[置顶] Map 综述（三）：彻头彻尾理解 ConcurrentHashMap

摘要：

ConcurrentHashMap 是 J.U.C(java.util.concurrent 包)的重要成员，它是 HashMap 的一个线程安全的、支持高效并发的版本。在默认理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作及任意数量线程的读操作。本文将结合 Java 内存模型，分析 JDK 源代码，探索 ConcurrentHashMap 高并发的具体实现机制，包括其在 JDK 中的定义和结构、并发存取、重哈希和跨段操作，并着重剖析了 ConcurrentHashMap 读操作不需要加锁和分段锁机制的内在奥秘和原理。

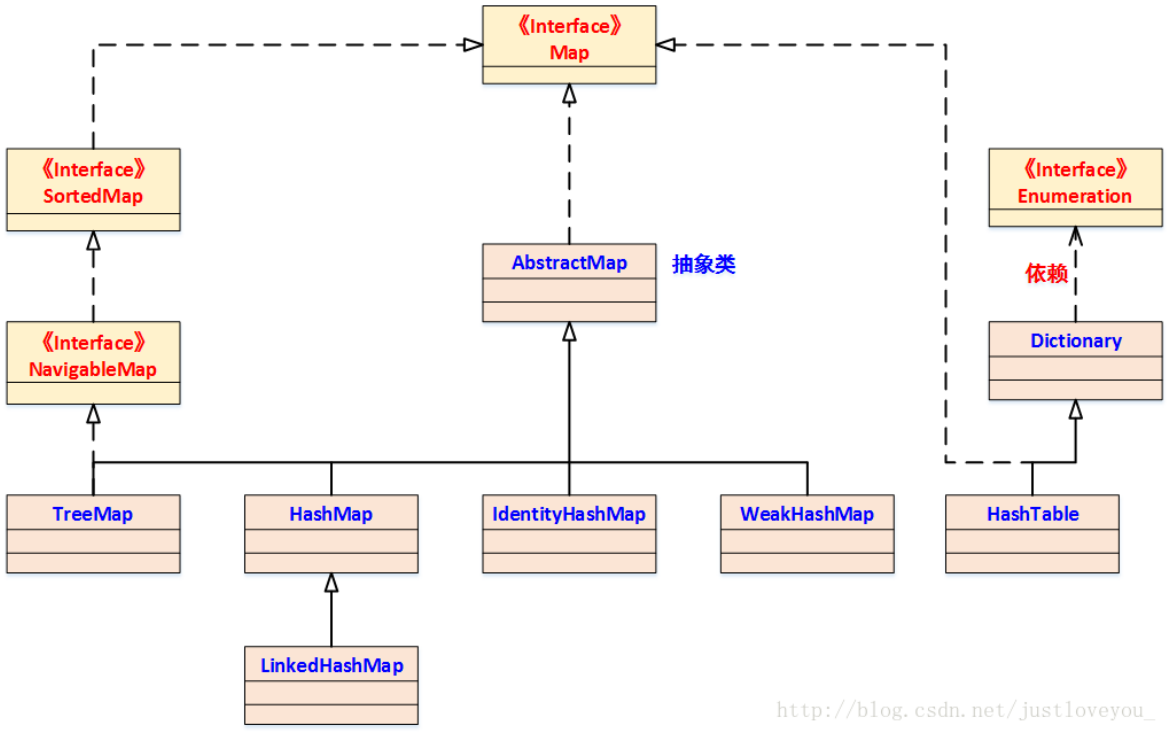
友情提示：

本文所有关于 ConcurrentHashMap 的源码都是基于 **JDK 1.6** 的，不同 JDK 版本之间会有些许差异，但不影响我们对 ConcurrentHashMap 的数据结构、原理等整体的把握和了解。

由于 ConcurrentHashMap 的源代码实现依赖于 Java 内存模型，所以阅读本文需要读者了解 Java 内存模型与 Volatile 语义，具体详见《Java 并发：volatile 关键字解析》一文。同时，ConcurrentHashMap 的源代码会涉及到散列算法和链表数据结构，所以，读者需要对散列算法和基于链表的数据结构有所了解，特别是对 HashMap 的进一步了解和回顾。关于 HashMap 的详细介绍，请移步我的博文《Map 综述（一）：彻头彻尾理解 HashMap》。

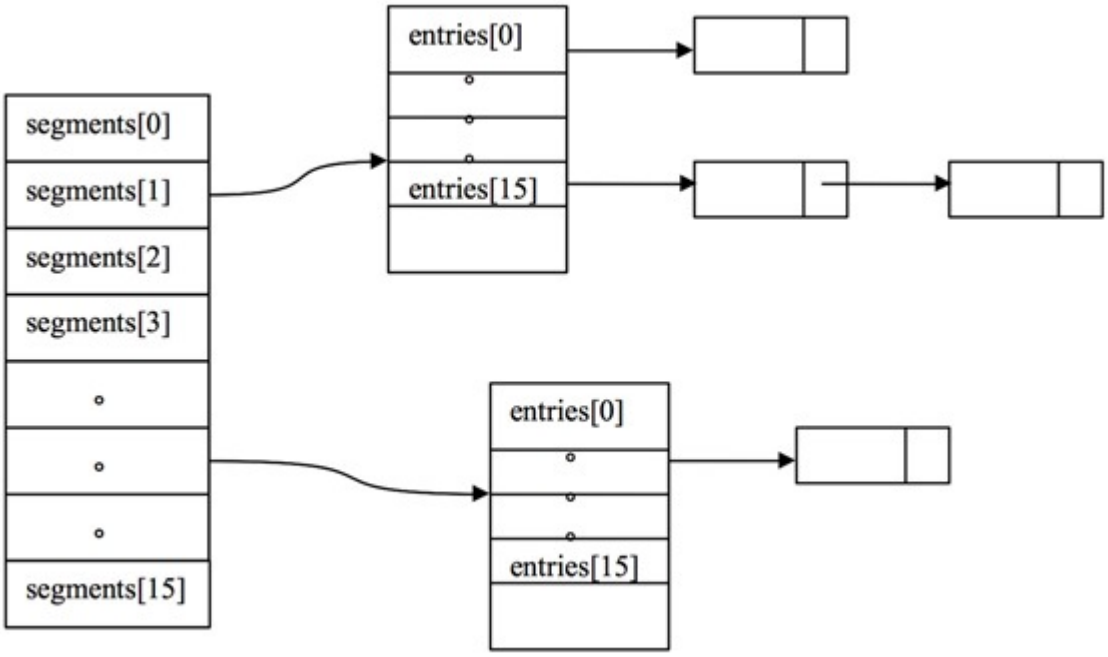
一. ConcurrentHashMap 概述

笔者曾在《Map 综述（一）：彻头彻尾理解 HashMap》一文中提到，HashMap 是 Java Collection Framework 的重要成员，也是 Map 族(如下图所示)中我们最为常用的一种。不过遗憾的是，HashMap 不是线程安全的。也就是说，在多线程环境下，操作 HashMap 会导致各种各样的线程安全问题，比如在 HashMap 扩容重哈希时出现的死循环问题，脏读问题等。HashMap 的这一缺点往往会造成诸多不便，虽然在并发场景下 HashTable 和由同步包装器包装的 HashMap(Collections.synchronizedMap(Map<K,V> m))可以代替 HashMap，但是它们都是通过使用一个全局的锁来同步不同线程间的并发访问，因此会带来不可忽视的性能问题。庆幸的是，JDK 为我们解决了这个问题，它为 HashMap 提供了一个线程安全的高效版本 —— ConcurrentHashMap 。在 ConcurrentHashMap 中，无论是读操作还是写操作都能保证很高的性能：在进行读操作时(几乎)不需要加锁，而在写操作时通过锁分段技术只对所操作的段加锁而不影响客户端对其它段的访问。特别地，**在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设为 16），及任意数量线程的读操作。**



如下图所示，ConcurrentHashMap 本质上是一个 Segment 数组，而一个 Segment 实例又包含若干个桶，每个桶中都包含一条由若干个 HashEntry 对象链接起来的链表。总的来说，ConcurrentHashMap 的高效并发机制是通过以下三方面来保证的(具体细节见后文阐述)：

- **通过锁分段技术保证并发环境下的写操作；**
- **通过 HashEntry 的不变性、Volatile 变量的内存可见性和加锁重读机制保证高效、安全的读操作；**
- **通过不加锁和加锁两种方案控制跨段操作的的安全性。**



二. HashMap 线程不安全的典型表现

我们先回顾一下 HashMap。HashMap 是一个数组链表，当一个 key/Value 对被加入时，首先会通过 Hash 算法定位出这个键值对要被放入的桶，然后就把它插到相应桶中。如果这个桶中已经有元素了，那么发生了碰撞，这样会在这个桶中形成一个链表。一般来说，当有数据要插入 HashMap 时，都会检查容量有

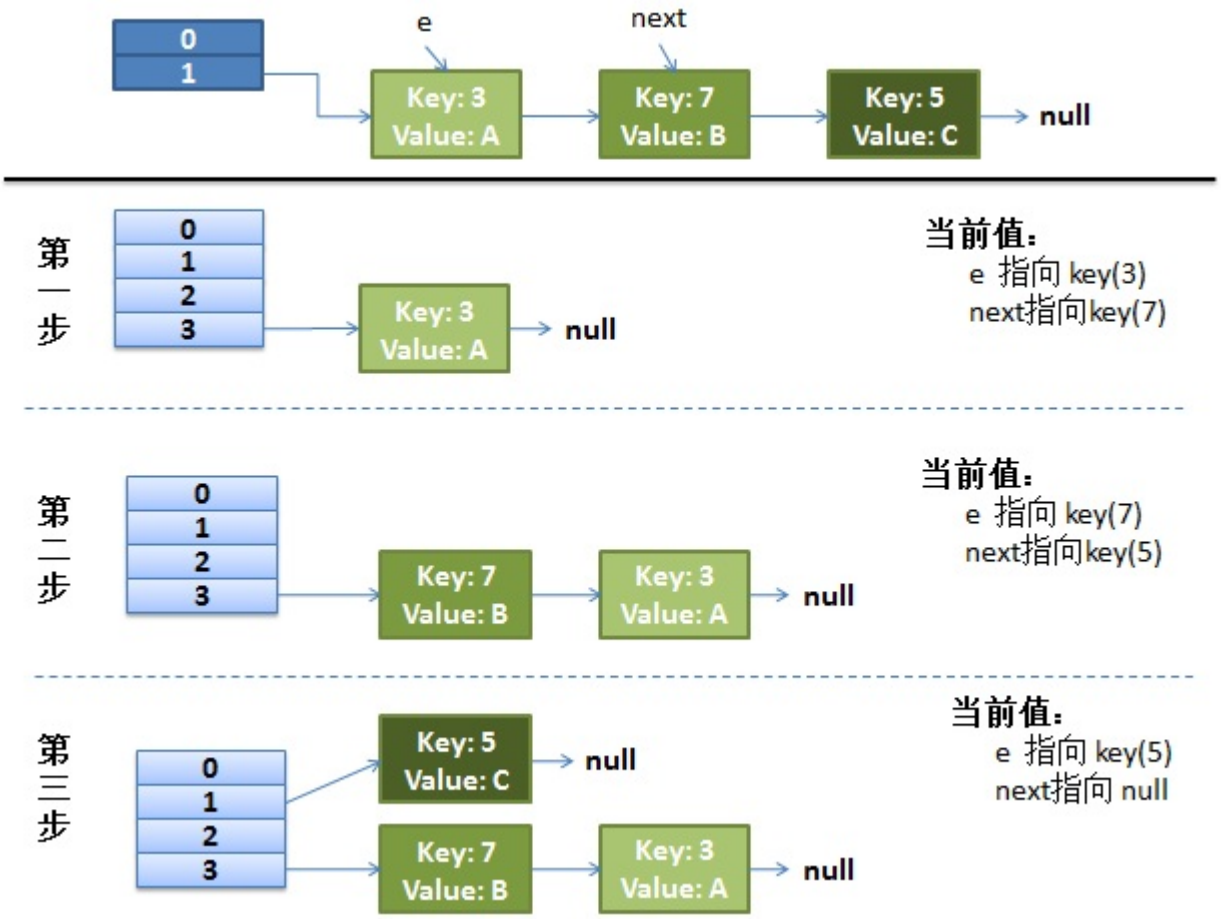
没有超过设定的 thredhold，如果超过，需要增大 HashMap 的尺寸，但是这样一来，就需要对整个 HashMap 里的节点进行重哈希操作。关于 HashMap 的重哈希操作本文不再详述，读者可以参考《Map 综述（一）：彻头彻尾理解 HashMap》一文。在此，笔者借助陈皓的《疫苗：JAVA HASHMAP 的死循环》一文说明 HashMap 线程不安全的典型表现 —— **死循环**。

HashMap 重哈希的关键源码如下：

```
/**
    * Transfers all entries from current table to newTable.
    */
   void transfer(Entry[] newTable) {

       // 将原数组 table 赋给数组 src
       Entry[] src = table;
       int newCapacity = newTable.length;

       // 将数组 src 中的每条链重新添加到 newTable 中
       for (int j = 0; j < src.length; j++) {
           Entry<K,V> e = src[j];
           if (e != null) {
               src[j] = null;   // src 回收

               // 将每条链的每个元素依次添加到 newTable 中相应的桶中
```

```
                do {
                    Entry<K,V> next = e.next;


                    // e.hash 指的是 hash(key.hashCode())的返回值;
                    // 计算在 newTable 中的位置，注意原来在同一条子链上的元素可能被分配到不同的桶中
                    int i = indexFor(e.hash, newCapacity);
                    e.next = newTable[i];
                    newTable[i] = e;
                    e = next;
                } while (e != null);
            }
        }
    }
```

1、单线程环境下的重哈希过程演示



单线程情况下，rehash 不会出现任何问题，如上图所示。假设 hash 算法就是最简单的 key mod table.length（也就是桶的个数）。最上面的是 old hash 表，其中的 Hash 表桶的个数为 2， 所以对于 key = 3、7、5 的键值对在 mod 2 以后都冲突在 table[1]这里了。接下来的三个步骤是，Hash 表 resize 成 4，然后对所有的键值对重哈希的过程。

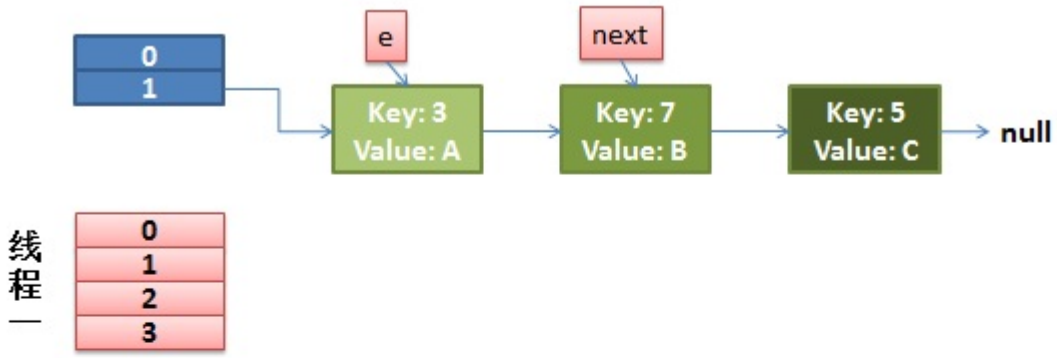
2、多线程环境下的重哈希过程演示

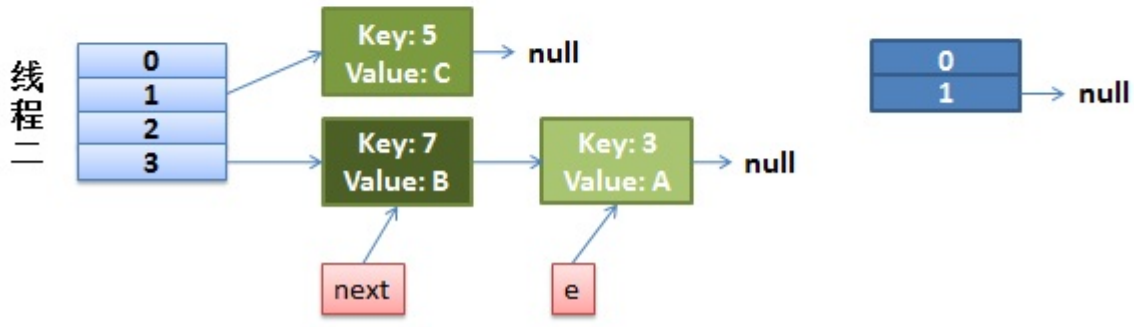
假设我们有两个线程，我用红色和浅蓝色标注了一下，被这两个线程共享的资源正是要被重哈希的原来 1 号桶中的 Entry 链。我们再回头看一下我们的 transfer 代码中的这个细节：

```
do {
```

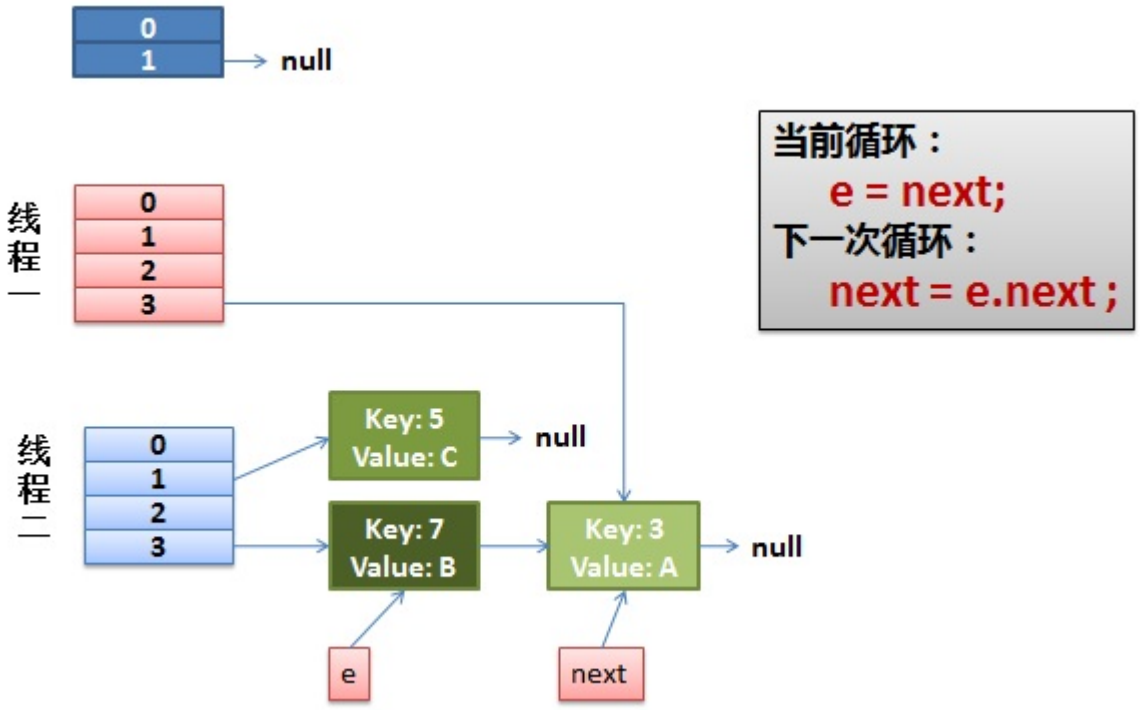
```
    Entry<K,V> next = e.next;       // <--假设线程一执行到这里就被调度挂起了
    int i = indexFor(e.hash, newCapacity);
    e.next = newTable[i];
    newTable[i] = e;
    e = next;
} while (e != null);
```

而我们的线程二执行完成了，于是我们有下面的这个样子：

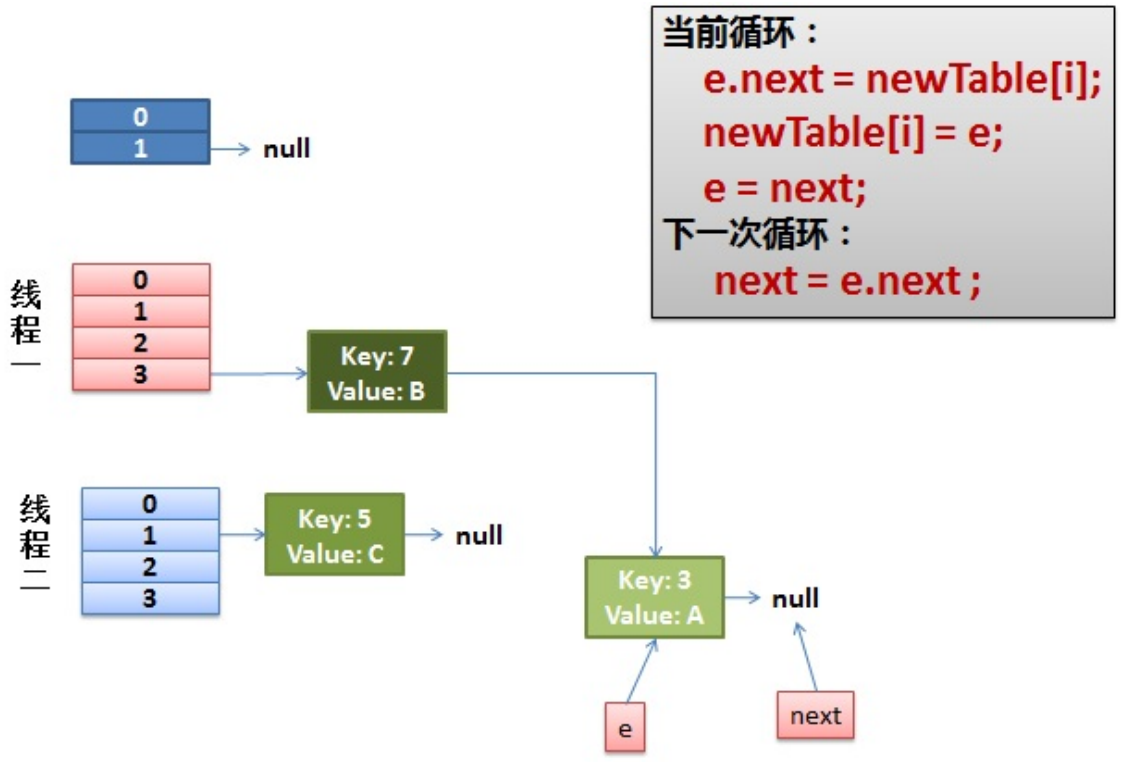




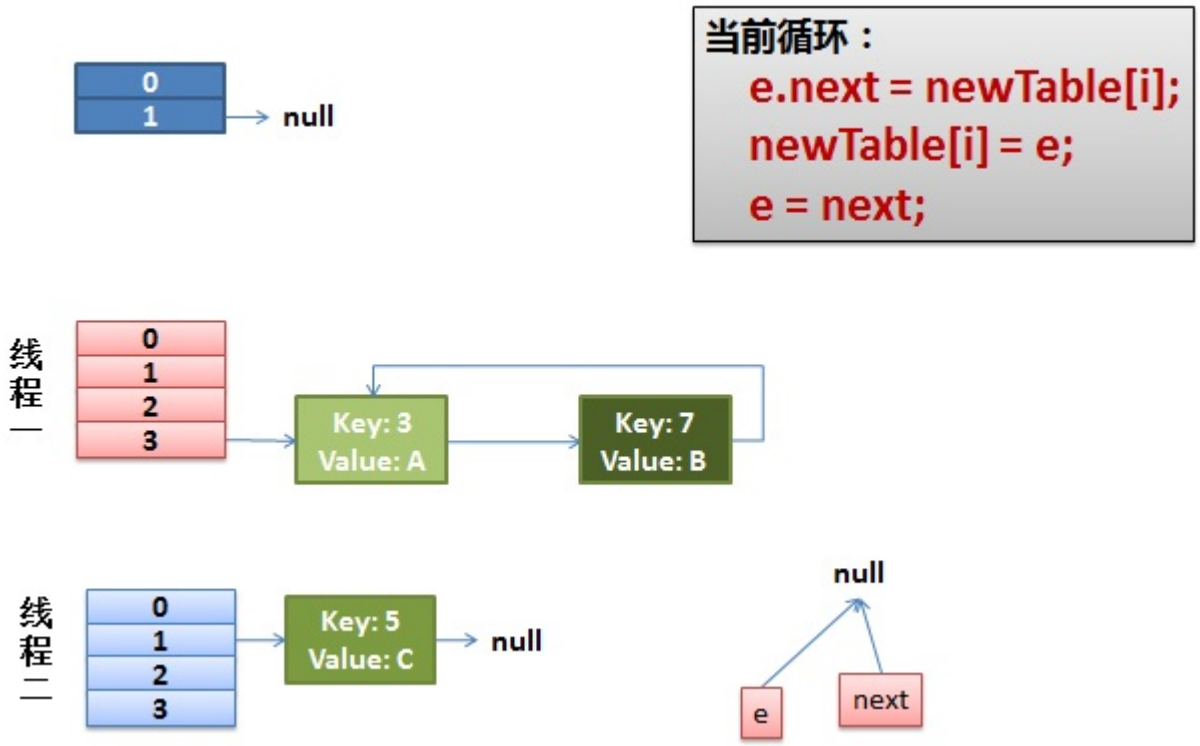
注意，在 Thread2 重哈希后，Thread1 的指针 e 和指针 next 分别指向了 Thread2 重组后的链表(e 指向了 key(3)，而 next 指向了 key(7))。此时，Thread1 被调度回来执行：Thread1 先是执行 newTalbe[i] = e;然后是 e = next，导致了 e 指向了 key(7)，而下一次循环的 next = e.next 导致了 next 指向了 key(3)，如下图所示：



这时，一切安好。Thread1 有条不紊的工作着：把 key(7)摘下来，放到 newTable[i]的第一个，然后把 e 和 next 往下移，如下图所示：



在此时，特别需要注意的是，当执行 e.next = newTable[i]后，会导致 key(3).next 指向了 key(7)，而此时的 key(7).next 已经指向了 key(3)，环形链表就这样出现了，如下图所示。于是，当我们的 Thread1 调用 HashMap.get(11)时，悲剧就出现了 —— Infinite Loop。



这是 HashMap 在并发环境下使用中最为典型的一个问题，就是在 HashMap 进行扩容重哈希时导致 Entry 链形成环。一旦 Entry 链中有环，势必会导致在同一个桶中进行插入、查询、删除等操作时陷入死循环。

三. ConcurrentHashMap 在 JDK 中的定义

为了更好的理解 ConcurrentHashMap 高并发的具体实现，我们先来了解它在 JDK 中的定义。ConcurrentHashMap 类中包含两个静态内部类 HashEntry

和 Segment，其中 HashEntry 用来封装具体的 K/V 对，是个典型的四元组；Segment 用来充当锁的角色，每个 Segment 对象守护整个 ConcurrentHashMap 的若干个桶 (可以把 Segment 看作是一个小型的哈希表)，其中每个桶是由若干个 HashEntry 对象链接起来的链表。总的来说，一个 ConcurrentHashMap 实例中包含由若干个 Segment 实例组成的数组，而一个 Segment 实例又包含由若干个桶，每个桶中都包含一条由若干个 HashEntry 对象链接起来的链表。特别地，ConcurrentHashMap 在默认并发级别下会创建 16 个 Segment 对象的数组，如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。

1、类结构定义

ConcurrentHashMap 继承了 AbstractMap 并实现了 ConcurrentMap 接口，其在 JDK 中的定义为：

```
public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>

       implements ConcurrentMap<K, V>, Serializable {



   ...

}
```

2、成员变量定义

与 HashMap 相比，ConcurrentHashMap 增加了两个属性用于定位段，分别是 segmentMask 和 segmentShift。此外，不同于 HashMap 的是，ConcurrentHashMap 底层结构是一个 Segment 数组，而不是 Object 数组，具体源码如下：

```
   /**

    * Mask value for indexing into segments. The upper bits of a
```

```
     * key's hash code are used to choose the segment.
     */
    final int segmentMask;  // 用于定位段，大小等于 segments 数组的大小减 1，是不可变的

    /**
     * Shift value for indexing within segments.
     */
    final int segmentShift;    // 用于定位段，大小等于 32(hash 值的位数)减去对 segments 的大小取以 2 为底的对数值，是不可变的

    /**
     * The segments, each of which is a specialized hash table
     */
    final Segment<K,V>[] segments;   // ConcurrentHashMap 的底层结构是一个 Segment 数组
```

3、段的定义：Segment

Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。每个 Segment 对象用来守护它的成员对象 table 中包含的若干个桶。table 是一个由 HashEntry 对象组成的链表数组，table 数组的每一个数组成员就是一个桶。

在 Segment 类中，count 变量是一个计数器，它表示每个 Segment 对象管理的 table 数组包含的 HashEntry 对象的个数，也就是 Segment 中包含的 HashEntry 对象的总数。特别需要注意的是，之所以在每个 Segment 对象中包

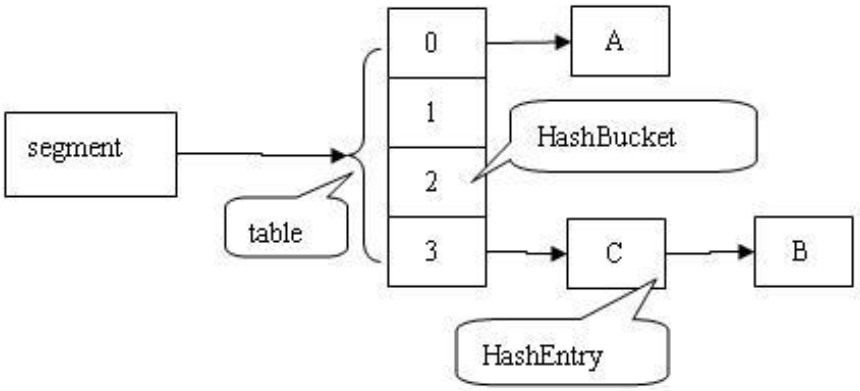
含一个计数器，而不是在 ConcurrentHashMap 中使用全局的计数器，是对 ConcurrentHashMap 并发性的考虑：**因为这样当需要更新计数器时，不用锁定整个 ConcurrentHashMap。**事实上，每次对段进行结构上的改变，如在段中进行增加/删除节点(修改节点的值不算结构上的改变)，都要更新 count 的值，此外，在 JDK 的实现中每次读取操作开始都要先读取 count 的值。特别需要注意的是，count 是 volatile 的，这使得对 count 的任何更新对其它线程都是立即可见的。modCount 用于统计段结构改变的次数，主要是为了检测对多个段进行遍历过程中某个段是否发生改变，这一点具体在谈到跨段操作时会详述。threashold 用来表示段需要进行重哈希的阈值。loadFactor 表示段的负载因子，其值等同于 ConcurrentHashMap 的负载因子的值。table 是一个典型的链表数组，而且也是 volatile 的，这使得对 table 的任何更新对其它线程也都是立即可见的。段(Segment)的定义如下：

```
    /**

     * Segments are specialized versions of hash tables.  This

     * subclasses from ReentrantLock opportunistically, just to

     * simplify some locking and avoid separate construction.

     */

    static final class Segment<K,V> extends ReentrantLock implements Ser
ializable {



        /**

         * The number of elements in this segment's region.

         */

        transient volatile int count;    // Segment 中元素的数量，可见的



        /**
```

```
         * Number of updates that alter the size of the table. This is
         * used during bulk-read methods to make sure they see a
         * consistent snapshot: If modCounts change during a traversal
         * of segments computing size or checking containsValue, then
         * we might have an inconsistent view of state so (usually)
         * must retry.
         */
        transient int modCount;  //对 count 的大小造成影响的操作的次数（比如 put 或者 remove 操作）


        /**
         * The table is rehashed when its size exceeds this threshold.
         * (The value of this field is always <tt>(int)(capacity *
         * loadFactor)</tt>.)
         */
        transient int threshold;      // 阈值，段中元素的数量超过这个值就会对 Segment 进行扩容


        /**
         * The per-segment table.
         */
```

```
        transient volatile HashEntry<K,V>[] table;  // 链表数组



        /**
         * The load factor for the hash table.  Even though this value
         * is same for all segments, it is replicated to avoid needing
         * links to outer object.
         * @serial
         */
        final float loadFactor;  // 段的负载因子，其值等同于 ConcurrentHash
Map 的负载因子



        ...
    }
```

我们知道，**ConcurrentHashMap 允许多个修改(写)操作并发进行，其关键在于使用了锁分段技术，它使用了不同的锁来控制对哈希表的不同部分进行的修改(写)，而 ConcurrentHashMap 内部使用段(Segment)来表示这些不同的部分。**实际上，每个段实质上就是一个小的哈希表，每个段都有自己的锁(Segment 类继承了 ReentrantLock 类)。这样，只要多个修改(写)操作发生在不同的段上，它们就可以并发进行。下图是依次插入 ABC 三个 HashEntry 节点后，Segment 的结构示意图：



4、基本元素：HashEntry

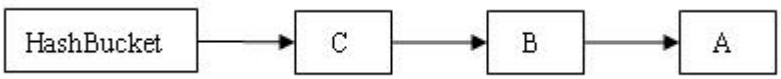
HashEntry 用来封装具体的键值对，是个典型的四元组。与 HashMap 中的 Entry 类似，HashEntry 也包括同样的四个域，分别是 key、hash、value 和 next。**不同的是，在 HashEntry 类中，key，hash 和 next 域都被声明为 final 的，value 域被 volatile 所修饰，因此 HashEntry 对象几乎是不可变的，这是 ConcurrentHashmap 读操作并不需要加锁的一个重要原因。**next 域被声明为 final 本身就意味着我们不能从 hash 链的中间或尾部添加或删除节点，因为这需要修改 next 引用值，因此所有的节点的修改只能从头部开始。对于 put 操作，可以一律添加到 Hash 链的头部。但是对于 remove 操作，可能需要从中间删除一个节点，这就需要将要删除节点的前面所有节点整个复制(重新 new)一遍，最后一个节点指向要删除结点的下一个结点(这在谈到 ConcurrentHashMap 的删除操作时还会详述)。特别地，**由于 value 域被 volatile 修饰，所以其可以确保被读线程读到最新的值，这是 ConcurrentHashmap 读操作并不需要加锁的另一个重要原因。**实际上，ConcurrentHashMap 完全允许多个读操作并发进行，读操作并不需要加锁。HashEntry 代表 hash 链中的一个节点，其结构如下所示：

```
/**
 * ConcurrentHashMap 中的 HashEntry 类
 *
 * ConcurrentHashMap list entry. Note that this is never exported
```

```
     * out as a user-visible Map.Entry.
     *
     * Because the value field is volatile, not final, it is legal wrt
     * the Java Memory Model for an unsynchronized reader to see null
     * instead of initial value when read via a data race.  Although a
     * reordering leading to this is not likely to ever actually
     * occur, the Segment.readValueUnderLock method is used as a
     * backup in case a null (pre-initialized) value is ever seen in
     * an unsynchronized access method.
     */
    static final class HashEntry<K,V> {
        final K key;                      // 声明 key 为 final 的
        final int hash;                   // 声明 hash 值为 final 的
        volatile V value;                 // 声明 value 被 volatile 所修饰
        final HashEntry<K,V> next;        // 声明 next 为 final 的


        HashEntry(K key, int hash, HashEntry<K,V> next, V value) {
            this.key = key;
            this.hash = hash;
            this.next = next;
```

```
            this.value = value;
        }

        @SuppressWarnings("unchecked")
        static final <K,V> HashEntry<K,V>[] newArray(int i) {
        return new HashEntry[i];
        }
    }
```

与 HashMap 类似，在 ConcurrentHashMap 中，如果在散列时发生碰撞，也会将碰撞的 HashEntry 对象链成一个链表。由于 HashEntry 的 next 域是 final 的，所以新节点只能在链表的表头处插入。下图是在一个空桶中依次插入 A，B，C 三个 HashEntry 对象后的结构图(由于只能在表头插入，所以链表中节点的顺序和插入的顺序相反)：



与 HashEntry 不同的是，HashMap 中的 Entry 类结构如下所示：

```
    /**
     * HashMap 中的 Entry 类
     */
    static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
```

```
        Entry<K,V> next;

        final int hash;



        /**

         * Creates new entry.

         */

        Entry(int h, K k, V v, Entry<K,V> n) {

            value = v;

            next = n;

            key = k;

            hash = h;

        }

        ...

    }
```

四. ConcurrentHashMap 的构造函数

ConcurrentHashMap 一共提供了五个构造函数，其中默认无参的构造函数和参数为 Map 的构造函数 为 Java Collection Framework 规范的推荐实现，其余三个构造函数则是 ConcurrentHashMap 专门提供的。

---

1 、 ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)

该构造函数意在构造一个具有指定容量、指定负载因子和指定段数目/并发级别(若不是 2 的幂次方，则会调整为 2 的幂次方)的空 ConcurrentHashMap，其相关源码如下：

```
    /**
     * Creates a new, empty map with the specified initial
     * capacity, load factor and concurrency level.
     *
     * @param initialCapacity the initial capacity. The implementation
     * performs internal sizing to accommodate this many elements.
     * @param loadFactor  the load factor threshold, used to control resi
zing.
     * Resizing may be performed when the average number of elements per
     * bin exceeds this threshold.
     * @param concurrencyLevel the estimated number of concurrently
     * updating threads. The implementation performs internal sizing
     * to try to accommodate this many threads.
     * @throws IllegalArgumentException if the initial capacity is
     * negative or the load factor or concurrencyLevel are
     * nonpositive.
     */
```

```
    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;           // 大小为 lg(ssize)
        int ssize = 1;           // 段的数目，segments 数组的大小(2 的幂次
方)
        while (ssize < concurrencyLevel) {
            ++sshift;
            ssize <<= 1;
        }
        segmentShift = 32 - sshift;      // 用于定位段
        segmentMask = ssize - 1;      // 用于定位段
        this.segments = Segment.newArray(ssize);   // 创建 segments 数组
```

```
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        int c = initialCapacity / ssize;     // 总的桶数/总的段数
        if (c * ssize < initialCapacity)
            ++c;
        int cap = 1;     // 每个段所拥有的桶的数目(2 的幂次方)
        while (cap < c)
            cap <<= 1;


        for (int i = 0; i < this.segments.length; ++i)      // 初始化 segments 数组
            this.segments[i] = new Segment<K,V>(cap, loadFactor);
    }
```

2、ConcurrentHashMap(int initialCapacity, float loadFactor)

该构造函数意在构造一个具有指定容量、指定负载因子和默认并发级别(16)的空 ConcurrentHashMap，其相关源码如下：

```
    /**
     * Creates a new, empty map with the specified initial capacity
     * and load factor and with the default concurrencyLevel (16).
     *
     * @param initialCapacity The implementation performs internal
```

```
     * sizing to accommodate this many elements.
     * @param loadFactor  the load factor threshold, used to control resi
zing.
     * Resizing may be performed when the average number of elements per
     * bin exceeds this threshold.
     * @throws IllegalArgumentException if the initial capacity of
     * elements is negative or the load factor is nonpositive
     *
     * @since 1.6
     */
    public ConcurrentHashMap(int initialCapacity, float loadFactor) {
        this(initialCapacity, loadFactor, DEFAULT_CONCURRENCY_LEVEL);  //
 默认并发级别为 16
    }
```

3、ConcurrentHashMap(int initialCapacity)

该构造函数意在构造一个具有指定容量、默认负载因子(0.75)和默认并发级别(16)的空 ConcurrentHashMap，其相关源码如下：

```
    /**
     * Creates a new, empty map with the specified initial capacity,
     * and with default load factor (0.75) and concurrencyLevel (16).
     *
```

```
     * @param initialCapacity the initial capacity. The implementation
     * performs internal sizing to accommodate this many elements.
     * @throws IllegalArgumentException if the initial capacity of
     * elements is negative.
     */
    public ConcurrentHashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR, DEFAULT_CONCURRENCY_L
EVEL);
    }
```

4、ConcurrentHashMap()

该构造函数意在构造一个具有默认初始容量(16)、默认负载因子(0.75)和默认并发级别(16)的空 ConcurrentHashMap，其相关源码如下：

```
    /**
     * Creates a new, empty map with a default initial capacity (16),
     * load factor (0.75) and concurrencyLevel (16).
     */
    public ConcurrentHashMap() {
        this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR, DEFAULT_CONC
URRENCY_LEVEL);
    }
```

5、ConcurrentHashMap(Map<? extends K, ? extends V> m)

该构造函数意在构造一个与指定 Map 具有相同映射的 ConcurrentHashMap，其初始容量不小于 16 (具体依赖于指定 Map 的大小)，负载因子是 0.75，并发级别是 16， 是 Java Collection Framework 规范推荐提供的，其源码如下：

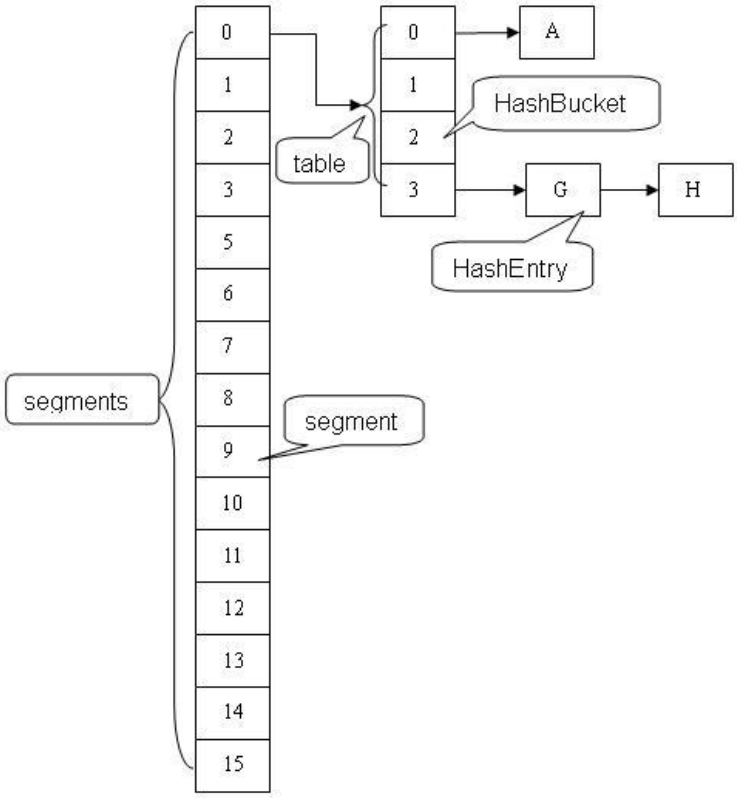
```
/**
 * Creates a new map with the same mappings as the given map.
 * The map is created with a capacity of 1.5 times the number
 * of mappings in the given map or 16 (whichever is greater),
 * and a default load factor (0.75) and concurrencyLevel (16).
 *
 * @param m the map
 */
public ConcurrentHashMap(Map<? extends K, ? extends V> m) {
    this(Math.max((int) (m.size() / DEFAULT_LOAD_FACTOR) + 1,
                  DEFAULT_INITIAL_CAPACITY),
         DEFAULT_LOAD_FACTOR, DEFAULT_CONCURRENCY_LEVEL);
    putAll(m);
}
```

在这里，我们提到了三个非常重要的参数：**初始容量**、**负载因子** 和 **并发级别**，这三个参数是影响 ConcurrentHashMap 性能的重要参数。从上述源码我们可以看出，ConcurrentHashMap 也正是通过 initialCapacity、loadFactor 和

concurrencyLevel 这三个参数进行构造并初始化 segments 数组、段偏移量 segmentShift、段掩码 segmentMask 和每个 segment 的。

五. ConcurrentHashMap 的数据结构

本质上，ConcurrentHashMap 就是一个 Segment 数组，而一个 Segment 实例则是一个小的哈希表。由于 Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色，这样，每个 Segment 对象就可以守护整个 ConcurrentHashMap 的若干个桶，其中每个桶是由若干个 HashEntry 对象链接起来的链表。**通过使用段(Segment)将 ConcurrentHashMap 划分为不同的部分，ConcurrentHashMap 就可以使用不同的锁来控制对哈希表的不同部分的修改，从而允许多个修改操作并发进行，这正是 ConcurrentHashMap 锁分段技术的核心内涵。**进一步地，如果把整个 ConcurrentHashMap 看作是一个父哈希表的话，那么每个 Segment 就可以看作是一个子哈希表，如下图所示：



注意，假设 ConcurrentHashMap 一共分为 2^n 个段，每个段中有 2^m 个桶，那么段的定位方式是将 key 的 hash 值的高 n 位与(2^n-1)相与。在定位到某个段后，再将 key 的 hash 值的低 m 位与(2^m-1)相与，定位到具体的桶位。

六. ConcurrentHashMap 的并发存取

在 ConcurrentHashMap 中，线程对映射表做读操作时，一般情况下不需要加锁就可以完成，对容器做结构性修改的操作(比如，put 操作、remove 操作等)才需要加锁。

1、用分段锁机制实现多个线程间的并发写操作: put(key, vlaue)

在 ConcurrentHashMap 中，典型结构性修改操作包括 put、remove 和 clear，下面我们首先以 put 操作为例说明对 ConcurrentHashMap 做结构性修改的过程。ConcurrentHashMap 的 put 操作对应的源码如下：

```
    /**
     * Maps the specified key to the specified value in this table.
     * Neither the key nor the value can be null.
     *
     * <p> The value can be retrieved by calling the <tt>get</tt> method
     * with a key that is equal to the original key.
     *
     * @param key key with which the specified value is to be associated
     * @param value value to be associated with the specified key
```

```
     * @return the previous value associated with <tt>key</tt>, or
     *         <tt>null</tt> if there was no mapping for <tt>key</tt>
     * @throws NullPointerException if the specified key or value is null
     */
    public V put(K key, V value) {
        if (value == null)
            throw new NullPointerException();
        int hash = hash(key.hashCode());
        return segmentFor(hash).put(key, hash, value, false);
    }
```

从上面的源码我们看到，**ConcurrentHashMap 不同于 HashMap，它既不允许 key 值为 null，也不允许 value 值为 null。**此外，我们还可以看到，实际上我们对 ConcurrentHashMap 的 put 操作被 ConcurrentHashMap 委托给特定的段来实现。也就是说，当我们向 ConcurrentHashMap 中 put 一个 Key/Value 对时，首先会获得 Key 的哈希值并对其再次哈希，然后根据最终的 hash 值定位到这条记录所应该插入的段，定位段的 segmentFor()方法源码如下：

```
    /**
     * Returns the segment that should be used for key with given hash
     * @param hash the hash code for the key
     * @return the segment
     */
    final Segment<K,V> segmentFor(int hash) {
        return segments[(hash >>> segmentShift) & segmentMask];
```

```
    }
```

segmentFor()方法根据传入的 hash 值向右无符号右移 segmentShift 位，然后和 segmentMask 进行与操作就可以定位到特定的段。在这里，假设 Segment 的数量(segments 数组的长度)是 2 的 n 次方(Segment 的数量总是 2 的倍数，具体见构造函数的实现)，那么 segmentShift 的值就是 32-n(hash 值的位数是 32)，而 segmentMask 的值就是 2^n-1（写成二进制的形式就是 n 个 1）。进一步地，我们就可以得出以下结论：**根据 key 的 hash 值的高 n 位就可以确定元素到底在哪一个 Segment 中。**紧接着，调用这个段的 put()方法来将目标 Key/Value 对插到段中，段的 put()方法的源码如下所示：

```
    V put(K key, int hash, V value, boolean onlyIfAbsent) {

            lock();    // 上锁

            try {

                int c = count;

                if (c++ > threshold) // ensure capacity

                    rehash();

                HashEntry<K,V>[] tab = table;    // table 是 Volatile 的

                int index = hash & (tab.length - 1);    // 定位到段中特定的桶

                HashEntry<K,V> first = tab[index];   // first 指向桶中链表的表头

                HashEntry<K,V> e = first;



                // 检查该桶中是否存在相同 key 的结点

                while (e != null && (e.hash != hash || !key.equals(e.key)))
```

```
                    e = e.next;



                V oldValue;
                if (e != null) {        // 该桶中存在相同 key 的结点
                    oldValue = e.value;
                    if (!onlyIfAbsent)
                        e.value = value;        // 更新 value 值
                }else {         // 该桶中不存在相同 key 的结点
                    oldValue = null;
                    ++modCount;     // 结构性修改，modCount 加 1
                    tab[index] = new HashEntry<K,V>(key, hash, first, valu
e);  // 创建 HashEntry 并将其链到表头
                    count = c;      //write-volatile，count 值的更新一定要放
在最后一步(volatile 变量)
                }
                return oldValue;    // 返回旧值(该桶中不存在相同 key 的结点，
则返回 null)
            } finally {
                unlock();       // 在 finally 子句中解锁
            }
        }
```

从源码中首先可以知道，ConcurrentHashMap 对 Segment 的 put 操作是加锁完成的。在第二节我们已经知道，Segment 是 ReentrantLock 的子类，因此 Segment 本身就是一种可重入的 Lock，所以我们可以直接调用其继承而来的 lock()方法和 unlock()方法对代码进行上锁/解锁。需要注意的是，这里的加锁操作是针对某个具体的 Segment，锁定的也是该 Segment 而不是整个 ConcurrentHashMap。因为插入键/值对操作只是在这个 Segment 包含的某个桶中完成，不需要锁定整个 ConcurrentHashMap。因此，其他写线程对另外 15 个 Segment 的加锁并不会因为当前线程对这个 Segment 的加锁而阻塞。故而 **相比较于 HashTable 和由同步包装器包装的 HashMap 每次只能有一个线程执行读或写操作，ConcurrentHashMap 在并发访问性能上有了质的提高。在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设置为 16），及任意数量线程的读操作。**

在将 Key/Value 对插入到 Segment 之前，首先会检查本次插入会不会导致 Segment 中元素的数量超过阈值 threshold，如果会，那么就先对 Segment 进行扩容和重哈希操作，然后再进行插入。重哈希操作暂且不表，稍后详述。第 8 和第 9 行的操作就是定位到段中特定的桶并确定链表头部的位置。第 12 行的 while 循环用于检查该桶中是否存在相同 key 的结点，如果存在，就直接更新 value 值；如果没有找到，则进入 21 行生成一个新的 HashEntry 并且把它链到该桶中链表的表头，然后再更新 count 的值(由于 count 是 volatile 变量，所以 count 值的更新一定要放在最后一步)。

到此为止，除了重哈希操作，ConcurrentHashMap 的 put 操作已经介绍完了。此外，在 ConcurrentHashMap 中，修改操作还包括 putAll()和 replace()。其中，putAll()操作就是多次调用 put 方法，而 replace()操作实现要比 put()操作简单得多，此不赘述。

2、ConcurrentHashMap 的重哈希操作 ：rehash()

上面叙述到，在 ConcurrentHashMap 中使用 put 操作插入 Key/Value 对之前，首先会检查本次插入会不会导致 Segment 中节点数量超过阈值 threshold，如果会，那么就先对 Segment 进行扩容和重哈希操作。**特别需要注意的是，ConcurrentHashMap 的重哈希实际上是对 ConcurrentHashMap 的某个段的**

重哈希，因此 ConcurrentHashMap 的每个段所包含的桶位自然也就不尽相同。

针对段进行 rehash()操作的源码如下：

```
    void rehash() {

          HashEntry<K,V>[] oldTable = table;    // 扩容前的 table

          int oldCapacity = oldTable.length;

          if (oldCapacity >= MAXIMUM_CAPACITY)   // 已经扩到最大容量，直接返回

              return;



          /*

           * Reclassify nodes in each list to new Map.  Because we are

           * using power-of-two expansion, the elements from each bin

           * must either stay at same index, or move with a power of two

           * offset. We eliminate unnecessary node creation by catching

           * cases where old nodes can be reused because their next

           * fields won't change. Statistically, at the default

           * threshold, only about one-sixth of them need cloning when

           * a table doubles. The nodes they replace will be garbage

           * collectable as soon as they are no longer referenced by any

           * reader thread that may be in the midst of traversing table

           * right now.
```

```
             */


            // 新创建一个 table，其容量是原来的 2 倍
            HashEntry<K,V>[] newTable = HashEntry.newArray(oldCapacity<<
1);
            threshold = (int)(newTable.length * loadFactor);   // 新的阈值
            int sizeMask = newTable.length - 1;     // 用于定位桶
            for (int i = 0; i < oldCapacity ; i++) {
                // We need to guarantee that any existing reads of old Map
 can
                //  proceed. So we cannot yet null out each bin.
                HashEntry<K,V> e = oldTable[i];  // 依次指向旧 table 中的每
个桶的链表表头


                if (e != null) {    // 旧 table 的该桶中链表不为空
                    HashEntry<K,V> next = e.next;
                    int idx = e.hash & sizeMask;   // 重哈希已定位到新桶
                    if (next == null)    //  旧 table 的该桶中只有一个节点
                        newTable[idx] = e;
                    else {
                        // Reuse trailing consecutive sequence at same slo
t
```

```
                        HashEntry<K,V> lastRun = e;
                        int lastIdx = idx;
                        for (HashEntry<K,V> last = next;
                             last != null;
                             last = last.next) {
                            int k = last.hash & sizeMask;
                            // 寻找 k 值相同的子链，该子链尾节点与父链的尾节点必须是同一个
                            if (k != lastIdx) {
                                lastIdx = k;
                                lastRun = last;
                            }
                        }

                        // JDK 直接将子链 lastRun 放到 newTable[lastIdx]桶中
                        newTable[lastIdx] = lastRun;

                        // 对该子链之前的结点，JDK 会挨个遍历并把它们复制到新桶中
                        for (HashEntry<K,V> p = e; p != lastRun; p = p.next) {
                            int k = p.hash & sizeMask;
```

```
                        HashEntry<K,V> n = newTable[k];

                        newTable[k] = new HashEntry<K,V>(p.key, p.has
h,

                                                         n, p.value);

                    }

                }

            }

        }

        table = newTable;   // 扩容完成

    }
```

其实 JDK 官方的注释已经解释的很清楚了。**由于扩容是按照 2 的幂次方进行的，所以扩展前在同一个桶中的元素，现在要么还是在原来的序号的桶里，或者就是原来的序号再加上一个 2 的幂次方，就这两种选择。**根据本文前面对 HashEntry 的介绍，我们知道链接指针 next 是 final 的，因此看起来我们好像只能把该桶的 HashEntry 链中的每个节点复制到新的桶中(这意味着我们要重新创建每个节点)，但事实上 JDK 对其做了一定的优化。因为在理论上原桶里的 HashEntry 链可能存在一条子链，这条子链上的节点都会被重哈希到同一个新的桶中，这样我们只要拿到该子链的头结点就可以直接把该子链放到新的桶中，从而避免了一些节点不必要的创建，提升了一定的效率。因此，JDK 为了提高效率，它会首先去查找这样的一个子链，而且这个子链的尾节点必须与原 hash 链的尾节点是同一个，那么就只需要把这个子链的头结点放到新的桶中，其后面跟的一串子节点自然也就连接上了。对于这个子链头结点之前的结点，JDK 会挨个遍历并把它们复制到新桶的链头(只能在表头插入元素)中。特别地，我们注意这段代码：

```
for (HashEntry<K,V> last = next;

     last != null;

     last = last.next) {
```

```
    int k = last.hash & sizeMask;

    if (k != lastIdx) {

        lastIdx = k;

        lastRun = last;

    }

}

newTable[lastIdx] = lastRun;
```

在该代码段中，JDK 直接将子链 lastRun 放到 newTable[lastIdx]桶中，难道这个操作不会覆盖掉 newTable[lastIdx]桶中原有的元素么？事实上，这种情形时不可能出现的，因为桶 newTable[lastIdx]在子链添加进去之前压根就不会有节点存在，这还是因为 table 的大小是按照 2 的幂次方的方式去扩展的。假设原来 table 的大小是 2^k 大小，那么现在新 table 的大小是 2^(k+1)大小，而定位桶的方式是:

```
// sizeMask = newTable.length - 1，即 sizeMask = 11...1，共 k+1 个 1。

int idx = e.hash & sizeMask;
```

因此这样得到的 idx 实际上就是 key 的 hash 值的低 k+1 位的值，而原 table 的 sizeMask 也全是 1 的二进制，不过总共是 k 位，那么原 table 的 idx 就是 key 的 hash 值的低 k 位的值。所以，**如果元素的 hashcode 的第 k+1 位是 0，那么元素在新桶的序号就是和原桶的序号是相等的；如果第 k+1 位的值是 1，那么元素在新桶的序号就是原桶的序号加上 2^k。**因此，JDK 直接将子链 lastRun 放到 newTable[lastIdx]桶中就没问题了，因为 newTable 中新序号处此时肯定是空的。

3、ConcurrentHashMap 的读取实现 ： get(Object key)

与 put 操作类似，当我们从 ConcurrentHashMap 中查询一个指定 Key 的键值对时，首先会定位其应该存在的段，然后查询请求委托给这个段进行处理，源码如下：

```
/**

     * Returns the value to which the specified key is mapped,

     * or {@code null} if this map contains no mapping for the key.

     *

     * <p>More formally, if this map contains a mapping from a key

     * {@code k} to a value {@code v} such that {@code key.equals(k)},

     * then this method returns {@code v}; otherwise it returns

     * {@code null}.  (There can be at most one such mapping.)

     *

     * @throws NullPointerException if the specified key is null

     */

    public V get(Object key) {

        int hash = hash(key.hashCode());

        return segmentFor(hash).get(key, hash);

    }
```

我们紧接着研读 Segment 中 get 操作的源码：

```
    V get(Object key, int hash) {

            if (count != 0) {            // read-volatile，首先读 count 变量
```

```
                HashEntry<K,V> e = getFirst(hash);   // 获取桶中链表头结点

                while (e != null) {

                    if (e.hash == hash && key.equals(e.key)) {    // 查找链
中是否存在指定 Key 的键值对

                        V v = e.value;

                        if (v != null)  // 如果读到 value 域不为 null，直接返
回

                            return v;

                        // 如果读到 value 域为 null，说明发生了重排序，加锁后重
新读取

                        return readValueUnderLock(e); // recheck

                    }

                    e = e.next;

                }

            }

            return null;  // 如果不存在，直接返回 null

        }
```

了解了 ConcurrentHashMap 的 put 操作后，上述源码就很好理解了。但是有一个情况需要特别注意，就是链中存在指定 Key 的键值对并且其对应的 Value 值为 null 的情况。在剖析 ConcurrentHashMap 的 put 操作时，我们就知道 ConcurrentHashMap 不同于 HashMap，它既不允许 key 值为 null，也不允许 value 值为 null。但是，此处怎么会存在键值对存在且的 Value 值为 null 的情形呢？JDK 官方给出的解释是，这种情形发生的场景是：初始化 HashEntry 时发生的指令重排序导致的，也就是在 HashEntry 初始化完成之前便返回了它的引用。这时，JDK 给出的解决之道就是加锁重读，源码如下：

```
/**
        * Reads value field of an entry under lock. Called if value
        * field ever appears to be null. This is possible only if a
        * compiler happens to reorder a HashEntry initialization with
        * its table assignment, which is legal under memory model
        * but is not known to ever occur.
        */
       V readValueUnderLock(HashEntry<K,V> e) {
           lock();
           try {
               return e.value;
           } finally {
               unlock();
           }
       }
```

4、ConcurrentHashMap 存取小结

在 ConcurrentHashMap 进行存取时，首先会定位到具体的段，然后通过对具体段的存取来完成对整个 ConcurrentHashMap 的存取。特别地，无论是 ConcurrentHashMap 的读操作还是写操作都具有很高的性能：在进行读操作时不需要加锁，而在写操作时通过锁分段技术只对所操作的段加锁而不影响客户端对其它段的访问。

七. ConcurrentHashMap 读操作不需要加锁的奥秘

在本文第二节，我们介绍到 HashEntry 对象几乎是不可变的(只能改变 Value 的值)，因为 HashEntry 中的 key、hash 和 next 指针都是 final 的。这意味着，我们不能把节点添加到链表的中间和尾部，也不能在链表的中间和尾部删除节点。这个特性可以保证：在访问某个节点时，这个节点之后的链接不会被改变，这个特性可以大大降低处理链表时的复杂性。与此同时，由于 HashEntry 类的 value 字段被声明是 Volatile 的，因此 Java 的内存模型就可以保证：某个写线程对 value 字段的写入马上就可以被后续的某个读线程看到。此外，由于在 ConcurrentHashMap 中不允许用 null 作为键和值，所以当读线程读到某个 HashEntry 的 value 为 null 时，便知道产生了冲突 —— 发生了重排序现象，此时便会加锁重新读入这个 value 值。这些特性互相配合，使得读线程即使在不加锁状态下，也能正确访问 ConcurrentHashMap。总的来说，ConcurrentHashMap 读操作不需要加锁的奥秘在于以下三点：

- **用 HashEntery 对象的不变性来降低读操作对加锁的需求；**
- **用 Volatile 变量协调读写线程间的内存可见性；**
- **若读时发生指令重排序现象，则加锁重读；**

由于我们在介绍 ConcurrentHashMap 的 get 操作时，已经介绍到了第三点，此不赘述。下面我们结合前两点分别从线程写入的两种角度 —— 对散列表做非结构性修改的操作和对散列表做结构性修改的操作来分析 ConcurrentHashMap 是如何保证高效读操作的。

1、用 HashEntery 对象的不变性来降低读操作对加锁的需求

非结构性修改操作只是更改某个 HashEntry 的 value 字段的值。由于对 Volatile 变量的写入操作将与随后对这个变量的读操作进行同步，所以当一个写

线程修改了某个 HashEntry 的 value 字段后，Java 内存模型能够保证读线程一定能读取到这个字段更新后的值。所以，写线程对链表的非结构性修改能够被后续不加锁的读线程看到。

对 ConcurrentHashMap 做结构性修改时，实质上是对某个桶指向的链表做结构性修改。如果能够确保在读线程遍历一个链表期间，写线程对这个链表所做的结构性修改不影响读线程继续正常遍历这个链表，那么读/写线程之间就可以安全并发访问这个 ConcurrentHashMap。在 ConcurrentHashMap 中，结构性修改操作包括 put 操作、remove 操作和 clear 操作，下面我们分别分析这三个操作：

- **clear 操作只是把 ConcurrentHashMap 中所有的桶置空，每个桶之前引用的链表依然存在，只是桶不再引用这些链表而已，而链表本身的结构并没有发生任何修改。**因此，正在遍历某个链表的读线程依然可以正常执行对该链表的遍历。
- 关于 put 操作的细节我们在上文已经单独介绍过，**我们知道 put 操作如果需要插入一个新节点到链表中时会在链表头部插入这个新节点，此时链表中的原有节点的链接并没有被修改。**也就是说，插入新的健/值对到链表中的操作不会影响读线程正常遍历这个链表。

下面来分析 remove 操作，先让我们来看看 remove 操作的源代码实现:

```
/**
 * Removes the key (and its corresponding value) from this map.
 * This method does nothing if the key is not in the map.
 *
 * @param  key the key that needs to be removed
 * @return the previous value associated with <tt>key</tt>, or
 *         <tt>null</tt> if there was no mapping for <tt>key</tt>
```

```
     * @throws NullPointerException if the specified key is null
     */
    public V remove(Object key) {
    int hash = hash(key.hashCode());
        return segmentFor(hash).remove(key, hash, null);
    }
```

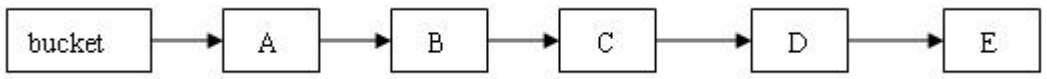
同样地，在 ConcurrentHashMap 中删除一个键值对时，首先需要定位到特定的段并将删除操作委派给该段。Segment 的 remove 操作如下所示：

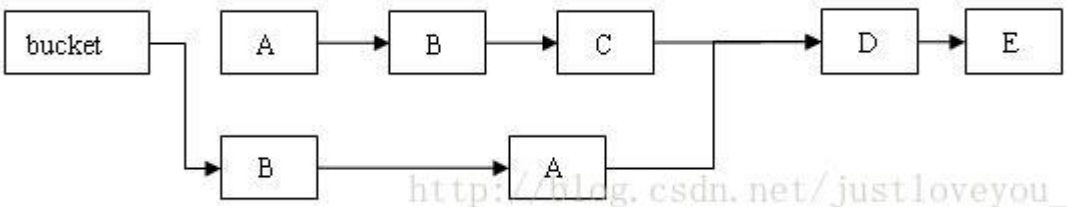
```
        /**
         * Remove; match on key only if value null, else match both.
         */
        V remove(Object key, int hash, Object value) {
            lock();     // 加锁
            try {
                int c = count - 1;
                HashEntry<K,V>[] tab = table;
                int index = hash & (tab.length - 1);        // 定位桶
                HashEntry<K,V> first = tab[index];
                HashEntry<K,V> e = first;
                while (e != null && (e.hash != hash || !key.equals(e.ke
y)))  // 查找待删除的键值对
```

```
                    e = e.next;



                V oldValue = null;
                if (e != null) {    // 找到
                    V v = e.value;
                    if (value == null || value.equals(v)) {
                        oldValue = v;
                        // All entries following removed node can stay
                        // in list, but all preceding ones need to be
                        // cloned.
                        ++modCount;
                        // 所有处于待删除节点之后的节点原样保留在链表中
                        HashEntry<K,V> newFirst = e.next;
                        // 所有处于待删除节点之前的节点被克隆到新链表中
                        for (HashEntry<K,V> p = first; p != e; p = p.next)
                            newFirst = new HashEntry<K,V>(p.key, p.hash,ne
wFirst, p.value);



                        tab[index] = newFirst;   // 将删除指定节点并重组后的
链重新放到桶中
                        count = c;      // write-volatile，更新 Volatile 变
量 count
```

```
                }

            }

            return oldValue;

        } finally {

            unlock();          // finally 子句解锁

        }

    }
```

Segment 的 remove 操作和前面提到的 get 操作类似，首先根据散列码找到具体的链表，然后遍历这个链表找到要删除的节点，最后把待删除节点之后的所有节点原样保留在新链表中，把待删除节点之前的每个节点克隆到新链表中。假设写线程执行 remove 操作，要删除链表的 C 节点，另一个读线程同时正在遍历这个链表，如下图所示：



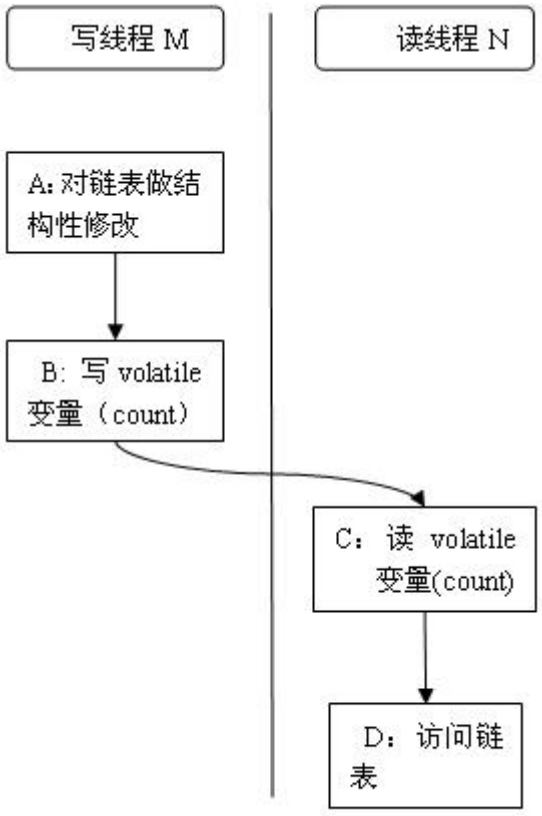


我们可以看出，删除节点 C 之后的所有节点原样保留到新链表中；删除节点 C 之前的每个节点被克隆到新链表中(它们在新链表中的链接顺序被反转了)。因此，**在执行 remove 操作时，原始链表并没有被修改，也就是说，读线程不会受同时执行 remove 操作的并发写线程的干扰。**

综合上面的分析我们可以知道，无论写线程对某个链表进行结构性修改还是非结构性修改，都不会影响其他的并发读线程对这个链表的访问。

2、用 Volatile 变量协调读写线程间的内存可见性

一般地，由于内存可见性问题，在未正确同步的情况下，对于写线程写入的值读线程可能并不能及时读到。下面以写线程 M 和读线程 N 来说明 ConcurrentHashMap 如何协调读/写线程间的内存可见性问题，如下图所示：



假设线程 M 在写入了 volatile 变量 count 后，线程 N 读取了这个 volatile 变量。根据 happens-before 关系法则中的程序次序法则，A appens-before 于 B，C happens-before D。根据 Volatile 法则，B happens-before C。结合传递性，则可得到：A appens-before 于 B； B appens-before C；C happens-before D。也就是说，写线程 M 对链表做的结构性修改对读线程 N 是可见的。虽然线程 N 是在未加锁的情况下访问链表，但 Java 的内存模型可以保证：**只要之前对链表**

做结构性修改操作的写线程 M 在退出写方法前写 volatile 变量 count，读线程 N 就能读取到这个 volatile 变量 count 的最新值。

事实上，ConcurrentHashMap 就是一个 Segment 数组，而每个 Segment 都有一个 volatile 变量 count 去统计 Segment 中的 HashEntry 的个数。并且，在 ConcurrentHashMap 中，所有不加锁读方法在进入读方法时，首先都会去读这个 count 变量。比如我们在上一节提到的 get 方法：

```
    V get(Object key, int hash) {

            if (count != 0) {            // read-volatile，首先读 count 变量

                HashEntry<K,V> e = getFirst(hash);   // 获取桶中链表头结点

                while (e != null) {

                    if (e.hash == hash && key.equals(e.key)) {    // 查找链中是否存在指定 Key 的键值对

                        V v = e.value;

                        if (v != null)  // 如果读到 value 域不为 null，直接返回

                            return v;

                        // 如果读到 value 域为 null，说明发生了重排序，加锁后重新读取

                        return readValueUnderLock(e); // recheck

                    }

                    e = e.next;

                }

            }
```

```
            return null;  // 如果不存在，直接返回 null

        }
```

3、小结

在 ConcurrentHashMap 中，所有执行写操作的方法（put、remove 和 clear）在对链表做结构性修改之后，在退出写方法前都会去写这个 count 变量；所有未加锁的读操作（get、contains 和 containsKey）在读方法中，都会首先去读取这个 count 变量。根据 Java 内存模型，对同一个 volatile 变量的写/读操作可以确保：写线程写入的值，能够被之后未加锁的读线程“看到”。这个特性和前面介绍的 HashEntry 对象的不变性相结合，使得在 ConcurrentHashMap 中读线程进行读取操作时基本不需要加锁就能成功获得需要的值。**这两个特性以及加锁重读机制的互相配合，不仅减少了请求同一个锁的频率（读操作一般不需要加锁就能够成功获得值），也减少了持有同一个锁的时间（只有读到 value 域的值为 null 时 ，读线程才需要加锁后重读）。**

八. ConcurrentHashMap 的跨段操作

在 ConcurrentHashMap 中，有些操作需要涉及到多个段，比如说 size 操作、containsValaue 操作等。以 size 操作为例，如果我们要统计整个 ConcurrentHashMap 里元素的大小，那么就必须统计所有 Segment 里元素的大小后求和。我们知道，Segment 里的全局变量 count 是一个 volatile 变量，那么在多线程场景下，我们是不是直接把所有 Segment 的 count 相加就可以得到整个 ConcurrentHashMap 大小了呢？显然不能，虽然相加时可以获取每个 Segment 的 count 的最新值，但是拿到之后可能累加前使用的 count 发生了变化，那么统计结果就不准了。所以最安全的做法，是在统计 size 的时候把所有 Segment 的 put，remove 和 clean 方法全部锁住，但是这种做法显然非常低效。那么，我们还是看一下 JDK 是如何实现 size()方法的吧：

```
/**

     * Returns the number of key-value mappings in this map.  If the
```

```
     * map contains more than <tt>Integer.MAX_VALUE</tt> elements, return
s

     * <tt>Integer.MAX_VALUE</tt>.

     *

     * @return the number of key-value mappings in this map

     */

    public int size() {

        final Segment<K,V>[] segments = this.segments;

        long sum = 0;

        long check = 0;

        int[] mc = new int[segments.length];

        // Try a few times to get accurate count. On failure due to

        // continuous async changes in table, resort to locking.

        for (int k = 0; k < RETRIES_BEFORE_LOCK; ++k) {

            check = 0;

            sum = 0;

            int mcsum = 0;

            for (int i = 0; i < segments.length; ++i) {

                sum += segments[i].count;

                mcsum += mc[i] = segments[i].modCount;  // 在统计 size 时记
录 modCount
```

```
            }
            if (mcsum != 0) {
                for (int i = 0; i < segments.length; ++i) {
                    check += segments[i].count;
                    if (mc[i] != segments[i].modCount) {  // 统计 size 后比较各段的 modCount 是否发生变化
                        check = -1; // force retry
                        break;
                    }
                }
            }
            if (check == sum)// 如果统计 size 前后各段的 modCount 没变，且两次得到的总数一致，直接返回
                break;
        }
        if (check != sum) { // Resort to locking all segments  // 加锁统计
            sum = 0;
            for (int i = 0; i < segments.length; ++i)
                segments[i].lock();
            for (int i = 0; i < segments.length; ++i)
                sum += segments[i].count;
```

```
            for (int i = 0; i < segments.length; ++i)
                segments[i].unlock();
        }
        if (sum > Integer.MAX_VALUE)
            return Integer.MAX_VALUE;
        else
            return (int)sum;
    }
```

size 方法主要思路是先在没有锁的情况下对所有段大小求和，这种求和策略最多执行 RETRIES_BEFORE_LOCK 次（默认是两次）：在没有达到 RETRIES_BEFORE_LOCK 之前，求和操作会不断尝试执行（这是因为遍历过程中可能有其它线程正在对已经遍历过的段进行结构性更新）；在超过 RETRIES_BEFORE_LOCK 之后，如果还不成功就在持有所有段锁的情况下再对所有段大小求和。事实上，在累加 count 操作过程中，之前累加过的 count 发生变化的几率非常小，所以 ConcurrentHashMap 的做法是先尝试 RETRIES_BEFORE_LOCK 次通过不锁住 Segment 的方式来统计各个 Segment 大小，如果统计的过程中，容器的 count 发生了变化，则再采用加锁的方式来统计所有 Segment 的大小。

那么，ConcurrentHashMap 是如何判断在统计的时候容器的段发生了结构性更新了呢？我们在前文中已经知道，Segment 包含一个 modCount 成员变量，在会引起段发生结构性改变的所有操作(put 操作、 remove 操作和 clean 操作)里，都会将变量 modCount 进行加 1，因此，**JDK 只需要在统计 size 前后比较 modCount 是否发生变化就可以得知容器的大小是否发生变化。**

至于 ConcurrentHashMap 的跨其他跨段操作，比如 contains 操作、containsValaue 操作等，其与 size 操作的实现原理相类似，此不赘述。

九. 更多

如果读者需要要深入了解 HashMap，请移步我的另一篇博文《Map 综述（一）：彻头彻尾理解 HashMap》。

更多关于哈希(Hash)和 equals 方法的介绍，请移步我的博文《Java 中的 ==, equals 与 hashCode 的区别与联系》。

更多关于 Java 内存模型与 Volatile 语义的介绍，请移步我的博文《Java 并发：volatile 关键字解析》。

更多关于 Java Collection Framework 各成员类的底层实现和原理的介绍，请见我的专栏《Java Collection Framework 源码剖析》。本专栏主要从源码角度剖析 Java Collection Framework 各成员类的底层实现原理和技巧，包括但不限于 List、Map 等经典容器类、ConcurrentHashMap 等并发容器类，并说明各容器类间的区别、联系以及应用场景，以便不断加深对 Java 容器框架的理解。

更多关于 Java SE 进阶 方面的内容，请关注我的专栏《Java SE 进阶之路》。本专栏主要研究 Java 基础知识、Java 源码和设计模式，从初级到高级不断总结、剖析各知识点的内在逻辑，贯穿、覆盖整个 Java 知识面，在一步步完善、提高把自己的同时，把对 Java 的所学所思分享给大家。万丈高楼平地起，基础决定你的上限，让我们携手一起勇攀 Java 之巅…