链滴

Java8 中 ConcurrentHashMap 是如何保证线程安全的

作者：Lord-X
原文链接：https://ld246.com/article/1564749911021
来源网站：链滴

rose rose ose rose 如果您觉得我的文章对您有帮助的话，记得在GitHub上star一波哈 rose rose GitHub_awesome-it-blog rose rose

---

HashMap是工作中使用频度非常高的一个K-V存储容器。在多线程环境下，使用HashMap是不安全，可能产生各种非期望的结果。

关于HashMap线程安全问题，可参考笔者的另一篇文章：

深入解读HashMap线程安全性问题

针对HashMap在多线程环境下不安全这个问题，HashMap的作者认为这并不是bug，而是应该使用程安全的HashMap。

目前有如下一些方式可以获得线程安全的HashMap：

- Collections.synchronizedMap
- HashTable
- ConcurrentHashMap

其中，前两种方式由于全局锁的问题，存在很严重的性能问题。所以，著名的并发编程大师Doug Le在JDK1.5的java.util.concurrent包下面添加了一大堆并发工具。其中就包含ConcurrentHashMap这线程安全的HashMap。

本文就来简单介绍一下ConcurrentHashMap的实现原理。

PS：基于JDK8

0 ConcurrentHashMap在JDK7中的回顾

ConcurrentHashMap在JDK7和JDK8中的实现方式上有较大的不同。首先我们先来大概回顾一下ConcurrentHashMap在JDK7中的原理是怎样的。

0.1 分段锁技术

针对HashTable会锁整个hash表的问题，ConcurrentHashMap提出了分段锁的解决方案。

分段锁的思想就是：锁的时候不锁整个hash表，而是只锁一部分。

如何实现呢？这就用到了ConcurrentHashMap中最关键的Segment。

ConcurrentHashMap中维护着一个Segment数组，每个Segment可以看做是一个HashMap。

而Segment本身继承了ReentrantLock，它本身就是一个锁。

在Segment中通过HashEntry数组来维护其内部的hash表。

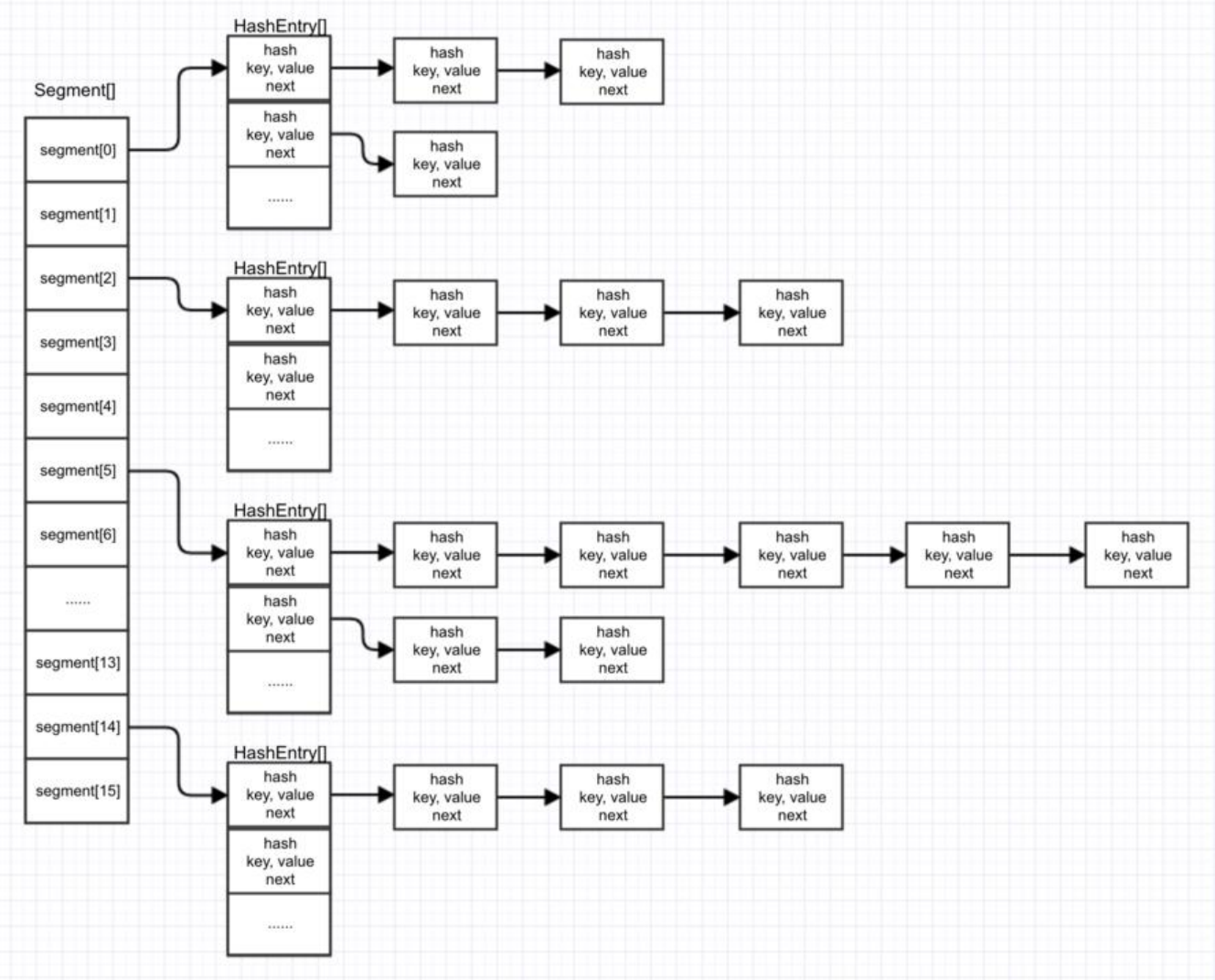
每个HashEntry就代表了map中的一个K-V，用HashEntry可以组成一个链表结构，通过next字段引到其下一个元素。

上述内容在源码中的表示如下：

```
public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
        implements ConcurrentMap<K, V>, Serializable {

    // ... 省略 ...
    /**
     * The segments, each of which is a specialized hash table.
     */
    final Segment<K,V>[] segments;

    // ... 省略 ...

    /**
     * Segment是ConcurrentHashMap的静态内部类
     *
     * 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 Serializable {
        // ... 省略 ...
        /**
         * The per-segment table. Elements are accessed via
         * entryAt/setEntryAt providing volatile semantics.
         */
        transient volatile HashEntry<K,V>[] table;
        // ... 省略 ...
    }
    // ... 省略 ...

    /**
```

```
     * ConcurrentHashMap list entry. Note that this is never exported
     * out as a user-visible Map.Entry.
     */
    static final class HashEntry<K,V> {
        final int hash;
        final K key;
        volatile V value;
        volatile HashEntry<K,V> next;
        // ... 省略 ...
    }
}
```

所以，JDK7中，ConcurrentHashMap的整体结构可以描述为下图这样子。



由上图可见，只要我们的hash值足够分散，那么每次put的时候就会put到不同的segment中去。

而segment自己本身就是一个锁，put的时候，当前segment会将自己锁住，此时其他线程无法操作个segment，

但不会影响到其他segment的操作。这个就是锁分段带来的好处。

0.2 线程安全的put

ConcurrentHashMap的put方法源码如下：

```
public V put(K key, V value) {
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    int hash = hash(key);
    int j = (hash >>> segmentShift) & segmentMask;

    // 根据key的hash定位出一个segment，如果指定index的segment还没初始化，则调用ensureSe
ment方法初始化
    if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
         (segments, (j << SSHIFT) + SBASE)) == null) //  in ensureSegment
        s = ensureSegment(j);
    // 调用segment的put方法
    return s.put(key, hash, value, false);
}
```

最终会调用segment的put方法，将元素put到HashEntry数组中，这里的注释中只给出锁相关的说明

```
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    // 因为segment本身就是一个锁
    // 这里调用tryLock尝试获取锁
    // 如果获取成功，那么其他线程都无法再修改这个segment
    // 如果获取失败，会调用scanAndLockForPut方法根据key和hash尝试找到这个node，如果不存
，则创建一个node并返回，如果存在则返回null
    // 查看scanAndLockForPut源码会发现他在查找的过程中会尝试获取锁，在多核CPU环境下，会
试64次tryLock()，如果64次还没获取到，会直接调用lock()
    // 也就是说这一步一定会获取到锁
    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))) {
                    oldValue = e.value;
                    if (!onlyIfAbsent) {
                        e.value = value;
                        ++modCount;
                    }
                    break;
                }
                e = e.next;
            }
            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)
                // 扩容
                rehash(node);
            else
                setEntryAt(tab, index, node);
            ++modCount;
            count = c;
            oldValue = null;
            break;
        }
    }
} finally {
    // 释放锁
    unlock();
}
return oldValue;
}
```

0.3 线程安全的扩容(Rehash)

HashMap的线程安全问题大部分出在扩容(rehash)的过程中。

ConcurrentHashMap的扩容只针对每个segment中的HashEntry数组进行扩容。

由上述put的源码可知，ConcurrentHashMap在rehash的时候是有锁的，所以在rehash的过程中，他线程无法对segment的hash表做操作，这就保证了线程安全。

1 JDK8中ConcurrentHashMap的初始化

以无参数构造函数为例，来看一下ConcurrentHashMap类初始化的时候会做些什么。

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
```

首先会执行静态代码块和初始化类变量。

主要会初始化以下这些类变量：

```
// Unsafe mechanics
private static final sun.misc.Unsafe U;
private static final long SIZECTL;
private static final long TRANSFERINDEX;
private static final long BASECOUNT;
private static final long CELLSBUSY;
private static final long CELLVALUE;
private static final long ABASE;
private static final int ASHIFT;

static {
    try {
        U = sun.misc.Unsafe.getUnsafe();
        Class<?> k = ConcurrentHashMap.class;
        SIZECTL = U.objectFieldOffset
```

```
            (k.getDeclaredField("sizeCtl"));
        TRANSFERINDEX = U.objectFieldOffset
            (k.getDeclaredField("transferIndex"));
        BASECOUNT = U.objectFieldOffset
            (k.getDeclaredField("baseCount"));
        CELLSBUSY = U.objectFieldOffset
            (k.getDeclaredField("cellsBusy"));
        Class<?> ck = CounterCell.class;
        CELLVALUE = U.objectFieldOffset
            (ck.getDeclaredField("value"));
        Class<?> ak = Node[].class;
        ABASE = U.arrayBaseOffset(ak);
        int scale = U.arrayIndexScale(ak);
        if ((scale & (scale - 1)) != 0)
            throw new Error("data type scale not a power of two");
        ASHIFT = 31 - Integer.numberOfLeadingZeros(scale);
    } catch (Exception e) {
        throw new Error(e);
    }
}
```

这里用到了Unsafe类，其中objectFieldOffset方法用于获取指定Field(例如sizeCtl)在内存中的偏移。

获取的这个偏移量主要用于干啥呢？不着急，在下文的分析中，遇到的时候再研究就好。

PS：关于Unsafe的介绍和使用，可以查看笔者的另一篇文章 Unsafe类的介绍和使用

2 内部数据结构

先来从源码角度看一下JDK8中是怎么定义的存储结构。

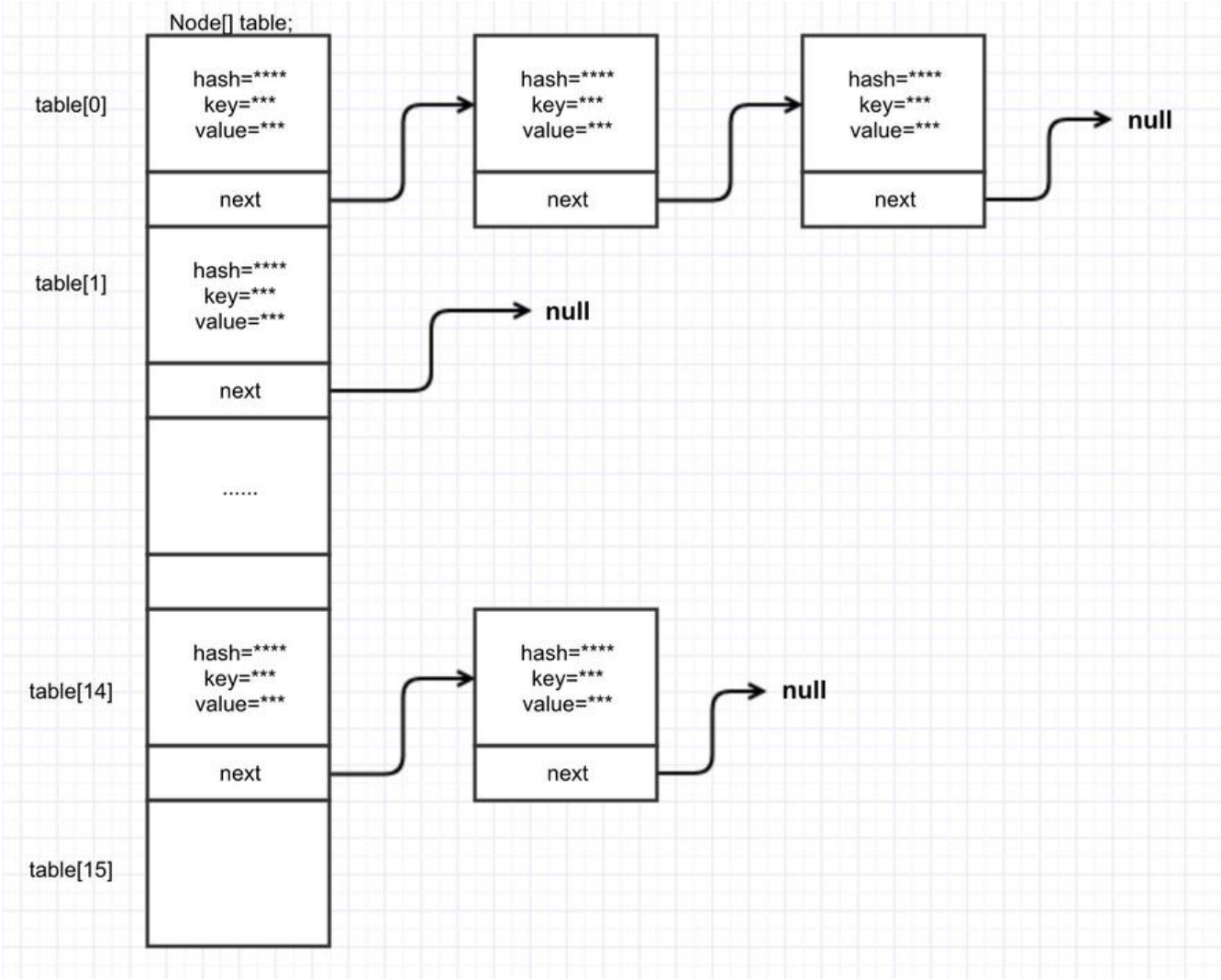
```
/**
 * The array of bins. Lazily initialized upon first insertion.
 * Size is always a power of two. Accessed directly by iterators.
 *
 * hash表，在第一次put数据的时候才初始化，他的大小总是2的倍数。
 */
transient volatile Node<K,V>[] table;

/**
 * 用来存储一个键值对
 *
 * Key-value entry.  This class is never exported out as a
 * user-mutable Map.Entry (i.e., one supporting setValue; see
 * MapEntry below), but can be used for read-only traversals used
 * in bulk tasks.  Subclasses of Node with a negative hash field
 * are special, and contain null keys and values (but are never
 * exported).  Otherwise, keys and vals are never null.
 */
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
```

```
    volatile Node<K,V> next;
}
```

可以发现，JDK8与JDK7的实现由较大的不同，JDK8中不在使用Segment的概念，他更像HashMap实现方式。

PS：关于HashMap的原理，可以参考笔者的另一篇文章 HashMap原理及内部存储结构

这个结构可以通过下图描述出来



3 线程安全的hash表初始化

由上文可知ConcurrentHashMap是用table这个成员变量来持有hash表的。

table的初始化采用了延迟初始化策略，他会在第一次执行put的时候初始化table。

put方法源码如下（省略了暂时不相关的代码）：

```
/**
 * 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 {@code get} 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 {@code key}, or
 *         {@code null} if there was no mapping for {@code key}
 * @throws NullPointerException if the specified key or value is null
 */
public V put(K key, V value) {
    return putVal(key, value, false);
}

/** Implementation for put and putIfAbsent */
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    // 计算key的hash值
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // 如果table是空，初始化之
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // 省略...
    }
    // 省略...
}
```

initTable源码如下

```
/**
 * Initializes table, using the size recorded in sizeCtl.
 */
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    // #1
    while ((tab = table) == null || tab.length == 0) {
        // sizeCtl的默认值是0，所以最先走到这的线程会进入到下面的else if判断中
        // #2
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        // 尝试原子性的将指定对象(this)的内存偏移量为SIZECTL的int变量值从sc更新为-1
        // 也就是将成员变量sizeCtl的值改为-1
        // #3
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                // 双重检查，原因会在下文分析
                // #4
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY; // 默认初始容量为16
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    // #5
                    table = tab = nt; // 创建hash表，并赋值给成员变量table
```

```
                sc = n - (n >>> 2);
            }
        } finally {
            // #6
            sizeCtl = sc;
        }
        break;
    }
  }
  return tab;
}
```

成员变量sizeCtl在ConcurrentHashMap中的其中一个作用相当于HashMap中的threshold，当has表中元素个数超过sizeCtl时，触发扩容；

他的另一个作用类似于一个标识，例如，当他等于-1的时候，说明已经有某一线程在执行hash表的初化了，一个小于-1的值表示某一线程正在对hash表执行resize。

这个方法首先判断sizeCtl是否小于0，如果小于0，直接将当前线程变为就绪状态的线程。

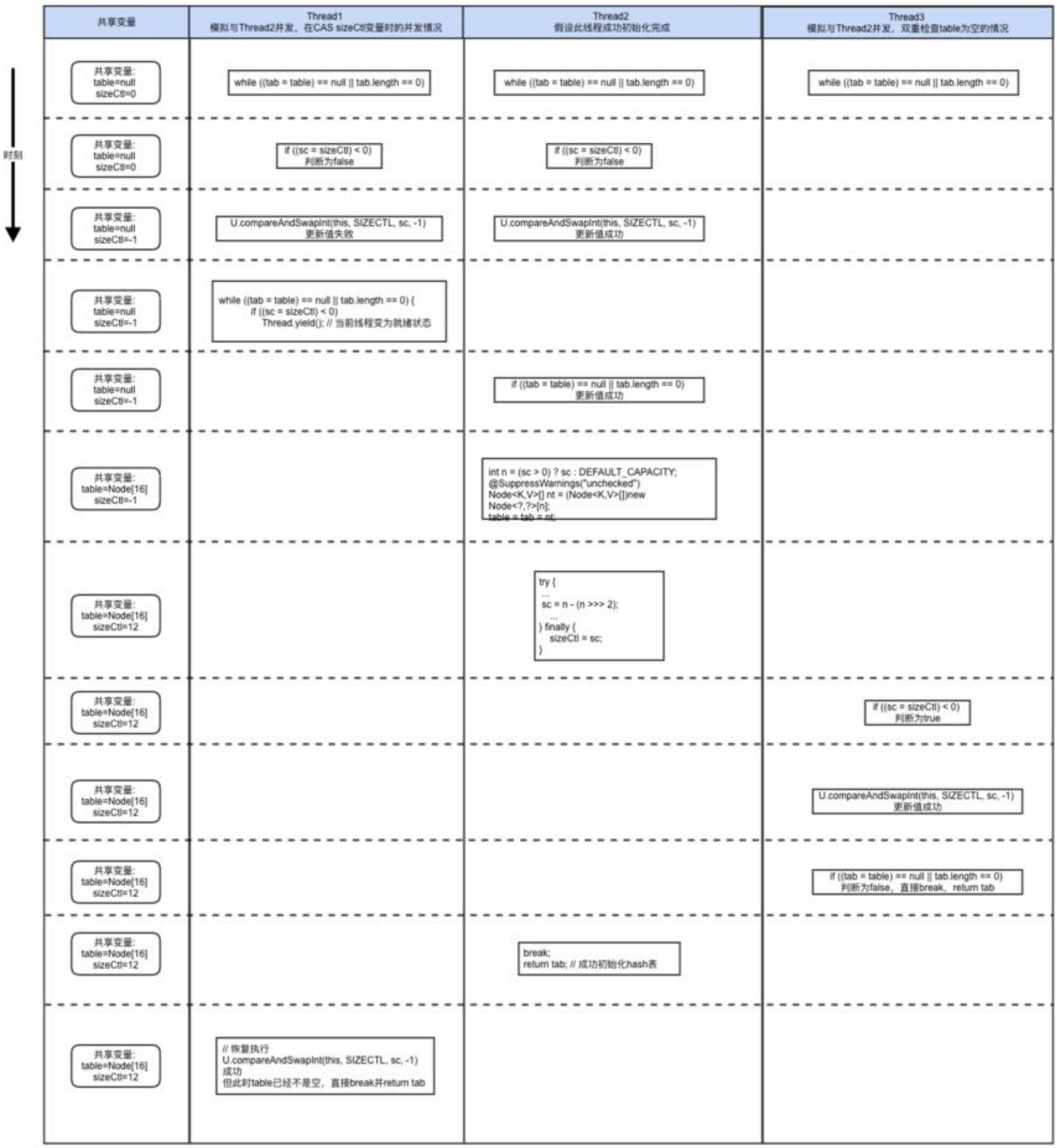
当sizeCtl大于等于0时，当前线程会尝试通过CAS的方式将sizeCtl的值修改为-1。修改失败的线程会入下一轮循环，判断sizeCtl<0了，被yield住；修改成功的线程会继续执行下面的初始化代码。

在new Node[]之前，要再检查一遍table是否为空，这里做双重检查的原因在于，如果另一个线程执完#1代码后挂起，此时另一个初始化的线程执行完了#6的代码，此时sizeCtl是一个大于0的值，那么切回这个线程执行的时候，是有可能重复初始化的。关于这个问题会在下图的并发场景中说明。

然后初始化hash表，并重新计算sizeCtl的值，最终返回初始化好的hash表。

下图详细说明了几种可能导致重复初始化hash表的并发场景，我们假设Thread2最终成功初始化has表。

- Thread1模拟的是CAS更新sizeCtl变量的并发场景
- Thread2模拟的是table的双重检查的必要性



由上图可以看出，在Thread1中如果不对sizeCtl的值更新做并发控制，Thread1是有可能走到new Node[]这一步的。

在Thread3中，如果不做双重判断，Thread3也会走到new Node[]这一步。

4 线程安全的put

put操作可分为以下两类

- 当前hash表对应当前key的index上没有元素时
- 当前hash表对应当前key的index上已经存在元素时(hash碰撞)

4.1 hash表上没有元素时

对应源码如下

```
else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
    if (casTabAt(tab, i, null,
                 new Node<K,V>(hash, key, value, null)))
        break;                   // no lock when adding to empty bin
}

static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
    return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
}

static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
                                    Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}
```

tabAt方法通过Unsafe.getObjectVolatile()的方式获取数组对应index上的元素，getObjectVolatile用于对应的内存偏移量上，是具备volatile内存语义的。

如果获取的是空，尝试用cas的方式在数组的指定index上创建一个新的Node。

4.2 hash碰撞时

对应源码如下

```
else {
    V oldVal = null;
    // 锁f是在4.1中通过tabAt方法获取的
    // 也就是说，当发生hash碰撞时，会以链表的头结点作为锁
    synchronized (f) {
        // 这个检查的原因在于：
        // tab引用的是成员变量table，table在发生了rehash之后，原来index上的Node可能会变
        // 这里就是为了确保在put的过程中，没有收到rehash的影响，指定index上的Node仍然是f
        // 如果不是f，那这个锁就没有意义了
        if (tabAt(tab, i) == f) {
            // 确保put没有发生在扩容的过程中，fh=-1时表示正在扩容
            if (fh >= 0) {
                binCount = 1;
                for (Node<K,V> e = f;; ++binCount) {
                    K ek;
                    if (e.hash == hash &&
                        ((ek = e.key) == key ||
                         (ek != null && key.equals(ek)))) {
                        oldVal = e.val;
                        if (!onlyIfAbsent)
                            e.val = value;
                        break;
                    }
                    Node<K,V> pred = e;
                    if ((e = e.next) == null) {
```

```
                    // 在链表后面追加元素
                    pred.next = new Node<K,V>(hash, key,
                                              value, null);
                    break;
                }
            }
        }
        else if (f instanceof TreeBin) {
            Node<K,V> p;
            binCount = 2;
            if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                          value)) != null) {
                oldVal = p.val;
                if (!onlyIfAbsent)
                    p.val = value;
            }
        }
    }
}
if (binCount != 0) {
    // 如果链表长度超过8个，将链表转换为红黑树，与HashMap相同，相对于JDK7来说，优化了
找效率
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
}
```

不同于JDK7中segment的概念，JDK8中直接用链表的头节点做为锁。

JDK7中，HashMap在多线程并发put的情况下可能会形成环形链表，ConcurrentHashMap通过这锁的方式，使同一时间只有有一个线程对某一链表执行put，解决了并发问题。

5 线程安全的扩容

put方法的最后一步是统计hash表中元素的个数，如果超过sizeCtl的值，触发扩容。

扩容的代码略长，可大致看一下里面的中文注释，再参考下面的分析。

其实我们主要的目的是弄明白ConcurrentHashMap是如何解决HashMap的并发问题的。

带着这个问题来看源码就好。关于HashMap存在的问题，参考本文一开始说的笔者的另一篇文章即。

其实HashMap的并发问题多半是由于put和扩容并发导致的。

这里我们就来看一下ConcurrentHashMap是如何解决的。

扩容涉及的代码如下：

```
/**
 * The array of bins. Lazily initialized upon first insertion.
 * Size is always a power of two. Accessed directly by iterators.
```

```
 * 业务中使用的hash表
 */
transient volatile Node<K,V>[] table;

/**
 * The next table to use; non-null only while resizing.
 * 扩容时才使用的hash表，扩容完成后赋值给table，并将nextTable重置为null。
 */
private transient volatile Node<K,V>[] nextTable;

/**
 * Adds to count, and if table is too small and not already
 * resizing, initiates transfer. If already resizing, helps
 * perform transfer if work is available.  Rechecks occupancy
 * after a transfer to see if another resize is already needed
 * because resizings are lagging additions.
 *
 * @param x the count to add
 * @param check if <0, don't check resize, if <= 1 only check if uncontended
 */
private final void addCount(long x, int check) {
    // ----- 计算键值对的个数 start -----
    CounterCell[] as; long b, s;
    if ((as = counterCells) != null ||
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            !(uncontended =
              U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        s = sumCount();
    }
    // ----- 计算键值对的个数 end -----
    // ----- 判断是否需要扩容 start -----
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        // 当上面计算出来的键值对个数超出sizeCtl时，触发扩容，调用核心方法transfer
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                // 如果有已经在执行的扩容操作，nextTable是正在扩容中的新的hash表
                // 如果并发扩容，transfer直接使用正在扩容的新hash表，保证了不会出现hash表覆盖的
况
```

```
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            // 更新sizeCtl的值，更新成功后为负数，扩容开始
            // 此时没有并发扩容的情况，transfer中会new一个新的hash表来扩容
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
            s = sumCount();
        }
    }
    // ----- 判断是否需要扩容 end -----
}

/**
 * Moves and/or copies the nodes in each bin to new table. See
 * above for explanation.
 */
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    int n = tab.length, stride;
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE; // subdivide range
    if (nextTab == null) {            // initiating
        try {
            @SuppressWarnings("unchecked")
            // 初始化新的hash表，大小为之前的2倍，并赋值给成员变量nextTable
            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
            nextTab = nt;
        } catch (Throwable ex) {      // try to cope with OOME
            sizeCtl = Integer.MAX_VALUE;
            return;
        }
        nextTable = nextTab;
        transferIndex = n;
    }
    int nextn = nextTab.length;
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab
    for (int i = 0, bound = 0;;) {
        Node<K,V> f; int fh;
        while (advance) {
            int nextIndex, nextBound;
            if (--i >= bound || finishing)
                advance = false;
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1;
                advance = false;
            }
            else if (U.compareAndSwapInt
                    (this, TRANSFERINDEX, nextIndex,
                     nextBound = (nextIndex > stride ?
                                  nextIndex - stride : 0))) {
                bound = nextBound;
```

```
            i = nextIndex - 1;
            advance = false;
        }
    }
    if (i < 0 || i >= n || i + n >= nextn) {
        int sc;
        // 扩容完成时，将成员变量nextTable置为null，并将table替换为rehash后的nextTable
        if (finishing) {
            nextTable = null;
            table = nextTab;
            sizeCtl = (n << 1) - (n >>> 1);
            return;
        }
        if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
            if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                return;
            finishing = advance = true;
            i = n; // recheck before commit
        }
    }
    else if ((f = tabAt(tab, i)) == null)
        advance = casTabAt(tab, i, null, fwd);
    else if ((fh = f.hash) == MOVED)
        advance = true; // already processed
    else {
        // 接下来是遍历每个链表，对每个链表的元素进行rehash
        // 仍然用头结点作为锁，所以在扩容的时候，无法对这个链表执行put操作
        synchronized (f) {
            if (tabAt(tab, i) == f) {
                Node<K,V> ln, hn;
                if (fh >= 0) {
                    int runBit = fh & n;
                    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;
                        }
                    }
                    if (runBit == 0) {
                        ln = lastRun;
                        hn = null;
                    }
                    else {
                        hn = lastRun;
                        ln = null;
                    }
                    for (Node<K,V> p = f; p != lastRun; p = p.next) {
                        int ph = p.hash; K pk = p.key; V pv = p.val;
                        if ((ph & n) == 0)
                            ln = new Node<K,V>(ph, pk, pv, ln);
                        else
                            hn = new Node<K,V>(ph, pk, pv, hn);
```

```
                    }
                    // setTabAt方法调用了Unsafe.putObjectVolatile来完成hash表元素的替换，具备vo
atile内存语义
                    setTabAt(nextTab, i, ln);
                    setTabAt(nextTab, i + n, hn);
                    setTabAt(tab, i, fwd);
                    advance = true;
                }
                else if (f instanceof TreeBin) {
                    TreeBin<K,V> t = (TreeBin<K,V>)f;
                    TreeNode<K,V> lo = null, loTail = null;
                    TreeNode<K,V> hi = null, hiTail = null;
                    int lc = 0, hc = 0;
                    for (Node<K,V> e = t.first; e != null; e = e.next) {
                        int h = e.hash;
                        TreeNode<K,V> p = new TreeNode<K,V>
                            (h, e.key, e.val, null, null);
                        if ((h & n) == 0) {
                            if ((p.prev = loTail) == null)
                                lo = p;
                            else
                                loTail.next = p;
                            loTail = p;
                            ++lc;
                        }
                        else {
                            if ((p.prev = hiTail) == null)
                                hi = p;
                            else
                                hiTail.next = p;
                            hiTail = p;
                            ++hc;
                        }
                    }
                    ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
                        (hc != 0) ? new TreeBin<K,V>(lo) : t;
                    hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
                        (lc != 0) ? new TreeBin<K,V>(hi) : t;
                    setTabAt(nextTab, i, ln);
                    setTabAt(nextTab, i + n, hn);
                    setTabAt(tab, i, fwd);
                    advance = true;
                }
            }
        }
    }
}
}
```

根据上述代码，对ConcurrentHashMap是如何解决HashMap并发问题这一疑问进行简要说明。

- 首先new一个新的hash表(nextTable)出来，大小是原来的2倍。后面的rehash都是针对这个新的hah表操作，不涉及原hash表(table)。
- 然后会对原hash表(table)中的每个链表进行rehash，此时会尝试获取头节点的锁。这一步就保证

在rehash的过程中不能对这个链表执行put操作。

- 通过sizeCtl控制，使扩容过程中不会new出多个新hash表来。
- 最后，将所有键值对重新rehash到新表(nextTable)中后，用nextTable将table替换。这就避免了HashMap中get和扩容并发时，可能get到null的问题。
- 在整个过程中，共享变量的存储和读取全部通过volatile或CAS的方式，保证了线程安全。

6 总结

多线程环境下，对共享变量的操作一定要小心。要充分从Java内存模型的角度考虑问题。

ConcurrentHashMap中大量的用到了Unsafe类的方法，我们自己虽然也能拿到Unsafe的实例，但生产中不建议这么做。

多数情况下，我们可以通过并发包中提供的工具来实现，例如Atomic包下面的可以用来实现CAS操作lock包下可以用来实现锁相关的操作。

善用线程安全的容器工具，例如ConcurrentHashMap、CopyOnWriteArrayList、ConcurrentLinkeQueue等，因为我们在工作中无法像ConcurrentHashMap这样通过Unsafe的getObjectVolatile和seObjectVolatile原子性的更新数组中的元素，所以这些并发工具是很重要的。