HashMap与ConcurrentHashMap面试要点

HashMap

HashMap底层数据结构

JDK7：数组+链表
JDK8：数组+链表+红黑树（看过源码的同学应该知道JDK8中即使用了单向链表，也使用了双向链表，双向链表主要是为了链表操作方便，应该在插入，扩容，链表转红黑树，红黑树转链表的过程中都要操作链表）

JDK8中的HashMap为什么要使用红黑树?

当元素个数小于一个阈值时，链表整体的插入查询效率要高于红黑树，当元素个数大于此阈值时，链表整体的插入查询效率要低于红黑树。此阈值在HashMap中为8

JDK8中的HashMap什么时候将链表转化为红黑树?

这个题很容易答错，大部分答案就是：当链表中的元素个数大于8时就会把链表转化为红黑树。但是其实还有另外一个限制：当发现链表中的元素个数大于8之后，还会判断一下当前数组的长度，如果数组长度小于

64时，此时并不会转化为红黑树，而是进行扩容。只有**当链表中的元素个数大于8，并且数组的长度大于等于64时才会将链表转为红黑树**。

上面扩容的原因是，如果数组长度还比较小，就先利用扩容来缩小链表的长度。

JDK8中HashMap的put方法的实现过程?

1. 根据key生成hashcode
2. 判断当前HashMap对象中的数组是否为空，如果为空则初始化该数组
3. 根据逻辑与运算，算出hashcode基于当前数组对应的数组下标i
4. 判断数组的第i个位置的元素（tab[i]）是否为空
    a. 如果为空，则将key，value封装为Node对象赋值给tab[i]
    b. 如果不为空：
        i. 如果put方法传入进来的key等于tab[i].key，那么证明存在相同的key
        ii. 如果不等于tab[i].key，则：
            1. 如果tab[i]的类型是TreeNode，则表示数组的第i位置上是一颗红黑树，那么将key和value插入到红黑树中，并且在插入之前会判断在红黑树中是否存在相同的key
            2. 如果tab[i]的类型不是TreeNode，则表示数组的第i位置上是一个链表，那么遍历链表寻找是否存在相同的key，并且在遍历的过程中会对链表中的结点数进行计数，当遍历到最后一个结点时，会将key,value封装为Node插入到链表的尾部，同时判断在插入新结点之前的链表结点个数是不是大于等于8，如果是，则将链表改为红黑树。
        iii. 如果上述步骤中发现存在相同的key，则根据onlyIfAbsent标记来判断是否需要更新value值，然后返回oldValue
5. modCount++
6. HashMap的元素个数size加1
7. 如果size大于扩容的阈值，则进行扩容

JDK8中HashMap的get方法的实现过程

1. 根据key生成hashcode
2. 如果数组为空，则直接返回空
3. 如果数组不为空，则利用hashcode和数组长度通过逻辑与操作算出key所对应的数组下标i
4. 如果数组的第i个位置上没有元素，则直接返回空
5. 如果数组的第1个位上的元素的key等于get方法所传进来的key，则返回该元素，并获取该元素的value
6. 如果不等于则判断该元素还有没有下一个元素，如果没有，返回空
7. 如果有则判断该元素的类型是链表结点还是红黑树结点
    a. 如果是链表则遍历链表
    b. 如果是红黑树则遍历红黑树
8. 找到即返回元素，没找到的则返回空

JDK7与JDK8中HashMap的不同点

1. JDK8中使用了红黑树
2. JDK7中链表的插入使用的**头插法**（扩容转移元素的时候也是使用的头插法，头插法速度更快，无需遍历链表，但是在多线程扩容的情况下使用头插法会出现循环链表的问题，导致CPU飙升），JDK8中链表使用的**尾插法**（JDK8中反正要去计算链表当前结点的个数，反正要遍历的链表的，所以直接使用尾插法）
3. JDK7的Hash算法比JDK8中的更复杂，Hash算法越复杂，生成的hashcode则更散列，那么hashmap中的元素则更散列，更散列则hashmap的查询性能更好，JDK7中没有红黑树，所以只能优化Hash算法使得元素更散列，而JDK8中增加了红黑树，查询性能得到了保障，所以可以简化一下Hash算法，毕竟Hash算法越复杂就越消耗CPU
4. 扩容的过程中JDK7中有可能会重新对key进行哈希（重新Hash跟哈希种子有关系），而JDK8中没有这部分逻辑
5. JDK8中扩容的条件和JDK7中不一样，除开判断size是否大于阈值之外，JDK7中还判断了tab[i]是否为空，不为空的时候才会进行扩容，而JDK8中则没有该条件了
6. JDK8中还多了一个API：putIfAbsent(key,value)
7. JDK7和JDK8扩容过程中转移元素的逻辑不一样，JDK7是每次转移一个元素，JDK8是先算出来当前位置上哪些元素在新数组的低位上，哪些在新数组的高位上，然后在一次性转移

ConcurrentHashMap

JDK7中的ConcurrentHashMap是怎么保证并发安全的?

主要利用Unsafe操作+ReentrantLock+分段思想。
主要使用了Unsafe操作中的：

1. compareAndSwapObject：通过cas的方式修改对象的属性
2. putOrderedObject：并发安全的给数组的某个位置赋值
3. getObjectVolatile：并发安全的获取数组某个位置的元素

分段思想是为了提高ConcurrentHashMap的并发量，分段数越高则支持的最大并发量越高，程序员可以通过concurrencyLevel参数来指定并发量。ConcurrentHashMap的内部类Segment就是用来表示某一个段的。

每个Segment就是一个小型的HashMap的，当调用ConcurrentHashMap的put方法是，最终会调用到Segment的put方法，而Segment类继承了ReentrantLock，所以Segment自带可重入锁，当调用到Segment的put方法时，会先利用可重入锁加锁，加锁成功后再将待插入的key,value插入到小型HashMap中，插入完成后解锁。

JDK7中的ConcurrentHashMap的底层原理

ConcurrentHashMap底层是由两层嵌套数组来实现的：

1. ConcurrentHashMap对象中有一个属性segments，类型为Segment[];
2. Segment对象中有一个属性table，类型为HashEntry[];

当调用ConcurrentHashMap的put方法时，先根据key计算出对应的Segment[]的数组下标j，确定好当前key,value应该插入到哪个Segment对象中，如果segments[j]为空，则利用自旋锁的方式在j位置生成一个Segment对象。

然后调用Segment对象的put方法。

Segment对象的put方法会先加锁，然后也根据key计算出对应的HashEntry[]的数组下标i，然后将key,value封装为HashEntry对象放入该位置，此过程和JDK7的HashMap的put方法一样，然后解锁。

在加锁的过程中逻辑比较复杂，先通过自旋加锁，如果超过一定次数就会直接阻塞等等加锁。（具体流程请求看vip视频.）

JDK8中的ConcurrentHashMap是怎么保证并发安全的?

主要利用Unsafe操作+synchronized关键字。
Unsafe操作的使用仍然和JDK7中的类似，主要负责并发安全的修改对象的属性或数组某个位置的值。

synchronized主要负责在需要操作某个位置时进行加锁（该位置不为空），比如向某个位置的链表进行插入结点，向某个位置的红黑树插入结点。

JDK8中其实仍然有分段锁的思想，只不过JDK7中段数是可以控制的，而JDK8中是数组的每一个位置都有一把锁。

当向ConcurrentHashMap中put一个key,value时，

1. 首先根据key计算对应的数组下标i，如果该位置没有元素，则通过自旋的方法去向该位置赋值。
2. 如果该位置有元素，则synchronized会加锁
3. 加锁成功之后，在判断该元素的类型
    a. 如果是链表节点则进行添加节点到链表中
    b. 如果是红黑树则添加节点到红黑树
4. 添加成功后，判断是否需要进行树化
5. addCount，这个方法的意思是ConcurrentHashMap的元素个数加1，但是这个操作也是需要并发安全的，并且元素个数加1成功后，会继续判断是否要进行扩容，如果需要，则会进行扩容，所以这个方法很重要。
6. 同时一个线程在put时如果发现当前ConcurrentHashMap正在进行扩容则会去帮助扩容。

扩容流程源码解析请看vip视频讲解。

JDK7和JDK8中的ConcurrentHashMap的不同点

这两个的不同点太多了...，既包括了HashMap中的不同点，也有其他不同点，比如：

1. JDK8中没有分段锁了，而是使用synchronized来进行控制
2. JDK8中的扩容性能更高，支持多线程同时扩容，实际上JDK7中也支持多线程扩容，因为JDK7中的扩容是针对每个Segment的，所以也可能多线程扩容，但是性能没有JDK8高，因为JDK8中对于任意一个线程都可以去帮助扩容
3. JDK8中的元素个数统计的实现也不一样了，JDK8中增加了CounterCell来帮助计数，而JDK7中没有，JDK7中是put的时候每个Segment内部计数，统计的时候是遍历每个Segment对象加锁统计（当然有一点点小小的优化措施，看视频吧..）。