1 ConcurrentHashMap

note author: heylichen@qq.com
JDK version: JDK 8

1.1 Basics

```
public class ConcurrentHashMap<K,V> extends AbstractMap<K,V>
    implements ConcurrentMap<K,V>, Serializable {
    private static final long serialVersionUID = 7249069246763182397L;

 //用于计算hash，低位，第0-30位为1，其余位是0.
 static final int HASH_BITS = 0x7fffffff; // usable bits of normal node hash
 /**
  * Spreads (XORs) higher bits of hash to lower and also forces top
  * bit to 0. Because the table uses power-of-two masking, sets of
  * hashes that vary only in bits above the current mask will
  * always collide. (Among known examples are sets of Float keys
  * holding consecutive whole numbers in small tables.)  So we
  * apply a transform that spreads the impact of higher bits
  * downward. There is a tradeoff between speed, utility, and
  * quality of bit-spreading. Because many common sets of hashes
  * are already reasonably distributed (so don't benefit from
  * spreading), and because we use trees to handle large sets of
  * collisions in bins, we just XOR some shifted bits in the
  * cheapest possible way to reduce systematic lossage, as well as
  * to incorporate impact of the highest bits that would otherwise
  * never be used in index calculations because of table bounds.
  */
 static final int spread(int h) {
     return (h ^ (h >>> 16)) & HASH_BITS;
 }
 //以下3个方法与HashMap一样，见HashMap解读
 private static final int tableSizeFor(int c);
 static Class<?> comparableClassFor(Object x);
 static int compareComparables(Class<?> kc, Object k, Object x);
```

1.2 Instance Fields

注意大部分 field 都是 volatile, 利用其可见性和 happens-before 特性。为可读性考虑，代码所在行数微做调整，内容与源代码一致。

```
    /**
     * The array of bins. Lazily initialized upon first insertion.
     * Size is always a power of two. Accessed directly by iterators.
     */
    // buckets数组
    transient volatile Node<K,V>[] table;

    /**
     * The next table to use; non-null only while resizing.
     */
     // nextTable和transferIndex用于resize，详见resize逻辑分析
    private transient volatile Node<K,V>[] nextTable;

    /**
     * The next table index (plus one) to split while resizing.
```

```
     */
     // 详见resize逻辑分析
    private transient volatile int transferIndex;

    /**
     * Table initialization and resizing control.  When negative, the
     * table is being initialized or resized: -1 for initialization,
     * else -(1 + the number of active resizing threads).  Otherwise,
     * when table is null, holds the initial table size to use upon
     * creation, or 0 for default. After initialization, holds the
     * next element count value upon which to resize the table.
     */
     // 1) when table is null, holds the initial table size to use upon creation
     // 2) After initialization (when >=0), holds the next element count value upon which to
     //    resize the table.
     // 3) when <0, table is being initialized or resized
     //   3.1) -1 for initialization, 正在初始化
     //   3.2) -(1 + the number of active resizing threads), 正在resize
    private transient volatile int sizeCtl;

    /**
    * Base counter value, used mainly when there is no contention,
    * but also as a fallback during table initialization
    * races. Updated via CAS.
    */
   //baseCount, cellsBusy, counterCells 三个变量用于Map.size计数。
   // 此处不展开，详见size逻辑分析
   private transient volatile long baseCount;

    /**
     * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
     */
    private transient volatile int cellsBusy;

    /**
     * Table of counter cells. When non-null, size is a power of 2.
     */
    private transient volatile CounterCell[] counterCells;

    // views
    private transient KeySetView<K,V> keySet;
    private transient ValuesView<K,V> values;
    private transient EntrySetView<K,V> entrySet;
```

1.3 Unsafe to access Instance Fields

1.3.1 Unsafe Api introduction

这里只介绍 ConcurrentHashMap 用到的 Unsafe Api, 详细了解 Unsafe 可参考文章"Java 魔法类: Unsafe 应用解析 - 美团技术团队".

对字段的访问。

```
// 对象访问相关API ----------------------------------------
//返回对象成员属性在内存地址相对于此对象的内存地址的偏移量
public native long objectFieldOffset(Field f);
//获得给定对象的指定地址偏移量的值，与此类似操作还有：getInt，getDouble，getLong，getChar等
public native Object getObject(Object o, long offset);
//给定对象的指定地址偏移量设值，与此类似操作还有：putInt，putDouble，putLong，putChar等
public native void putObject(Object o, long offset, Object x);
//从对象的指定偏移量处获取变量的引用，使用volatile的加载语义
public native Object getObjectVolatile(Object o, long offset);
//存储变量的引用到对象的指定的偏移量处，使用volatile的存储语义
```

```
public native void putObjectVolatile(Object o, long offset, Object x);

/**
* CAS 具有volatile更新语义
* @param o          包含要修改field的对象
* @param offset     对象中某field的偏移量
* @param expected   期望值
* @param update     更新值
* @return          true | false
*/
public final native boolean compareAndSwapObject(Object o, long offset,
      Object expected, Object update);

public final native boolean compareAndSwapInt(Object o, long offset, int expected,int update);

public final native boolean compareAndSwapLong(Object o, long offset,
       long expected, long update);
```

用法示例

```
//字段偏移
private static final long SIZECTL;

U = sun.misc.Unsafe.getUnsafe();
SIZECTL = U.objectFieldOffset(k.getDeclaredField("sizeCtl"));
U.compareAndSwapInt(this, SIZECTL, sc, sc + 1); //CAS操作
```

CAS 操作的 volatile 语义

CAS 操作 (Unsafe.compareAndSwap*, atomic 包下面的各种实现类的 compareAndSet) 应该具有原子性和 volatile 语义。我找了下官方文档的相关说明。

在 java.util.concurrent.atomic 包的 javaDoc 文档 (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html) 中，有如下描述。

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in The Java Language Specification (17.4 Memory Model):
compareAndSet and all other read-and-update operations such as getAndIncrement have the **memory effects of both reading and writing volatile variables**.

看一下 atomic 包里相关类 compareAnsSet 的实现。AtomicReferenceFieldUpdaterImpl 的代码

```
public final boolean compareAndSet(T obj, V expect, V update) {
  accessCheck(obj);
  valueCheck(update);
  //U就是sun.misc.Unsafe
  return U.compareAndSwapObject(obj, offset, expect, update);
}
```

由此可确认，Unsafe.compareAndSwap* 具有 volatile 更新语义.

Unsafe array access

```
// 返回数组类型的第一个元素的偏移地址(基础偏移地址)。如果arrayIndexScale方法返回的比例因子不为
// 0，你可以通过结合基础偏移地址和比例因子访问数组的所有元素。Unsafe中已经初始化了很多类似的常
// 量如ARRAY_BOOLEAN_BASE_OFFSET等。
public native int arrayBaseOffset(Class<?> arrayClass);
// 返回数组类型的比例因子(其实就是数据中元素偏移地址的增量，因为数组中的元素的地址是连续的)。
// 此方法不适用于数组类型为"narrow"类型的数组，"narrow"类型的数组类型使用此方法会返回0(这里
// narrow应该是狭义的意思，但是具体指哪些类型暂时不明确，笔者查了很多资料也没找到结果)。
// Unsafe中已经初始化了很多类似的常量如ARRAY_BOOLEAN_INDEX_SCALE等。
public native int arrayIndexScale(Class<?> arrayClass);
```

例：arrayBaseOffset，获取数组第一个元素的偏移地址。arrayIndexScale，数组中元素的大小，占用多少个字节。arrayBaseOffset 与 arrayIndexScale 配合起来使用，就可以定位数组中每个元素在内存中的位置。

$$floor(log_2(x)) = 31 - numberOfLeadingZeros(x)$$

如果这是一个 int 型数组，indexScale 等于 4，那么 shift 值为 2，所以乘以 4 和向左移 2 位，结果是一样的。

1.3.2 constant offset fields in source

```
// Unsafe mechanics
private static final sun.misc.Unsafe U;
private static final long SIZECTL;        //sizeCtl field offset
private static final long TRANSFERINDEX;  //transferIndex
private static final long BASECOUNT;      //baseCount
private static final long CELLSBUSY;      //cellsBusy
private static final long CELLVALUE;      //value field offset in CounterCell
private static final long ABASE;          //Node[] offset
private static final int ASHIFT;          //Node[] index scale

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);
    }
}

//对于ABASE和ASHIFT，使用时如下
U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
//等价于
U.getObjectVolatile(tab, ((long)i * arrayIndexScale) + ABASE);
```

1.3.3 Table element access

```
/*
 * Volatile access methods are used for table elements as well as
 * elements of in-progress next table while resizing.  All uses of
 * the tab arguments must be null checked by callers.  All callers
 * also paranoically precheck that tab's length is not zero (or an
 * equivalent check), thus ensuring that any index argument taking
 * the form of a hash value anded with (length - 1) is a valid
 * index.  Note that, to be correct wrt arbitrary concurrency
 * errors by users, these checks must operate on local variables,
 * which accounts for some odd-looking inline assignments below.
 * Note that calls to setTabAt always occur within locked regions,
 * and so in principle require only release ordering, not
 * full volatile semantics, but are currently coded as volatile
 * writes to be conservative.
```

```
 */

@SuppressWarnings("unchecked")
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);
}

static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {
    U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);
}
```

1.4 constructors

```
/**
 * Creates a new, empty map with the default initial table size (16).
 */
public ConcurrentHashMap() {}

/**
 * Creates a new, empty map with an initial table size
 * accommodating the specified number of elements without the need
 * to dynamically resize.
 *
 * @param initialCapacity The implementation performs internal
 * sizing to accommodate this many elements.
 * @throws IllegalArgumentException if the initial capacity of
 * elements is negative
 */
public ConcurrentHashMap(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY : //MAXIMUM_CAPACITY OR initialCapacity*1.5+1
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap; //初始大小存入sizeCtl, 详见sizeCtl注释
}

/**
 * Creates a new map with the same mappings as the given map.
 *
 * @param m the map
 */
public ConcurrentHashMap(Map<? extends K, ? extends V> m) {
    this.sizeCtl = DEFAULT_CAPACITY;
    putAll(m);
}

/**
 * Creates a new, empty map with an initial table size based on
 * the given number of elements ({@code initialCapacity}), table
 * density ({@code loadFactor}), and number of concurrently
 * updating threads ({@code concurrencyLevel}).
 *
 * @param initialCapacity the initial capacity. The implementation
 * performs internal sizing to accommodate this many elements,
 * given the specified load factor.
 * @param loadFactor the load factor (table density) for
 * establishing the initial table size
```

```
     * @param concurrencyLevel the estimated number of concurrently
     * updating threads. The implementation may use this value as
     * a sizing hint.
     * @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 > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)
            throw new IllegalArgumentException();
        if (initialCapacity < concurrencyLevel)   // Use at least as many bins
            initialCapacity = concurrencyLevel;   // as estimated threads
        long size = (long)(1.0 + (long)initialCapacity / loadFactor);
        int cap = (size >= (long)MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : tableSizeFor((int)size);
        this.sizeCtl = cap;
    }
```

1.5 get

```
    /**
     * 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) {
        Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
        //key不允许为null spread计算内部使用的hash值
        int h = spread(key.hashCode());
        if ((tab = table) != null && (n = tab.length) > 0 && (e = tabAt(tab, (n - 1) & h)) != null) {
            //如果有对应bucket节点
            if ((eh = e.hash) == h) {
                if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                    return e.val; //hash相等且key相等，则找到节点
            }
            else if (eh < 0) //hash为负数，则节点为TreeBin或者ForwardingNode，代理给node.find
                return (p = e.find(h, key)) != null ? p.val : null;

            //以上条件不满足，则e为普通链表节点，继续找
            while ((e = e.next) != null) {
                if (e.hash == h && ((ek = e.key) == key || (ek != null && key.equals(ek))))
                    return e.val; //对比hash和key的范式
            }
        }
        return null;
    }
```

1.6 put

```
    /**
     * 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();//不允许null
      int hash = spread(key.hashCode());
      int binCount = 0;
      for (ConcurrentHashMap.Node<K,V>[] tab = table;;) { //cas需要循环重试
        ConcurrentHashMap.Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
          tab = initTable(); // case1 tab为空则初始化
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
          //case 2 正常更新bucket时要对首节点加synchronzed锁，但首节点为空则直接CAS新增
          if (casTabAt(tab, i, null, new ConcurrentHashMap.Node<K,V>(hash, key, value, null)))
            break;  // no lock when adding to empty bin. CAS成功，则已完成put操作，退出循环
        }
        else if ((fh = f.hash) == MOVED)
          //case 3 tab正在resize，且此buket节点为已MOVED状态，则尝试参与resize操作。
          tab = helpTransfer(tab, f); //避免空等，加速resize。
        else {
          //case 4 对应buket首节点不为空且是普通节点(链表或者tree)
          V oldVal = null;
          synchronized (f) { //更新bucket，用synchronized对bucket的第一个节点加锁
            if (tabAt(tab, i) == f) { //再校验一下f是否还是tabAt(tab, i)的引用，即f是否还是首节点?
              //一个bucket的首节点一旦成为首节点，就不会变化，除非:1 删除该节点 2 resize后
              if (fh >= 0) { // case 4.1 是普通链表节点
                binCount = 1;
                for (ConcurrentHashMap.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;
                  }
                  ConcurrentHashMap.Node<K,V> pred = e;
                  if ((e = e.next) == null) {
                    pred.next = new ConcurrentHashMap.Node<K,V>(hash, key, value, null);
                    break;
                  }
                } //for退出时，要么新增节点要么修改节点，一定是完成put value更新操作
              }
              else if (f instanceof ConcurrentHashMap.TreeBin) { //红黑树
                ConcurrentHashMap.Node<K,V> p;
                binCount = 2; //已经是treeBin，避免下面走到treeifyBin
                if ((p = ((ConcurrentHashMap.TreeBin<K,V>)f).putTreeVal(hash, key,
                    value)) != null) { //代理给putTreeVal
                  oldVal = p.val; //找到已有node
                  if (!onlyIfAbsent)
                    p.val = value;
                }
              }
```

```
        }
      }
      if (binCount != 0) { //只要上面更新了binCount, 则已完成value的put, 退出外层for.
        if (binCount >= TREEIFY_THRESHOLD) //TREEIFY_THRESHOLD ==8
          treeifyBin(tab, i);
        if (oldVal != null)
          return oldVal; // 是更新已有节点不是新增节点, 直接返回。不走下面addCount逻辑
        break;
      }
    }
  }
  addCount(1L, binCount); //有新增节点, 维护map的size
  return null;
}
```

下面展开 addCount，详细分析 ConcurrentHashMap 中 size 的维护逻辑。

1.6.1 initTable

```
/**
 * Initializes table, using the size recorded in sizeCtl.
 */
private final Node<K,V>[] initTable() {
  Node<K,V>[] tab; int sc;
  // table是实例变量
  while ((tab = table) == null || tab.length == 0) {
    //sizeCtl为实例变量, 初始化时设置为-1
    //通过sizeCtl spinLock 来保证只有1个线程初始化table
    if ((sc = sizeCtl) < 0)
        Thread.yield(); // lost initialization race; just spin
    else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
        try {
            if ((tab = table) == null || tab.length == 0) { //double check
                // sc>0, 用户设置了初始大小, sc==0使用默认大小
                int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                @SuppressWarnings("unchecked")
                Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                table = tab = nt; //volatile 变量table保证happerns-before可见性
                sc = n - (n >>> 2); // sc = n * 0.75, as threshold size for resize
            }
        } finally {
            sizeCtl = sc; //由sizeCtl spinLock保证此时只有1个线程在访问sizeCtl
        }
        break;
    }
  }
  return tab;
}
```

上述代码中，table = tab = nt 对 volatile 变量 table 赋值，由于 volatile 具有 happens-before 语义，其他线程在此语句后执行读取 table 操作后，能够读到 new Node<>[] 的数组内容。

1.7 size

1.7.1 how size is maintained

对 baseCount 和 counterCells 求和就是 map.size.

```
//baseCount, cellsBusy, counterCells 三个变量用于Map.size计数。
private transient volatile long baseCount;

/**
```

```
 * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
 */
private transient volatile int cellsBusy;

/**
 * Table of counter cells. When non-null, size is a power of 2.
 */
private transient volatile CounterCell[] counterCells;
```

这些字段和addCount的前半部分逻辑主要是从Stripped64复制过来的。详见笔记"Stripped64AndLongAdder". **为很么不直接用 LongAdder 来维护 size，而是本地维护这些状态和方法**？下面是 432 行的代码注释:

The element count is maintained using a specialization of LongAdder. We need to incorporate a specialization rather than just use a LongAdder in order to access implicit contention-sensing that leads to creation of multiple CounterCells. The counter mechanics avoid contention on updates but can encounter cache thrashing if read too frequently during concurrent access. To avoid reading so often, resizing under contention is attempted only upon adding to a bin already holding two or more nodes. Under uniform hash distributions, the probability of this occurring at threshold is around 13%, meaning that only about 1 in 8 puts check threshold (and after resizing, many fewer do so).

大意是 Stripped64 在并发更新性能好，但频繁的读操作时，例如 sumCount 操作，会导致缓存捶打现象 (cache thrashing). 为避免此种场景，只在部分条件下尝试 resize, 因此大大降低 sumCount 的调用频率，从而降低相关 baseCount 和 cells 的读操作。

```
/**
 * 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) {
  ConcurrentHashMap.CounterCell[] as; long b, s;
  if ((as = counterCells) != null ||
      !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
    ConcurrentHashMap.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); //fullAddCount 对应于Stripped64.longAccumulate
      return;
    }
    if (check <= 1)
      return;
    s = sumCount();
  }
  //上面的代码主要维护baseCount和cells，他们的和就是map的size。
  if (check >= 0) {
    ConcurrentHashMap.Node<K,V>[] tab, nt; int n, sc;
    while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
        (n = tab.length) < MAXIMUM_CAPACITY) {
      int rs = resizeStamp(n);
      if (sc < 0) {
        // sizeCtl为负，正在初始化或resize
        //  如果已resize结束，则直接退出，详见resize分析
        if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
            sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
            transferIndex <= 0)
          break;
```

```
            // 通过sizeCtl维护正在resize的worker线程数+1, 参与resize
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
              transfer(tab, nt);
          }
          else if (U.compareAndSwapInt(this, SIZECTL, sc, (rs << RESIZE_STAMP_SHIFT) + 2))
            transfer(tab, null); // 设置工作线程数为1, 发起resze. 详见resize分析
          s = sumCount();
        }
      }
    }

    //--------------- Counter support 从Stripped64复制过来
    //---------------以下代码详见笔记"Stripped64AndLongAdder"
    /**
    * A padded cell for distributing counts.  Adapted from LongAdder
    * and Striped64.  See their internal docs for explanation.
    */
    @sun.misc.Contended static final class CounterCell {
       volatile long value;
       CounterCell(long x) { value = x; }
    }

    final long sumCount() {
       CounterCell[] as = counterCells; CounterCell a;
       long sum = baseCount;
       if (as != null) {
           for (int i = 0; i < as.length; ++i) {
               if ((a = as[i]) != null)
                   sum += a.value;
           }
       }
       return sum;
    }

    // See LongAdder version for explanation, 对应于Stripped64.longAccumulate, 详见笔记。
    private final void fullAddCount(long x, boolean wasUncontended)
```

1.7.2 size

基于上述逻辑，size 相关方法实现就比较简单了。

```
    /**
     * Returns the number of key-value mappings in this map. If the map contains more than
     * Integer.MAX_VALUE elements, returns Integer.MAX_VALUE.
     */
    public int size() {
      long n = sumCount();
      return ((n < 0L) ? 0 : (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE : (int)n);
    }

    /**
     * Returns true if this map contains no key-value mappings.
     */
    public boolean isEmpty() {
      return sumCount() <= 0L; // ignore transient negative values
    }
```

1.8 transfer operation

1.8.1 transfer 整体逻辑

1. 第一个进程 cas 更新 sizeCtl, 发起 transfer 操作。并尝试 cas 更新 transferIndex, 取 bucket[bound,i] 作为处理的节点。
2. 线程处理完 bucket[bound,i], 会尝试 cas 更新 transerIndex, 继续取 bucket[bound,i](上下边界都减 stride).
3. 如果来了第二个 thread 尝试 helpTransfer，会尝试 cas 更新 transerIndex, 继续取 bucket[bound,i](上下边界都减 stride).
4. 多个线程之间处理的 buckets 范围，由 cas 更新 transferIndex 来调度。如果多个线程竞争，一次只有 1 个线程 CAS 更新 transferIndex 能成功 (成功后将负责 transfer buckets 的范围 [bound, i], bound 和 i 都是 stack 局部变量)，也就保证了一个 buckets 范围只会由 1 个线程负责 transfer.
5. 每次 transfer(resize), buckets 数组大小变为原来 2 倍。tranfer 进行中，会同时存在 table 和 nextTable。
6. 原来的 table 的某个 bucket 处理完，则 bucket 里的所有 node 都已到 nextTable。原来 table 对应位置的 bucket 的第一个节点设置为 ForwardingNode, hash 值为 MOVED, 并有实例变量 nextTable.
7. transfer 结束，nextTable 赋值给 table, nextTable 设置为 null.

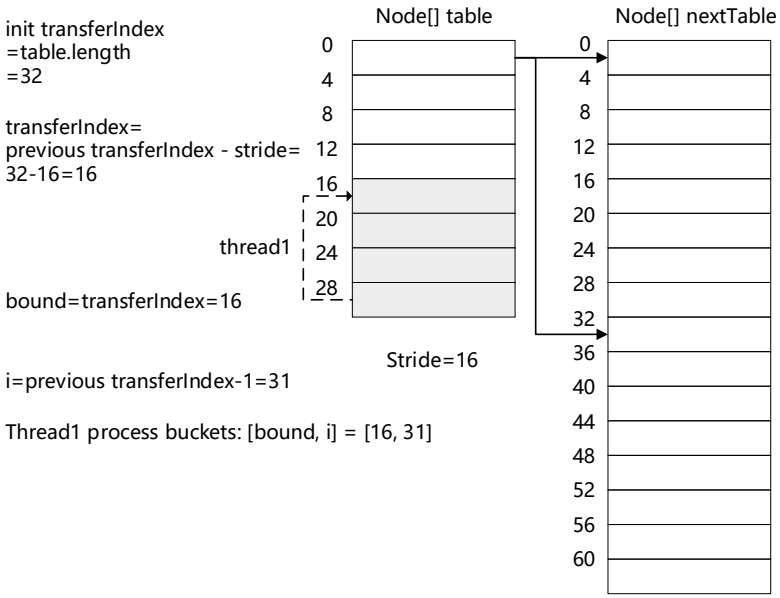

Figure 1.1: the thread init transfer

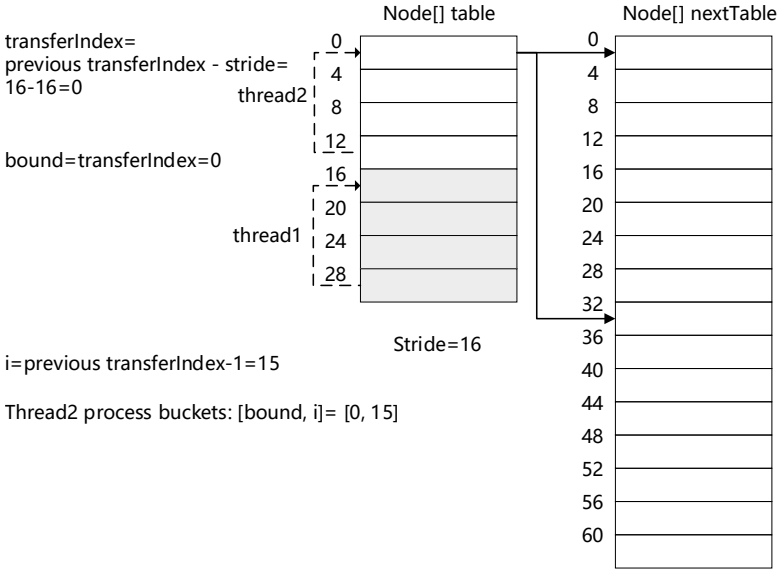

Figure 1.2: another thread help transfer

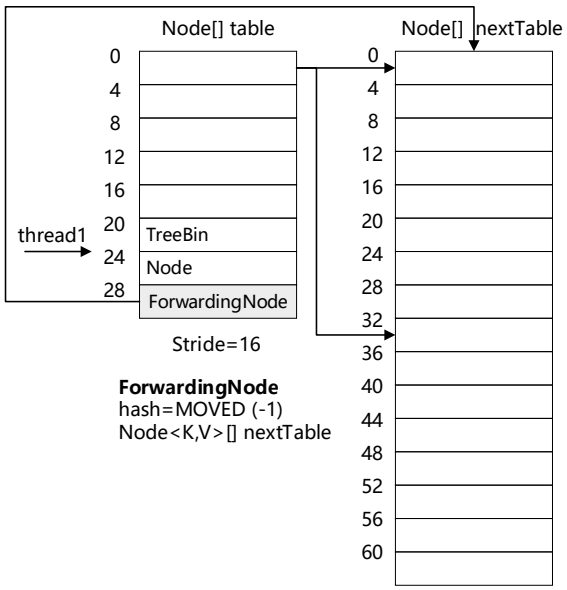

Figure 1.3: Node, TreeBin, ForwardingNode

1.8.2 发起 transfer

```
  private final void addCount(long x, int check) {
    CounterCell[] as; long b, s;
    if ((as = counterCells) != null ||
        // 计数相关逻辑，此处省略
        s = sumCount();
    }
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) { // sizeCtl<0，transfer进行中，尝试参与transfer. 详见下一个小节,helpTransfer
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            else if (U.compareAndSwapInt(this, SIZECTL, sc, (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null); //发起transfer
            s = sumCount();
        }
    }
}

 /**
  * Returns the stamp bits for resizing a table of size n.
  * Must be negative when shifted left by RESIZE_STAMP_SHIFT.
  */
 static final int resizeStamp(int n) {
   //RESIZE_STAMP_BITS为16， 1 << (RESIZE_STAMP_BITS - 1) = 1<< 15
     return Integer.numberOfLeadingZeros(n) | (1 << (RESIZE_STAMP_BITS - 1));
 }
```

sizeCtl 的编码

我们看 resizeStamp 方法，入参 n 是 table 的 length, 必然是 2 的幂。Integer.numberOfLeadingZeros(n) 的返回值介于 0 到 32 之间，如果转换成二进制

1. Integer.numberOfLeadingZeros(n) 最大值为 00000000 00000000 00000000 00100000.
2. Integer.numberOfLeadingZeros(n) 最小值是 00000000 00000000 00000000 00000000.

因此 resizeStamp 的返回值也就介于 00000000 00000000 10000000 00000000 到 00000000 00000000 10000000 00100000 之间. 从这个返回值的范围可以看出来 resizeStamp 的返回值高 16 位全都是 0，是不包含任何信息的。

以 table.length = 32 为例，Integer.numberOfLeadingZeros(32) = 26.

```
RESIZE_STAMP_SHIFT = 16
Integer.numberOfLeadingZeros(n) = 00000000 00000000 00000000 00011010 = 26
1 << (RESIZE_STAMP_BITS - 1)    = 00000000 00000000 10000000 00000000 = 1 << 15
rs                              = 00000000 00000000 10000000 00011010
rs << RESIZE_STAMP_SHIFT)       = 10000000 00011010 00000000 00000000
rs << RESIZE_STAMP_SHIFT) + 2   = 10000000 00011010 00000000 00000002 assign to sizeCtl when init transfer
```

因此 sizeCtl 在 transfer 时，为负数，编码为

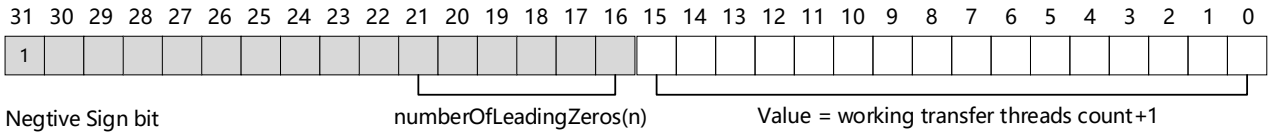

Figure 1.4: sizeCtl-bis-format

总结 sizeCtl 编码:

1. 低 16 位存 transfer 线程数 +1.
2. 负数符号位表示 trasnfer 进行中。
3. 高 16 位存 transfer 开始时，tables.length. 它还用于表示这是哪一次的 transfer(stamp bits), 保证所有 transfer threads 工作于同一次 transfer(高 16 位相同)，避免 transfer overlap. 如果 tables 从 8 扩展到 16，和从 16 扩展到 32 的 2 次 transfer 同时进行中，就是 transfer overlap.

发起 transfer 操作时，sizeCtl 设置为 `(rs << RESIZE_STAMP_SHIFT) + 2`.

```
U.compareAndSwapInt(this, SIZECTL, sc, (rs << RESIZE_STAMP_SHIFT) + 2)
```

即 cas 更新 sizeCtl，把 transfer threads count=1 编码到低 16 位。把 transfer 进行中的信息编码为 sizeCtl 负数符号位。把 resize 开始时的 table.length 编码进 sizeCtl 高 16 位。

if (sc < 0) // sizeCtl<0，transfer 进行中，尝试参与 transfer. 详见下一个小节,helpTransfer.

1.8.3 helpTransfer

在 putVal 中如果发现对应 bucket 的首节点是 MOVED，则调用 helpTransfer。下面展开 helpTransfer 的逻辑。

```
/**
 * Helps transfer if a resize is in progress.
 */
final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
  Node<K,V>[] nextTab; int sc;
  if (tab != null && (f instanceof ForwardingNode) &&
      (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
    int rs = resizeStamp(tab.length);
    // while条件中 == 右边的变量是实例变量
    while (nextTab == nextTable && table == tab && (sc = sizeCtl) < 0) {
        //读取的变量没有发生变化，即没有被其他线程更新
        if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
            sc == rs + MAX_RESIZERS || transferIndex <= 0)
            break;
```

```
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) { //将transfer线程数+1
                transfer(tab, nextTab);
                break;
            }
        }
        return nextTab;
    }
    return table;
}
```

- `(sc >>> RESIZE_STAMP_SHIFT) != rs` 如果 resizeStamp 变化，退出循环。保证所有线程要基于同一个旧的桶数组扩容。避免 transfer overlap。
- transferIndex <= 0 已经有线程完成扩容任务了。

至于 sc == rs + 1 || sc == rs + MAX_RESIZERS 这两个判断条件如果是细心的同学一定会觉得难以理解，这个地方确实是 JDK 的一个 BUG，这个 BUG 已经在 JDK 12 中修复，详细情况可以参考一下 Oracle 的官网：https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8214427。这两个判断条件应该写成这样：`sc == (rs << RESIZE_STAMP_SHIFT) + 1 || sc == (rs << RESIZE_STAMP_SHIFT) + MAX_RESIZERS`, 因为直接比较 rs 和 sc 是没有意义的，必须要有移位操作。它表达的含义是

- `sc == (rs << RESIZE_STAMP_SHIFT) + 1` 当前扩容的线程数为 0，即已经扩容完成了，就不需要再新增线程扩容
- `sc == (rs << RESIZE_STAMP_SHIFT) + MAX_RESIZERS` 参与扩容的线程数已经到了最大，就不需要再新增线程扩容

1.8.4 transfer

```
/**
 * 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 发起transfer操作
        //如何保证只有1个线程发起transfer? 通过sizeCtl spinLock
        try {
            @SuppressWarnings("unchecked")
            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; //见下图Case advanceA, 递减i循环处理[bound, i]
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1; //transfer整体任务已完成，当前线程考虑退出或者做收尾工作
```

```
            advance = false; //见下图Case advanceC
        }
        else if (U.compareAndSwapInt
                 (this, TRANSFERINDEX, nextIndex,
                  nextBound = (nextIndex > stride ?
                               nextIndex - stride : 0))) {
            bound = nextBound; //[bound, i]当前stride buckets范围处理完
            i = nextIndex - 1; //通过sizeCtl spinLock争抢下一个[bound, i]
            advance = false;   //见下图Case advanceB
        }
    }
    if (i < 0 || i >= n || i + n >= nextn) {
       //i==-1, transfer整体任务已完成. i >= n || i + n >= nextn没有走到的场景
        int sc;
        if (finishing) { //只有收尾线程会看到true
            nextTable = null;
            table = nextTab;
            sizeCtl = (n << 1) - (n >>> 1);
            return;
        }
        if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
           // cas 更新sizeCtl, 将transfer线程数减1
            if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                return; //如果当前线程是最后一个transfer线程
            finishing = advance = true; // 设置finish, 注意只有最后线程会设置此值
            i = n; // recheck before commit
            // i = n会导致将所有table的buckets遍历一遍, 作为recheck. 没有必要。
        }
    }
    else if ((f = tabAt(tab, i)) == null)
        advance = casTabAt(tab, i, null, fwd); //null bucket, 无需转移node
    else if ((fh = f.hash) == MOVED)
        advance = true; // already processed. 之前已处理过, 直接继续
    else {
        synchronized (f) {
            if (tabAt(tab, i) == f) {
                Node<K,V> ln, hn;
                if (fh >= 0) { //解读见 "transfer a bucket of normal Node list"小节
                    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(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;
                }
            }
        }
    }
}
```

transfer advance cases

下图是 adcance 的逻辑详解。多线程同时进行 transfer 时，如何协调一致的?

每个线程 transfer 时，首先通过 transferIndex spinLock CAS 操作，来锁定一个 buckets 区间 [bound, i]，由该线程专属处理这一批 buckets。由于是 CAS 操作，并且 bound 和 i 是局部变量，保证了只有 1 个线程会处理 [bound, i]。处理完 [bound, i]，通过 transferIndex spinLock 再次锁定更新后的区间 [bound, i]。通过 transferIndex spinLock 来保证多线程 transfer 更新时，不会出现数据竞争。

transferIndex==0 还标志了整个 transfer 过程已完成。

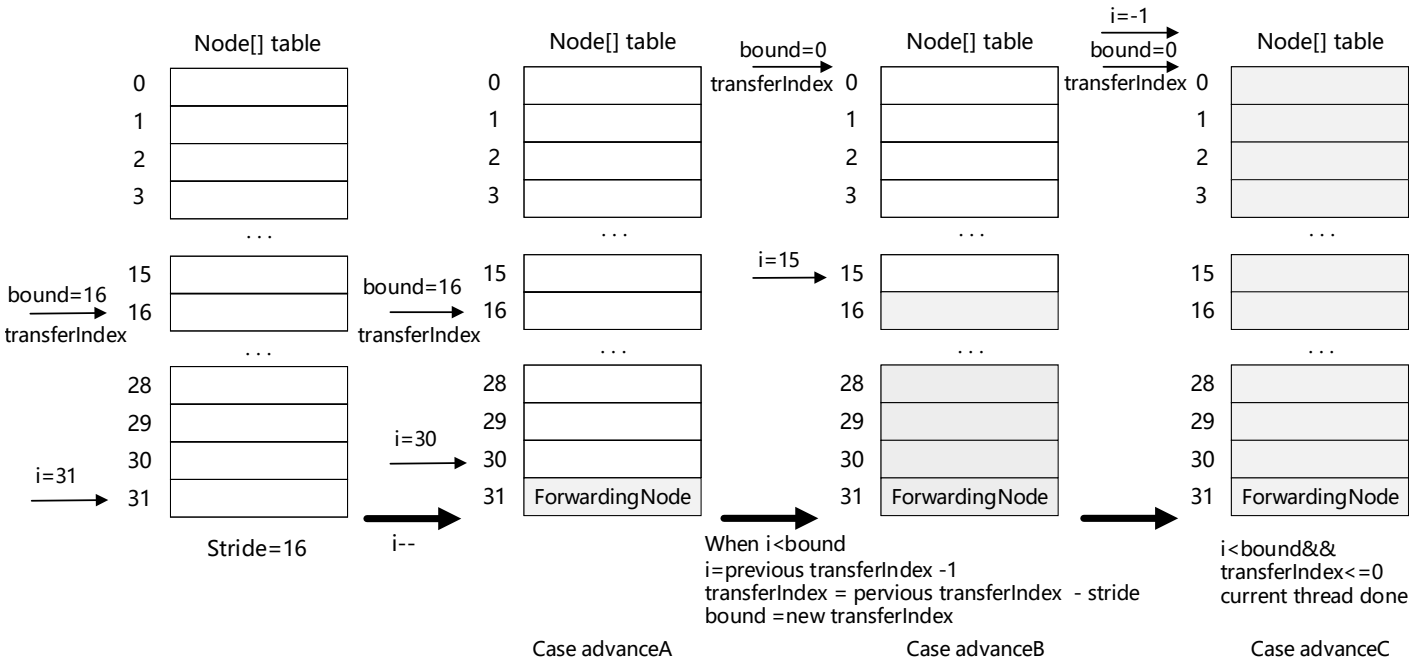

Figure 1.5: transfer advance cases

transfer a bucket of normal Node list

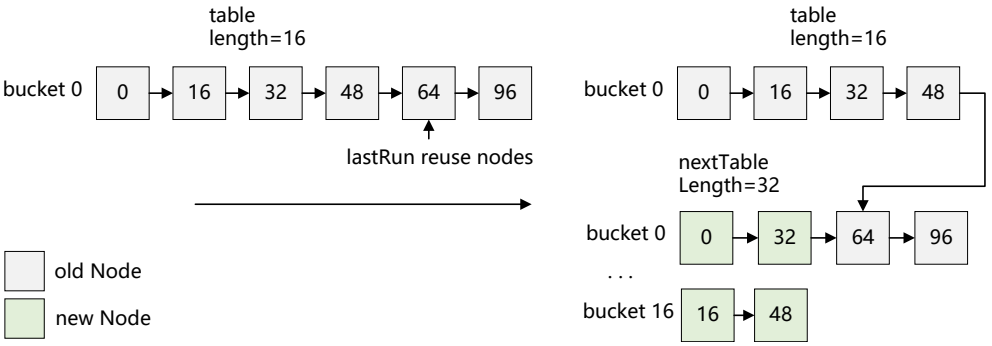

Figure 1.6: transfer normal node bucket

上图是 transfer 过程中，对某个 bucket 第一个节点加锁后，判断 fh>0, 即第一个节点为普通链表节点的逻辑示意图, 以 bucket 0 示例。可以看到 1 个 bucket 可能是 1 个节点链表，包含多个节点。在 transfer 到新 nextTable 的过程中，老的 table 对应的 bucket 结构完全没有变化，因此保证了 transfer 线程和 reader 线程互不干扰。即 transfer 过程中，reader 线程读到的数据仍然是有效的，如果没有 put 和 delete 操作，reader 读到的数据与 transfer 之前是一致的。

而原 bucket 节点链表的节点分为 2 部分，前面的节点，都是新建 Node，维护到新的 bucket 链表结构。后面的从 lastRun 开始的节点，由于都属于新 table 的同一个 bucket，没有新建节点，采用复用老节点的思路。第一个 for 循环，计算 runBit 和 lastRun, 目的就是为了实现这种复用。作者注释中对此有说明:

```
We eliminate unnecessary node creation by catching cases where old nodes can be reused because their
next fields won't change.  On average, only about one-sixth of them need cloning when a table doubles.
```

recheck before commit

在代码 finishing = advance = true 的下一行, i = n, 注释为 recheck before commit。有一个问题，按照我们前面的分析，扩容线程能够通力协作，保证各自负责的桶数组的分段不重不漏，这里为什么还需要做二次确认么？有一个开发者在 concurrency-interest 这个邮件列表中也关于这件事咨询了 Doug Lea (地址: http://cs.oswego.edu/pipermail/concurrency-interest/2020-July/017171.html)，他给出的回复是:

```
Yes, this is a valid point; thanks. The post-scan was needed in a previous version, and could be removed.
It does not trigger often enough to matter though, so is for now another minor tweak that might be
included next time CHM is updated.
```

虽然 Doug 在邮件中的措辞用了 could be, not often enough 等，但也确认了最后一个扩容线程的二次检查是没有必要的。

transfer 与 putVal 并发更新安全

目前位置,transfer 和 put 的代码都已解读，在"transfer advance cases" 小节中，我们也分析了 transfer 过程中，reader 线程读到的仍是一致的 map 数据。那么 transfer 和 putVal 同时进行时（多线程更新），如何保证线程安全的？此小节详解。

可以看到，无论是 put 还是 transfer，在处理 bucket 时，首先通过 synchronized 对 bucket 的第一个节点加锁。代码截取如下:

```
//f = tabAt(tab, i = (n - 1) & hash)
synchronized (f) {
  if (tabAt(tab, i) == f) {
    //内部逻辑省略
  }
```

设想可能发生的问题:

- transfer 线程抢到节点 f 的 intrinsic lock 后，如果在抢锁的过程中，别的线程已经删除了节点 f. 此时 transfer 线程不能继续以 f 为锁，进行后续操作。
- putVal 线程抢到节点 f 的 intrinsic lock 后，如果在抢锁的过程中 transfer 线程完成此 bucket 的 transfer，`tabAt(tab, i = (n - 1) & hash)` 的节点已经变成 ForwardingNode，此时 putVal 线程不能继续以 f 为锁，进行后续操作。当然 putVal 可以处理 ForwardingNode，但不是在 synchronized (f) 内部的那部分代码逻辑处理的。

代码中可以看到在 synchronized (f) 内部，首先检查当前 bucket 的第一个节点仍是 f: if (tabAt(tab, i) == f)。这样就避免了上述问题。此做法类似于 double check locking。

1.9 remove and replaceNode

remove 等方法调用了 replaceNode。在看过 put 代码后，replaceNode 的逻辑就比较简单直接了。

```
/**
 * 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 {@code key}, or
 *         {@code null} if there was no mapping for {@code key}
 * @throws NullPointerException if the specified key is null
 */
public V remove(Object key) {
    return replaceNode(key, null, null);
}

/**
 * Implementation for the four public remove/replace methods:
 * Replaces node value with v, conditional upon match of cv if
 * non-null.  If resulting value is null, delete.
 */
final V replaceNode(Object key, V value, Object cv) {
  int hash = spread(key.hashCode());
  for (Node<K,V>[] tab = table;;) {
    Node<K,V> f; int n, i, fh;
    if (tab == null || (n = tab.length) == 0 ||
        (f = tabAt(tab, i = (n - 1) & hash)) == null)
```

```
            break; //找不到对应节点，直接返回
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            boolean validated = false; //validated表示已处理
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        validated = true;
                        for (Node<K,V> e = f, pred = null;;) {
                            K ek;
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                V ev = e.val;
                                if (cv == null || cv == ev ||
                                    (ev != null && cv.equals(ev))) {
                                    oldVal = ev;
                                    if (value != null)
                                        e.val = value;           //case 1 修改值
                                    else if (pred != null)
                                        pred.next = e.next;      //case 2 非首节点，改链接
                                    else
                                        setTabAt(tab, i, e.next);//case 3 首节点，CAS
                                }
                                break;
                            }
                            pred = e;
                            if ((e = e.next) == null)
                                break;
                        }
                    }
                    else if (f instanceof TreeBin) {
                        validated = true;
                        TreeBin<K,V> t = (TreeBin<K,V>)f;
                        TreeNode<K,V> r, p;
                        if ((r = t.root) != null &&
                            (p = r.findTreeNode(hash, key, null)) != null) {
                            V pv = p.val;
                            if (cv == null || cv == pv ||
                                (pv != null && cv.equals(pv))) {
                                oldVal = pv;
                                if (value != null)
                                    p.val = value;
                                else if (t.removeTreeNode(p))
                                    setTabAt(tab, i, untreeify(t.first));
                            }
                        }
                    }
                }
            }
            if (validated) { //已修改，可以跳出循环
                if (oldVal != null) {
                    if (value == null)
                        addCount(-1L, -1);
                    return oldVal;
                }
                break;
            }
        }
    }
    return null;
}
```

1.10 traverser

1.10.1 stack used by traverser

实例变量维护了 stack, spare. spare 的存在是用于回收节点，避免创建和 GC Stack 节点的开销，是一种内存使用的优化。与 traverser 的逻辑没什么关系 (不关心 spare 的逻辑也不影响 Traverser 的理解)，这里关键还是实现了一个 stack，供 Traverser 使用。

```
// Traverser的实例变量
  TableStack<K,V> stack, spare

// Traverser static inner class
/**
 * Records the table, its length, and current traversal index for a
 * traverser that must process a region of a forwarded table before
 * proceeding with current table.
 */
static final class TableStack<K,V> {
    int length;
    int index;
    Node<K,V>[] tab;
    TableStack<K,V> next;
}

/**
* Traverser.pushState
* Saves traversal state upon encountering a forwarding node.
*/
private void pushState(Node<K,V>[] t, int i, int n) {
   TableStack<K,V> s = spare;  // reuse if possible
   if (s != null)
       spare = s.next;
   else
       s = new TableStack<K,V>();
   s.tab = t;
   s.length = n;
   s.index = i;
   s.next = stack;
   stack = s;
}

/**
* Possibly pops traversal state.
* Traverser.pushState
* @param n length of current table
*/
private void recoverState(int n) {
   TableStack<K,V> s; int len;
   while ((s = stack) != null && (index += (len = s.length)) >= n) {
       n = len;
       index = s.index;
       tab = s.tab;
       s.tab = null;
       TableStack<K,V> next = s.next;
       s.next = spare; // save for reuse
       stack = next;
       spare = s;
   }
   if (s == null && (index += baseSize) >= n)
       index = ++baseIndex;
}
```

上述逻辑参见下图。为便于理解，recoverState 只画出 while 循环内执行 1 次的逻辑。目的在于展示 stack 和 spare 的运行逻辑，Node 如何重用的, stack 如何实现先进后出的。

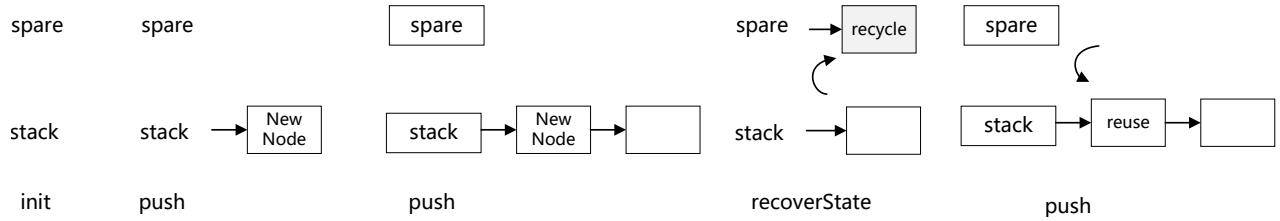

Figure 1.7: traverser stack spare

1.10.2 advance

下面详细分析 advance 的逻辑。采取从简单到复杂场景，从具体到抽象的方式展开。

如果不考虑 map 扩容 (transfer 操作), 则不需要 stack，直接循环遍历就可以了。这就是场景 1 展开的。考虑 transfer 操作的话，首先考虑到可能要同时访问初始 table（Traverser 构造函数传入的 tab）, 和 ForwardingNode 指向的 nextTable。进一步考虑，nextTable 中可能也有 ForwardingNode，即 traverse 的过程中，Map 进行了多次 transfer。由于多级 ForwardingNode 存在的可能性，Traverser 借助 Stack 来追溯多个 tab, 以及对应的 index, length。

1.10.3 advance 场景 1: traverse 过程没有遇到 ForwardingNode

```
static class Traverser<K,V> {
    Node<K,V>[] tab;        // current table; updated if resized
    Node<K,V> next;         // the next entry to use
    TableStack<K,V> stack, spare; // to save/restore on ForwardingNodes
    int index;              // index of bin to use next
    //base开头的都是创建时传入的tab的信息
    int baseIndex;          // current index of initial table
    int baseLimit;          // index bound for initial table
    final int baseSize;     // initial table size

    Traverser(Node<K,V>[] tab, int size, int index, int limit) {
        this.tab = tab;
        this.baseSize = size;
        this.baseIndex = this.index = index;
        this.baseLimit = limit;
        this.next = null;
    }

    /**
     * Advances if possible, returning next valid node, or null if none.
     */
    final Node<K,V> advance() {
        Node<K,V> e;
        // TAG 1 第一次为null. 后续index对应的bucket如果还有节点，
        // 则遍历此bucket链表直到遍历完。index对应的bucket链表处理完时，会更新index
        if ((e = next) != null)
            e = e.next;
        for (;;) {
            Node<K,V>[] t; int i, n;  // must use locals in checks
            if (e != null)
                return next = e; //index对应bucket还有节点，则直接返回

            // baseIndex >= baseLimit 时遍历完成
            if (baseIndex >= baseLimit || (t = tab) == null ||
                (n = t.length) <= (i = index) || i < 0)
                return next = null;
            // 取bucket=i的第一个节点，赋值e，但e.hash>0不会进入if分支
            if ((e = tabAt(t, i)) != null && e.hash < 0) {
                if (e instanceof ForwardingNode) {
                    tab = ((ForwardingNode<K,V>)e).nextTable;
                    e = null;
```

```
                    pushState(t, i, n);
                    continue;
                }
                else if (e instanceof TreeBin)
                    e = ((TreeBin<K,V>)e).first;
                else
                    e = null;
            }
            if (stack != null)
                recoverState(n); //本场景，stack始终为null，每次advance都不会进入分支
            else if ((index = i + baseSize) >= n)
                index = ++baseIndex; // visit upper slots if present
                //TAG2 index对应的bucket链表处理完时，尝试更新index
        }
    }
}
```

本场景比较简单直接, TAG2 切换 bucket，TAG1 遍历 bucket 中所有节点。这样就把所有 bucket 中的所有节点都处理完。

1.10.4 advance 场景 2: 有 1 级 ForwardingNode

下面我们跟踪下 traverse 过程中，遇到 1 级 ForwardingNode，但在 nextTab 中不再遇到 ForwardingNode 的执行情况。举个例子, Traverser 初始化时，tab 数组大小为 16，第一次执行 advance 时，ConcurrentHashMap 进行了 transfer，tab[0] 是一个 ForwardingNode。

```
1  static class Traverser<K,V> {
2      Node<K,V>[] tab;        // current table; updated if resized
3      Node<K,V> next;         // the next entry to use
4      TableStack<K,V> stack, spare; // to save/restore on ForwardingNodes
5      int index;              // index of bin to use next
6      int baseIndex;          // current index of initial table
7      int baseLimit;          // index bound for initial table
8      final int baseSize;     // initial table size
9
10     Traverser(Node<K,V>[] tab, int size, int index, int limit) {
11         this.tab = tab;
12         this.baseSize = size;
13         this.baseIndex = this.index = index;
14         this.baseLimit = limit;
15         this.next = null;
16     }
17
18     /**
19      * Advances if possible, returning next valid node, or null if none.
20      */
21     final Node<K,V> advance() {
22         Node<K,V> e;
23         if ((e = next) != null)
24             e = e.next;
25         for (;;) {
26             Node<K,V>[] t; int i, n;  // must use locals in checks
27             if (e != null)
28                 return next = e;
29             if (baseIndex >= baseLimit || (t = tab) == null ||
30                 (n = t.length) <= (i = index) || i < 0)
31                 return next = null;
32             if ((e = tabAt(t, i)) != null && e.hash < 0) {
33                 if (e instanceof ForwardingNode) {
34                     //tab和e是Traverser实例变量，更新
35                     tab = ((ForwardingNode<K,V>)e).nextTable;
36                     e = null;
37                     pushState(t, i, n);
38                     continue;
```

```
39                  }
40                  else if (e instanceof TreeBin)
41                      e = ((TreeBin<K,V>)e).first;
42                  else
43                      e = null;
44              }
45              if (stack != null)
46                  recoverState(n);
47              else if ((index = i + baseSize) >= n)
48                  index = ++baseIndex; // visit upper slots if present
49          }
50      }
51
52      private void pushState(Node<K,V>[] t, int i, int n) {
53          TableStack<K,V> s = spare;  // reuse if possible
54          if (s != null)
55              spare = s.next;
56          else
57              s = new TableStack<K,V>();
58          s.tab = t;
59          s.length = n;
60          s.index = i;
61          s.next = stack;
62          stack = s;
63      }
64
65      private void recoverState(int n) {
66          TableStack<K,V> s; int len;
67          while ((s = stack) != null && (index += (len = s.length)) >= n) {
68              n = len;
69              index = s.index;
70              tab = s.tab;
71              s.tab = null;
72              TableStack<K,V> next = s.next;
73              s.next = spare; // save for reuse
74              stack = next;
75              spare = s;
76          }
77          if (s == null && (index += baseSize) >= n)
78              index = ++baseIndex;
79      }
80  }
```

- 32 行, 取出 tab 的 bucket[0] 节点赋值 e, 它是 ForwardingNode, e.hash<0，进入 if 分支。更新 tab 为 nextTab, 为便于表述，后面称其为 tab1，初始 tab 称为 tab0。同时 e 赋值为 null。
- 然后调用 pushState，然后马上 continue 继续下一次循环。
- 下一次循环 (图中的 loop round 2)，e==null. 更新 tab=tab1, n = 32, i=index =0. e 取 tab1 的 bucket[0], e.hash>0 为普通节点。此次没有进入 32 行的分支内。
- stack!=null，执行 recoverState。index = 0, s.length=16, n=32, (index += (len = s.length)) >= n 为 false. 所以此次执行 recoverState 的效果只是更新了实例变量 index=16。后续的循环中，e 为当前遍历状态，index 保存 e 遍历完的下一个 bucket 信息。
- 下一次循环 (图中的 loop round 3), 以及后续对 advance 的调用，遍历以 e 为第一个节点的 bucket 链表中所有节点直到 null。然后取 tabAt(tab, index) 图中例子为 tab1[16]。
- 后续的循环（对应 loop round i），tabAt(tab1, 0) 的 bucket 节点全部 advance 完后的 advance. e 取 tabAt(tab1, 16)。此时 index=16, len=16, n=32, (index += (len = s.length)) >= n 成立。执行 recoverState 内部逻辑，pop stack, 更新 tab, index, baseIndex 等 Traverser 的实例变量，如图中斜体字所示。tab 回到了 tab0, index 变为 1, 前进了一个索引位置。
- 后续的 advance，则遍历 tabAt(tab1, 16) bucket 中的所有节点，直到 null。此时 e 取 tabAt(tab0, 1)，又是一个 ForwardingNode。后续逻辑与处理 tabAt(tab0, 0) 类似，不再赘述。

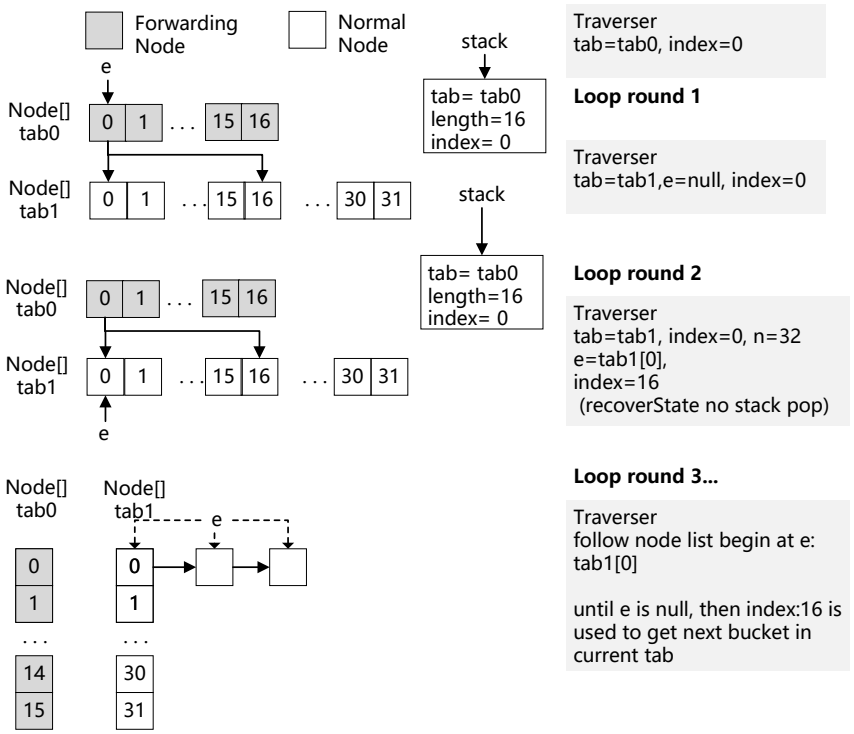

Figure 1.8: one level forward part 1

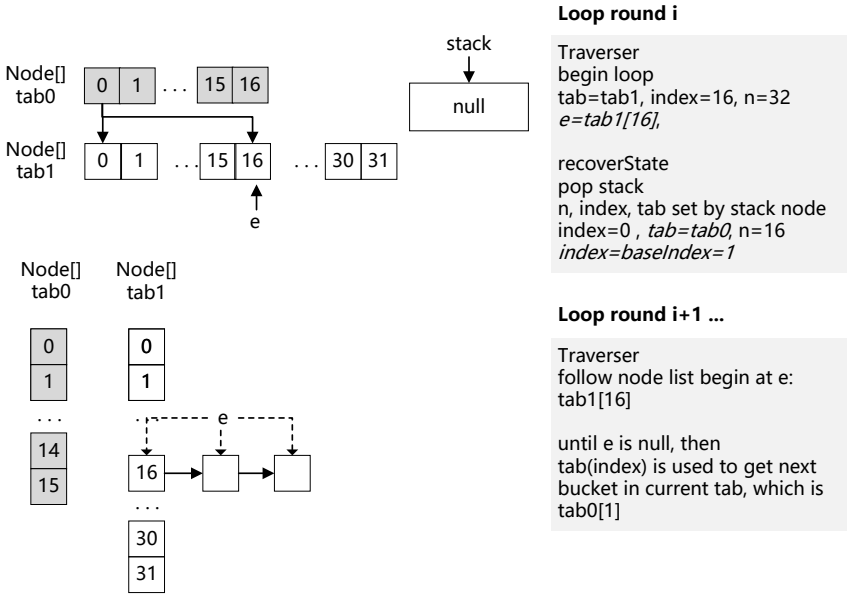

Figure 1.9: one level forward part 2

总结

- 局部变量 e 是当前遍历中的 bucket 节点。实例变量 index 存当 e 遍历完，下一个 bucket 的索引。
- 每次处理 ForwardingNode，push stack 存老的 tab, 需要处理最多 2 个 nextTable 的节点后，pop stack. 回到老的 tab, index+1.
- stack == null 时，Traverser 实例变量 tab 就是 tab0。

1.10.5 advance 场景 3: 有 2 级 ForwardingNode

为便于对照，此处再次贴上代码。

```
static class Traverser<K,V> {
    Node<K,V>[] tab;        // current table; updated if resized
    Node<K,V> next;         // the next entry to use
    TableStack<K,V> stack, spare; // to save/restore on ForwardingNodes
    int index;              // index of bin to use next
    int baseIndex;          // current index of initial table
    int baseLimit;          // index bound for initial table
    final int baseSize;     // initial table size

    Traverser(Node<K,V>[] tab, int size, int index, int limit) {
        this.tab = tab;
        this.baseSize = size;
        this.baseIndex = this.index = index;
        this.baseLimit = limit;
        this.next = null;
    }

    /**
     * Advances if possible, returning next valid node, or null if none.
     */
    final Node<K,V> advance() {
        Node<K,V> e;
        if ((e = next) != null)
            e = e.next;
        for (;;) {
            Node<K,V>[] t; int i, n;  // must use locals in checks
            if (e != null)
                return next = e;
            if (baseIndex >= baseLimit || (t = tab) == null ||
                (n = t.length) <= (i = index) || i < 0)
                return next = null;
            if ((e = tabAt(t, i)) != null && e.hash < 0) {
                if (e instanceof ForwardingNode) {
                    //tab和e是Traverser实例变量，更新
                    tab = ((ForwardingNode<K,V>)e).nextTable;
                    e = null;
                    pushState(t, i, n);
                    continue;
                }
                else if (e instanceof TreeBin)
                    e = ((TreeBin<K,V>)e).first;
                else
                    e = null;
            }
            if (stack != null)
                recoverState(n);
            else if ((index = i + baseSize) >= n)
                index = ++baseIndex; // visit upper slots if present
                //没有遇到ForwardingNode会走到此分支
        }
    }

    //push逻辑比较直接，就是push节点。
    //push的节点是当前tab, index, length, push后会更新实例变量tab
    private void pushState(Node<K,V>[] t, int i, int n) {
        TableStack<K,V> s = spare;  // reuse if possible
        if (s != null)
            spare = s.next;
        else
            s = new TableStack<K,V>();
        s.tab = t;
        s.length = n;
        s.index = i;
        s.next = stack;
```

```
            stack = s;
        }

        private void recoverState(int n) {
            TableStack<K,V> s; int len;
            while ((s = stack) != null && (index += (len = s.length)) >= n) {
                n = len;
                index = s.index;
                tab = s.tab;
                s.tab = null;
                TableStack<K,V> next = s.next;
                s.next = spare; // save for reuse
                stack = next;
                spare = s;
            }
            // s==null时，tab指向初始的table，此时index每次加1
            if (s == null && (index += baseSize) >= n)
                index = ++baseIndex;
        }
    }
```

存在 2 级 forwardingNode，示例如图。

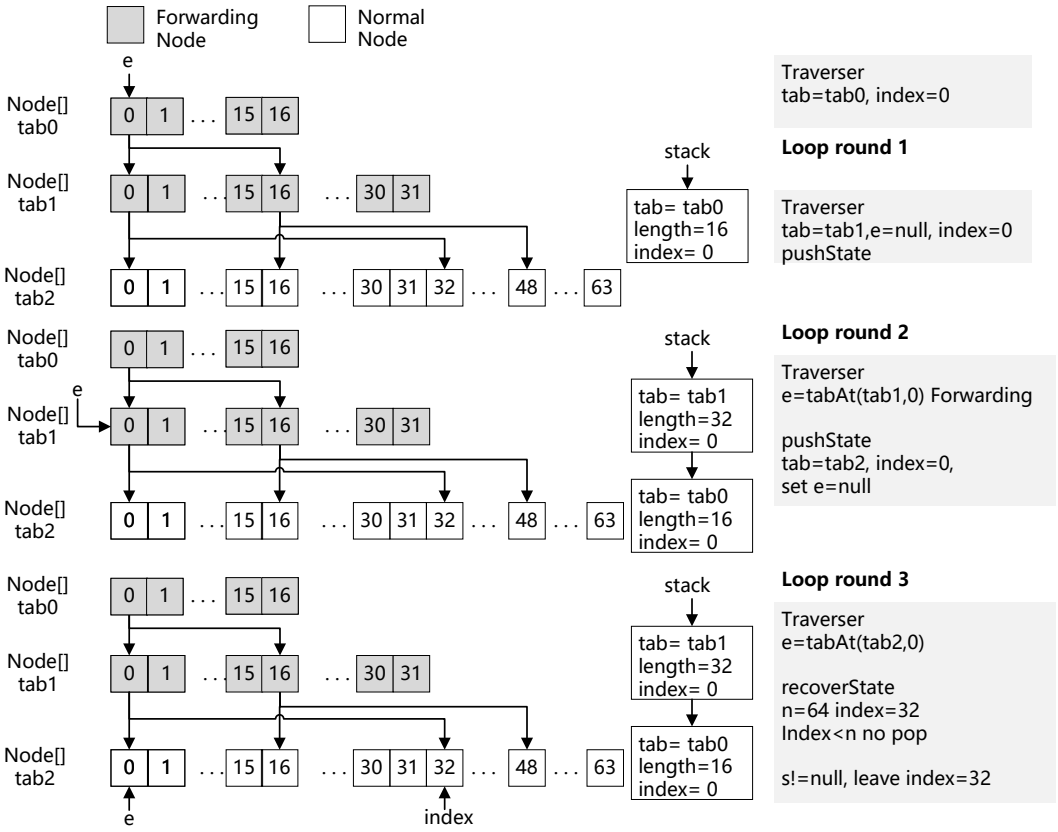

Figure 1.10: traver 2level forwarding part 1

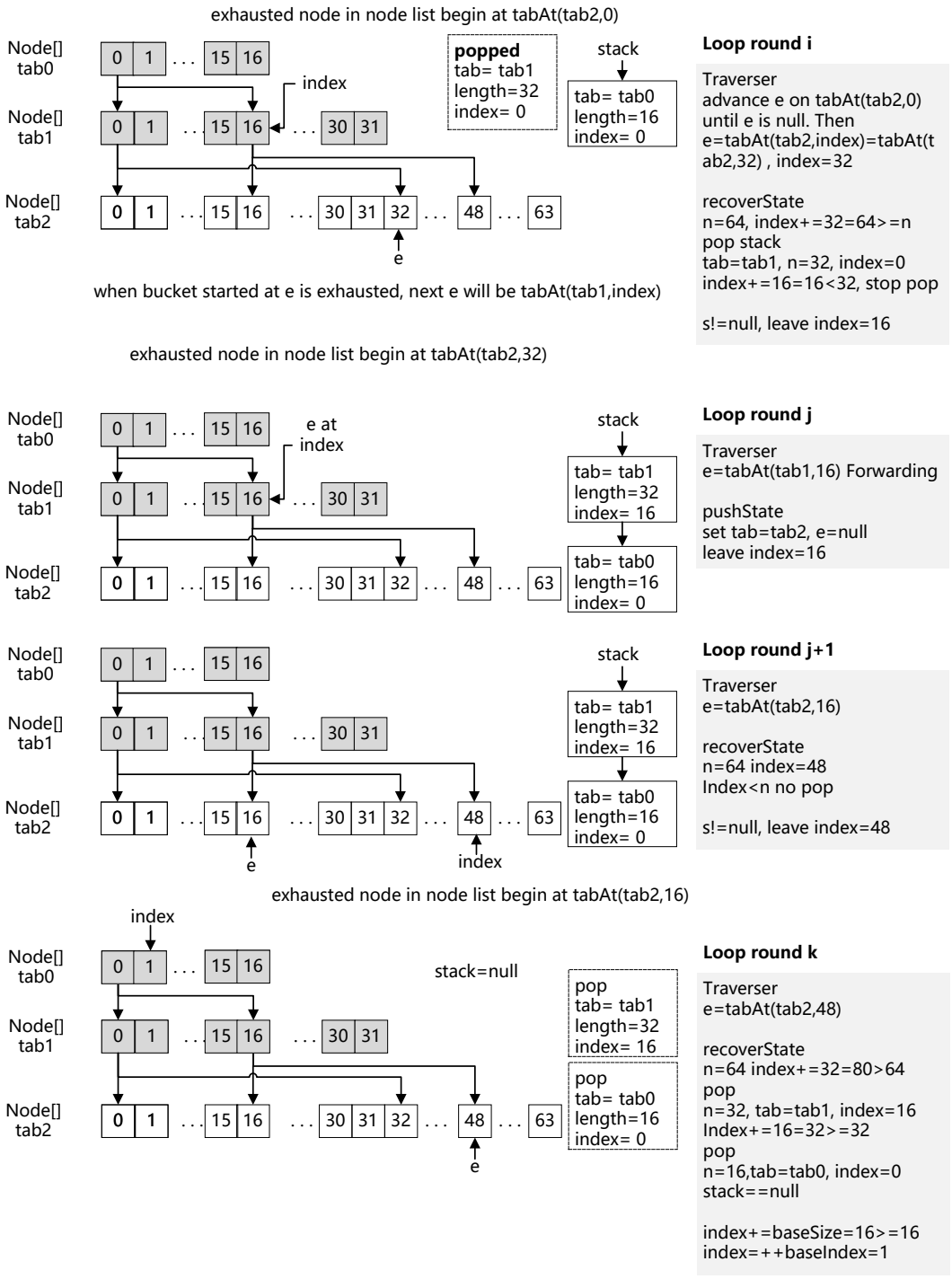

Figure 1.11: traver 2level forwarding part 2

1.10.6 traverser 逻辑理解总结

代码及注释见 advance 场景 3: 有 2 级 ForwardingNode。

追踪过一些具体场景，我们来理解 traverser 的思路。Traverser 利用了 stack 保存中间数据，可以联想到这是一个递归结构。执行逻辑图如下

- tab0 比较特殊，它是 Traverser 构造器传入的 tab, 每次处理完一个节点，index 前进 1. 它的一个 ForwardingNode，例如 tab0[j], 对应于下一级 tab1 的 2 个节点, 分别位于 tab1[j], tab1[j+tab0.len].
- 对于 tabi, i>=1, 它的一个 ForwardingNode，例如 tabi[j], 对应于下一级 tab(i+1) 的 2 个节点, 分别位于 tab(i+1)[j], tab(i+1)[j+tabi.len]. tabi 的 index 移动的偏移量是 tab(i-1).length, 存在 stack 的顶部节点。
- 上面总结过，每次获取普通节点 e, 则开始移动 index. 新的 index 指向 e 对应的节点链表处理完时，下一个 e 从哪个 tab 的哪个 index 取。

- 下面来看 index 的移动。对于对于 tabi 的 ForwardingNode tabi[j], 如果刚获取 e=tab(i+1)[j], 则更新 index=j+tabi.len, tab=tab(i+1)，stack 保持不变. 如果刚获取 e=tab(i+1)[j+tabi.len]，则表示 tabi[j] 对应的 ForwardingNode 已全部处理完 (因为一个 ForwardingNode 只对应 2 个下一级的普通 node). 此时 pop stack 中的节点信息: tab=i, index=j. 继续，看 stack 中剩余的节点，tab=i-1, 以及 index 的值，看是否需要继续 pop。
- stack 为空时 (即 stack==null), Traverser 的实例变量 tab 指向 tab0. 因 tab0 的 index 移动偏移量与其他 tabi 不一样, recoverState 的 `if (s == null && (index += baseSize) >= n)` 就是用于特殊处理 index 移动量的。

从 push pop 的角度看执行逻辑

- 每次看到 ForwardingNode 则 push, 存入当前 tab, index, length.
- 每次处理完当前一级的 2 个节点，则 pop，弹出上一级 tab, index, length, 表示弹出的这一级 tab 的对应 index 的节点处理完毕。并将实例变量 tab, index 更新为刚弹出的。继续看是在当前 tab 移动 index 还是从 stack pop 节点。

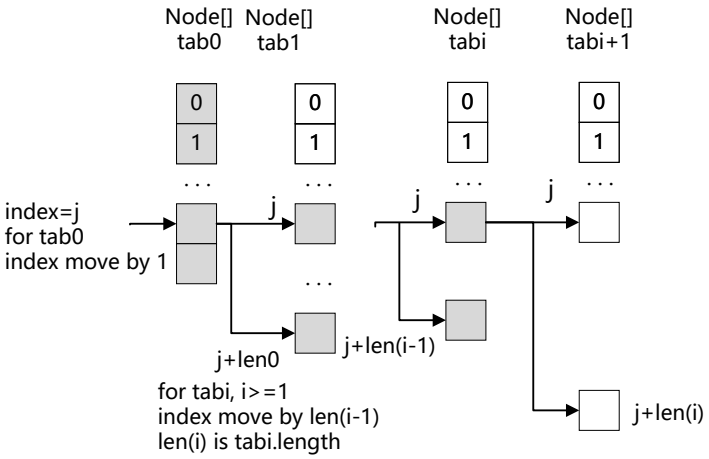

Figure 1.12: traver adance logic

1.11 summary

1.11.1 How ConcurrentHashMap implements Thread Safety

volatile arrays in java

因为 ConcurrentHashMap 用 volatile array 实例变量 table 来维护内部数据，作为知识预备，这里先提一下 volatile array 的内存可见性。本小节主要摘自笔记"Volatile Arrays in Java", 全文可见笔记或者原作者网址: http://jeremymanson.blogspot.com/2009/06/volatile-arrays-in-java.html.

Basically, if you write to a volatile field, and then you have a later read that sees that write, then the actions that happened before that write are guaranteed to be ordered before and visible to the actions that happen after the read. In practice, what this means is that the compiler and the processor can't do any sneaky reordering to move actions that come before the write to after it, or actions that come after the write to before it. See my post on What Volatile Means in Java for more detail.

With that out of the way, let's go through some examples of what you can do with volatile arrays:

```
volatile int [] arr = new int[SIZE];

arr = arr;
int x = arr[0];
arr[0] = 1;
```

The first lesson to learn, which will guide us here, is that arr is a volatile reference to **an array, not a reference to an array of volatile elements**! As a result, the write to arr[0] is not a volatile write. If you write to arr[0], you will get none of the ordering or visibility guarantees that you want from a volatile write.

What examples are there of a volatile write in the code above? Well, both of the writes to arr —the self-assignment and the write of new int[SIZE] —are volatile writes, because they are writing to arr, not one of its elements.

The astute reader will notice that there is no actual way to get volatile write semantics by writing to an element of an array referenced by a volatile field. The easiest way to get volatile array semantics is to use the Atomic[Reference/Long/Integer]Array classes in java.util.concurrent.atomic, which provide volatile semantics for reads and writes of individual elements.

get operation

```
public V get(Object key) {
  Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
  //key不允许为null spread计算内部使用的hash值
  int h = spread(key.hashCode());
  // -> tabAt读取element, U.getObjectVolatile具有volatile happens-before 语义
  if ((tab = table) != null && (n = tab.length) > 0 && (e = tabAt(tab, (n - 1) & h)) != null) {
    //如果有对应bucket节点
    if ((eh = e.hash) == h) {
      if ((ek = e.key) == key || (ek != null && key.equals(ek)))
        return e.val; //hash相等且key相等，则找到节点
    }
    else if (eh < 0) //hash为负数，则节点为TreeBin或者ForwardingNode，代理给node.find
      return (p = e.find(h, key)) != null ? p.val : null;

    //以上条件不满足，则e为普通链表节点，继续找
    while ((e = e.next) != null) {
      if (e.hash == h && ((ek = e.key) == key || (ek != null && key.equals(ek))))
        return e.val; //对比hash和key的范式
    }
  }
  return null;
}

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) {
  //具有volatile更新语义
 return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}

static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {
  U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);
}
```

get 操作的访问可见性

table 是 volatile array 实例变量。因此 table reference 的访问是可以保证内存可见性并具有 happens-before 语义的。但并不能保证 table 数组元素的访问可见性 (参见小节: volatile arrays in java)。作者用 U.getObjectVolatile 和 CAS 操作的 volatile 语义来保证对 table 数组元素访问的 volatile 语义。对于更新操作后的可见性:

- put 时，如果 key hash 后对应的 bucket 为 null, 则调用 casTabAt 设置 bucket 节点。由 cas 操作的 volatile 语义保证更新后 reader 线程可见。
- put 时，如果 key hash 后对应的 bucket 不为 null。对 bucket 的第一个节点加 intrinsic lock, 如果首节点是 ForwardingNode，则说明 transfer 进行中，put 线程参与到 transfer 活动中。对于 reader 线程，transfer 过程中 reader 线程始终能看到有效的 table 内容。参见小节"transfer/transfer a bucket of normal Node list". 有效的 table 内容是指，如果 transfer 过程中没有任何线程 put 或 remove, reader 将看到与 transfer 发起前一样的 table 内容。如果有 put 或 remove 操作 (如果与正在 transfer 的在同一个 bucket，由 synchronized 保护)，reader 将能够看到更新后的节点内容。

- put 时如果 key hash 后对应的 bucket 不为 null。对 bucket 的第一个节点加 intrinsic lock, 如果首节点不是 ForwardingNode. 则可能是红黑树或者是普通链表节点。如果是链表节点，遍历链表，由于 Node.next 是 volatile 变量，则保证读到的是最新的。对于 Node 的其他属性 key 和 hash，由于是 final 修饰，正确构造对象后即可以保证可见性。

对于上面 final 限定可见性，我们回顾下 java concurrency in practice 书中对于 safe publication 的范式总结:

To publish an object safely, both the reference to the object and the object's state must be made visible to other threads at the same time. A properly constructed object can be safely published by:

- Initializing an object reference from a static initializer;
- Storing a reference to it into a volatile field or AtomicReference;
- Storing a reference to it into a final field of a properly constructed object; or
- Storing a reference to it into a field that is properly guarded by a lock.