【哈希表和 Java 的前世今生】

主要内容

1. 开篇
2. 哈希表原理
3. JDK7 HashMap
4. JDK8 HashMap
5. Hashtable
6. JDK7 ConcurrentHashMap
7. JDK8 ConcurrentHashMap
8. 最后的总结和交流

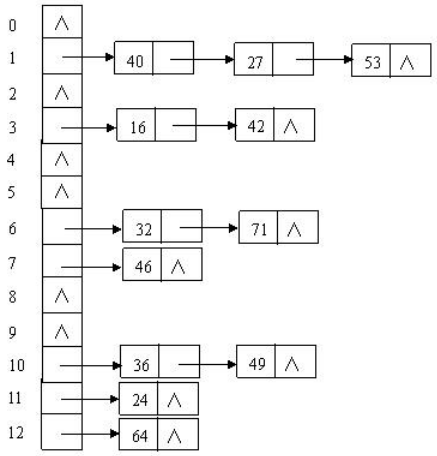
学习目标

知识点	要求
哈希表原理	理解
JDK7 HashMap	掌握
JDK8 HashMap	理解
Hashtable	了解
JDK7 ConcurrentHashMap	理解
JDK8 ConcurrentHashMap	掌握

一、开篇

1.1 问题 1：哈希表和 Java 的前世今生是什么

哈希表（hashtable 散列表）是一种数据结构，是一种神奇的数据结构，查询、添加、删除效率非常快，时间复杂度可以达到 O（1）。



Java 的集合中给出了底层结构采用哈希表数据结构的实现类，按照时间顺序分别为第一代 Hashtable、第二代 HashMap、第三代 ConcurrentHashMap（concurrent 并发）。相同点：底层结构都是哈希表，都是用来存储 key-value 映射，都实现了 Map 接口。

第一代	Hashtable	前世	线程安全	JDK1.0				
第二代	HashMap	今生 1	线程不安全		JDK1.2	JDK5	JDK7	JDK8
第三代	ConcurrentHashMap	今生 2	线程安全			JDK5	JDK7	JDK8

1) Hashtable 线程安全，但是效率太低，底层使用 synchronized 同步方法，已不再使用。
2) HashMap 线程不安全，效率提升，适用单线程情况下。可以借助 Collections.synchronziedMap()保证线程安全，底层使用 synchronized 同步代码块，效率比 Hashtable 高。
3) 在大量并发情况下如何提高集合的效率和安全呢？ ConcurrentHashMap：JDK7 底层采用 Lock 锁，但是 JDK8 的 ConcurrentHashMap 不使用 Lock 锁，而是使用了 CAS + synchronized 代码块锁。保证安全的同时，性能均也很高。

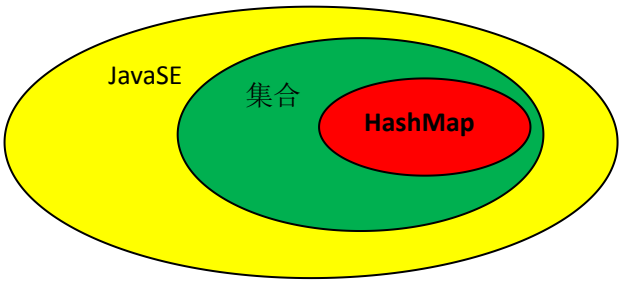
注意：

- ConcurrentHashMap 推出后，HashMap 并未过时，适用不同场景。两者同时进行性能提高和结构完善。
- 和三种线程同步技术的发展密切相关
- Hashtable 不可 null key-value，HashMap 可以，ConcurrentHashMap 不可

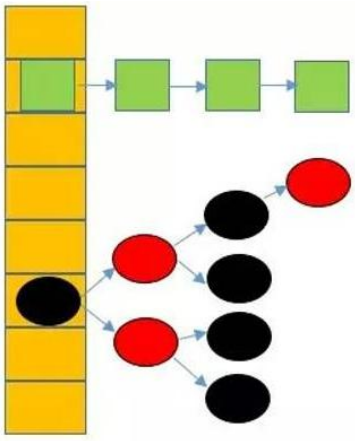
1.2 问题 2：为什么选择这个题目

这是一个重要技能点、更是一个异常重要的面试点，不管是大企业还是小企业，尤其是 BAT。

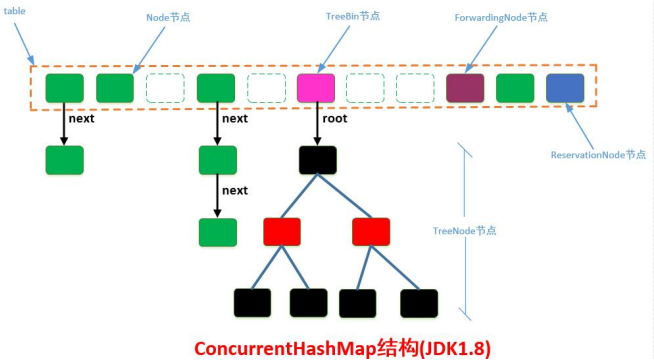
- **重点：**JavaSE 是 Java 的技术基础，集合是 JavaSE 的重点，HashMap 是集合的重点



- **难点**：综合了数组、链表、红黑树等多种数据结构。其设计和实现充满着哲学、智慧的光芒



- **新点**：HashMap 在 7，8 中有较大变化；ConcurrentHashMap 是新一代并发集合类，在 JDK7、JDK8 中也有很大的变化



ConcurrentHashMap结构(JDK1.8)

- **综合点**：单独这一个技能点就可以聊上 1 小时，并且还可以辐射到其他集合类、数据结构、多线程等多个技能点

从这个题目，可以看出你的技术功底有多厚，能够看到你对技术有多执着，对新知识有多敏锐；争取在这个问题上，或在该问题的某个细节上能够干过面试官！！大大加分的题目！！

1.3 问题 3：如何保证今天公开课的效果

参加今天公开课的同学技术水平不一，我的讲课争取让所有同学都有收获

- ❑ 先讲明白特征、流程、变化、结论，再看具体源码的实现
- ❑ 充分利用文档，按部就班。不仅听，而且看。课下有文档可以复习
- ❑ 化大为小、分为 16+7=23 个问题各个击破
- ❑ 多加交互。几个问题结束可以简单沟通一下，看看理解情况
- ❑ 条理清晰，抽丝剥茧、步步推进，争取让大家一次听明白。

1.4 问题 4：分成的 23 小问题是什么

1. 哈希表的由来，她究竟解决了什么问题，为什么神奇。
2. 哈希表的原理（结构、添加步骤、查询步骤）

3. JDK7 中 HashMap 的关键点
   1) 为什么要把 hash 也放到 Entity 中
   2) 第一步为什么还要多次散列，为什么这样实现
   3) 为什么求索引不使用 h%length，而是使用 h&(length-1)
   4) 为什么主数组的长度必须是 2 的幂
   5) 为什么加载因子选择 0.75
   6) JDK7 的死循环问题（并不是死锁）
   7) 多线程 put 的时候为什么可能导致元素丢失
4. JDK8 中 HashMap 的变化
5. JDK8 HashMap 为什么是当链表长度>=8 后变成红黑树，而不是其他值

6. Hashtable 和 HashMap 的不同之处
7. Hashtable 的缺点
8. 为什么 Hashtable 主数组默认长度是 11，为何扩容 2 倍还要+1

9. JDK7 ConcurrentHashMap 关键技能点
10. JDK7 ConcurrentHashMap 通过无参构造方法创建对象的结果
11. Unsafe 类是怎么回事
12. JDK7 ConcurrentHashMap 的缺点是什么
13. JDK8 中 ConcurrentHashMap 变化
14. JDK8 ConcurrentHashMap 怎么放弃 Lock 使用 synchronized 了
15. JDK8 中 ConcurrentHashMap 的 sizeCtl 属性的作用
16. 折腾什么？Hashtable 不可存储 null key-value，HashMap 可以，ConcurrentHashMap 不可

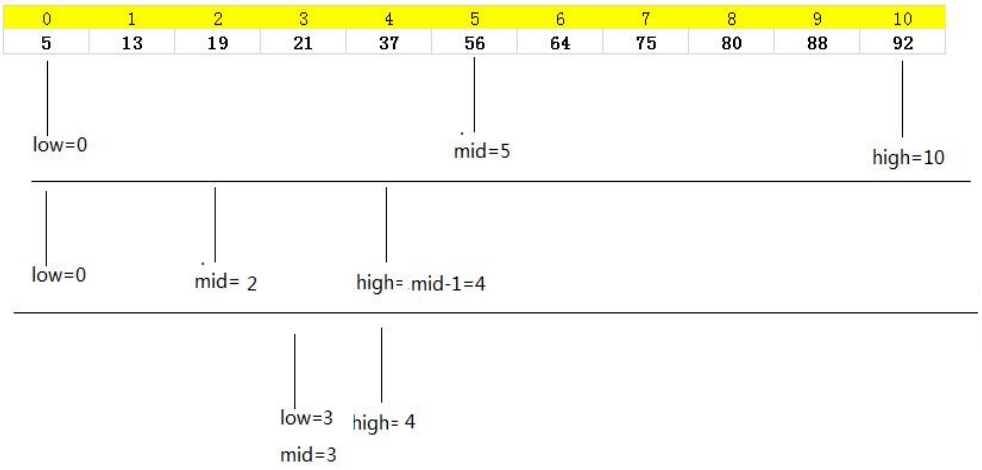
二、哈希表的原理

2.1 问题 1：哈希表的由来，她究竟解决了什么问题，为什么神奇

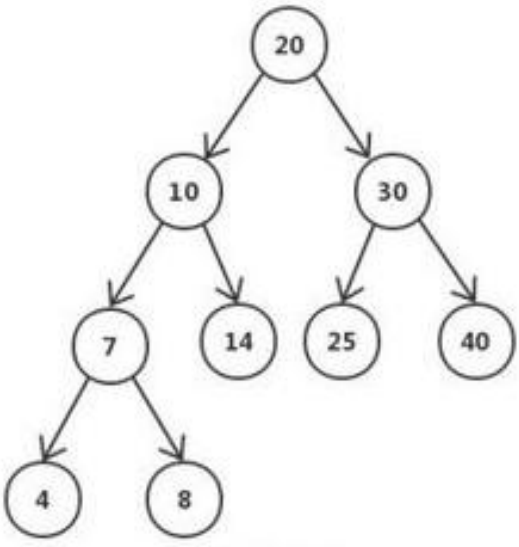
如何解决数据查询、添加、删除等效率呢？此处以查询为例进行说明。

- 在无序数组中按照内容查找，效率低下，时间复杂度是 O（n）

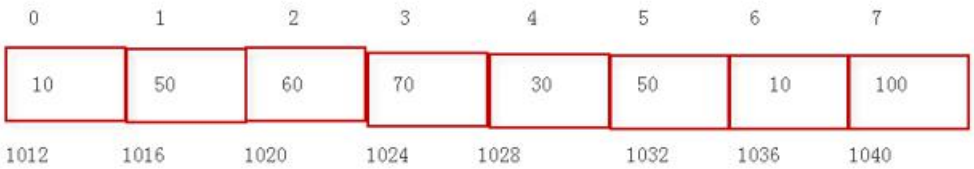
- 在有序数组中按照内容查找，可以使用折半查找，时间复杂度 O（log_2n）



- 在二叉平衡树中按照内容查找，时间复杂度 O（log_2n）



- 在数组中按照**索引**查找，不进行比较和计数，直接计算得到，效率最高，时间复杂度 O（1）



索引是i的元素的地址 = 起始地址+每个元素大小*索引
arr[4] = 1012 +4*4 = 1012+16 = 1028
arr[0] = 1012+ 4*0 = 1012

问题：按照内容查找，能否也不进行比较，而是通过计算得到地址，实现类似数组按照 索引查询的高效率呢 O（1）

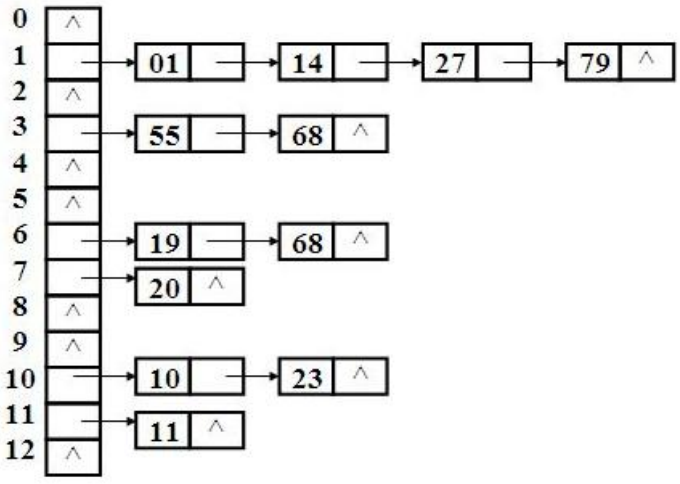
有！！！哈希表来实现

前面查找方法共同特点：通过将关键字值与给定值比较，来确定位置。效率取决比较次数。
理想的方法是：不需要比较，根据给定值能直接定位记录的存储位置。
这样，需要在记录的存储位置与该记录的关键字之间建立一种确定的对应关系，使每个记录的关键字与一个存储位置相对应。

2.2 问题 2：哈希表的原理（结构、添加步骤、查询步骤）

1. 哈希表的结构和特点

- hashtable 也叫散列表
- 特点：快 很快 神奇的快
- 结构：结构有多种
  - 最流行、最容易理解：顺序表+链表
  - 主结构：顺序表
  - 每个顺序表的节点在单独引出一个链表



桶 bucket ：每个元素后面可以拉一个链表，成为一个桶。

bucketIndex：桶索引。数组元素的索引。

2. 哈希表是如何添加数据的

1.计算哈希码(调用 hashCode(),结果是一个 int 值，整数的哈希码取自身即可)

2.计算在哈希表中的存储位置 y=k(x)=x%11

x:哈希码 k(x) 函数 y：在哈希表中的存储位置 （位置就是数组的索引）

3.存入哈希表

- 情况 1：一次添加成功
- 情况 2：多次添加成功（出现了冲突（碰撞 collision），调用 equals()和对应链表的元素进行比较，比较到最后，结果都是 false，创建新节点，存储数据，并加入链表）
- 情况 3：不添加（出现了冲突，调用 equals()和对应链表的元素进行比较， 经过一次或者多次比较后，结果是 true，表明重复，不添加）

结论 1：哈希表添加数据快（3 步即可，不考虑冲突）

结论 2：唯一

结论 3：无序

3. 哈希表是如何查询数据的

和添加数据的过程是相同的

- 情况 1：一次找到 23 86 76
- 情况 2：多次找到 67 56 78
- 情况 3：找不到 100 200

结论 1：哈希表查询数据快

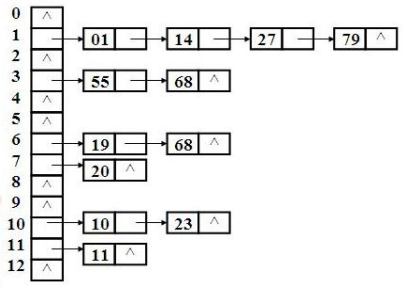
三、JDK7 HashMap

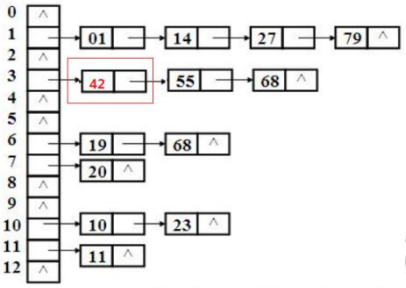
3.1 问题 3：JDK7 中 HashMap 的关键点

- JDK7 及之前，HashMap 底层是一个 table 数组+链表的哈希表存储结构
- 链表上每个节点的就是一个 Entry，字段包括四部分

hash（哈希码）	key（键）	value（值）	next（指向下一个 Entry 节点）

- 默认主数组长度 16；
- 主数组的长度可以直接指定，但最终长度会变为刚刚大于指定值的 2 的幂。
- 默认装填因子 0.75（元素个数达到主数组长度 75%时扩容）
- 每次主数组扩容为原来的 2 倍





- 发生冲突，经过比较不存在相同 key 的元素，要添加一个新的节点。不是加到链表最后，而是添加到链表最前
- 发生冲突，经过比较存在相同 key 的元素，使用新的 value 替换旧的 value，并返回旧的 value。

3.2 源码阅读

- 如果 key 是 null，直接存入到索引是 0 的桶中。不进行第一步和第二步操作。

```
if (key == null)
    return putForNullKey(value);
```

- 第一步计算哈希码，不仅调用了 hashCode()，有进行了多次散列。目的在于 key 不同，哈希码尽量不同，减少冲突

```
final int hash(Object k) {
    int h = 0;
    if (useAltHashing) {
        if (k instanceof String) {
            return sun.misc.Hashing.stringHash32((String) k);
        }
        h = hashSeed;
    }

    h ^= k.hashCode();

    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

极端情况下，对 String 的哈希值的判断，采用 JDK 内部的特殊算法计算。了解即可。

- 第二步，计算存储位置，使用了位运算来提高效率

```
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

- 第三步判断 key 是否存在的条件是比较哈希码 && 内容。其实直接调用 equals()即可，这个条件是为了提高效率。

```
if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
    V oldValue = e.value;
    e.value = value;
    e.recordAccess( m: this);
    return oldValue;
}
```

- 扩容的条件是：节点数量达到阈值 && 新元素的位置已经有节点

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize( newCapacity: 2 * table.length);
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = indexFor(hash, table.length);
    }

    createEntry(hash, key, value, bucketIndex);
}
```

尽量减少扩容的次数，因为扩容会导致原来的节点要重新散列到新数组的位置。如果能够预估到节点的数量，可以直接指定哈希表主数组的长度。

- 真正的扩容是有 transfer()实现的。扩容需要重新创建一个新的哈希表（主数组），原来的节点 Entry 都要重新计算存储位置并添加到新哈希表中。

```
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];
    boolean oldAltHashing = useAltHashing;
    useAltHashing |= sun.misc.VM.isBooted() &&
            (newCapacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);
    boolean rehash = oldAltHashing ^ useAltHashing;
    transfer(newTable, rehash);
    table = newTable;
    threshold = (int)Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1);
}
```

- 发生冲突，经过比较不存在相同 key 的元素，要添加一个新的节点。不是加到链表最后，而是添加到链表最前

```
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++;
}
```

3.3 更多的细节问题

- 问题 3-1：为什么要把 hash 也放到 Entity 中
  - 扩容时不用重新计算 hash，省去第一步，直接使用来计算存储位置即可
  - 判断 key 是否存在时，可以先判断 hash，只是一个整数，效率高。Hash 不同，直接短路，提高效率。

- 问题 3-2：第一步为什么还要多次散列，为什么这样实现

```
// This function ensures that hashCodes that differ only by
// constant multiples at each bit position have a bounded
// number of collisions (approximately 8 at default load factor).
h ^= (h >>> 20) ^ (h >>> 12);
return h ^ (h >>> 7) ^ (h >>> 4);
```

目的：保证高低 bit 位都能参与到 Hash 的计算中，一句话就是为了减少 hash 冲突的几率。保证在默认的加载因子下，每个链表的长度一般不会超过 8.

- 问题 3-3：为什么求索引不使用 h%length，而是使用 h&(length-1)

```
/**
 * Returns index for hash code h.
 */
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

使用位运算可以提升效率。直接取模，如果 h 是负数，计算的索引会是负数

- 问题 3-4：为什么主数组的长度必须是 2 的幂；

因为计算存储位置的公式：h&(length-1)。如果主数组的长度必须是 2 的幂，该表达式实现不了取模的效果。

length	2	4	8	16	32	64	128
length-1	1	3	7	15	31	63	127
length	10	100	1000	10000	100000	1000000	10000000
length-1	1	11	111	1111	11111	111111	1111111

保证 length-1 的二进制的低 x 位都是 1，进行&运算。因为 0&1=0，1&1=1，可以让 h 的低 x 位值完整保存下来，正好作为余数，还不冲突。

如果 length 不是 2 的幂，使用该表达式会导致不同数据产生相同哈希码，进而导致地址冲突。

```
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);

    // Find a power of 2 >= initialCapacity
    int capacity = 1;
    while (capacity < initialCapacity)
        capacity <<= 1;

    this.loadFactor = loadFactor;
    threshold = (int)Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
    table = new Entry[capacity];
    useAltHashing = sun.misc.VM.isBooted() &&
            (capacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);
    init();
}
```

❑ 问题 3-5：为什么加载因子选择 0.75

```
* <p>As a general rule, the default load factor (.75) offers a good tradeoff
* between time and space costs.  Higher values decrease the space overhead
* but increase the lookup cost (reflected in most of the operations of the
* <tt>HashMap</tt> class, including <tt>get</tt> and <tt>put</tt>).  The
* expected number of entries in the map and its load factor should be taken
* into account when setting its initial capacity, so as to minimize the
* number of rehash operations.  If the initial capacity is greater
* than the maximum number of entries divided by the load factor, no
* rehash operations will ever occur.
```

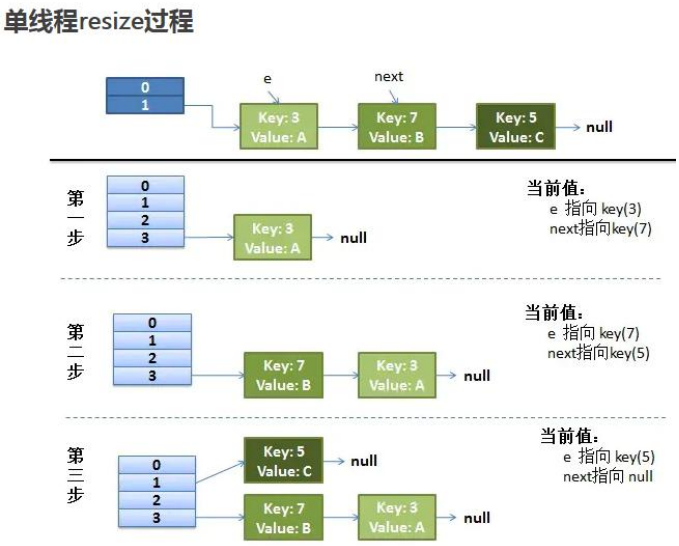
通常，默认负载因子（0.75）在时间和空间成本之间提供了一个很好的权衡。 较高的值会减少空间开销，但会增加查找成本。 设置其初始容量时，应考虑映射中的预期条目数及其负载因子，以最大程度地减少重新哈希操作的数量。 如果初始容量大于最大条目数除以负载因子，则不会发生任何哈希操作。

❑ 问题 3-6：JDK7 的死循环问题（并不是死锁）。

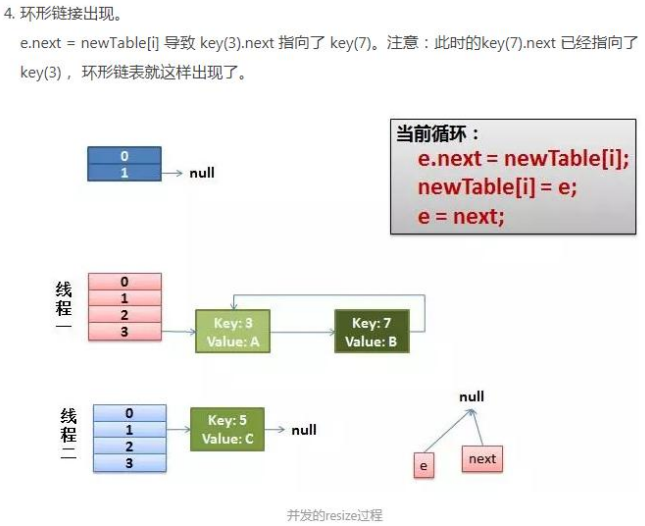
这也是为什么 JDK8 对 HashMap 进行大手术的原因所在。问题出在 HashMap 扩容时。扩容对用户是透明的，却是一切坑的原因。

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

因为 JDK7 的新节点是添加到链表头部，导致重新散列后，链表的节点顺序会颠倒。如果是单线程情况下，这不算问题。



多线程扩容时，同时执行 transfer 方法，可能扩容后形成循环列表。如果下步要查询 get()一个不存在的 key，或者 put 不存在的 key，悲剧出现了——Infinite Loop。



参考：https://www.jianshu.com/p/e1c020d37c6a

❑ 问题 3-7 多线程 put 的时候为什么可能导致元素丢失

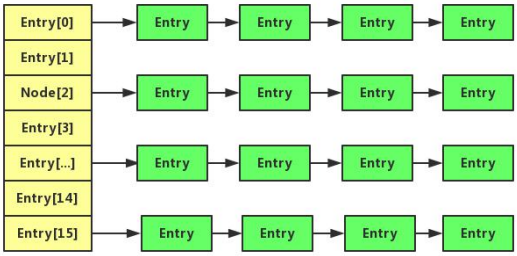
主要问题出在 addEntry 方法的 new Entry (hash, key, value, e)，如两个线程都同时取得了 e,则他们下一个元素都是 e，然后赋值给 table 元素时有一个成功有一个丢失。

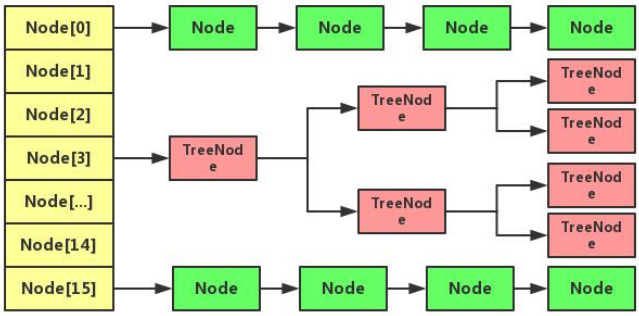
虽然 HashMap 是线程不安全的，还是有人在多线程情况下用，其实是开发者自己的原因。有人把这个问题报给了 Oracle，不过 Oracle 起初不认为这是一个问题。因为 HashMap 本来就不支持并发。要并发就用 ConcurrentHashmap 或者使用 Collections 类的 synchronizedMap()方法来同步。别看 Oracle 嘴硬，在 JDK8 中还是进行了修补。

四、JDK8 HashMap

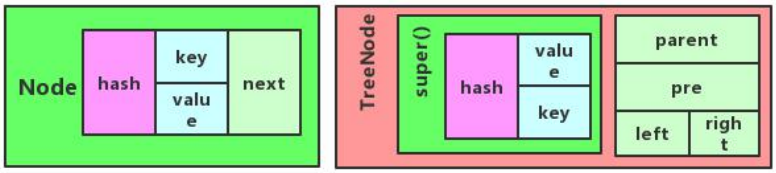
4.1 问题 4：JDK8 中 HashMap 的变化

- 结构变化：由数组+链表变成了数组+链表+红黑树。

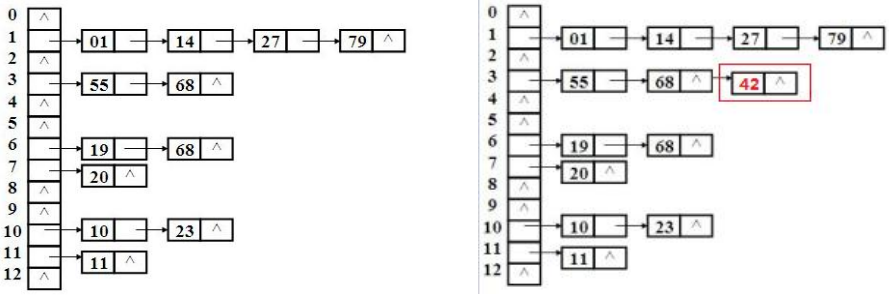




- 链表长度>=8，转换为红黑树；链表长度减少为 6，红黑树再变回链表。只有总的节点数量>=64 的时候，才会有红黑树，否则直接进行主数组扩容
- 链表节点为 Node，红黑树节点为 TreeNode。Node 是 TreeNode 的父类。



- 添加到链表后面：JDK7 中新的节点是加到最前，JDK8 后新节点是加到最后（也是避免死循环的一种解决方式）



- 主数组的创建不是构造方法中搞定，而是 put 元素时通过 resize()搞定。
- 哈希表扩容后原来链表节点重新散列后不改变之前顺序，也不会形成循环链表

注意：JDK8 HashMap 虽然针对 JDK7 的缺点做了某些修改，但是仍旧是线程不安全的，并发情况下建议使用 ConcurrentHashMap，或者使用 Collections 加锁。

4.2 问题 5：为什么是当链表长度>=8 后变成红黑树，而不是其他值

```
* rarely used.  Ideally, under random hashCodes, the frequency of
* nodes in bins follows a Poisson distribution
* (http://en.wikipedia.org/wiki/Poisson_distribution) with a
* parameter of about 0.5 on average for the default resizing
* threshold of 0.75, although with a large variance because of
* resizing granularity. Ignoring variance, the expected
* occurrences of list size k are (exp(-0.5) * pow(0.5, k) /
* factorial(k)). The first values are:
*
* 0:    0.60653066
* 1:    0.30326533
* 2:    0.07581633
* 3:    0.01263606
* 4:    0.00157952
* 5:    0.00015795
* 6:    0.00001316
* 7:    0.00000094
* 8:    0.00000006
* more: less than 1 in ten million
```

因为泊松分布 Poisson distribution（概率和数理统计内容）。

在理想的随机 hashCodes 下，容器中节点的频率遵循泊松分布，对于 0.75 的默认调整阈值，泊松分布的概率质量函数中参数λ（事件发生的平均次数）的值约为 0.5，尽管λ的值会因为 load factor 值的调整而产生较大变化。

PMF(probability Mass Function/概率质量函数): 点击了解PMF

$$P(k \text{ events in interval}) = \frac{\lambda^k e^{-\lambda}}{k!}$$

1. Lambda 是 在 一段特定时间/空间内 事件发生的平均值.
2. e 是自然常数 2.71828...
3. k 是 事件在这 一段发生的次数
4. P(K event in interval) 是该事件在这一段时间/空间发生的 概率

即链表中出现 8 个节点的概率是非常低的，仅有 0.00000006，所以不用担心产生大量红黑树导致结构复杂的问题

4.3 源码阅读

- 依旧要求主数组容量还是 2 的幂

```
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;
}
```

这是一个小巧但精妙的方法，这里通过异或的位运算将两个字节的 n 打造成比 cap 大但最接近 2 的 n 次幂的一个数值。例如：

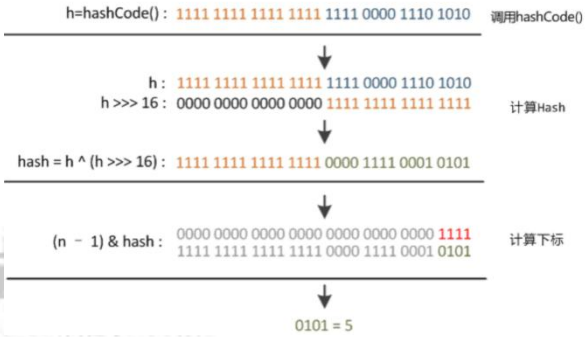
n=000...01xxxxxxx

```
n |= n >>> 1 : 000...01xxxxxxx | 000...001xxxxxx=000...011xxxxxx
n |= n >>> 2 : 000...011xxxxxx | 000...00011xxxx=000...01111xxxx
n |= n >>> 4 : 000...01111xxxx | 000...000001111=000...011111111
n |= n >>> 8 : 000...011111111 | 000....00000000=000...011111111
n |= n >>> 16 : 000...011111111 | 000.....000000=000...011111111
```

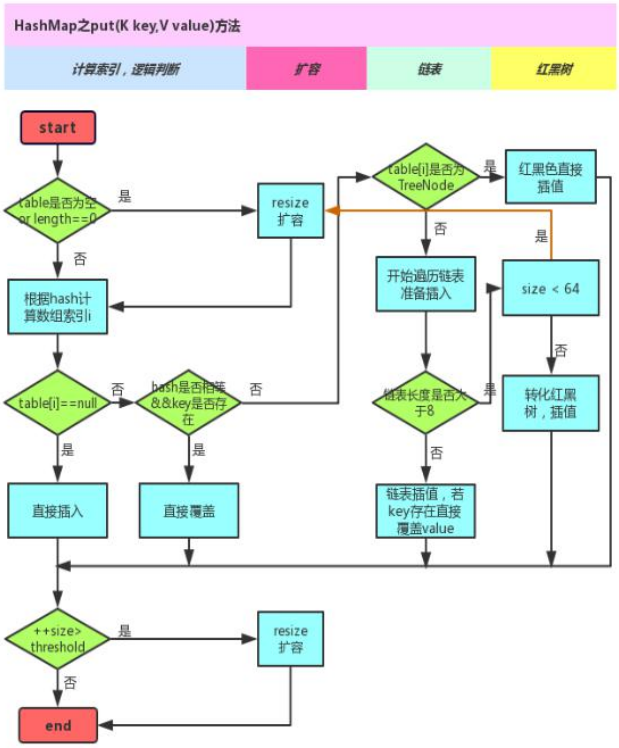
- 计算哈希码的方法简单了

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

JDK7 中，hash 计算的时候会对操作数进行右移操作，计算复杂，目的是将高位也参与运算，减少 hash 碰撞；在 JDK8 中，链表可以转变成红黑树，所以 hash 计算也变得简单。下面的图为 JDK8 中的 hash 计算和索引计算。



- 添加数据的步骤：put()



图片来自：https://blog.csdn.net/goosson/article/details/81029729

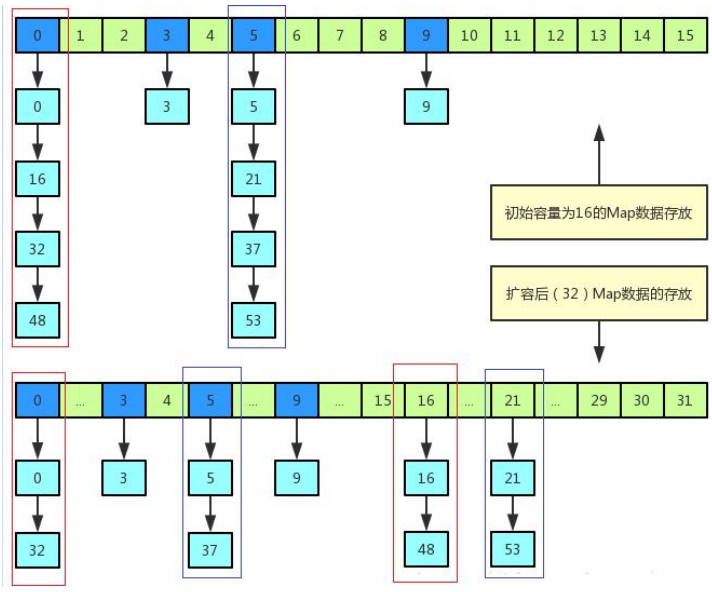
```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
// 第三个参数 onlyIfAbsent 如果是 true，那么只有在不存在该 key 时或者 value 是 null 才会进行
// put 操作，第四个参数 evict 我们这里不关心
 final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {
    Node<K, V>[] tab;//指向哈希表主数组的数组名
    Node<K, V> p;//链表
    int n, i; //n 永远存放数组长度，i 表示 key 在数组中的索引
    // 第一次 put 值的时候，会触发下面的 resize(),第一次 resize 和后续的扩容有些不一样，
    // 因为这次是数组从 null 初始化到默认的 16 或自定义的初始容量
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 找到具体的数组下标，如此位置没有值，那么直接初始化一下 Node 并放置在这个位置就可以了
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {// 数组该位置有数据
        Node<K, V> e;
        K k;
        // 首先，判断该位置的第一个数据和要插入的数据，key 是不是"相等"，如果是，取出这个节点
        if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
            // 如果该节点是代表红黑树的节点，调用红黑树的插值方法，本文不展开说红黑树
        else if (p instanceof TreeNode)
            e = ((TreeNode<K, V>) p).putTreeVal(this, tab, hash, key, value);
        else {
            // 到这里，说明数组该位置上是一个链表
            for (int binCount = 0; ; ++binCount) {
                // 如果不存在相同 key 的，插入到链表的最后面(Java7 是插入到链表的最前面)
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    // TREEIFY_THRESHOLD 为 8，所以，如果新插入的值是链表中的第 8 个
                    // 会触发下面的 treeifyBin，也就是将链表转换为红黑树
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                // 如果在该链表中找到了"相等"的 key(== 或 equals)
                if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                    // 此时 break，那么 e 为链表中[与要插入的新值的 key "相等"]的 node
                    break;
```

```
                    p = e;
                }
            }
            // e!=null 说明存在旧值的 key 与要插入的 key"相等"
            // 对于我们分析的 put 操作，下面这个 if 其实就是进行 "值覆盖"，然后返回旧值
            if (e != null) {
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        // 如果 HashMap 由于新插入这个值导致 size 已经超过了阈值，需要进行扩容
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

■ 主数组容量扩容为原来的二倍，原来元素的索引会有什么变化吗？

索引要么是原来的索引，要么是原来的索引+原来的数组容量。在 JDK8 中，不进行存储位置的重新计算，而是判断应该在原位置还是新位置。判断条件为：

```
if ((e.hash & oldCap) == 0) {//如果是原来的索引
 }  else{//如果是原来的索引+原来的数组容量
 }
```



■ 扩容到底是怎么实现的

扩容后，原来的一个链表可能会变成两个链表。实现思路是定义四个指针，在对原来链表进行逐个重新散列的过程中

✧ Node loHead,loTail 分别指向索引不变的新链表的首节点、尾节点

✧ Node hiHead,hiTail 分别指向索引改变的新链表的首节点、尾节点

```
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;
    }
}
```

■ 单链表变成红黑树是怎么实现的

简单来说，是先调用 treeifyBin()方法，将单链表变成双向链表（但节点已经是红黑树的节点 TreeNode），再将双向链表转换为红黑树。

```
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, next: 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);
    }
}
```

五、简单来看一下 Hashtable

```
* Hash table based implementation of the <tt>Map</tt> interface.  This
* implementation provides all of the optional map operations, and permits
* <tt>null</tt> values and the <tt>null</tt> key.  (The <tt>HashMap</tt>
* class is roughly equivalent to <tt>Hashtable</tt>, except that it is
* unsynchronized and permits nulls.)  This class makes no guarantees as to
* the order of the map; in particular, it does not guarantee that the order
* will remain constant over time.
```

Hashmap 类大致相当于 Hashtable，只是它不同步并且允许空值。

5.1 问题 6：Hashtable 和 HashMap 的不同之处

- Hashtable 的方法是同步的

```
/** Returns the number of keys in this hashtable. ...*/
public synchronized int size() {
    return count;
}

/** Tests if this hashtable maps no keys to values. ...*/
public synchronized boolean isEmpty() { return count == 0; }
```

- 父类是 Dictionary
- 主数组长度不要求是 2 的幂
- 默认长度 11
- Key 和 value 都不能是 null
- 计算哈希值就是直接调用 hashCode()
- 计算存储位置就是使用%取模

```
public synchronized V put(K key, V value) {
    // Make sure the value is not null
    if (value == null) {
        throw new NullPointerException();
    }

    // Makes sure the key is not already in the hashtable.
    Entry<?,?> tab[] = table;
    int hash = key.hashCode();
    int index = (hash & 0x7FFFFFFF) % tab.length;
    /unchecked/
    Entry<K,V> entry = (Entry<K,V>)tab[index];
    for(; entry != null ; entry = entry.next) {
        if ((entry.hash == hash) && entry.key.equals(key)) {
            V old = entry.value;
            entry.value = value;
            return old;
        }
    }

    addEntry(hash, key, value, index);
    return null;
}
```

- 每次扩容为原来容量的 2 倍再+1

```
protected void rehash() {
    int oldCapacity = table.length;
    Entry<?,?>[] oldMap = table;

    // overflow-conscious code
    int newCapacity = (oldCapacity << 1) + 1;
    if (newCapacity - MAX_ARRAY_SIZE > 0) {
        if (oldCapacity == MAX_ARRAY_SIZE)
            // Keep running with MAX_ARRAY_SIZE buckets
            return;
        newCapacity = MAX_ARRAY_SIZE;
    }
    Entry<?,?>[] newMap = new Entry<?,?>[newCapacity];
```

- 使用 Enumeration 进行迭代，而不是使用 Iterator（JDK1.2 才有）。

5.2 问题 7：为什么 Hashtable 主数组默认长度是 11，为何扩容 2 倍还要+1

按照哈希表的经典理论：哈希表主数组的长度应该是一个素数，这样会产生最分散的余数，尽可能减少哈希冲突。11 就是素数。扩容 2 倍肯定不是素数，+1 可能变成素数。

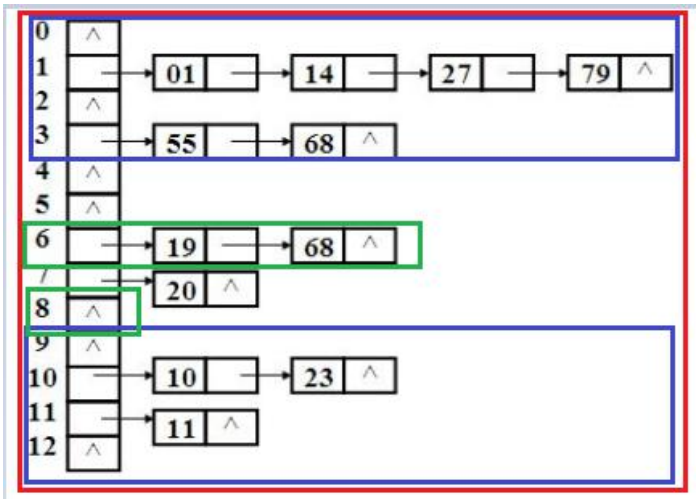
5.3 问题 8：Hashtable 的缺点

Hashtable 的方法使用的 synchronized 同步方法锁，非静态方法的锁是 this（当前的 Hashtable 对象，即整个哈希表）。一个线程上锁，就会锁住所有的访问同步方法的线程，并且是挡在了方法之外，效率太低。

HashMap 是非线程同步的，可以借助 Collections. synchronziedMap()保证线程安全，底层使用 synchronized 同步代码块，同步监视器也是当前的对象 this，也是锁住了整个哈希表，但是是将其他线程所在了方法之内，同步代码之外，实际性能比 Hashtable 有提高，但是提高有限。

在大量高并发情况下如何提高集合的效率和安全呢？能否降低锁的粒度，锁住哈希表的一部分而不是全部呢？可以的

- **JDK7 ConcurrentHashMap 锁住主数组的一部分（几个桶）**
- **JDK8 ConcurrentHashMap 锁住主数组的一部分（一个桶）**

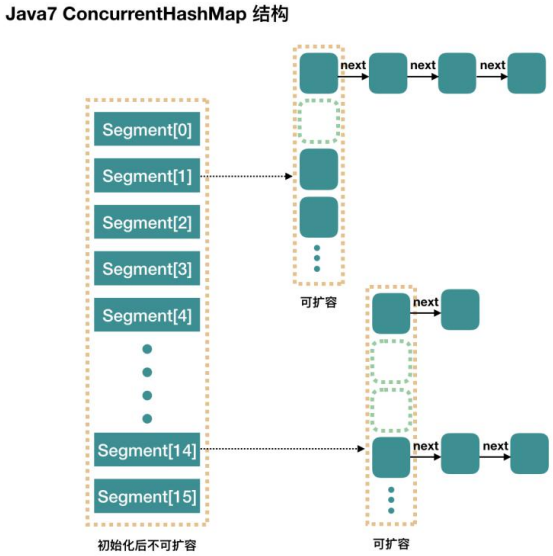


red：锁住整个表（所有桶）； blue：锁住几个桶 green：锁住一个桶

六、JDK7 ConcurrentHashMap

6.1 问题 9：JDK7 ConcurrentHashMap 关键技能点

- 使用分段锁 Segment。由 Hashtable 的锁住整个表，HashMap 的不锁，到锁住表的一部分。线程同步使用的是 Lock 锁。
- 结构图：



- ConcurrentHashMap 是由 Segment 数组和 HashEntry 数组结构组成。
- Segment 是一种可重入锁 ReentrantLock ,在 ConcurrentHashMap 里扮演锁的角色，HashEntry 则用于存储键值对数据。
- 一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的结构和 HashMap 类似，是一种数组和链表结构。一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素。每个 Segment 守护着一个 HashEntry 数组里的元素,当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁。

6.2 问题 10：JDK7 ConcurrentHashMap 通过无参构造方法创建对象的结果

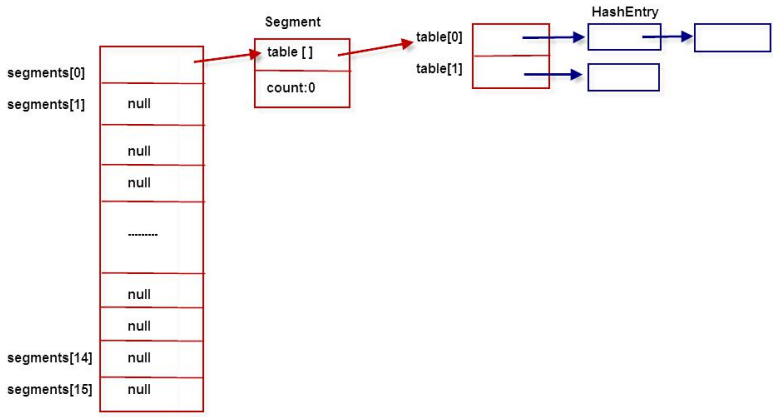
```
class ConcurrentHashMap{
        static final int DEFAULT_INITIAL_CAPACITY = 16;
        static final float DEFAULT_LOAD_FACTOR = 0.75f;
        static final int DEFAULT_CONCURRENCY_LEVEL = 16;

        public ConcurrentHashMap() {
                this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR,
                        DEFAULT_CONCURRENCY_LEVEL);
        }
        public ConcurrentHashMap(int initialCapacity,
                                 float loadFactor, int concurrencyLevel) {

        }
}
```

```
public ConcurrentHashMap(int initialCapacity,
                         float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    if (concurrencyLevel > MAX_SEGMENTS)
        concurrencyLevel = MAX_SEGMENTS;
    // Find power-of-two sizes best matching arguments
    int sshift = 0;
    int ssize = 1;
    while (ssize < concurrencyLevel) {
        ++sshift;
        ssize <<= 1;
    }
    this.segmentShift = 32 - sshift;
    this.segmentMask = ssize - 1;
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    int c = initialCapacity / ssize;
    if (c * ssize < initialCapacity)
        ++c;
    int cap = MIN_SEGMENT_TABLE_CAPACITY;
    while (cap < c)
        cap <<= 1;
    // create segments and segments[0]
    Segment<K,V> s0 =
        new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
                         (HashEntry<K,V>[])new HashEntry[cap]);
    Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
    UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]
    this.segments = ss;
}
```

- initialCapacity :初始容量 ,这个值指的是整个 ConcurrentHashMap 的初始容量，实际操作的时候需要平均分给每个 Segment
- loadFactor :加载因子。Segment 数组不扩容 ,所以是针对每个 Segment 内部的加载因子。
- concurrencyLevel：并发级别，segment 数组容量，默认 16。要求是 2 的幂。
- SEGMENT_TABLE_CAPACITY :每个 Segment 的内部数组的容量 ,最小是 2。可以扩容，必须是 2 的幂。每次扩容 100%。(int newCapacity = oldCapacity << 1;)
- 采用无参数构造方法创建对象后的内存结构如图所示(不包含蓝色部分)。Segment 数组长度 16 ,只给 segments[0]分配空间。发现每个 Segment 其实就是一个 HashMap，其中的数组 table 的容量是 2。



- **认识 Segment 和 HashEntry：**
  - Segment 其实就是之前的一个 HashMap ,本身继承了 ReentrantLock，可以直接复用加锁、解锁等操作
  - HashEntry 就是 HashMap 中的 Entry，是链表的节点类型，存储具体的键值对信息。
  - 注意：Segment 的 table、HashEntry 的 value、next 都使用 volatile 修饰，其修改在各个线程之间具有可见性。

```
class ConcurrentHashMap{
        static final class Segment<K,V>
                extends ReentrantLock implements Serializable {

                transient volatile HashEntry<K, V>[] table;
                transient int count;
                transient int threshold;
                final float loadFactor;
                Segment(float lf, int threshold, HashEntry<K, V>[] tab) {
                        this.loadFactor = lf;
                        this.threshold = threshold;
                        this.table = tab;
                }
        }
        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;
                }
        }
}
```

6.3 JDK7 ConcurrentHashMap 源码阅读

- **put 操作：**

1) 计算 key 所在的 Segments 数组的索引 j。如果 segments[j]==null，需要先分配空间。

```
public V put( @NotNull K key,  @NotNull V value) {
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    int hash = hash(key);
    int j = (hash >>> segmentShift) & segmentMask;
    if ((s = (Segment<K,V>)UNSAFE.getObject          //
         (segments, (j << SSHIFT) + SBASE)) == null)
        s = ensureSegment(j);
    return s.put(key, hash, value, onlyIfAbsent: false);
}
```

2) 计算 key 所在的 HashEntry 数组的索引，并完成添加操作(使用 Lock 锁保证并发安全)

```
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    HashEntry<K,V> node = tryLock() ? null :
        scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        HashEntry<K,V>[] tab = table;
        int index = (tab.length - 1) & hash;
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
            if (e != null) {
                K k;
                if ((k = e.key) == key ||
                    (e.hash == hash && key.equals(k))) {...}
                e = e.next;
            }
            else {...}
        }
    } finally {
        unlock();
    }
    return oldValue;
}
```

3) 注意 1：新的节点会添加到链表的头部（JDK7 的 HashMap 和 ConcurrentHashMap 都是添加到头部）。

4) 注意 2：添加了新节点再判断是否扩容（JDK7 的 HashMap 是先判断是否扩容，再添加）。


```
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    HashEntry<K,V> node = tryLock() ? null :
        scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        HashEntry<K,V>[] tab = table;
        int index = (tab.length - 1) & hash;
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
            if (e != null) {...}
            else {
                if (node != null)
                    node.setNext(first);
                else
                    node = new HashEntry<K,V>(hash, key, value, first);
                int c = count + 1;
                if (c > threshold && tab.length < MAXIMUM_CAPACITY)
                    rehash(node);
                else
                    setEntryAt(tab, index, node);
                ++modCount;
                count = c;
                oldValue = null;
                break;
            }
        }
    } finally {
        unlock();
    }
    return oldValue;
}
```

- **get 操作：**

1) 计算 key 所在的 Segments 数组的索引 j。如果是 null，直接返回 null，不存在。
2) 计算 key 所在的 HashEntry 数组的索引。找到了返回 HashEntry 的 value，找不到，返回 null。
3) **查询不加锁，但是支持并发查询。需要使用 UNSAFE 的方法**

```
public V get(Object key) {
    Segment<K,V> s; // manually integrate access methods to reduce overhead
    HashEntry<K,V>[] tab;
    int h = hash(key);
    long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
    if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
        (tab = s.table) != null) {
        for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
                 (tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE);
             e != null; e = e.next) {
            K k;
            if ((k = e.key) == key || (e.hash == h && key.equals(k)))
                return e.value;
        }
    }
    return null;
}
```


- **size()操作：**

1) 有些方法需要跨段，比如 size()和 containsValue()。需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。
2) 这里“按顺序”是很重要的，否则极有可能出现死锁

```
public int size() {
    //...
    final Segment<K,V>[] segments = this.segments;
    int size;
    boolean overflow; // true if size overflows 32 bits
    long sum;         // sum of modCounts
    long last = 0L;   // previous sum
    int retries = -1; // first iteration isn't retry
    try {
        for (;;) {
            if (retries++ == RETRIES_BEFORE_LOCK) {
                for (int j = 0; j < segments.length; ++j)
                    ensureSegment(j).lock(); // force creation
            }
            sum = 0L;
            size = 0;
            overflow = false;
            for (int j = 0; j < segments.length; ++j) {...}
            if (sum == last)
                break;
            last = sum;
        }
    } finally {
        if (retries > RETRIES_BEFORE_LOCK) {
            for (int j = 0; j < segments.length; ++j)
                segmentAt(segments, j).unlock();
        }
    }
    return overflow ? Integer.MAX_VALUE : size;
}
```

6.4 问题 11：Unsafe 类是怎么回事

- Unsafe 类是在 sun.misc 包下，不属于 Java 标准。但是很多 Java 的基础类库，包括一些被广泛使用的高性能开发库都是基于 Unsafe 类开发的，比如 Netty、Hadoop、Kafka 等。
- 使用 Unsafe 可用来直接访问系统内存资源并进行自主管理，大部分 API 都是 native 的方法。Unsafe 类在提升 Java 运行效率，增强 Java 语言底层操作能力方面起了很大的作用。
- Unsafe 可认为是 Java 中留下的后门，提供了一些低层次操作，如直接内存访问、线程调度等。官方并不建议使用 Unsafe。

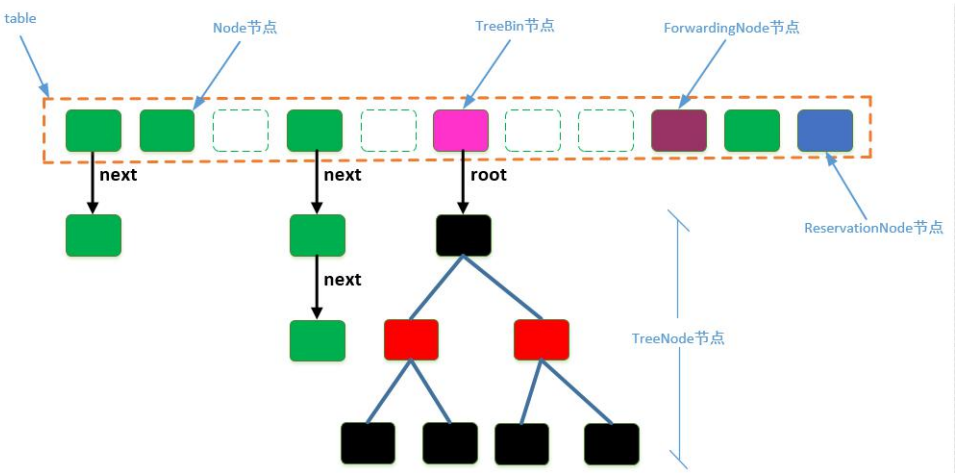
6.5 问题 12：JDK7 ConcurrentHashMap 的缺点是什么

- 结构复杂了：由一个 Segment 数组和多个 HashEntry 组成的两级结构组成
- 查询效率低了：需要先查询到 Segment 的索引，再查询到 HashEntry 的索引。统计 size()需要遍历整个 Segment 数组。
- 锁的粒度也不算小：concurrentLevel（并发数）基本上是固定的，其实还是锁住了一个哈希表，哪怕是一个小的 HashMap。能否只锁住 HashMap 的一个桶呢？concurrentLevel 就可以和数组大小保持一致了。

七、JDK8 中 ConcurrentHashMap

7.1 问题 13：JDK8 中 ConcurrentHashMap 变化

1) 结构简单：JDK8 抛弃 JDK7 的 Segment 分段锁机制，由 JDK7 的两级数组变回了原来的一级数组。链表长度>=8，该链表转换为红黑树。

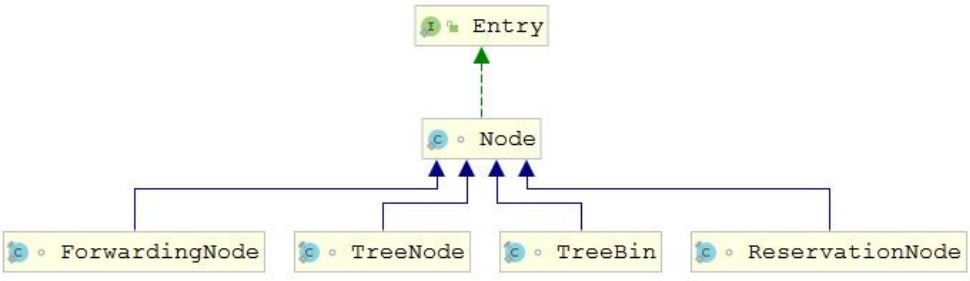


ConcurrentHashMap结构(JDK1.8)

2) 降低锁的粒度：锁住数组的每个桶的头结点，锁粒度更小。（Hashtable 是锁住整个表、JDK7 的 ConcurrrentHashMap 是锁住一个段 Segment。而这里是锁住一个链表或者一个红黑树）
3) 锁变化：不使用 Segment 锁（继承 ReentrantLock），利用 CAS+Synchronized 来保证并发安全。
4) 并发扩容,多个线程参与。（JDK7 的 ConncurrentHashMap 的 Segement 数组长度固定不扩容，扩容的每个 HashEntry 数组的容量，此时不需要考虑并发，因为到这里的时候，是持有该 Segment 的独占锁的）

   注意：JDK8 中，Segment 类依旧存在，但只是为了兼容，只有在序列化和反序列化时才会被用到

5) 更多的 Node 类型

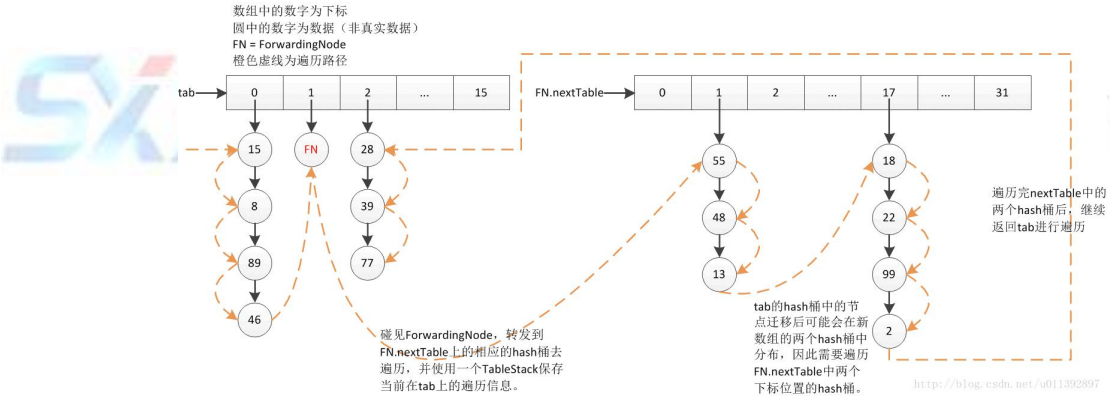


a. Node<K,V> :基本结点/普通节点。当 table 中的 Entry 以链表形式存储时才使用，存储实际数据。该类的 key 和 value 不为 null（其子类可为 null）

b. TreeNode：红黑树结点。当 table 中的 Entry 以红黑树的形式存储时才会使用，存储实际数据。ConcurrentHashMap 中对 TreeNode 结点的操作都会由 TreeBin 代理执行。

c. TreeBin：代理操作 TreeNode 结点。该节点的 hash 值固定为-2，存储实际数据的红黑树的根节点。因为红黑树进行写入操作整个树的结构可能发生很大变化，会影响到读线程。因此 TreeBin 需要维护一个简单的读写锁，不用考虑写-写竞争的情况。当然并不是全部的写操作都需要加写锁，只有部分 put/remove 需要加写锁。

d. ForwardingNode：转发结点。该节点是一种临时结点，只有在扩容进行中才会出现，该节点的 hash 值固定为-1，并且他不存储实际数据。如果旧 table 的一个 hash 桶中全部结点都迁移到新的数组中，旧 table 就在桶中放置一个 ForwardingNode。当读操作或者迭代操作遇到 ForwardingNode 时，将操作转发到扩容后新的 table 数组中去执行，当写操作遇见 ForwardingNode 时，则尝试帮助扩容。



e. ReservationNode :保留结点，也被称为空节点。该节点的 hash 值固定为-3，不保存实际数据。正常的写操作都需要对 hash 桶的第一个节点进行加锁，如果 hash 桶的第一个节点为 null 时是无法加锁的，因此需要 new 一个 ReservationNode 节点，作为 hash 桶的第一个节点，对该节点进行加锁。

7.2 问题 14：JDK8 ConcurrentHashMap 怎么放弃 Lock 使用 synchronized 了

1) synchronized 之前一直都是重量级锁，但是 JDK6 中官方是对他进行过升级，引入了偏向锁，轻量级锁，重量级锁，现在采用的是锁升级的方式去做的。针对 synchronized 获取锁的方式，JVM 使用了锁升级的优化方式，就是先使用**偏向锁**优先同一线程然后再次获取锁，如果失败，就升级为 **CAS 轻量级锁**，如果失败就会短暂自旋，防止线程被系统挂起。最后如果以上都失败就升级为**重量级锁**。所

以是一步步升级上去的，最初也是通过很多轻量级的方式锁定的。

2) ReentantLock 是 JDK 层面的，synchronized 是 JVM 层面的。相对而言，synchronized 的性能优化空间更大，这就使得 synchronized 能够随着 JDK 版本的升级而不改动代码的前提下获得性能上的提升。

3) 另外此处 synchronized 锁住的是单个链表的头结点，粒度小，而不是 Hashtable、Collections 等锁整个哈希表。低粒度下，synchronized 和 Lock 的差异没有高粒度下明显。

对象头 Mark Word（标记字段）

锁状态	31bit(25bit unused)		4bit	1bit	2bit
	54bit	2bit		是否偏向锁	锁标志位
无锁	对象的HashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向轻量级锁的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空				11

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程使用自旋会消耗 CPU	追求响应时间，锁占用时间很短
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量，锁占用时间较长

7.3 问题 15：sizeCtl 属性的作用

sizeCtl 属性是 ConcurrentHashMap 中出镜率很高的一个属性，因为它是一个控制标识符，在不同的地方有不同用途，而且它的取值不同，也代表不同的含义。

1) -1 代表正在初始化

2) -N 表示有 N-1 个线程正在进行扩容操作

3) 正数或 0 代表 hash 表还没有被初始化，这个数值表示初始化或下一次进行扩容的大小，这一点类似于扩容阈值的概念。还后面可以看到，它的值始终是当前 ConcurrentHashMap 容量的 0.75 倍，这与 loadfactor 是对应的。

7.4 问题 16：折腾什么？Hashtable 不可存储 null key-value，HashMap 可以，ConcurrentHashMap 不可

其实你发现 Hashtable、ConcurrentHashMap 不允许 null，但是都是线程安全的，适用于多线程环境。而 HashMap 却是线程不安全的，适用于单线程环境。和此有关吗？其实有关系的。

无法容忍的歧义。如果 map.get(key) return null，是 key 不存在呢，还是 value 是 null 呢？单线程情况下可以区分，可以通过先调用 map.contains(key)来辨别，但在并行映射中，两次调用 map.contains(key)和 map.get(key) 之间，映射可能已更改。

7.5 JDK8 ConcurrentHashMap 源码阅读：put( )

```
public V put(K key, V value) {
    return putVal(key, value, false);
}
final V putVal(K key, V value, boolean onlyIfAbsent) {
    // key 和 value 均不能为空
    if (key == null || value == null) throw new NullPointerException();
    // 得到 hash 值
    int hash = spread(key.hashCode());
    // 用于记录相应链表的长度
    int binCount = 0;
    for (Node<K, V>[] tab = table; ; ) {
        Node<K, V> f;
        int n, i, fh;
        // 如果数组"空"，进行数组初始化
        if (tab == null || (n = tab.length) == 0)
            // 初始化数组，后面会详细介绍
            tab = initTable();
            // 找该 hash 值对应的数组下标，得到第一个节点 f
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 如果数组该位置为空，用一次 CAS 操作将这个新值放入其中即可，
            //这个 put 操作差不多就结束了，可以拉到最后面了
            // 如果 CAS 失败，那就是有并发操作，进到下一个循环就好了
            if (casTabAt(tab, i, null,
                    new Node<K, V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
```

```
            }
            // hash 等于 MOVED，表示正在扩容
            else if ((fh = f.hash) == MOVED)
                // 帮助数据迁移
                tab = helpTransfer(tab, f);
            else { // 到这里就是说，f 是该位置的头结点，而且不为空
                V oldVal = null;
                // 获取数组该位置的头结点的监视器锁
                synchronized (f) {
                    if (tabAt(tab, i) == f) {
                        if (fh >= 0) { // 头结点的 hash 值大于 0，说明是链表
                            // 用于累加，记录链表的长度
                            binCount = 1;
                            // 遍历链表
                            for (Node<K, V> e = f; ; ++binCount) {
                                K ek;
                                // 如果发现了"相等"的 key，进行值覆盖，
                                if (e.hash == hash &&
                                        ((ek = e.key) == key ||
                                                (ek != null && key.equals(ek)))) {
                                    oldVal = e.val;
                                    if (!onlyIfAbsent)
                                        e.val = value;
                                    break;
                                }
                                // 到了链表的最末端，将这个新值放到链表的最后面
                                Node<K, V> pred = e;
                                if ((e = e.next) == null) {
                                    pred.next = new Node<K, V>(hash, key,
                                            value, null);
                                    break;
                                }
                            }
                        } else if (f instanceof TreeBin) { // 红黑树
                            Node<K, V> p;
                            binCount = 2;
                            // 调用红黑树的插值方法插入新节点
                            if ((p = ((TreeBin<K, V>) f).putTreeVal(hash, key,
```

```
                                    value)) != null) {
                                oldVal = p.val;
                                if (!onlyIfAbsent)
                                    p.val = value;
                            }
                        }
                    }
                }
                // binCount != 0 说明上面在做链表操作
                if (binCount != 0) {
                    // 判断是否要将链表转换为红黑树,临界值和 HashMap 一样,也是 8
                    if (binCount >= TREEIFY_THRESHOLD)
                        treeifyBin(tab, i);
                    if (oldVal != null)
                        return oldVal;
                    break;
                }
            }
        }
        //节点数增加了 1 个
        addCount(1L, binCount);
        return null;
    }
```

八、最后的总结和交流

8.1 三代 HashMap 代码行数的变化

类名	代码行数（含注释）	重要性	作者
Hashtable	1400	*	
JDK7 HashMap	1150	***	Doug Lea...
JDK7 ConcurrentHashMap	1600	**	Doug Lea
JDK8 HashMap	2400	**	Doug Lea...
JDK8 ConcurrentHashMap	6300	***	Doug Lea

编程不识 Doug Lea,写尽 Java 也枉然。整个 JUC（java.util.concurrent）就是他的杰作。

Doug Lea 编辑 讨论

Doug Lea，中文名为道格·利。美国国籍，现担任纽约州立大学Oswego分校教师。

中文名	道格·利	国　籍	美国
外文名	Doug Lea	职　业	教师
		隶　属	纽约州立大学Oswego分校

如果IT的历史，是以人为主体串接起来的话，那么肯定少不了Doug Lea。这个鼻梁挂着眼镜，留着德王威廉二世的胡子，脸上永远挂着谦逊腼腆笑容，服务于纽约州立大学Oswego分校计算机科学系的老大爷。

说他是这个世界上对Java影响力最大的一个人，一点也不为过。因为两次Java历史上的大变革，他都间接或直接的扮演了举足轻重的角色。2004年所推出的Tiger。Tiger广纳了15项JSRs(Java Specification Requests)的语法及标准，其中一项便是JSR-166。JSR-166是来自于Doug编写的util.concurrent包。

值得一提的是: Doug Lea也是JCP (Java社区项目)中的一员。

Doug是一个无私的人，他深知分享知识和分享苹果是不一样的，苹果会越分越少，而自己的知识并不会因为给了别人就减少了，知识的分享更能激荡出不一样的火花。《Effective JAVA》这本Java经典之作的作者Joshua Bloch便在书中特别感谢Doug Lea是此书中许多构想的共鸣板，感谢Doug Lea大方分享丰富而又宝贵的知识。

8.2 多阅读源码

1. 理解的更加明白透彻，知其然且知其所以然
2. 阅读的是顶级程序员架构师的编码，开拓思路，增长智慧，无形之中提高编码能力。熟读唐诗三百首，不会做诗也会吟。
3. 面试的优势，脱颖而出。
4. 日常工作的优势，物以类聚，人以群分。你会认识很多技术牛人。
5. 如何看源码
   (1) 广度优先遍历，而不是深度优先遍历。先明白主要步骤，再研究每步骤细节
   (2) 有所为有所不为，研究关键点，懂得放弃。
   (3) 充分借助网络资源提高效率，大量的源码爱好者已经写好了详细的注释并分享
   (4) idea 中查看源码的跳转键 ctrl+alt+(左右方向健)

8.3 更多并发编程技能点

1. Java 内存模型（可见性、有序性、原子性）
2. AQS 抽象的队列式同步器
3. 锁的类型和升级：自旋锁、偏向锁、锁消除、锁粗化、悲观锁、乐观锁
4. JUC
5. CAS
6. ABA 问题
7. volatile
8. sun.misc.Unsafe 类
9. 安全失败（fail—safe）和快速失败（fail—fast）