ConcurrentHashMap（1.8）面试题

Author：郑金维

一、存储结构（常识）

数组+链表+红黑树

JDK1.7：数组+链表

JDK1.8：数组+链表+红黑树

为什么 1.8 中追加了红黑树：

- 链表的话，查询的时间复杂度为 On，链表过长，查询速度慢
- 当链表长度达到了 8 的时候，就要从链表转换为红黑树，红黑树查询的时间复杂度是 Ologn

链表长度到 8，一定会转换为红黑树嘛？

- 必须达到数组长度>=64，并且某一个桶下的链表长度到 8，才会转换为红黑树，因为数组查询效率更快

为什么链表长度为 8 才会转为红黑树？

- 泊松分布

红黑树什么时候回转换为链表

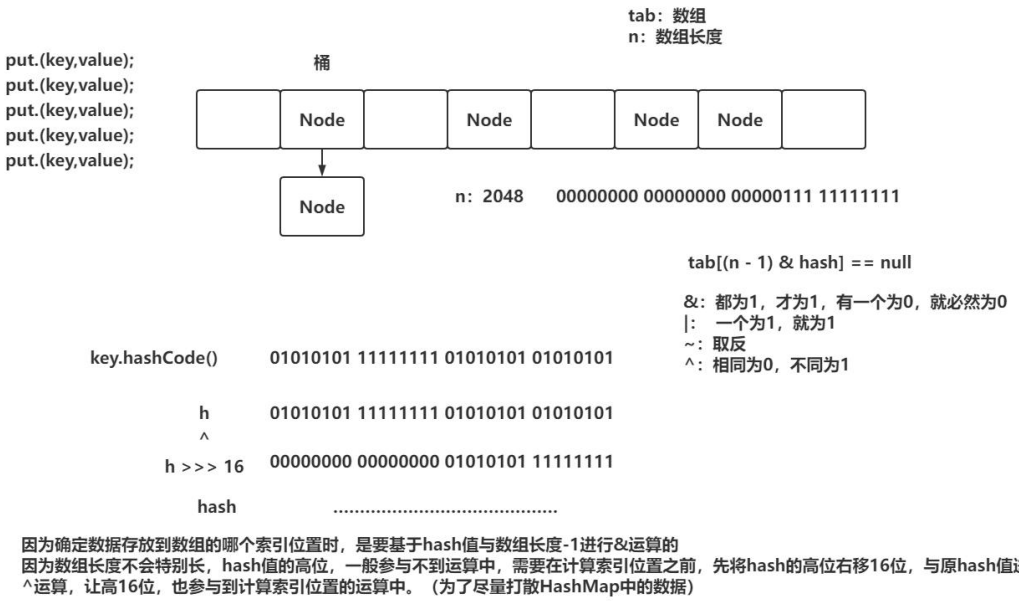
- 6 个

二、散列算法（hash 运算的方式）

散列算法：就是 HashMap、ConcurrentHashMap 如何基于 key 进行运算，并将 key-value 存储到数组的某一个节点上，或者是挂载到下面的链表或者红黑树上

2.1 散列算法介绍

```
// ConcurrentHashMap 的散列算法 int hash = spread(key.hashCode());// 具体实现 static final int spread(int h) {
    return (h ^ (h >>> 16)) & HASH_BITS;
}
```



2.2 为什么要执行一个&运算在散列算法中

```
h           01010101 11111111 01010101 01010101
  ^
h >>> 16    00000000 00000000 01010101 11111111
            11010101 01010101 01010101 01010101
  &
HASH_BITS   01111111 11111111 11111111 11111111
hash        01010101 01010101 01010101 01010101
```

&：都为1，才为1，有一个为0，就必然为0
|：一个为1，就为1
~：取反
^：相同为0，不同为1

正数 = (h ^ (h >>> 16)) & HASH_BITS;　　2进制的最高位是符号位，0为正数，1为负数

运算的目的是为了保证hash值，一定是正数，因为hash值为负数，有特殊含义！

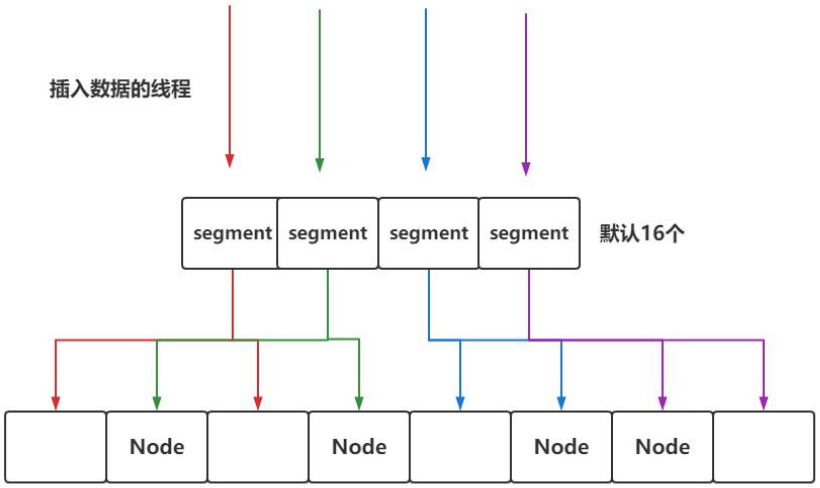
2.3 hash 有什么特殊含义

```
// Hash 值为-1，代表当前位置数据已经被迁移到新数组中（正在扩容！）static final int MOVED     = -1; // hash for forwarding nodes// Hash 值为-2，代表当前索引位置下是一颗红黑树!static final int TREEBIN   = -2; // hash for roots of trees// Hash 值为-3，代表当前索引位置已经被占座了 static final int RESERVED = -3; // hash for transient reservations
```

三、保证安全的方式

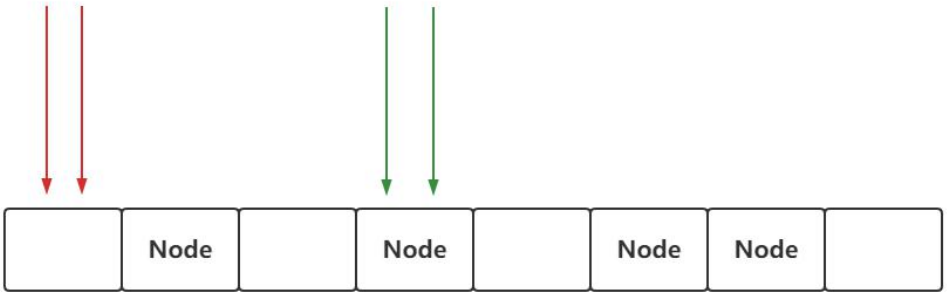
Hashtable：是将方法追加上 synchronized 保证线程安全（速度巨慢）

JDK1.7 的 ConcurrentHashMap：使用分段锁，Segment，原理就是 ReentrantLock。



JDK1.8 的 ConcurrentHashMap：基于 CAS 和 synchronized 同步代码块实现的线程安全

卷卷单单~~



1. 插入数据时，数组索引位置没数据，那就使用CAS的方式，将数据插入到索引位置

2. 插入数据时，数组索引位置有数据，可能需要追加到链表或者红黑树上，这时锁住当前桶

```
for (Node<K,V>[] tab = table;;) {
    // f 就是数组上的数据。
    if ((f = tab[(n - 1) & hash]) == null) {
        if (CAS 插入数据))
            break;
    }
    else {
        V oldVal = null;
        synchronized (f) {
            // 基于当前索引位置数据作为锁，插入
        }
    }
}
```

四、ConcurrentHashMap 扩容

4.1 sizeCtl 是啥?

sizeCtl = -1：代表当前 ConcurrentHashMap 的数组正在初始化

sizeCtl < -1：代表当前 ConcurrentHashMap 正在扩容，低 16 位的值为-2，代表有 1 个线程在扩容

sizeCtl = 0：代表当前还没初始化呢

sizeCtl > 0：如果数组还没初始化，代表初始数组长度。 如果数组已经初始化了，就代表扩容阈值

ConcurrentHashMap 在第一次 put 操作时，才会初始化数组。

sizeCtl = -2 时，代表有 1 个线程在扩容。-1 已经代表初始化状态了，而且在扩容时，-2 也有妙用!

4.2 ConcurrentHashMap 扩容触发条件

- 数组长度达到了扩容的阈值
- 链表达到了 8，但是数组长度没到 64，触发扩容
- 在执行 putAll 操作时，会直接优先判断是否需要扩容

在一些方法扩容时，有的会先执行 tryPresize，有的会自行判断逻辑，计算扩容戳，执行 transfer 方法开始扩容

4.3 扩容戳

ConcurrentHashMap 会触发 helpTransfer 操作，也就是多线程扩容。

就要保证在扩容时，多个线程扩容是的长度都是一样的 A（32 - 64），B（32 - 64），C（64 - 128）

基于这个方式计算扩容标识：

```
static final int resizeStamp(int n) {
    return Integer.numberOfLeadingZeros(n) | (1 << (RESIZE_STAMP_BITS - 1));
}
```

结果跟原数组长度是绑定到一起的，如果原数组长度不一样，那么结果必然不一样！

ConcurrentHashMap 扩容处的 BUG：https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8214427

在 JDK12 中，修复了一部分。

4.4 扩容流程

（比如从 32 长度扩容到 64 长度）

ConcurrentHashMap 在扩容时，会先指定每个线程每次扩容的长度，最小值为 16（根据数组长度和 CPU 内核去指定每次扩容长度）。

开始扩容，而开始扩容的线程只有一个，第一个扩容的线程需要把新数组 new 出来。

有了新数组之后，其他线程再执行 transfer 方法（可能从 helpTransfer 方法进来），其他线程进来后，对扩容戳进行+1 操作，也就是如果 1 个线程低位是-2，那么 2 个线程低位为-3

每次迁移时，会从后往前迁移数据，也就是说两个线程并发扩容：

线程 A 负责索引位置：16~31

线程 B 负责索引位置：15~0

是一个桶一个桶的去迁移数据，每次迁移完一个桶之后，会将，会将 ForwardingNode 设置到老数组中，证明当前老数组的数据已经迁移到新数组了！

在迁移链表数据时，会基于 lastRun 机制，提升效率

lastRun：提前将链表数据进行计算，算出链表的节点需要存放到哪个新数组位置，将不同位置算完打个标记

```
Node<K,V> lastRun = f;for (Node<K,V> p = f.next; p != null; p = p.next) {
```

```
    int b = p.hash & n;
    if (b != runBit) {
        runBit = b;
        lastRun = p;
    }
}
```

五、加个钟

老数组数据放到新数组的哪个位置上：

```
// HashMap 和 ConcurrentHashMap 计算原理一致  oldCap=16   newCap=32
hash & (oldCap - 1)01010101 01010101 01010101 0101010101010101 01010101 01010101 01000101


00000000 00000000 00000000 00010000
```

结果只有两种情况：要么是 0，要么是老数组长度// lo 就是放到新数组的原位置。（老数组放到索引为 1 的位置，新数组也放到索引为 1 的位置。）// hi 就是放到新数组的原位置 + 老数组长度的位置。（老数组放到索引为 1 的位置，新数组放到 17 位置）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);
```