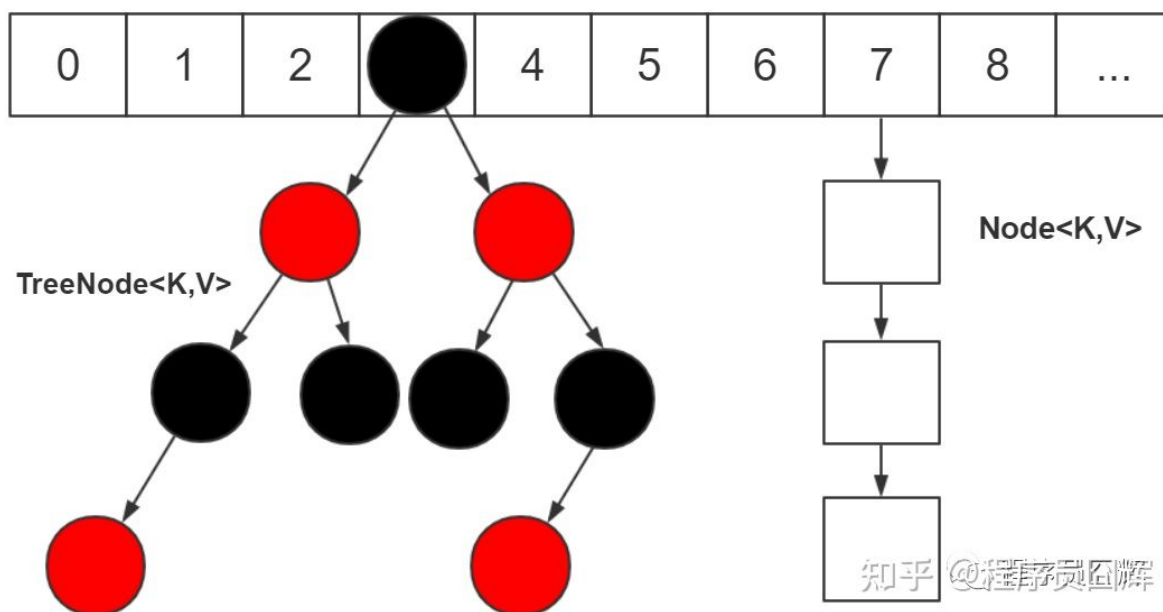


## 1、介绍下 HashMap 的底层数据结构吧。

在 JDK 1.8, HashMap 底层是由“数组+链表+红黑树”组成，如下图所示，而在 JDK 1.8 之前是由“数组+链表”组成，就是下图去掉红黑树。

Node<K,V>[] table



## 2、为什么使用“数组+链表”？

使用“数组+链表”是为了解决 hash 冲突的问题。

数组和链表有如下特点：

数组：查找容易，通过 index 快速定位；插入和删除困难，需要移动插入和删除位置之后的节点；

链表：查找困难，需要从头结点或尾节点开始遍历，直到寻找到目标节点；插入和删除容易，只需修改目标节点前后节点的 next 或 prev 属性即可；

HashMap 巧妙的将数组和链表结合在一起，发挥两者各自的优势，使用一种叫做“拉链法”的方式来解决哈希冲突。

首先通过 index 快速定位到索引位置，这边利用了数组的优点；然后遍历链表找到节点，进行节点的新增/修改/删除操作，这边利用了链表的优点。简直，完美。

# 完美!



### 3、为什么要改成“数组+链表+红黑树”？

通过上题可以看出，“数组+链表”已经充分发挥了这两种数据结构的优点，是个很不错的组合了。

但是这种组合仍然存在问题，就是在定位到索引位置后，需要先遍历链表找到节点，这个地方如果链表很长的话，也就是 hash 冲突很严重的时候，会有查找性能问题，因此在 JDK1.8 中，通过引入红黑树，来优化这个问题。

使用链表的查找性能是  $O(n)$ ，而使用红黑树是  $O(\log n)$ 。

### 4、那在什么时候用链表？什么时候用红黑树？

对于插入，默认情况下是使用链表节点。当同一个索引位置的节点在新增后超过 8 个（阈值 8）：如果此时数组长度大于等于 64，则会触发链表节点转红黑树节点（treeifyBin）；而如果数组长度小于 64，则不会触发链表转红黑树，而是会进行扩容，因为此时的数据量还比较小。

对于移除，当同一个索引位置的节点在移除后达到 6 个（阈值 6），并且该索引位置的节点为红黑树节点，会触发红黑树节点转链表节点（untreeify）。

### 5、为什么链表转红黑树的阈值是 8？

我们平时在进行方案设计时，必须考虑的两个很重要的因素是：时间和空间。对于 HashMap 也是同样的道理，简单来说，阈值为 8 是在时间和空间上权衡的结果（这 B 我装定了）。

就算打死我，这逼装定了



红黑树节点大小约为链表节点的 2 倍，在节点太少时，红黑树的查找性能优势并不明显，付出 2 倍空间的代价作者觉得不值得。

理想情况下，使用随机的哈希码，节点分布在 hash 桶中的频率遵循泊松分布，按照泊松分布的公式计算，链表中节点个数为8时的概率为 0.00000006（**就我们这QPS不到10的系统，根本不可能遇到嘛**），这个概率足够低了，并且到8个节点时，红黑树的性能优势也会开始展现出来，因此8是一个较合理的数字。

## 6、那为什么转回链表节点是用的6而不是复用8？

如果我们设置节点多于8个转红黑树，少于8个就马上转链表，当节点个数在8徘徊时，就会频繁进行红黑树和链表的转换，造成性能的损耗。

## 7、HashMap 有哪些重要属性？分别用于做什么的？

除了用来存储我们的节点 table 数组外，HashMap 还有以下几个重要属性：

- 1) size: HashMap 已经存储的节点个数；
- 2) threshold: 1) 扩容阈值（主要），当 HashMap 的个数达到该值，触发扩容。2) 初始化时的容量，在我们新建 HashMap 对象时，threshold 还会被用来存初始化时的容量。HashMap 直到我们第一次插入节点时，才会对 table 进行初始化，避免不必要的空间浪费。
- 3) loadFactor: 负载因子，扩容阈值 = 容量 \* 负载因子。

## 8、HashMap 的默认初始容量是多少？HashMap 的容量有什么限制吗？

默认初始容量是16。HashMap 的容量必须是2的N次方，HashMap 会根据我们传入的容量计算一个“大于等于该容量的最小的 2 的 N 次方”，例如传 16，容量为16；传17，容量为32。

## 9、“大于等于该容量的最小的2的N次方”是怎么算的？

```
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

我们先不看第一行“int n = cap - 1”，先看下面的5行计算。

|=（或等于）：这个符号比较少见，但是“+=”应该都见过，看到这你应该明白了。例如：a |= b，可以转成：a = a | b。

a	1 0 1
b	1 0 0
<hr/>	
a   b	1 0 1

>>> (无符号右移)：例如  $a \gg b$  指的是将  $a$  向右移动  $b$  指定的位数，右移后左边空出的位用零来填充，移出右边的位被丢弃。

a	1 1 0
<hr/>	
$a \gg 1$	0 1 1
$a \gg 2$	0 0 1

假设  $n$  的值为 0010 0001，则该计算如下图：

		最高位的1							
		↓							
n	0 0 1 0 0 0 0 1								
$n \gg 1$	0 0 0 1 0 0 0 0 1								
<hr/>									
$n   n \gg 1$	0 0 1 1 0 0 0 1								
<hr/>									
n	0 0 1 1 0 0 0 1								
$n \gg 2$	0 0 0 0 1 1 0 0 1								
<hr/>									
$n   n \gg 2$	0 0 1 1 1 1 0 1								

最高位的1右移到次高位

由于是或运算，所以此时  $n$  的最高和次高位都必然为1

同理，再右移2位进行相同操作后，可得到4个1

相信你应该看出来，这5个公式会通过最高位的1，拿到2个1、4个1、8个1、16个1、32个1。当然，有多少个1，取决于我们的入参有多大，但我们肯定的是经过这5个计算，得到的值是一个低位全是1的值，最后返回的时候 +1，则会得到1个比  $n$  大的 2 的  $N$  次方。

这时再看开头的  $\text{cap} - 1$  就很简单了，这是为了处理  $\text{cap}$  本身就是 2 的 N 次方的情况。

计算机底层是二进制的，移位和或运算是非常快的，所以这个方法的效率很高。

PS：这是 HashMap 中我个人最喜欢的设计，非常巧妙。

## 10、HashMap 的容量必须是 2 的 N 次方，这是为什么？

核心目的是：实现节点均匀分布，减少 hash 冲突。

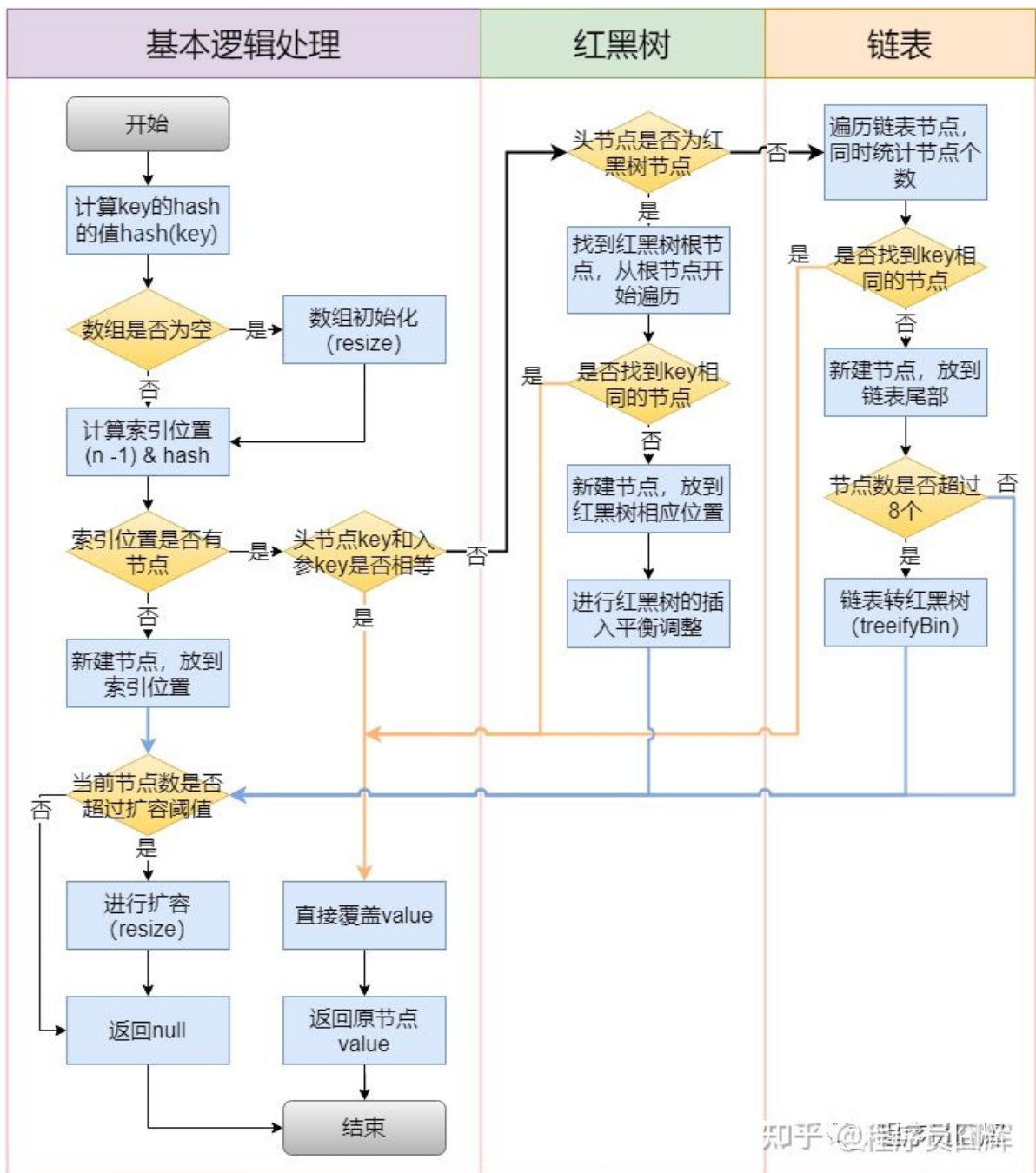
计算索引位置的公式为： $(n - 1) \& \text{hash}$ ，当  $n$  为 2 的 N 次方时， $n - 1$  为低位全是 1 的值，此时任何值跟  $n - 1$  进行  $\&$  运算的结果为该值的低 N 位，达到了和取模同样的效果，实现了均匀分布。实际上，这个设计就是基于公式： $x \bmod 2^n = x \& (2^n - 1)$ ，因为  $\&$  运算比  $\bmod$  具有更高的效率。

如下图，当  $n$  不为 2 的 N 次方时，hash 冲突的概率明显增大。

hash	0 1 0 1	hash	0 1 1 1
$n - 1$	0 1 1 1	$n - 1$	0 1 1 1
hash & ( $n - 1$ )	0 1 0 1	hash & ( $n - 1$ )	0 1 1 1
hash	0 1 0 1	hash	0 1 1 1
$n - 1$	0 1 0 1	$n - 1$	0 1 0 1
hash & ( $n - 1$ )	0 1 0 1	hash & ( $n - 1$ )	0 1 0 1

## 11、HashMap 的插入流程是怎么样的？

真香，建议收藏。



## 12、插入流程的图里刚开始有个计算 key 的 hash 值，是怎么设计的？

源码如下：拿到 key 的 hashCode，并将 hashCode 的高16位和 hashCode 进行异或（XOR）运算，得到最终的 hash 值。

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



### 13、为什么要将 hashCode 的高16位参与运算？

主要是为了在 table 的长度较小的时候，让高位也参与运算，并且不会有太大的开销。

例如下图，如果**高位不参与运算**，由于  $n - 1$  是 0000 0111，所以结果只取决于 hash 值的低3位，无论高位怎么变化，结果都是一样的。

hash = a.hashCode()	1	0	1	0	0	1	0	1
n - 1	0	0	0	0	0	1	1	1
<hr/>								
(n - 1) & hash	0	0	0	0	0	1	0	1

hash = b.hashCode()	1	1	1	0	0	1	0	1
n - 1	0	0	0	0	0	1	1	1
<hr/>								
(n - 1) & hash	0	0	0	0	0	1	0	1

如果我们将**高位参与运算**，则索引计算结果就不会仅取决于低位。

$h = a.hashCode()$	1	0	1	0	0	1	0	1				
$h >>> 4$	0	0	0	0	1	0	1	0	<del>0</del>	<del>1</del>	<del>0</del>	<del>1</del>
$hash = h \wedge (h >>> 4)$	1	0	1	0	1	1	1	1				
$n - 1$	0	0	0	0	0	1	1	1				
$(n - 1) \& hash$	0	0	0	0	0	1	1	1				

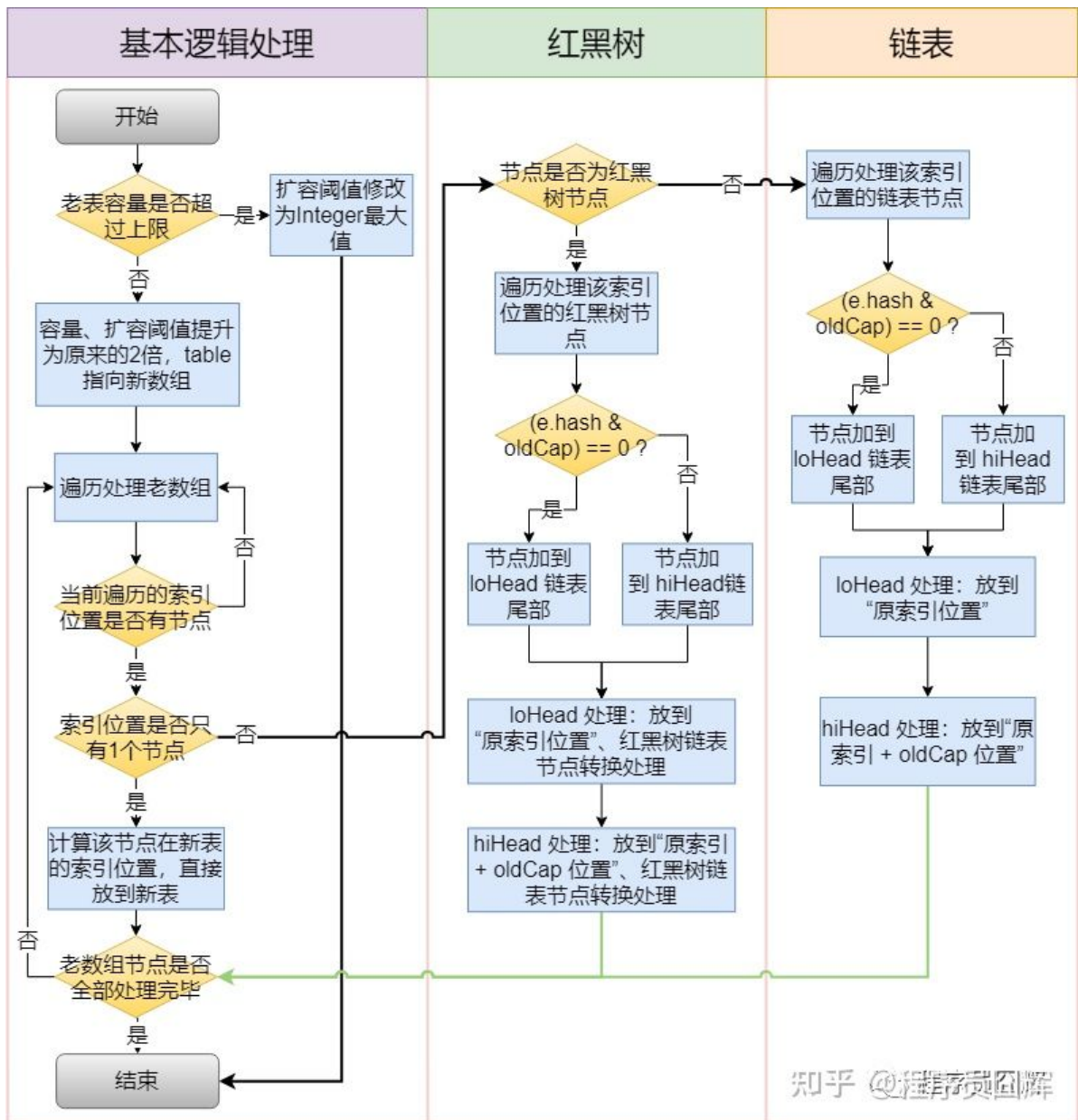
  

$h = b.hashCode()$	1	1	1	0	0	1	0	1				
$h >>> 4$	0	0	0	0	1	1	1	0	<del>0</del>	<del>1</del>	<del>0</del>	<del>1</del>
$hash = h \wedge (h >>> 4)$	1	1	1	0	1	0	1	1				
$n - 1$	0	0	0	0	0	1	1	1				
$(n - 1) \& hash$	0	0	0	0	0	0	1	1				

## 14、扩容 (resize) 流程介绍下?

真香，建议收藏。





## 15、红黑树和链表都是通过 `e.hash & oldCap == 0` 来定位在新表的索引位置，这是为什么？

请看对下面的例子。

扩容前 table 的容量为16，a 节点和 b 节点在扩容前处于同一索引位置。

计算在老表索引位置

a.hash	0 0 0 0 0 1 0 1
n - 1	0 0 0 0 1 1 1 1
<hr/>	
(n - 1) & a.hash	0 0 0 0 0 1 0 1
<hr/>	
b.hash	0 0 0 1 0 1 0 1
n - 1	0 0 0 0 1 1 1 1
<hr/>	
(n - 1) & b.hash	0 0 0 0 0 1 0 1

扩容后，table 长度为32，新表的 n - 1 只比老表的 n - 1 在高位多了一个1（图中标红）。

计算在新表索引位置

a.hash	0 0 0 0 0 1 0 1
n - 1	0 0 0 1 1 1 1 1
<hr/>	
(n - 1) & a.hash	0 0 0 0 0 1 0 1
<hr/>	
b.hash	0 0 0 1 0 1 0 1
n - 1	0 0 0 1 1 1 1 1
<hr/>	
(n - 1) & b.hash	0 0 0 1 0 1 0 1

因为 2 个节点在老表是同一个索引位置，因此计算新表的索引位置时，只取决于新表在高位多出来的这一位（图中标红），而这一位的值刚好等于 oldCap。

因为只取决于这一位，所以只会存在两种情况：1)  $(e.hash \& oldCap) == 0$ ，则新表索引位置为“原索引位置”；2)  $(e.hash \& oldCap) != 0$ ，则新表索引位置为“原索引 + oldCap 位置”。

## 16、HashMap 是线程安全的吗？

不是。HashMap 在并发下存在数据覆盖、遍历的同时进行修改会抛出 ConcurrentModificationException 异常等问题，JDK 1.8 之前还存在死循环问题。

## 17、介绍一下死循环问题？

导致死循环的根本原因是 JDK 1.7 扩容采用的是“头插法”，会导致同一索引位置的节点在扩容后顺序反掉，在并发插入触发扩容时形成环，从而产生死循环。

而 JDK 1.8 之后采用的是“尾插法”，扩容后节点顺序不会反掉，不存在死循环问题。

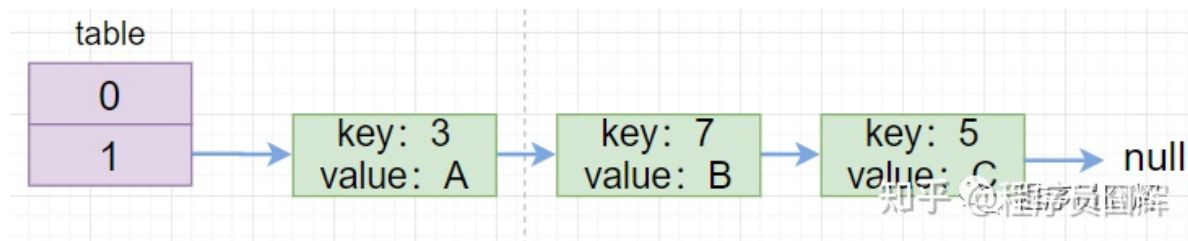
JDK 1.7.0 的扩容代码如下，用例子来看会好理解点。

```
void transfer(Entry[] newTable) {
    Entry[] src = table;
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            } while (e != null);
        }
    }
}
```

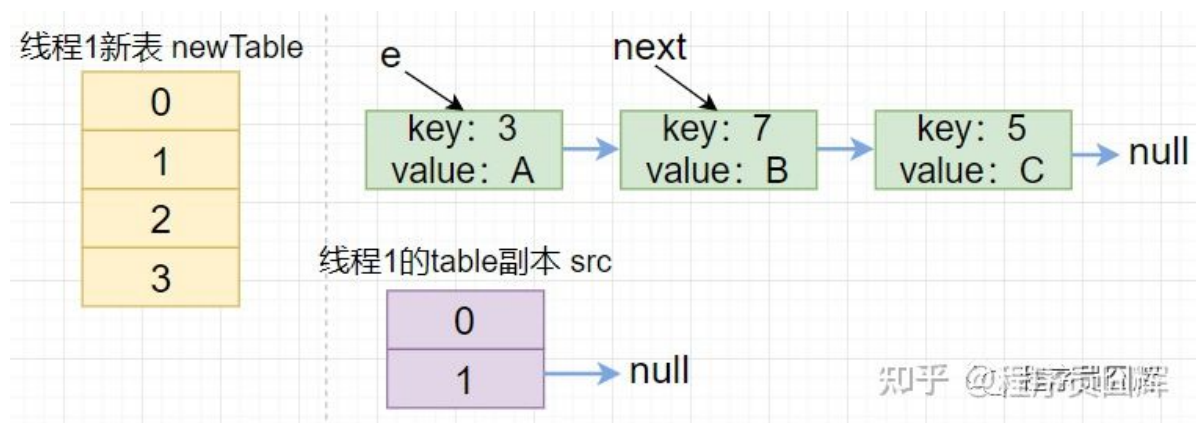
PS：这个流程较难理解，建议对着代码自己模拟走一遍。

例子：我们有 1 个容量为 2 的 HashMap，loadFactor=0.75，此时线程 1 和线程 2 同时往该 HashMap 插入一个数据，都触发了扩容流程，接着有以下流程。

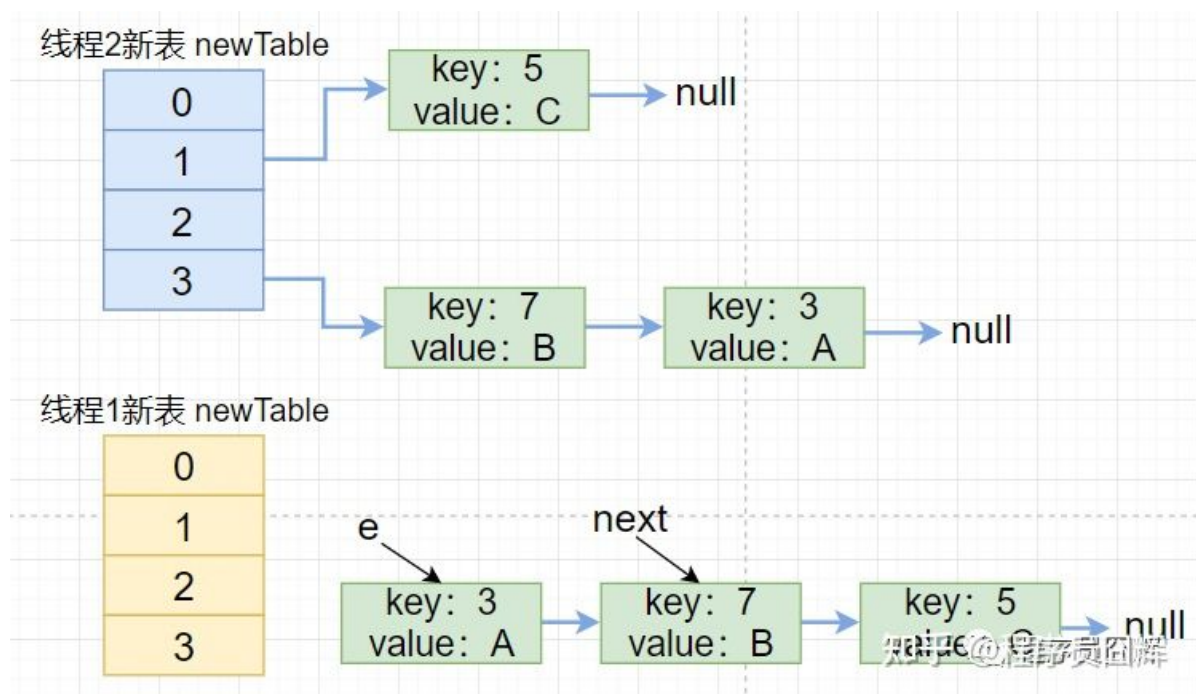
1) 在 2 个线程都插入节点，触发扩容流程之前，此时的结构如下图。



2) 线程1进行扩容，执行到代码：Entry<K,V> next = e.next 后被调度挂起，此时的结构如下图。

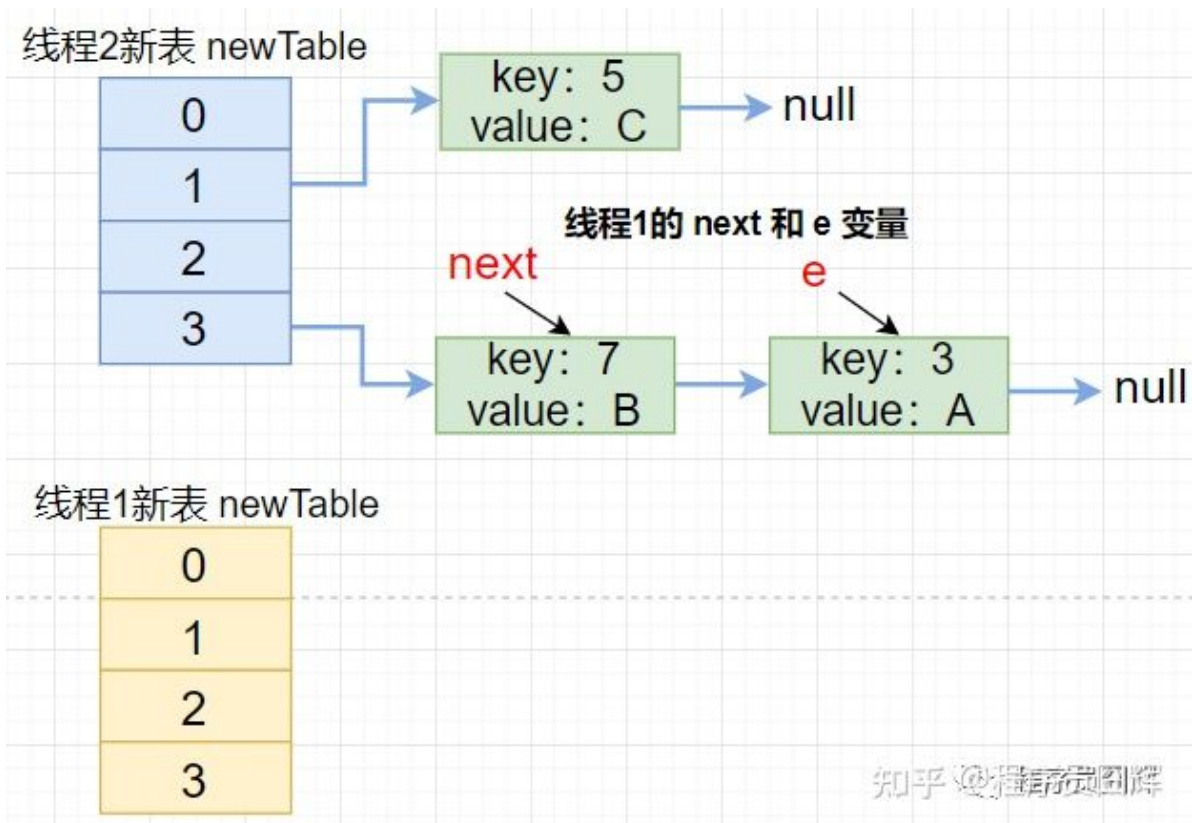


3) 线程1被挂起后，线程2进入扩容流程，并走完整个扩容流程，此时的结构如下图。

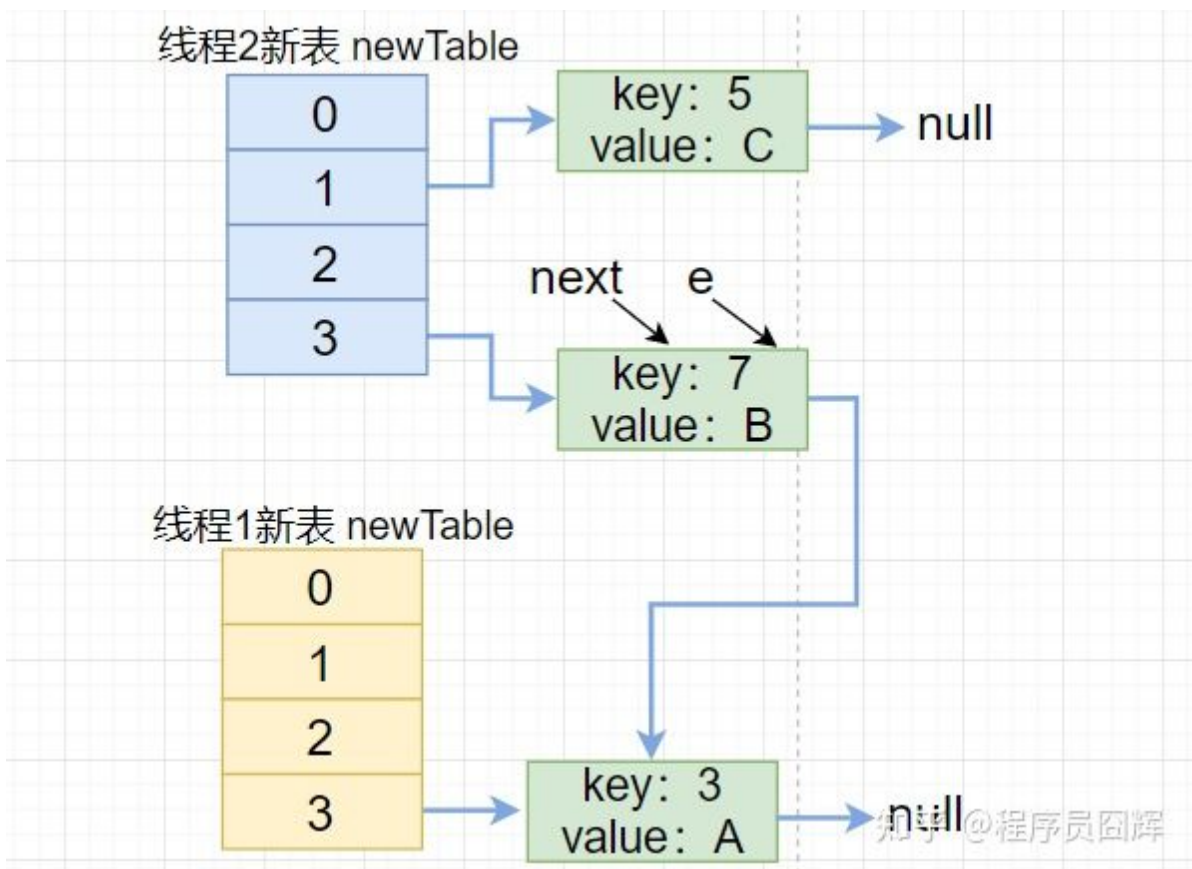


由于两个线程操作的是同一个 table，所以该图又可以画成如下图。





4) 线程1恢复后，继续走完第一次的循环流程，此时的结构如下图。



5) 线程1继续走完第二次循环，此时的结构如下图。





## 18、总结下 JDK 1.8 主要进行了哪些优化？

JDK 1.8 的主要优化刚才我们都聊过了，主要有以下几点：

- 1) 底层数据结构从“数组+链表”改成“数组+链表+红黑树”，主要是优化了 hash 冲突较严重时，链表过长的查找性能： $O(n) \rightarrow O(\log n)$ 。
- 2) 计算 table 初始容量的方式发生了改变，老的方式是从1开始不断向左进行移位运算，直到找到大于等于入参容量的值；新的方式则是通过“5个移位+或等于运算”来计算。

```
// JDK 1.7.0
public HashMap(int initialCapacity, float loadFactor) {
    // 省略
    // Find a power of 2 >= initialCapacity
    int capacity = 1;
    while (capacity < initialCapacity)
        capacity <<= 1;
    // ... 省略
}

// JDK 1.8.0_191
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

- 3) 优化了 hash 值的计算方式，老的通过一顿瞎JB操作，新的只是简单的让高16位参与了算。
- 4) 扩容时插入方式从“头插法”改成“尾插法”，避免了并发下的死循环。
- 5) 扩容时计算节点在新表的索引位置方式从“ $h \& (\text{length}-1)$ ”改成“ $\text{hash} \& \text{oldCap}$ ”，性能可能提升不大，但设计更巧妙、更优雅。

## 19、Hashtable 是怎么加锁的？

Hashtable 通过 synchronized 修饰方法来加锁，从而实现线程安全。

```
public synchronized V get(Object key) {
    // ...
}

public synchronized V put(K key, V value) {
    // ...
}
```

## 20、LinkedHashMap 和 TreeMap 排序的区别？

LinkedHashMap 和 TreeMap 都是提供了排序支持的 Map，区别在于支持的排序方式不同：

LinkedHashMap：保存了数据的插入顺序，也可以通过参数设置，保存数据的访问顺序。

TreeMap：底层是红黑树实现。可以指定比较器（Comparator 比较器），通过重写 compare 方法来定义排序；如果没有指定比较器，TreeMap 默认是按 Key 的升序排序（如果 key 没有实现 Comparable 接口，则会抛异常）。

## 21、HashMap 和 Hashtable 的区别？

HashMap 允许 key 和 value 为 null，Hashtable 不允许。

HashMap 的默认初始容量为 16，Hashtable 为 11。

HashMap 的扩容为原来的 2 倍，Hashtable 的扩容为原来的 2 倍加 1。

HashMap 是非线程安全的，Hashtable 是线程安全的，使用 synchronized 修饰方法实现线程安全。

HashMap 的 hash 值重新计算过，Hashtable 直接使用 hashCode。

HashMap 去掉了 Hashtable 中的 contains 方法。

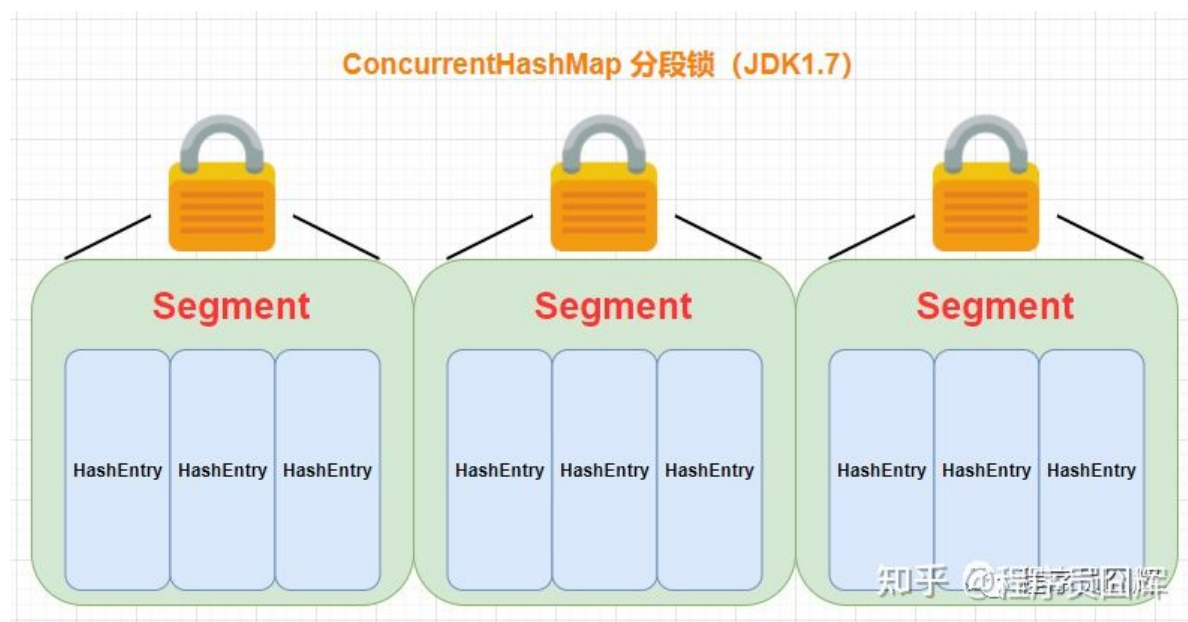
HashMap 继承自 AbstractMap 类，Hashtable 继承自 Dictionary 类。

HashMap 的性能比 Hashtable 高，因为 Hashtable 使用 synchronized 实现线程安全，还有就是 HashMap 1.8 之后底层数据结构优化成“数组+链表+红黑树”，在极端情况下也能提升性能。

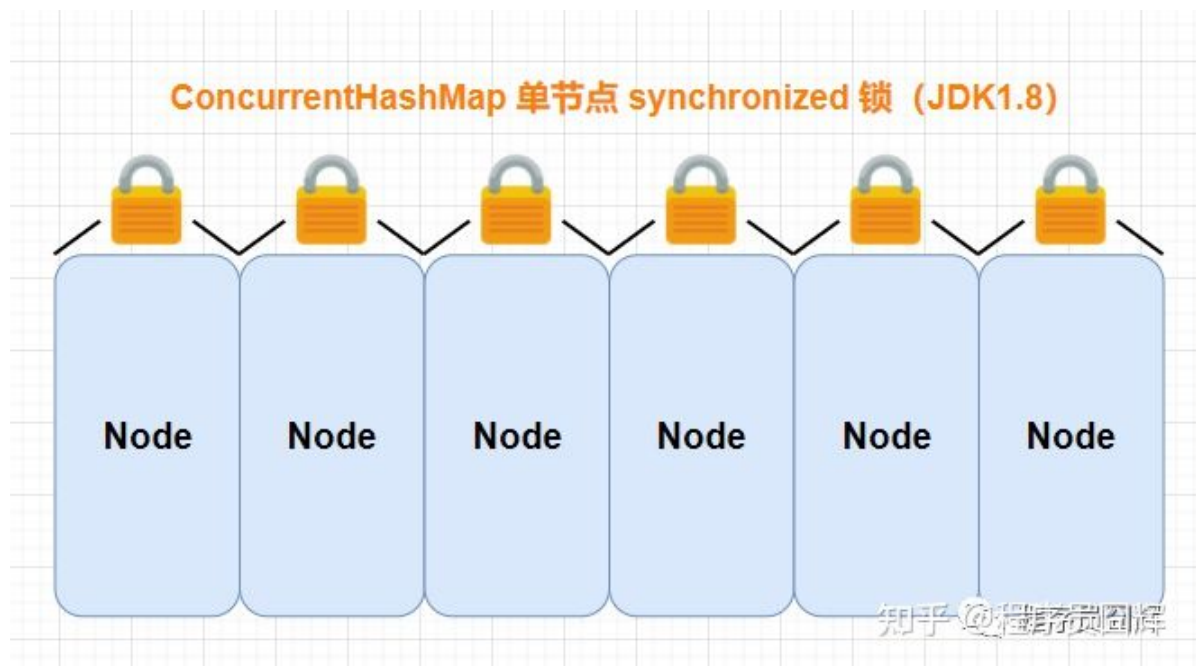
## 22、介绍下 ConcurrentHashMap，要讲出 1.7 和 1.8 的区别？

ConcurrentHashMap 是 HashMap 的线程安全版本，和 HashMap 一样，在 JDK 1.8 中进行了较大的优化。

JDK1.7：底层结构为：分段的数组+链表；实现线程安全的方式：分段锁（Segment，继承了 ReentrantLock），如下图所示。



JDK1.8：底层结构为：数组+链表+红黑树；实现线程安全的方式：CAS + Synchronized



区别：

- 1、JDK1.8 中降低锁的粒度。JDK1.7 版本锁的粒度是基于 Segment 的，包含多个节点（HashEntry），而 JDK1.8 锁的粒度就是单节点（Node）。
- 2、JDK1.8 版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用 synchronized 来进行同步，所以不需要分段锁的概念，也就不需要 Segment 这种数据结构了，当前还保留仅为了兼容。
- 3、JDK1.8 使用红黑树来优化链表，跟 HashMap 一样，优化了极端情况下，链表过长带来的性能问题。
- 4、JDK1.8 使用内置锁 synchronized 来代替重入锁 ReentrantLock，synchronized 是官方一直在不断优化的，现在性能已经比较可观，也是官方推荐使用的加锁方式。

## 23、ConcurrentHashMap 的并发扩容

ConcurrentHashMap 在扩容时会计算出一个步长（stride），最小值是16，然后给当前扩容线程分配“一个步长”的节点数，例如16个，让该线程去对这16个节点进行扩容操作（将节点从老表移动到新表）。

如果在扩容结束前又来一个线程，则也会给该线程分配一个步长的节点数让该线程去扩容。依次类推，以达到多线程并发扩容的效果。

例如：64要扩容到128，步长为16，则第一个线程会负责第113（索引112）~128（索引127）的节点，第二个线程会负责第97（索引96）~112（索引111）的节点，依次类推。

具体处理（该流程后续可能会替换成流程图）：

- 1) 如果索引位置上为null，则直接使用 CAS 将索引位置赋值为 ForwardingNode（hash值为-1），表示已经处理过，这个也是触发并发扩容的关键点。
- 2) 如果索引位置的节点 f 的 hash 值为 MOVED（-1），则代表节点 f 是 ForwardingNode 节点，只有 ForwardingNode 的 hash 值为 -1，意味着该节点已经处理过了，则跳过该节点继续往下处理。

3) .否则，对索引位置的节点 f 对象使用 synchronized 进行加锁，遍历链表或红黑树，如果找到 key 和入参相同的，则替换掉 value 值；如果没找到，则新增一个节点。如果是链表，同时判断是否需要转红黑树。处理完在索引位置的节点后，会将该索引位置赋值为 ForwardingNode，表示该位置已经处理过。

**ForwardingNode**：一个特殊的 Node 节点，hash 值为-1（源码中定义成 MOVED），其中存储 nextTable 的引用。只有发生扩容的时候，ForwardingNode才会发挥作用，作为一个占位符放在 table 中表示当前节点已经被处理（或则为 null）。

## 24、ConcurrentHashMap 和 Hashtable 的区别？

### 1) 底层数据结构：

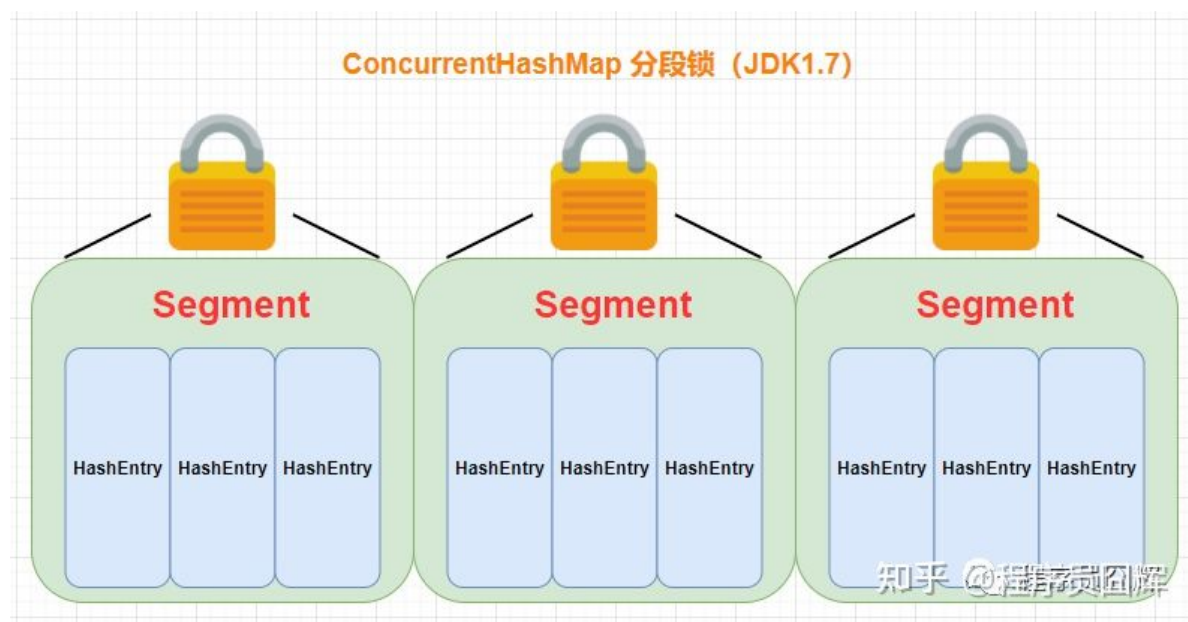
ConcurrentHashMap：1) JDK1.7 采用 分段的数组+链表 实现；2) JDK1.8 采用 数组+链表+红黑树，跟 JDK1.8 的 HashMap 的底层数据结构一样。

Hashtable：采用 数组+链表 的形式，跟 JDK1.8 之前的 HashMap 的底层数据结构类似。

### 2) 实现线程安全的方式（重要）：

ConcurrentHashMap：

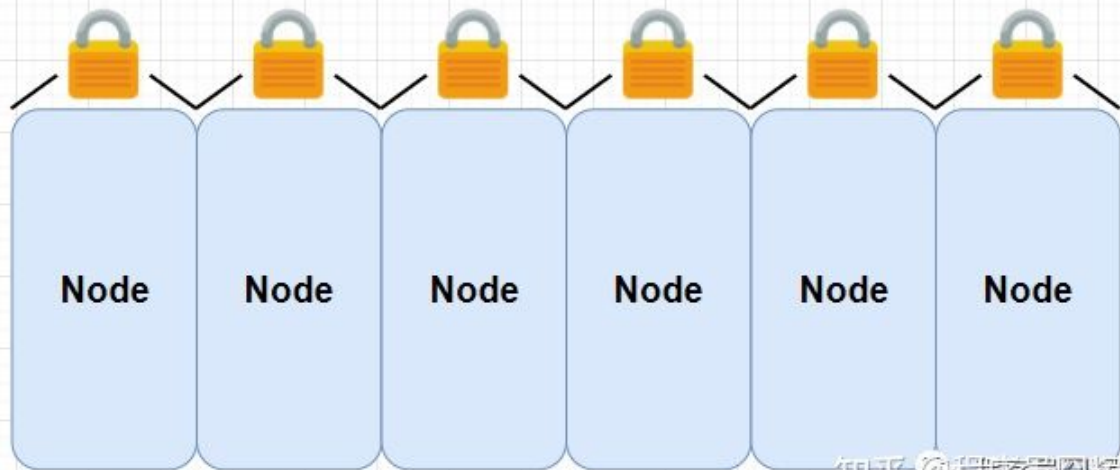
1) JDK1.7：使用分段锁（Segment）保证线程安全，每个分段（Segment）包含若干个 HashEntry，当并发访问不同分段的数据时，不会产生锁竞争，从而提升并发性能。



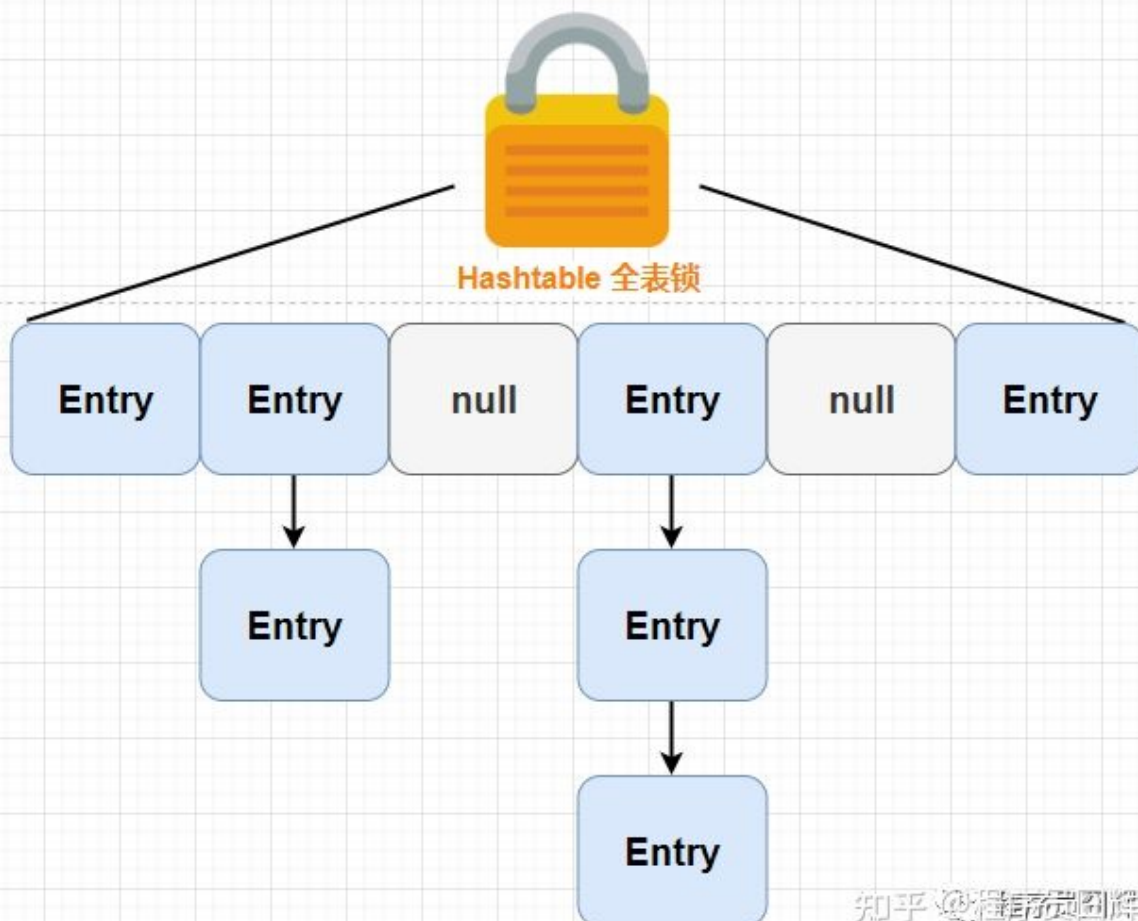
2) JDK1.8：使用 synchronized + CAS 的方式保证线程安全，每次只锁一个节点（Node），进一步降低锁粒度，降低锁冲突的概率，从而提升并发性能。



### ConcurrentHashMap 单节点 synchronized 锁 (JDK1.8)



Hashtable: 使用 synchronized 修饰方法来保证线程安全，每个实例对象只有一把锁，并发性能较低，相当于串行访问。



## 25、ConcurrentHashMap 的 size() 方法怎么实现的？

JDK 1.7：先尝试在不加锁的情况下尝试进行统计 size，最多统计3次，如果连续两次统计之间没有任何对 segment 的修改操作，则返回统计结果。否则，对每个segment 进行加锁，然后统计出结果，返回结果。

JDK 1.8：直接统计 baseCount 和 counterCells 的 value 值，返回的是一个近似值，如果有并发的插入或删除，实际的数量可能会有所不同。

该统计方式改编自 LongAdder 和 Striped64，这两个类在 JDK 1.8 中被引入，出自并发大神 Doug Lea 之手，是原子类（AtomicLong 等）的优化版本，主要优化了在并发竞争下，AtomicLong 由于 CAS 失败的带来的性能损耗。

值得注意的是，JDK1.8中，提供了另一个统计的方法 mappingCount，实现和 size 一样，只是返回的类型改成了 long，这也是官方推荐的方式。

```
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}
// 一个ConcurrentHashMap包含的映射数量可能超过int上限，
// 所以应该使用这个方法代替size()
public long mappingCount() {
    long n = sumCount();
    return (n < 0L) ? 0L : n; // ignore transient negative values
}
final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}
```

## 26、比较下常见的几种 Map，在使用时怎么选择？

类	介绍	使用场景
Hashtable	早期的线程安全 Map，直接通过在方法加 synchronized 实现线程安全	现在理论上不会使用
Cocurrent HashMap	线程安全的 Map，通过 synchronized + CAS 实现线程安全	需要保证线程安全
LinkedHashMap	能记录访问顺序或插入顺序的 Map，通过 head、tail 属性维护有序双向链表，通过 Entry 的 after、before 属性维护节点的顺序	需要记录访问顺序或插入顺序
TreeMap	通过实现 Comparator 实现自定义顺序的 Map，如果没有指定 Comparator 则会按 key 的升序排序，key 如果没有实现 Comparable 接口，则会抛异常	需要自定义排序
HashMap	最通用的 Map，非线程安全、无序	无特殊需求都可使用

知乎 @程序员画解

## 30、ArrayList 和 Vector 的区别。

Vector 和 ArrayList 的实现几乎是一样的，区别在于：

- 1) 最重要的区别：Vector 在方法上使用了 synchronized 来保证线程安全，同时由于这个原因，在性能上 ArrayList 会有更好的表现。
- 2) Vector 扩容后容量默认变为原来 2 倍，而 ArrayList 为原来的 1.5 倍。

有类似关系的还有：StringBuilder 和 StringBuffer、HashMap 和 Hashtable。

## 31、ArrayList 和 LinkedList 的区别？

- 1、ArrayList 底层基于动态数组实现，LinkedList 底层基于双向链表实现。
- 2、对于随机访问（按 index 访问，get/set 方法）：ArrayList 通过 index 直接定位到数组对应位置的节点，而 LinkedList 需要从头结点或尾节点开始遍历，直到寻找到目标节点，因此在效率上 ArrayList 优于 LinkedList。
- 3、对于随机插入和删除：ArrayList 需要移动目标节点后面的节点（使用 System.arraycopy 方法移动节点），而 LinkedList 只需修改目标节点前后节点的 next 或 prev 属性即可，因此在效率上 LinkedList 优于 ArrayList。

4、对于顺序插入和删除：由于 ArrayList 不需要移动节点，因此在效率上比 LinkedList 更好。这也是为什么在实际使用中 ArrayList 更多，因为大部分情况下我们的使用都是顺序插入。

5、两者都不是线程安全的。

6、内存空间占用：ArrayList 的空间浪费主要体现在在 list 列表的结尾会预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和直接前驱以及数据）。

## 32、HashSet 是如何保证不重复的？

HashSet 底层使用 HashMap 来实现，见下面的源码，元素放在 HashMap 的 key 里，value 为固定的 Object 对象。当 add 时调用 HashMap 的 put 方法，如果元素不存在，则返回 null 表示 add 成功，否则 add 失败。

由于 HashMap 的 Key 值本身就不允许重复，HashSet 正好利用 HashMap 中 key 不重复的特性来校验重复元素，简直太妙了。

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

## 33、TreeSet 清楚吗？能详细说下吗？

“TreeSet 和 TreeMap 的关系”和上面说的“HashSet 和 HashMap 的关系”几乎一致。

TreeSet 底层默认使用 TreeMap 来实现。而 TreeMap 通过实现 Comparator（或 Key 实现 Comparable）来实现自定义顺序。

```
private transient NavigableMap<E, Object> m;
private static final Object PRESENT = new Object();

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

public TreeSet() {
    this(new TreeMap<E, Object>());
}

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

## 34、介绍下 CopyOnWriteArrayList？

CopyOnWriteArrayList 是 ArrayList 的线程安全版本，也是大名鼎鼎的 copy-on-write（COW，写时复制）的一种实现。

在读操作时不加锁，跟 ArrayList 类似；在写操作时，复制出一个新的数组，在新数组上进行操作，操作完了，将底层数组指针指向新数组。适合使用在读多写少的场景。

例如 add(E e) 方法的操作流程如下：使用 ReentrantLock 加锁，拿到原数组的length，使用 Arrays.copyOf 方法从原数组复制一个新的数组（length+1），将要添加的元素放到新数组的下标 length位置，最后将底层数组指针指向新数组。

## 35、Comparable 和 Comparator 比较？

- 1、Comparable 是排序接口，一个类实现了 Comparable接口，意味着“该类支持排序”。Comparator 是比较器，我们可以实现该接口，自定义比较算法，创建一个“该类的比较器”来进行排序。
- 2、Comparable 相当于“内部比较器”，而 Comparator 相当于“外部比较器”。
- 3、Comparable 的耦合性更强，Comparator 的灵活性和扩展性更优。
- 4、Comparable 可以用作类的默认排序方法，而 Comparator 则用于默认排序不满足时，提供自定义排序。

耦合性和扩展性的问题，举个简单的例子：

当实现类实现了 Comparable 接口，但是已有的 compareTo 方法的比较算法不满足当前需求，此时如果想对两个类进行比较，有两种办法：

- 1) 修改实现类的源代码，修改 compareTo 方法，但是这明显不是一个好方案，因为这个实现类的默认比较算法可能已经在其他地方使用了，此时如果修改可能会造成影响，所以一般不会这么做。
- 2) 实现 Comparator 接口，自定义一个比较器，该方案会更优，自定义的比较器只用于当前逻辑，其他已有的逻辑不受影响。

## 36、List、Set、Map三者的区别？

List（对付顺序的好帮手）：存储的对象是可重复的、有序的。

Set（注重独一无二的性质）：存储的对象是不可重复的、无序的。

Map（用 Key 来搜索的专业户）：存储键值对（key-value），不能包含重复的键（key），每个键只能映射到一个值。

## 37、Map、List、Set 分别说下你了解到它们有的线程安全类和线程不安全的类？

### Map

线程安全：ConcurrentHashMap、Hashtable

线程不安全：HashMap、LinkedHashMap、TreeMap、WeakHashMap

### List

线程安全：Vector（线程安全版的ArrayList）、Stack（继承Vector，增加pop、push方法）、CopyOnWriteArrayList

线程不安全：ArrayList、LinkedList



## Set

线程安全：CopyOnWriteArraySet（底层使用CopyOnWriteArrayList，通过在插入前判断是否存在实现Set不重复的效果）

线程不安全：HashSet（基于HashMap）、LinkedHashSet（基于LinkedHashMap）、TreeSet（基于TreeMap）、EnumSet

## 38、Collection 与 Collections的区别

Collection：集合类的一个顶级接口，提供了对集合对象进行基本操作的通用接口方法。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，常见的List与Set就是直接继承Collection接口。

Collections：集合类的一个工具类/帮助类，提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。