常见面试题

HashMap常见面试题：

- HashMap的底层数据结构?
- HashMap的存取原理?
- Java7和Java8的区别?
- 默认初始化大小是多少？为啥是这么多？为啥大小都是2的幂?
- HashMap的扩容方式？负载因子是多少？为什是这么多?

HashMap的底层数据结构?

1.7 数组+链表

- 节点名

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

1.8 数组+红黑树

- 节点名

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

HashMap存和取

1.7

```
public V put(K key, V value) {
  //先判断当前的数组是否为空
    if (table == EMPTY_TABLE) {
      inflateTable(threshold);
    }
  //如果key是空,则put到第一个下标
    if (key == null)
      return putForNullKey(value);
  //计算 hash
    int hash = hash(key);
```

```
    //获得存入的下标
        int i = indexFor(hash, table.length);
    //遍历当前下标的链表
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            //如果 hash 相同,并且   key 相同或者  equals 相同
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                //覆盖原本的值
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }

    //没有相同的 对象存储在此 hashmap 中
        modCount++;
        addEntry(hash, key, value, i);
        return null;
    }
```

```
/**
     * 当 key 为 null 的时候
     */
    private V putForNullKey(V value) {
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
            if (e.key == null) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }
        modCount++;
        addEntry(0, null, value, 0);
        return null;
    }
```

```
/**
     * 与运算
     */
    static int indexFor(int h, int length) {
        // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";
        return h & (length-1);
    }
```

例如 数组容量为16

h=5　　: 0000 0101

length=16　: 0000 1111

return　　: 0000 0101

```
/**
    * The number of times this HashMap has been structurally modified
    * Structural modifications are those that change the number of mappings in
    * the HashMap or otherwise modify its internal structure (e.g.,
    * rehash).  This field is used to make iterators on Collection-views of
    * the HashMap fail-fast.  (See ConcurrentModificationException).
    * 对该HashMap进行结构修改的次数*结构修改是指更改* HashMap中的映射数或以其他方式修改其内部结构（例如*重新哈希）的修改。此字段用于使HashMap的Collection-view上的迭代器快速失败。 （请参阅ConcurrentModificationException）
    */
   transient int modCount;
```

foreach : 删对象出问题,原因在于底层的 fail-fast 机制

```
/**
    * Adds a new entry with the specified key, value and hash code to
    * the specified bucket.  It is the responsibility of this
    * method to resize the table if appropriate.
    *
    * Subclass overrides this to alter the behavior of put method.
    */
//添加的方法
   void addEntry(int hash, K key, V value, int bucketIndex) {
      //如果当前的 size 大于 阈值 同时 数组的当前下标的对象 不为空
      if ((size >= threshold) && (null != table[bucketIndex])) {
         //扩容两倍长度
         resize(2 * table.length);
         //如果这个 key 是 null  那么hash就等于 0
         hash = (null != key) ? hash(key) : 0;
         //index 下标也就等于 0
         bucketIndex = indexFor(hash, table.length);
      }

      //新增一个节点
      createEntry(hash, key, value, bucketIndex);
   }
```

```
void createEntry(int hash, K key, V value, int bucketIndex) {
      Entry<K,V> e = table[bucketIndex];
      table[bucketIndex] = new Entry<>(hash, key, value, e);
      size++;
   }
```

1.8

```
/**
    * 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;
        //如果数组为空，则新生成一个数组
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            //链表的方式插入数据
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            //如果是树节点，就按照红黑树的方式插入节点数据
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        //变成红黑树
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    //重复则覆盖
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        //大于阈值，重新生成节点
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

```
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; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;

  //为 0 时，运行的代码
    else {               // zero initial threshold signifies using defaults
        //初始化空间大小
        newCap = DEFAULT_INITIAL_CAPACITY;
        //扩容阈值
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
```

```
/**
   * Replaces all linked nodes in bin at index for given hash unless
   * table is too small, in which case resizes instead.
   */
  final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        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);
    }
  }
```

扩容

```
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;
        }
```

```
        //两倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
}
```

Java7和Java8的区别?

红黑树和链表区别

1.7死锁问题

```
//重新设置容量大小
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
      threshold = Integer.MAX_VALUE;
      return;
    }

    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable, initHashSeedAsNeeded(newCapacity));
    table = newTable;
    threshold = (int)Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1);
  }
```

```
/**
   * Transfers all entries from current table to newTable.
   */
  void transfer(Entry[] newTable, boolean rehash) {
    //初始容量为原来的数组长度的两倍
    int newCapacity = newTable.length;
    //遍历原本的ｔａｂｌｅ
    for (Entry<K,V> e : table) {
      //将此位置下的链表放入新的数组的下标中
      while(null != e) {
        Entry<K,V> next = e.next;
        //重新hash
        if (rehash) {
          e.hash = null == e.key ? 0 : hash(e.key);
        }
        //获得下标
        int i = indexFor(e.hash, newCapacity);
        //出现死锁问题出，链表倒置
        e.next = newTable[i];
        newTable[i] = e;
        e = next;
      }
    }
  }
```

1.8的 resize

核心（非红黑树插入）

```
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;
    }
}
```

总之就是，尾插法，并不是像１．７的头插法，会倒置链表

默认初始化大小是多少？为啥是这么多？为啥大小都是2的幂？

1.7

```
/**
     * The default initial capacity - MUST be a power of two.
     */
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

    /**
     * The maximum capacity, used if a higher value is implicitly specified
     * by either of the constructors with arguments.
     * MUST be a power of two <= 1<<30.
     */
    static final int MAXIMUM_CAPACITY = 1 << 30;
```

```
    /**
     * The load factor used when none specified in constructor.
     */
    static final float DEFAULT_LOAD_FACTOR = 0.75f;

    /**
     * An empty table instance to share when the table is not inflated.
     */
    static final Entry<?,?>[] EMPTY_TABLE = {};

    /**
     * The table, resized as necessary. Length MUST Always be a power of two.
     */
    transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY_TABLE;

    /**
     * The number of key-value mappings contained in this map.
     */
    transient int size;

    /**
     * The next size value at which to resize (capacity * load factor).
     * @serial
     */
    // If table == EMPTY_TABLE then this is the initial capacity at which the
    // table will be created when inflated.
    int threshold;

    /**
     * The load factor for the hash table.
     *
     * @serial
     */
    final float loadFactor;

    /**
     * The number of times this HashMap has been structurally modified
     * Structural modifications are those that change the number of mappings in
     * the HashMap or otherwise modify its internal structure (e.g.,
     * rehash).  This field is used to make iterators on Collection-views of
     * the HashMap fail-fast.  (See ConcurrentModificationException).
     */
    transient int modCount;
```

1.8

```
    /**
     * The default initial capacity - MUST be a power of two.
     */
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

    /**
     * The maximum capacity, used if a higher value is implicitly specified
     * by either of the constructors with arguments.
     * MUST be a power of two <= 1<<30.
     */
```

```
static final int MAXIMUM_CAPACITY = 1 << 30;

/**
 * The load factor used when none specified in constructor.
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;

/**
 * The bin count threshold for using a tree rather than list for a
 * bin.  Bins are converted to trees when adding an element to a
 * bin with at least this many nodes. The value must be greater
 * than 2 and should be at least 8 to mesh with assumptions in
 * tree removal about conversion back to plain bins upon
 * shrinkage.
 */
static final int TREEIFY_THRESHOLD = 8;

/**
 * The bin count threshold for untreeifying a (split) bin during a
 * resize operation. Should be less than TREEIFY_THRESHOLD, and at
 * most 6 to mesh with shrinkage detection under removal.
 */
static final int UNTREEIFY_THRESHOLD = 6;

/**
 * The smallest table capacity for which bins may be treeified.
 * (Otherwise the table is resized if too many nodes in a bin.)
 * Should be at least 4 * TREEIFY_THRESHOLD to avoid conflicts
 * between resizing and treeification thresholds.
 */
static final int MIN_TREEIFY_CAPACITY = 64;
```

DEFAULT_INITIAL_CAPACITY：默认的初始化容量，1<<4位运算的结果是16，也就是默认的初始化容量为16。当然如果对要存储的数据有一个估计值，最好在初始化的时候显示的指定容量大小，减少扩容时的数据搬移等带来的效率消耗。同时，容量大小需要是2的整数倍。

MAXIMUM_CAPACITY：容量的最大值.

DEFAULT_LOAD_FACTOR：默认的加载因子，设计HashMap和HashTable的那些大叔说这个数值是基于时间和空间消耗上最好的数值，也不知道是不是看我读书少糊弄我。这个值和容量的乘积是一个很重要的数值，也就是阈值，当达到这个值时候会产生扩容，扩容的大小大约为原来的二倍。

TREEIFY_THRESHOLD：因为jdk8以后，HashMap底层的存储结构改为了数组+链表+红黑树的存储结构（之前是数组+链表），刚开始存储元素产生碰撞时会在碰撞的数组后面挂上一个链表，当链表长度大于这个参数时，链表就可能会转化为红黑树，为什么是可能，em。。。。后面还有一个参数，需要他们两个都满足的时候才会转化。

UNTREEIFY_THRESHOLD：介绍上面的参数时，我们知道当长度过大时可能会产生从链表到红黑树的转化，但是，元素不仅仅只能添加还可以删除，或者另一种情况，扩容后该数组槽位置上的元素数据不是很多了，还使用红黑树的结构就会很浪费，所以这时就可以把红黑树结构变回链表结构，什么时候变，就是元素数量等于这个值也就是6的时候变回来（元素数量指的是一个数组槽内的数量，不是HashMap中所有元素的数量）。

MIN_TREEIFY_CAPACITY：链表树化的一个标准，前面说过当数组槽内的元素数量大于8时可能会转化为红黑树，之所以说是可能就是因为这个值，当数组的长度小于这个值是，会先去进行扩容，扩容之后就有很大的可能让数组槽内的数据可以更分散一些了，也就不用转化数组槽后的存储结构了。当然，长度大于这个值并且槽内数据大于8时，那就老老实实的转化为红黑树吧。

table：HashMap中存储数据的格式是内部声明的一个Node<K,V>类，而存储这些Node<K,V>是使用的数组。

entrySet：存储的键值对形成一个entrySet，用一个set集合存储。

size：指的是整个HashMap中存储的数据的个数。

modCount：由于HashMap是线程不安全的类，所以在操作HashMap中的数据时，会记录这个修改的次数，当使用迭代器遍历HashMap中的数据时，先把这个值赋给迭代器的expectedModCount，迭代的过程中比较这两个值，如果不相等直接抛异常，也就是源码注释中写的fail-fast机制。

threshold：扩容时的阈值。

loadFactor：加载因子，和容量的乘积就是阈值。