Android HashMap 底层原理

deepwaterooo

August 30, 2022

Contents

1 HashMap 的原理与实现

1.1 版本之更迭:

- –》JDK 1.7 ： Table 数组 + Entry 链表;
- –》JDK1.8 : Table 数组 + Entry 链表/红黑树; (为什么要使用红黑树?)

- 一问 HashMap 的实现原理
  - 你看过 HashMap 源码吗，知道底层的原理吗
  - 为什么使用数组 + 链表
  - 用 LinkedList 代替数组可以吗
  - 既然是可以的，为什么不用反而用数组。
- 重要变量介绍：
  - ps：都是重要的变量记忆理解一下最好。
  - DEFAULT_INITIAL_CAPACITY Table 数组的初始化长度：1 « 42^4=16（为什么要是 2 的 n 次方?）
  - MAXIMUM_CAPACITY Table 数组的最大长度：1«302^30=1073741824
  - DEFAULT_LOAD_FACTOR 负载因子：默认值为 0.75。当元素的总个数 > 当前数组的长度 * 负载因子。数组会进行扩容，扩容为原来的两倍（todo：为什么是两倍?）
  - TREEIFY_THRESHOLD 链表树化阙值：默认值为 8 。表示在一个 node（Table）节点下的值的个数大于 8 时候，会将链表转换成为红黑树。
  - UNTREEIFY_THRESHOLD 红黑树链化阙值：默认值为 6 。表示在进行扩容期间，单个 Node 节点下的红黑树节点的个数小于 6 时候，会将红黑树转化成为链表。
  - MIN_TREEIFY_CAPACITY = 64 最小树化阈值，当 Table 所有元素超过改值，才会进行树化（为了防止前期阶段频繁扩容和树化过程冲突）。

1.2 实现原理：

- 我们都知道，在 HashMap 中，采用数组 + 链表的方式来实现对数据的储存。
- HashMap 采用 Entry 数组来存储 key-value 对，每一个键值对组成了一个 Entry 实体，Entry 类实际上是一个单向的链表结构，它具有 Next 指针，可以连接下一个 Entry 实体。只是在 JDK1.8 中，链表🄴度大于 8 的时候，链表会转成红黑树！

1.2.1 第一问：为什么使用链表 + 数组：要知道为什么使用链表首先需要知道 Hash 冲突是如何来的：

- 答：由于我们的数组的值是限制死的，我们在对 key 值进行散列取到下标以后，放入到数组中时，难免出现两个 key 值不同，但是却放入到下标相同的格子中，此时我们就可以使用链表来对其进行链式的存放。

1.2.2 第二问我用 LinkedList 代替数组结构可以吗？

- 对于题目的意思是说，在源码中我们是这样的

```
Entry[] table=new Entry[capacity];
// entry 就是一个链表的节点
```

- 现在进行替换，进行如下的实现

```
List<Entry> table=new LinkedList<Entry>();
```

- 是否可以行得通？答案当然是肯定的。

1.2.3 第三问那既然可以使用进行替换处理，为什么有偏偏使用到数组呢？

- 因为用数组效率最高！在 HashMap 中，定位节点的位置是利用元素的 key 的哈希值对数组 F度取模得到。此时，我们已得到节点的位置。显然数组的查找效率比 LinkedList 大（底层是链表结构）。
- 那 ArrayList，底层也是数组，查找也快啊，为啥不用 ArrayList? 因为采用基本数组结构，扩容机制可以自己定义，HashMap 中数组扩容刚好是 2 的次幂，在做取模运算的效率高。而 ArrayList 的扩容机制是 1.5 倍扩容（这一点我相信学习过的都应该清楚），那 ArrayList 为什么是 1.5 倍扩容这就不在本文说明了。

1.3 Hash 冲突：得到下标值：

- 我们都知道在 HashMap 中使用数组加链表，这样问题就来了，数组使用起来是有下标的，但是我们平时使用 HashMap 都是这样使用的：

```
HashMap<Integer,String> hashMap=new HashMap<>();
hashMap.put(2,"dd");
```

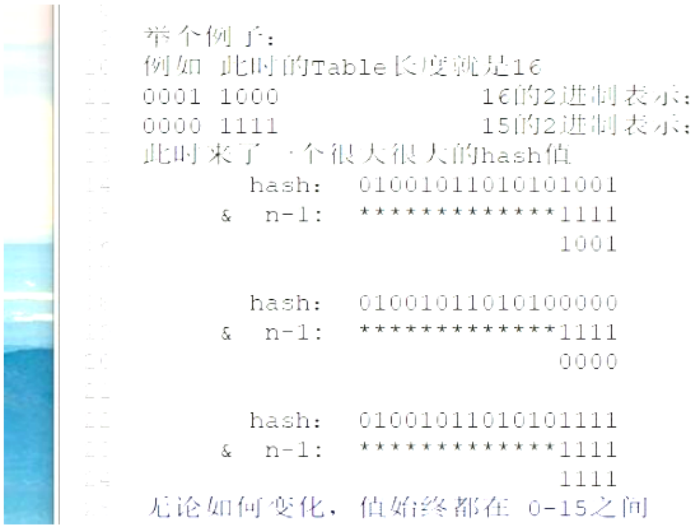
- 可以看到的是并没有特地为我们存放进来的值指定下标，那是因为我们的 hashMap 对存放进来的 key 值进行了 hashcode()，生成了一个值，但是这个值很大，我们不可以直接作为下标，此时我们想到了可以使用取余的方法，例如这样：

```
key.hashcode()%Table.length;
```

- 即可以得到对于任意的一个 key 值，进行这样的操作以后，其值都落在 0-Table.length-1 中，但是 HashMap 的源码却不是这样做?
- 它对其进行了与操作，对 Table 的表长度减一再与生产的 hash 值进行相与：

```
if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
```

- 我们来画张图进行进一步的了解；



- 这里我们也就得知为什么 Table 数组的长度要一直都为 2 的 n 次方，只有这样，减一进行相与时候，才能够达到最大的 n-1 值。
- 举个栗子来反证一下：

- 我们现在数组的长度为 15 减一为 14 ，二进制表示 0000 1110 进行相与时候，最后一位永远是 0，这样就可能导致，不能够完完全全的进行 Table 数组的使用。违背了我们最开始的想要对 Table 数组进行最大限度的无序使用的原则，因为 HashMap 为了能够存取高效，，要尽量较少碰撞，就是要尽量把数据分配均匀，每个链表F度大致相同。
- 此时还有一点需要注意的是：我们对 key 值进行 hashcode 以后，进行相与时候都是只用到了后四位，前面的很多位都没有能够得到使用，这样也可能会导致我们所生成的下标值不能够完全散列。
- 解决方案：将生成的 hashcode 值的高 16 位于低 16 位进行异或运算，这样得到的值再进行相与，一得到最散列的下标值。

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

1.4 二问讲一讲 HashMap 的 get/put 过程

- 知道 HashMap 的 put 元素的过程是什么样吗?
- 知道 get 过程是是什么样吗?
- 你还知道哪些的 hash 算法?
- 说一说 String 的 hashcode 的实现

1.4.1 Put 方法

- 1. 对 key 的 hashCode() 做 hash 运算，计算 index;
- 2. 如果没碰撞直接放到 bucket 里；
- 3. 如果碰撞了，以链表的形式存在 buckets 后；
- 4. 如果碰撞导致链表过F (大于等于 TREEIFY_THRESHOLD)，就把链表转换成红黑树 (JDK1.8 中的改动)；
- 5. 如果节点已经存在就替换 old value(保证 key 的唯一性)
- 6. 如果 bucket 满了 (超过 load factor*current capacity)，就要 resize
- 在得到下标值以后，可以开始 put 值进入到数组 + 链表中，会有三种情况：
  - 数组的位置为空。
  - 数组的位置不为空，且面是链表的格式。
  - 数组的位置不为空，且下面是红黑树的格式。
- 同时对于 Key 和 Value 也要经历一下步骤
  - 通过 Key 散列获取到对于的 Table；’
  - 遍历 Table 下的 Node 节点，做更新/添加操作；
  - 扩容检测；

```
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)
// HashMap 的懒加载策略，当执行 put 操作时检测 Table 数组初始化。
           n = (tab = resize()).length;
       if ((p = tab[i = (n - 1) & hash]) == null)
//通过 ``Hash``函数获取到对应的 Table，如果当前 Table 为空，则直接初始化一个新的 Node 并放入该 Table 中。
           tab[i] = newNode(hash, key, value, null);
       else {
           Node<K,V> e; K k;
           //进行值的判断：判断对于是不是对于相同的 key 值传进来不同的 value，若是如此，将原来的 value 进行返回
           if (p.hash == hash &&
               ((k = p.key) == key || (key != null && key.equals(k))))
               e = p;
           else if (p instanceof TreeNode)
          // 如果当前 Node 类型为 TreeNode，调用 PutTreeVal 方法。
               e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
           else {
//如果不是 TreeNode，则就是链表，遍历并与输入 key 做命中碰撞。
               for (int binCount = 0; ; ++binCount) {
                   if ((e = p.next) == null) {
//如果当前 Table 中不存在当前 key，则添加。
                       p.next = newNode(hash, key, value, null);
                       if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
//超过了 ``TREEIFY_THRESHOLD``则转化为红黑树。
                           treeifyBin(tab, hash);
                       break;
                   }
                   if (e.hash == hash &&
                       ((k = e.key) == key || (key != null && key.equals(k))))
//做命中碰撞，使用 hash、内存和 equals 同时判断（不同的元素 hash 可能会一致）。
                       break;
                   p = e;
               }
           }
           if (e != null) { // existing mapping for key
           //如果命中不为空，更新操作。
               V oldValue = e.value;
               if (!onlyIfAbsent || oldValue == null)
                   e.value = value;
               afterNodeAccess(e);
               return oldValue;
           }
       }
       ++modCount;
       if (++size > threshold)
       //扩容检测!
           resize();
       afterNodeInsertion(evict);
       return null;
   }
```

- 以上就是 HashMap 的 Put 操作，若是对其中的红黑树的添加，以及 Node 链表和红黑树的转换过程我们暂时不进行深入的讨论，这个流程大概还是可以进行理解，下面来深入讨论扩容问题。

1.4.2 resise 方法

- HashMap 的扩容实现机制是将老 table 数组中所有的 Entry 取出来，重新对其 Hashcode 做 Hash 散列到新的 Table 中，可以看到注解 Initializes or doubles table size. resize 表示的是对数组进行初始化或
- 进行 Double 处理。现在我们来一步一步进行分析。

```
/**
     * Initializes or doubles table size.  If null, allocates in
     * accord with initial capacity target held in field threshold.
     * Otherwise, because we are using power-of-two expansion, the
     * elements from each bin must either stay at same index, or move
     * with a power of two offset in the new table.
     *
     * @return the table
     */
```

```
final Node<K,V>[] resize() {
//先将老的 Table 取别名，这样利于后面的操作。
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    //表示之前的数组容量不为空。
    if (oldCap > 0) {
    // 如果 此时的数组容量大于最大值
        if (oldCap >= MAXIMUM_CAPACITY) {
        // 扩容 阈值为 Int 类型的最大值，这种情况很少出现
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        //表示 old 数组的长度没有那么大，进行扩容，两倍（这里也是有讲究的）对阈值也进行扩容
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    //表示之前的容量是 0 但是之前的阈值却大于零，此时新的 hash 表长度等于此时的阈值
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
    //表示是初始化时候，采用默认的 数组长度 * 负载因子
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    //此时表示若新的阈值为 0 就得用 新容量 * 加载因子重新进行计算。
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    // 开始对新的 hash 表进行相对应的操作。
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
    //遍历旧的 hash 表，将之内的元素移到新的 hash 表中。
        for (int j = 0; j < oldCap/*** 此时旧的 hash 表的阈值 */; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
            //表示这个格子不为空
                oldTab[j] = null;
                if (e.next == null)
                // 表示当前只有一个元素，重新做 hash 散列并赋值计算。
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                // 如果在旧哈希表中，这个位置是树形的结果，就要把新 hash 表中也变成树形结构，
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order
                //保留 旧 hash 表中是链表的顺序
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {// 遍历当前 Table 内的 Node 赋值给新的 Table。
                        next = e.next;
                        // 原索引
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        // 原索引 +oldCap
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    // 原索引放到 bucket 里面
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
```

```
                    }
                    // 原索引 +oldCap 放到 bucket 里面
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
        return newTab;
    }
```

1.4.3 get 方法

- 1. 对 key 的 hashCode() 做 hash 运算，计算 index;
- 2. 如果在 bucket 里的第一个节点里直接命中，则直接返回；
- 3. 如果有冲突，则通过 key.equals(k) 去查找对应的 Entry;
- 4. 若为树，则在树中通过 key.equals(k) 查找，O(logn)；
- 5. 若为链表，则在链表中通过 key.equals(k) 查找，O(n)。
- 在进行取值时候，因为对于我们传进来的 key 值进行了一系列的 hash 操作，首先，在传进来 key 值时候，先进性 hash 操作，

```
final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
        // 判断 表是否为空，表重读是否大于零，并且根据此 key 对应的表内是否存在 Node 节点。
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                // 检查第一个 Node 节点，若是命中则不需要进行 do... whirle 循环。
                return first;
            if ((e = first.next) != null) {
                if (first instanceof TreeNode)
                //树形结构，采用 对应的检索方法，进行检索。
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do {
                //链表方法 做 while 循环，直到命中结束或者遍历结束。
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

1.4.4 containsKey 方法

- 根据 get 方法的结果，判断是否为空，判断是否包含该 key

```
public boolean containsKey(Object key) {
        return getNode(hash(key), key) != null;
    }
```

- 还知道哪些 hash 算法
- 先说一下 hash 算法干嘛的，Hash 函数是指把一个大范围映射到一个小范围。把大范围映射到一个小范围的目的往往是为了节省空间，使得数据容易保存。
- 比较出名的有 MurmurHash、MD4、MD5 等等

1.4.5 String 中 hashcode 的实现

```
public int hashCode() {
        int h = hash;
        if (h == 0 && value.length > 0) {
            char val[] = value;
            for (int i = 0; i < value.length; i++) {
                h = 31 * h + val[i];
            }
            hash = h;
        }
        return h;
    }
```

- String 类中的 hashCode 计算方法还是比较简单的，就是以 31 为权，每一位为字符的 ASCII 值进行运算，用自然溢出来等效取模。
- 哈希计算公式可以计为 ss003311^^((nn–11)) + ss113311^^((nn–22)) + ……++ ssnn–11
- 那为什么以 31 为质数呢? 主要是因为 31 是一个奇质数，所以 31i=32i-i=(i«5)-i，这种位移与减法结合的计算相比一般的运算快很多

1.5 三问为什么 hashmap 的在链表元素数量超过 8 时候改为红黑树

- 知道 jdk1.8 中 hashmap 改了什么吗。
- 说一下为什么会出现线程的不安全性
- 为什么在解决 hash 冲突时候，不直接用红黑树，而是先用链表，再用红黑树
- 当链表转为红黑树，什么时候退化为链表

1.5.1 第一问改动了什么

- 1. 由数组 + 链表的结构改为数组 + 链表 + 红黑树。
- 2. 优化了高位运算的 hash 算法：h^(h»>16)
- 3. 扩容后，元素要么是在原位置，要么是在原位置再移动 2 次幂的位置，且链表顺序不变。
- 注意：最后一条是重点，因为最后一条的变动，hashmap 在 1.8 中，不会在出现死循环问题。

1.5.2 HashMap 的线程不安全性

- HashMap 在 jdk1.7 中使用数组加链表的方式，并且在进行链表插入时候使用的是头结点插入的方法。
- 注：这里为什么使用头插法的原因是我们若是在散列以后，判断得到值是一样的，使用头插法，不用每次进行遍历链表的长度。但是这样会有一个缺点，在进行扩容时候，会导致进入新数组时候出现倒序的情况，也会在多线程时候出现线程的不安全性。
- 但是对与 jdk1.8 而言，还是要进行阙值的判断，判断在什么时候进行红黑树和链表的转换。所以无论什么时候都要进行遍历，于是插入到尾部，防止出现扩容时候还会出现倒序情况。
- **所以当在多线程的使用场景中,尽量使用线程安全的 ConcurrentHashMap。至于 Hashtable 而言，使用效率太低。**

1.5.3 线程安全

```
// 扩容
void transfer(Entry[] newTable, boolean rehash) {
                    int newCapacity = newTable.length;
                    for (Entry<K,V> e : table) { // A
                         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; }
                          }
 }
```

- 在 jdk1.7 若是产生了多线程，例如 thread1，和 thread2，同时想要进入到 transfer 中，此时会出现如下图所示的情况：
- 此时对于我们的 1 会拥有两个临时变量，我们称为 e1 与 e2。这个时候，线程一会先执行上述的函数，进行数组的翻倍，并且，会进入逆序的状态，此时的临时变量 e1 和 next1 都已经消失，但是对于每个节点上面所拥有的连接不会更改，这个时候，1 上还有一个 e2 临时变量，2 上有一个 next2 临时变量。如下图所示：
- 完成了线程一的扩容以后，线程二也会创建一个属于自己的数组，长度也是 6。这个时候开始又执行一遍以上的程序。

```
// 第一遍过来
e.next = newTable[i];
newTable[i] = e;
e = next;
```

- 此时完成了第一次的循环以后，进入到以上的情况，这个时候执行 e.next = newTable[i]; 寓意为：2 所表示的下一个指向 newTable[i], 此时我们就发现了问题的所在，在执行完第一遍循环以后，2 所表示的下一下就已经指向了 newTable[i], 就是我们的 1 ，当然这样我们就不用动，那我们就不动就好了，然后完成以后就如下图所示。

```
// 第二遍来
e.next = newTable[i];
newTable[i] = e;
e = next;
```

- 这个时候开始第三次的循环，首先执行 Entry<K,V> next = e.next; ，这个时候我们就发现了问题，e2 和 e2 的 next2 都执行了 1，这个时候我们再度，执行以上的语句就会指向一个空的节点，当然空就空了，暂时也还不会出现差错，但是执行到 e.next = newTable[i]; 时候，会发现，执行到如下图所示的情况。这个时候出现了循环链表，若是不加以控制，就会耗尽我们的 cpu。

1.5.4 第三问为什么不一开始就使用红黑树，不是效率很高吗?

- 因为红黑树需要进行左旋，右旋，变色这些操作来保持平衡，而单链表不需要。
- 当元素小于 8 个当时候，此时做查询操作，链表结构已经能保证查询性能。
- 当元素大于 8 个的时候，此时需要红黑树来加快查询速度，但是新增节点的效率变慢了。
- 因此，如果一开始就用红黑树结构，元素太少，新增效率又比较慢，无疑这是浪费性能的。

1.5.5 第四问什么时候退化为链表

- 为 6 的时候退转为链表。中间有个差值 7 可以防止链表和树之间频繁的转换。
- 假设一下，如果设计成链表个数超过 8 则链表转换成树结构，链表个数小于 8 则树结构转换成链表，
- 如果一个 HashMap 不停的插入、删除元素，链表个数在 8 左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。

1.6 四问 HashMap 的并发问题

- HashMap 在并发环境下会有什么问题
- 一般是如何解决的

1.6.1 问题的出现

- (1) 多线程扩容，引起的死循环问题
- (2) 多线程 put 的时候可能导致元素丢失
- (3)put 非 null 元素后 get 出来的却是 null

1.6.2 不安全性的解决方案

- 在之前使用 hashtable。在每一个函数前面都加上了 synchronized 但是效率太低我们现在不常用了。
- 使用 ConcurrentHashmap 函数，对于这个函数而言我们可以每几个元素共用一把锁。用于提高效率。

1.7 五问你一般用什么作为 HashMap 的 key 值

- key 可以是 null 吗，value 可以是 null 吗
- 一般用什么作为 key 值
- 用可变类当 Hashmap1 的 Key 会有什么问题
- 让你实现一个自定义的 class 作为 HashMap 的 Key 该如何实现

1.7.1 key 可以是 null 吗，value 可以是 null 吗

- 当然都是可以的，但是对于 key 来说只能运行出现一个 key 值为 null，但是可以出现多个 value 值为 null

1.7.2 一般用什么作为 key 值

- 一般用 Integer、String 这种不可变类当 HashMap 当 key，而且 String 最为常用。
  - (1) 因为字符串是不可变的，所以在它创建的时候 hashcode 就被缓存了，不需要重新计算。这就使得字符串很适合作为 Map 中的键，字符串的处理速度要快过其它的键对象。这就是 HashMap 中的键往往都使用字符串。
  - (2) 因为获取对象的时候要用到 equals() 和 hashCode() 方法，那么键对象正确的重写这两个方法是非常重要的，这些类已经很规范的覆写了 hashCode() 以及 equals() 方法。

1.7.3 用可变类当 Hashmap1 的 Key 会有什么问题

- hashcode 可能会发生变化，导致 put 进行的值，无法 get 出来，如下代码所示：

```
HashMap<List<String>,Object> map=new HashMap<>();
        List<String> list=new ArrayList<>();
        list.add("hello");
        Object object=new Object();
        map.put(list,object);
        System.out.println(map.get(list));
        list.add("hello world");
        System.out.println(map.get(list));
```

- 输出值如下：

```
java.lang.Object@1b6d3586
null
```

1.7.4 实现一个自定义的 class 作为 Hashmap 的 key 该如何实现

- 对于这个问题考查到了下面的两个知识点
  - 重写 hashcode 和 equals 方法需要注意什么?
  - 如何设计一个不变的类。

1. 针对问题一，记住下面四个原则即可
   - (1) 两个对象相等，hashcode 一定相等
   - (2) 两个对象不等，hashcode 不一定不等
   - (3)hashcode 相等，两个对象不一定相等
   - (4)hashcode 不等，两个对象一定不等
2. 针对问题二，记住如何写一个不可变类
   - (1) 类添加 final 修饰符，保证类不被继承。如果类可以被继承会破坏类的不可变性机制，只要继承类覆盖父类的方法并且继承类可以改变成员变量值，那么一旦子类以父类的形式出现时，不能保证当前类是否可变。
   - (2) 保证所有成员变量必须私有，并且加上 final 修饰通过这种方式保证成员变量不可改变。但只做到这一步还不够，因为如果是对象成员变量有可能再外部改变其值。所以第 4 点弥补这个不足。
   - (3) 不提供改变成员变量的方法，包括 setter 避免通过其他接口改变成员变量的值，破坏不可变特性。
   - (4) 通过构造器初始化所有成员，进行深拷□ (deep copy)
   - (5) 在 getter 方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷□这种做法也是防止对象外泄，防止通过 getter 获得内部可变成员对象后对成员变量直接操作，导致成员变量发生改变

1.8 后记

- 对于 HashMap 而言，扩容是一个特别消耗内存的操作。所以当程序员在使用 HashMap 的时候，估算 map 的大小，初始化的时候给一个大致的数值，避免 map 进行频繁的扩容。
- 负载因子是可以修改的，也可以大于 1，但是建议不要轻易修改，除非情况非常特殊。
- HashMap 是线程不安全的，不要在并发的环境中同时操作 HashMap，建议使用 ConcurrentHashMap。

2 SparseArray、ArrayMap 实现原理

- SparseArray 与 ArrayMap 是 Android 提供的两个列表数据结构。
- **SparseArray** 相比于 HashMap 采用的是，**时间换取空间的方式来提高手机 App 的运行效率**。而 **ArrayMap 实现原理上也类似于 SparseArray**。
- SparseArray 源码来自： android-25/java/util/SparseArray
- ArrayMap 源码来自：25.3.1/support-compat-25.3.1/android/android.support.v4.util.ArrayMap

2.1 一、SparseArray 实现源码学习

- **SparseArray 采用时间换取空间的方式来提高手机 App 的运行效率**，这也是其与 HashMap 的区别；**HashMap 通过空间换取时间，查找迅速**；HashMap 中当 table 数组中内容达到总容量 0.75 时，则扩展为当前容量的两倍，关于 HashMap 可查看 HashMap 实现原理学习)
- SparseArray 的 key 为 int，value 为 Object。
- **在 Android 中，数据长度小于千时，用于替换 HashMap**
- 相比与 HashMap，其采用时间换空间的方式，使用更少的内存来提高手机 APP 的运行效率 (HashMap 中当 table 数组中内容达到总容量 0.75 时，则扩展为当前容量的两倍，关于 HashMap 可查看 HashMap 实现原理学习)