

2）transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。

3）被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。

4）使用场景举例，在实际开发过程中，我们常常会遇到这样的问题，这个类的有些属性需要序列化，而其他属性不需要被序列化，打个比方，如果一个用户有一些敏感信息（如密码，银行卡号等），为了安全起见，不希望在网络操作（主要涉及到序列化操作，本地序列化缓存也适用）中被传输，这些信息对应的变量就可以加上transient关键字。换句话说，这个字段的生命周期仅存于调用者的内存中而不会写到磁盘里持久化。

60. ArrayList的Fail-Fast机制是什么原理。

采用了Fail-Fast机制，面对并发的修改时，迭代器很快就会完全失败，报异常concurrentModificationException(并发修改一次),而不是冒着在将来某个不确定时间发生任意不确定行为的风险。 **----来自题目54中的一段话**

ArrayList的父类 AbstractList中有一个类属性： protected transient int **modCount** = 0; 这个属性代表了这个list被结构性修改的次数。
结构性修改是指：改变list的size大小，或者，以其他方式改变他导致正在进行迭代时出现错误的结果。
这个字段用于迭代器和列表迭代器的实现类中，由迭代器和列表迭代器方法返回。如果这个值被意外改变，这个迭代器将会抛出 ConcurrentModificationException的异常来响应：
next,remove,previous,set,add 这些操作。
在迭代过程中，他提供了fail-fast行为而不是不确定行为来处理并发修改。子类使用这个字段是可选的，如果子类希望提供fail-fast迭代器，它仅仅需要在add(int, E),remove(int)方法（或者它重写的其他任何会结构性修改这个列表的方法）中添加这个字段。调用一次add(int,E)或者remove(int)方法时必须且仅仅给这个字段加1，否则迭代器会抛出伪装的ConcurrentModificationExceptions错误。如果一个实现类不希望提供fail-fast迭代器，则可以忽略这个字段。

```java
private class Itr implements Iterator<E> {
        int cursor;       // index of next element to return
        int lastRet = -1; // index of last element returned; -1 if no such
        int expectedModCount = modCount;

        public boolean hasNext() {
            return cursor != size;
        }

        @SuppressWarnings("unchecked")
        public E next() {
            checkForComodification();
            int i = cursor;
            if (i >= size)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;
            return (E) elementData[lastRet = i];
        }

        public void remove() {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();

            try {
                ArrayList.this.remove(lastRet);
                cursor = lastRet;
                lastRet = -1;
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }

        final void checkForComodification() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
    }
```

1、expectedModCount的初值为modCount

2、hasNext的判断条件为cursor!=size，就是当前迭代的位置不是数组的最大容量值就返回true

3、next和remove操作之前都会先调用checkForComodification来检查expectedModCount和modCount是否相等

如果没checkForComodification去检查expectedModCount与modCount相等，这个程序肯定会报越界异常ArrayIndexOutOfBoundsException

因为有modCount的存在，在使用多线程对非线程安全的集合进行操作时，使用迭代器循环会产生modCount != expectedModCount的情况，会抛出异常。

61. ArrayList add（E e）方法源码原理

add主要的执行逻辑如下：
1）确保数组已使用长度（size）加1之后足够存下 下一个数据
2）修改次数modCount 标识自增1，如果当前数组已使用长度（size）加1后的大于当前的数组长度，则调用grow方法，增长数组，grow方法会将当前数组的长度变为原来容量的1.5倍。
3）确保新增的数据有地方存储之后，则将新元素添加到位于size的位置上。
4）返回添加成功布尔值。

方法入口：

```
public boolean add(E e) {
    ensureCapacityInternal( minCapacity: size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

可以看到 主要是ensureCapacityInternal方法的使用需要深入研究

```
private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}
```

继续看，应该要先看 *calculateCapacity 方法*

```
private static int calculateCapacity(Object[] elementData, int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity;
    }
```

最后一步，添加完元素，modCount要自增1， 计算的minCapacity大于 elementData的长度，则进行扩用。

```
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

注意： 扩容是一件非常耗时且损耗空间的事情，如果能在使用arrayList之前能够确定List的容量大小，那就不会调用grow方法进行扩容。

61. ArrayList add(int index, E element) 源码有了解过吗，说说你对这个方法的利弊看法

```
public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal( minCapacity: size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData,  destPos: index + 1,
                     length: size - index);
    elementData[index] = element;
    size++;
}
```

唯一有必要深入看看的就是 rangeCheckForAdd方法（简单地角标越界判断）：

```
private void rangeCheckForAdd(int index) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

该方法可以按照元素的位置，指定位置插入元素，具体的执行逻辑如下：

1）确保数插入的位置小于等于当前数组长度，并且不小于0，否则抛出异常

2）确保数组已使用长度（size）加1之后足够存下下一个数据

3）修改次数（modCount）标识自增1，如果当前数组已使用长度（size）加1后的大于当前的数组长度，则调用grow方法，增长数组

4）grow方法会将当前数组的长度变为原来容量的1.5倍。

5）确保有足够的容量之后，使用System.arraycopy将需要插入的位置（index）后面的元素统统往后移动一位。

6）将新的数据内容存放到数组的指定位置（index）上

好处：

这个方法的好处，完全体现在第一个参数 index ， 它能够让使用者按照自己的意愿，在不超过数组的角标上界时插入元素到指定位置。

坏处：

这个方法的坏处，完全体现在源码中的System.arrayCopy方法。ArrayList底层是数组，进行插入操作的时候，如果插入的索引位置处之后还有其他元素，必须进行后续元素的角标调整（角标调整的方式，就是将元素后移。） 这样会造成频繁的元素移动，速率会打一定的折扣。

62. ArrayList的扩容原理

private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8 = 2^31-1-8 ;
Integer.MAX_VALUE = 0x7fffffff = 2^31-1;

```
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);
}
```

老的长度=当前的elementData的长度

newCapacity = 老的长度+ 老的长度>>1 .右移1即除以2

若扩容1.5倍依然不满足，那newCapacity=minCapacity

若newCapacity比MAX_ARRAY_SIZE还大，用hugeCapacity

老的copy到新的数组

看看hugeCapacity方法代码：

```
private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
        throw new OutOfMemoryError();
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
}
```

我们知道 *MAX_ARRAY_SIZE=* 2^31-1-8；
Integer.MAX_VALUE = 2^31-1;
从这里来看，如果*MAX_ARRAY_SIZE* 达不到要求，则赋值为*Integer.MAX_VALUE*

所以理论上来说： arrayList的最高容量 是= 2^31-1

63. 为什么ArrayList的MAX_ARRAY_SIZE 是Integer.MAX_VALUE 减8，而不是减别的

数组在java里是一种特殊类型，既不是基本数据类型也不是引用数据类型。有别于普通的“类的实例”对象，java里数组不是类，所以也就没有对应的class文件，数组类型是由jvm从元素类型合成出来的；

在jvm中获取数组的长度是用arraylength这个专门的字节码指令的；在数组的对象头里有一个_length字段，记录数组长度，只需要去读_length字段就可以了。

所以ArrayList中定义的最大长度为Integer最大值减8，这个8就是就是存了数组_length字段。

64. ArrayList的remove方法有了解过吗，如果长度为1的Arraylist，移除这个元素后，方法是怎么考虑后续垃圾回收的。

```
public E remove(int index) {
    rangeCheck(index);

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, srcPos: index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;
}
```

我们看到，移除元素的时候也会改变modCount，并且是++操作

判断是否是移除的最后一个元素，>0 表示不是

如果移除的不是最后一个元素，进行arrayCopy进行元素向前移动

将最后一个元素设置为null，为GC做准备。
这个操作非常细节，可以看到设计者的用心。

65. ArrayList的contains方法的时间复杂是多少

```
public boolean contains(Object o) { return indexOf(o) >= 0; }
```

```
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

一次循环，O(n)的时间复杂度

66. ArrayList如果在循环中删除一个元素，有什么办法避开fail-fast机制吗

```java
public class DemoTest {
    public static void main(String[] args) {
        ArrayList list = new ArrayList( initialCapacity: 0);
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.add(6);
        list.add(7);
        Iterator iterator = list.iterator();
        while (iterator.hasNext()) {
            int tmp = (int) iterator.next();
            if (tmp==2) {
                iterator.remove();
            }

        }
        System.out.println(list);

    }
}
```

使用迭代器?

这个可行，因为迭代器使用的自己实现的remove方法

```java
public class DemoTest {
    public static void main(String[] args) {
        ArrayList list = new ArrayList( initialCapacity: 0);
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.add(6);
        list.add(7);
        for (int i = 0; i < list.size(); i++) {
            int tmp =2 ;
            if (tmp ==  (int)list.get(i)) {
                list.remove(i);
            }
        }
    }
}
```

使用For循环?

这个也可行。

使用Foreach呢？？？

为什么使用迭代器可以解决问题？我们直接看看迭代器的remove方法是如何实现的。

```
private class Itr implements Iterator<E> {
    int cursor;       // index of next element to return
    int lastRet = -1; // index of last element returned; -1 if no such
    int expectedModCount = modCount;

    Itr() {}

    public boolean hasNext() { return cursor != size; }

    /unchecked/
    public E next() {
        checkForComodification();
        int i = cursor;
        if (i >= size)
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i + 1;
        return (E) elementData[lastRet = i];
    }

    public void remove() {
        if (lastRet < 0)
            throw new IllegalStateException();
        checkForComodification();

        try {
            ArrayList.this.remove(lastRet);
            cursor = lastRet;
            lastRet = -1;
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
```

cursor为游标，指向下一个元素的索引，默认初始化为0
lastRet 也为游标，指向已被迭代过的元素，默认初始化为-1.

expectedModCount，赋值为modCount，删除元素后重新赋值

每调用一次next方法，cursor=i+1，指向下一个元素

lastRet 指向刚刚被迭代过的元素，lastRet=i
我们可以看到，多数情况下，lastRet与cursor的角标是连续的，只差1

lastRet<0. 代表lastRet没有被i赋值，说明是初始值-1. 说明没有被迭代过，没有被迭代过就删除，这是不允许的。也就是说，iterator是靠lastRet的值来判断是否可以进行remove操作的。

如果lastRet > 0,说明已经被迭代过，可以删除，这时候cursor的角标需要减去1，cursor-1 = lastRet，所以对cursor 进行lastRet的赋值操作

lastRet的位置被成功的remove了，自己的位置被cursor替代了。把自己置成初始值-1，等待下次的赋值删除操作。

67. ArrayList和LinkedList的区别

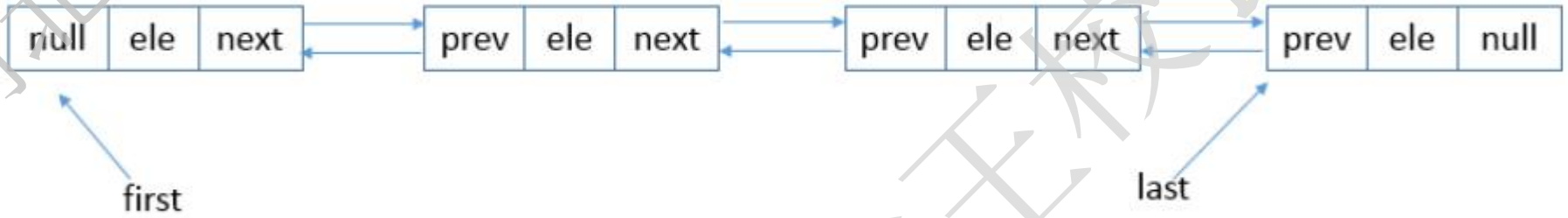
1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于双向链表的数据结构。

2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。

3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

4. Arraylist的额外空间占用是1.5倍扩容导致的空间资源预留，LinkedList是需要对前后指针进行保存，单个元素比arraylist占用更大的空间。

68. 描述下LinkedList的数据结构



如上图所示，LinkedList底层使用的**双向链表结构**，有一个头结点和一个尾结点，双向链表意味着我们可以从头开始正向遍历，或者是从尾开始逆向遍历，并且可以针对头部和尾部进行相应的操作。

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    transient int size = 0;

    /**...*/
    transient Node<E> first;

    /**...*/
    transient Node<E> last;

    /**
     * Constructs an empty list.
     */
    public LinkedList() {
    }
```

69. ArrayList和Vector的区别

1、同步性：
Vector 是线程安全的，也就是说是它的方法之间是线程同步的，而 ArrayList 是线程是不安全的，它的方法之间是线程不同步的。如果只有一个线程会访问到集合，那最好是使用 ArrayList，因为它不考虑线程安全，效率会高些；如果有多个线程会访问到集合，那最好是使用 Vector，因为不需要我们自己再去考虑和编写线程安全的代码。
备注：对于 Vector&ArrayList、Hashtable&HashMap，要记住线程安全的问题，记住 Vector 与 Hashtable 是旧的，是 java 一诞生就提供了的，它们是线程安全的，ArrayList 与 HashMap 是 java2 时才提供的，它们是线程不安全的。

2、数据增长：
ArrayList 与 Vector 都有一个初始的容量大小，当存储进它们里面的元素的个数超过了容量时，就需要增加 ArrayList 与 Vector 的存储空间，每次要增加存储空间时，不是只增加一个存储单元，而是增加多个存储单元，每次增加的存储单元的个数在内存空间利用与程序效率之间要取得一定的平衡。Vector 默认增长为原来两倍，而 ArrayList 的增长为原来的 1.5 倍。ArrayList 与 Vector 都可以设置初始的空间大小，Vector 还可以设置增长的空间大小，而 ArrayList 没有提供设置增长空间的方法。

70. 说说List，Map和Set三者的区别

List，主要是为顺序存储诞生的，List接口是为了存储一组不唯一的（允许重复）有序的对象。

Set，主要特性是不允许重复的集合。对象存储不可重复性，且无序。

Map，主要特征是Key-value。Map会维护与Key对应的值。