深入理解 ConcurrentHashmap（JDK1.6到1.7）


作者 codertom (/u/cd1316ad27cf) + 关注
2016.07.30 22:06 字数 4993 阅读 225 评论 0 喜欢 4
(/u/cd1316ad27cf)

concurrentHashmap是JDK提供的一个线程安全的Map容器类，因为它是线程安全的，同时获取和释放锁的代价很低，所以被广泛的应用在各种场景下。在开源项目中随处可见。对于concurrentHashmap，以前都是只会用，但是从来没有深入了解和学习，最近抽出时间分析一番。ps：对于concurrentHashmap，JDK1.6和JDK1.7的实现是不一样的，这里主要以JDK1.7的分析为主。

concurrentHashmap和HashMap的区别：

concurrentHashmap和HashMap大多数下的使用场景基本一致，但最大的区别就是concurrentHashmap是线程安全的HashMap则不是，在并发的场景下HashMap存在死循环的问题。具体的成因，我会总结一篇这样的笔记。

concurrentHashmap和HashTable的区别:

HashTable是一个线程安全的容器类，在HashTable所有方法都是用synchronized关键字修饰的，也就是说它是线程安全的。但是HashTable的性能十分低下，对于每一个操作都要做家锁操作，即使操作的是不同的bucket内的Entry也要全局枷锁，在高并发场景下性能低下。而concurrentHashmap引入了分段锁的概念，对于不同Bucket的操作不需要全局锁来保证线程安全。

concurrentHashmap在JDK1.6和JDK1.7的实现异同点：

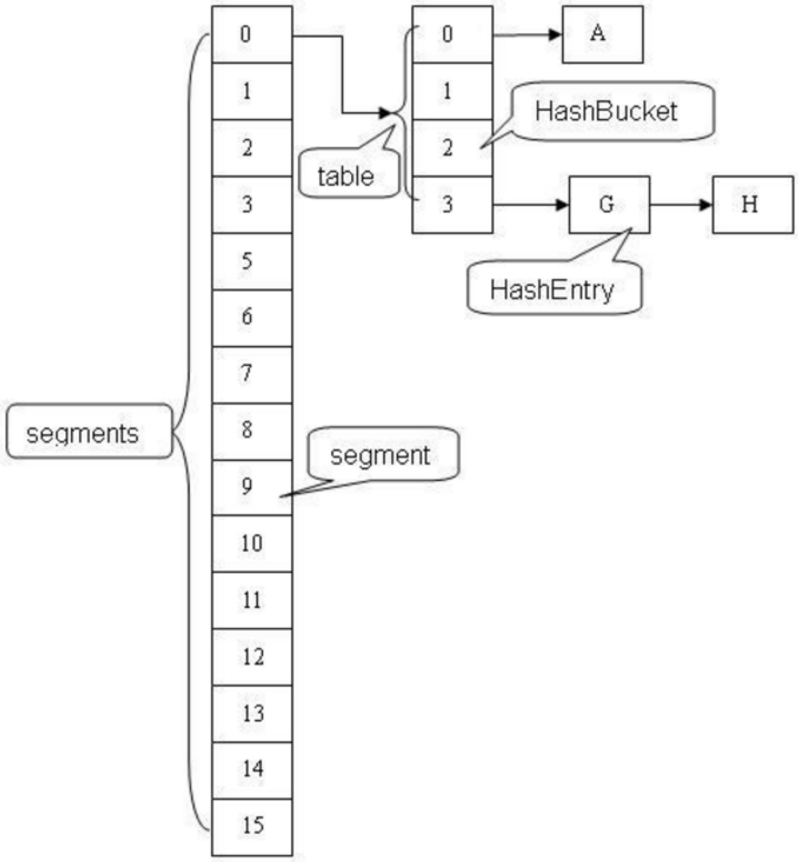
在学习源码之前，我也看了很多博客，发现上面说法不一，后来对比了代码才知道，原来JDK1.7将concurrentHashmap的实现机制改变了，但是代码确实比原来好懂了一下。

初始化：

```
/**
 * The default initial capacity for this table,
 * used when not otherwise specified in a constructor.
 * 默认的初始化容量
 */
static final int DEFAULT_INITIAL_CAPACITY = 16;

/**
 * The default load factor for this table, used when not
 * otherwise specified in a constructor.
 * 默认负载因子
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;

/**
 * The default concurrency level for this table, used when not
 * otherwise specified in a constructor.
 * 默认的并发等级
 */
static final int DEFAULT_CONCURRENCY_LEVEL = 16;

/**
 * The maximum capacity, used if a higher value is implicitly
 * specified by either of the constructors with arguments.  MUST
 * be a power of two <= 1<<30 to ensure that entries are indexable
 * using ints.
 * 最大容量
 */
static final int MAXIMUM_CAPACITY = 1 << 30;

/**
 * The minimum capacity for per-segment tables.  Must be a power
 * of two, at least two to avoid immediate resizing on next use
 * after lazy construction.
 * 一个Segment中Table数组最小长度为2
 */
static final int MIN_SEGMENT_TABLE_CAPACITY = 2;

/**
 * The maximum number of segments to allow; used to bound
 * constructor arguments. Must be power of two less than 1 << 24.
 * Segment的最大数
 */
static final int MAX_SEGMENTS = 1 << 16; // slightly conservative
```

```
/**
     * 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 resizing.
     * 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.
     */
    @SuppressWarnings("unchecked")
    public ConcurrentHashMap(int initialCapacity,
                             float loadFactor, int concurrencyLevel) {
        //首先检查入参的有效性
        if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
            throw new IllegalArgumentException();
        //限制并发度
        if (concurrencyLevel > MAX_SEGMENTS)
            concurrencyLevel = MAX_SEGMENTS;
        // Find power-of-two sizes best matching arguments
        //Segment的段寻址的因子
        int sshift = 0;
        //Segments数组的长度
        int ssize = 1;
        //根据并发等级来确定Segment的数组长度和段段寻址的因子
        while (ssize < concurrencyLevel) {
            ++sshift;
            ssize <<= 1;
        }
        //默认并发等级下ssize为16，sshift为4，这里有一个关系就是2的sshift次方等于ssize，主要是为了
        //segmentShift为Segment寻址的偏移量
        this.segmentShift = 32 - sshift;
        //Segment掩码，ssize为16时，segmentMask为0xFF
        this.segmentMask = ssize - 1;
        //判断初始化容量的有效性
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        //计算一个Segment的容量
        int c = initialCapacity / ssize;
        //保证容量足够。ps： /是整除，所以需要通过下面语句保证
        if (c * ssize < initialCapacity)
            ++c;
        //计算Segment中的table容量，最小为2,如果小于c，那么x2
        int cap = MIN_SEGMENT_TABLE_CAPACITY;
        while (cap < c)
            cap <<= 1;
        // create segments and segments[0]
        //创建一个Segment0，以后以此为镜像，新建Segment
        Segment<K,V> s0 =
            new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
                             (HashEntry<K,V>[])new HashEntry[cap]);
        //创建Segment数组，长度为ssize
        Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
        //用UNSAFE的方法将S0放到ss[0],相当于初始化ss
        UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]
        this.segments = ss;
    }
```

ConcurrentHashmap的结构图：



Paste_Image.png

元素定位：

初始化之后，我们需要看看concurrentHashmap是怎么定位元素的，比较关键的是hash算法。

```
/**
     * Applies a supplemental hash function to a given hashCode, which
     * defends against poor quality hash functions.  This is critical
     * because ConcurrentHashMap uses power-of-two length hash tables,
     * that otherwise encounter collisions for hashCodes that do not
     * differ in lower or upper bits.
     */
    private int hash(Object k) {
        int h = hashSeed;

        if ((0 != h) && (k instanceof String)) {
            return sun.misc.Hashing.stringHash32((String) k);
        }

        h ^= k.hashCode();

        // Spread bits to regularize both segment and index locations,
        // using variant of single-word Wang/Jenkins hash.
        h += (h <<  15) ^ 0xffffcd7d;
        h ^= (h >>> 10);
        h += (h <<   3);
        h ^= (h >>>  6);
        h += (h <<   2) + (h << 14);
        return h ^ (h >>> 16);
    }
```

看到这里，绝大多数人都和我一样是懵逼的，的确我现在也没弄明白是什么逻辑，但是这里有一个疑问，就是Object本身是有hashcode，那么为什么不用Object的HashCode呢?看过《算法导论》的人应该明白，这种算法可能是有问题的，那就是在hash取模的时候，主要是根据后几位确定取模之后的index，所以会很不均匀。所以需要重新设计hash算法。

put的实现：

在了解了重新设计的Hashcode之后，我们需要知道是怎么根据hash定位到Segment和Segment里面table的索引。那么我们通过学习put方法，附带看一下元素定位的规则：

```
/**
     * 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
     */
    @SuppressWarnings("unchecked")
    public V put(K key, V value) {
        Segment<K,V> s;
        if (value == null)
            throw new NullPointerException();
        int hash = hash(key);
        //定位Segment，让Hash右移动segmentShift位，默认情况下就是28位(总长32位)，之后和segmentMa
        int j = (hash >>> segmentShift) & segmentMask;
        //利用UNSAFE.getObject中的方法获取到目标的Segment。
        if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
             (segments, (j << SSHIFT) + SBASE)) == null) //  in ensureSegment
            //如果没有取到目标Segment，所以需要保证能取到这个Segment，没有的话创建一个Segment
            s = ensureSegment(j);
        //代理到Segment的put方法
        return s.put(key, hash, value, false);
    }
```

上面的代码中其实是有一些点比较难理解，首先是(Segment<K,V>)UNSAFE.getObject(segments, (j << SSHIFT) + SBASE)), UNSAFE这种用法是在JDK1.6中没有的，主要是利用Native方法来快速的定位元素。看下SSHIFT和SBASE。

```
// Unsafe mechanics
    private static final sun.misc.Unsafe UNSAFE;
    private static final long SBASE;
    private static final int SSHIFT;
    private static final long TBASE;
    private static final int TSHIFT;
    private static final long HASHSEED_OFFSET;

    static {
        int ss, ts;
        try {
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class tc = HashEntry[].class;
            Class sc = Segment[].class;
            TBASE = UNSAFE.arrayBaseOffset(tc);
            SBASE = UNSAFE.arrayBaseOffset(sc);
            ts = UNSAFE.arrayIndexScale(tc);
            ss = UNSAFE.arrayIndexScale(sc);
            HASHSEED_OFFSET = UNSAFE.objectFieldOffset(
                ConcurrentHashMap.class.getDeclaredField("hashSeed"));
        } catch (Exception e) {
            throw new Error(e);
        }
        if ((ss & (ss-1)) != 0 || (ts & (ts-1)) != 0)
            throw new Error("data type scale not a power of two");
        SSHIFT = 31 - Integer.numberOfLeadingZeros(ss);
        TSHIFT = 31 - Integer.numberOfLeadingZeros(ts);
    }
```

这里我是有一些迷惑的,SBASE是基址，但是SSHIFT是什么其实我是不理解的，但是猜测应该是一种计算偏移量的方式（ps：如果有明白的大神，请留言我）。这样就获得了指定索引的Segment。

还有一个点是：ensureSegment（ ）

```
/**
 * Returns the segment for the given index, creating it and
 * recording in segment table (via CAS) if not already present.
 *
 * @param k the index
 * @return the segment
 */
@SuppressWarnings("unchecked")
private Segment<K,V> ensureSegment(int k) {
    final Segment<K,V>[] ss = this.segments;
    long u = (k << SSHIFT) + SBASE; // raw offset
    Segment<K,V> seg;
    //getObjectVolatile是以Volatile的方式获得目标的Segment，Volatile是为了保证可见性。
    if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) {
        //如果没有取到，那么证明指定的Segment不存在，那么需要新建Segment,方式是以ss[0]为镜像创
        Segment<K,V> proto = ss[0]; // use segment 0 as prototype
        int cap = proto.table.length;
        float lf = proto.loadFactor;
        int threshold = (int)(cap * lf);
        HashEntry<K,V>[] tab = (HashEntry<K,V>[])new HashEntry[cap];
        if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
            == null) { // 再次检查
            Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);//创建新Segment
            //以CAS的方式，将新建的Segment，set到指定的位置。
            while ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
                   == null) {
                if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s))
                    break;
            }
        }
    }
    return seg;
}
```

上面的代码就是保证，在put之前，要保证目标的Segment是存在的，不存在需要创建一个Segment。
put方法代理到了Segment的put方法，Segment extends 了ReentrantLock，以至于它能当做一个Lock使用。那么我们看一下Segment的put的实现：

```
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    //因为put操作会改变整体的结构，所以需要保证段的线程安全性，所以首先tryLock
    HashEntry<K,V> node = tryLock() ? null :
        scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        //新建tab引用，避免直接引用Volatile导致性能损耗，
        HashEntry<K,V>[] tab = table;
        int index = (tab.length - 1) & hash;
        //Volatile读，保证可见性
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
            if (e != null) {
                K k;
                //遍历HashEntry数组，寻找可替换的HashEntry
                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 {
                //如果不存在可替换的HashEntry，如果在scanAndLockForPut中建立了此Node直接
                if (node != null)
                    node.setNext(first);
                else
                    //如果没有则新建一个Node，添加到链表头
                    node = new HashEntry<K,V>(hash, key, value, first);
                //容量计数+1
                int c = count + 1;
                //如果容量不足，那么扩容
                if (c > threshold && tab.length < MAXIMUM_CAPACITY)
                    rehash(node);
                else
                    //以Volatile写的方式，替换tab[index]的引用
                    setEntryAt(tab, index, node);
                ++modCount;
                count = c;
                oldValue = null;
                break;
            }
        }
    } finally {
        unlock();
    }
    return oldValue;
}
```

put方法是做了加锁操作的，所以不用过多的考虑线程安全的问题，但是get操作为了保证性能是没有加锁的，所以需要尽量的保证数据的可见性，能让get得到最新的数据。上面的方法里有一点是比较难理解的：

1.scanAndLockForPut(key, hash, value)在做什么：

```
/**
         * Scans for a node containing given key while trying to
         * acquire lock, creating and returning one if not found. Upon
         * return, guarantees that lock is held. UNlike in most
         * methods, calls to method equals are not screened: Since
         * traversal speed doesn't matter, we might as well help warm
         * up the associated code and accesses as well.
         *
         * @return a new node if key not found, else null
         */
        private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {
            HashEntry<K,V> first = entryForHash(this, hash);
            HashEntry<K,V> e = first;
            HashEntry<K,V> node = null;
            int retries = -1; // negative while locating node
            while (!tryLock()) {
                HashEntry<K,V> f; // to recheck first below
                if (retries < 0) {
                    if (e == null) {
                        if (node == null) // speculatively create node
                            node = new HashEntry<K,V>(hash, key, value, null);
                        retries = 0;
                    }
                    else if (key.equals(e.key))
                        retries = 0;
                    else
                        e = e.next;
                }
                else if (++retries > MAX_SCAN_RETRIES) {
                    lock();
                    break;
                }
                else if ((retries & 1) == 0 &&
                         (f = entryForHash(this, hash)) != first) {
                    e = first = f; // re-traverse if entry changed
                    retries = -1;
                }
            }
            return node;
        }
```

从上面的逻辑可以看出来，其实就是在获取锁的时候顺便检查一下指定index的HashEntry有没有变化，同时如果目标节点不存在创建一个新的目标节点。但是为什么做这样的检查，查了很多资料结合注释理解是，为了事先做数据的缓存，让这些数据缓存在CPU的cache中，这样后续在使用时能避免Cache missing。ps：scanAndLockForPut有个孪生兄弟scanAndLock，作用都差不多。

和JDK1.6的实现的不同：

```
V put(K key, int hash, V value, boolean onlyIfAbsent) {
    lock();
    try {
        int c = count;
        if (c++ > threshold) // ensure capacity
            rehash();
        HashEntry<K,V>[] tab = table;
        int index = hash & (tab.length - 1);
        HashEntry<K,V> first = tab[index];
        HashEntry<K,V> e = first;
        while (e != null && (e.hash != hash || !key.equals(e.key)))
            e = e.next;

        V oldValue;
        if (e != null) {
            oldValue = e.value;
            if (!onlyIfAbsent)
                e.value = value;
        }
        else {
            oldValue = null;
            ++modCount;
            tab[index] = new HashEntry<K,V>(key, hash, first, value);
            count = c; // write-volatile
        }
        return oldValue;
    } finally {
        unlock();
    }
}
```

JDK1.6的实现和JDK1.7的实现比较相似，但是主要区别是，没有使用一些UNSAFE的方法去保证内存的可见性，而是通过一个Volatile变量——count去实现。在开始的时候读count保证lock的内存语意，最后写count实现unlock的内存语意。
但是这里存在一个问题，new HashEntry操作存在重排序问题，导致在getValue的时候tab[index]不为null，但是value为null。

get方法：

看过了put方法之后，接下来我们看比较关键的方法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) {
      Segment<K,V> s; // manually integrate access methods to reduce overhead
      HashEntry<K,V>[] tab;
      int h = hash(key);
      long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
      if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
          (tab = s.table) != null) {
          for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
                   (tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE);
               e != null; e = e.next) {
              K k;
              if ((k = e.key) == key || (e.hash == h && key.equals(k)))
                  return e.value;
          }
      }
      return null;
  }
```

可以看出来，get方法很简单，同时get是没有加锁的，那么get是如何保证可见性的呢？首先获取指定index的Segment，利用getObjectVolatile获取指定index的first HashEntry，之后遍历HashEntry链表，这里比较关键的是HashEntry的数据结构：

```
volatile V value;
volatile HashEntry<K,V> next;
```

两个变量是volatile的，也就是说，两个变量的读写能保证数据的可见性。
所以在变量HashEntry时，总能保证得到最新的值。

JKD1.6的get方法的实现：

```
V get(Object key, int hash) {
if (count != 0) { // read-volatile 当前桶的数据个数是否为0
HashEntry<K,V> e = getFirst(hash); 得到头节点
while (e != null) {
if (e.hash == hash && key.equals(e.key)) {
V v = e.value;
if (v != null)
return v;
return readValueUnderLock(e); // recheck
}
e = e.next;
}
}
return null;
}
```

首先是读取count变量，因为内存的可见性，总是能返回最新的结构，但是对于getFirst可能得到的是过时的HashEntry。接下来获取到HashEntry之后getValue。但是这里为什么要做一个value的判空，原因就是上一步put的重排序问题，如果为null，那么只能加锁，加锁之后进行重新读取。但是这样确实会带来一些开销。

为什么JDK1.6的实现是弱一致性的？

这里比较重要的一点就是，为什么JDK1.6的是弱一致性的？因为JDK1.6的所有可见性都是以count实现的，当put和get并发时，get可能获取不到最新的结果，这就是JDK1.6中ConcurrentHashMap弱一致性问题，主要问题是 tab[index] = new HashEntry<K,V>(key, hash, first, value);不一定 happened before getFirst(hash)；盗图一张：

执行put的线程	执行get的线程
⑧tab[index] = new HashEntry<K,V>(key, hash, first, value)	
	③if (count != 0)
②count = c	
	⑨HashEntry e = getFirst(hash);

Paste_Image.png

而JDK1.7的实现，对于每一个操作都是Volatile变量的操作，能保证线程之间的可见性，所以不存在弱一致性的问题。

remove方法：

看了put方法之后，接下来看一下同样能改变结构的remove方法：

```
/**
     * 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);
        Segment<K,V> s = segmentForHash(hash);
        return s == null ? null : s.remove(key, hash, null);
    }
```

```
final V remove(Object key, int hash, Object value) {
    if (!tryLock())
        scanAndLock(key, hash);
    V oldValue = null;
    try {
        HashEntry<K,V>[] tab = table;
        int index = (tab.length - 1) & hash;
        HashEntry<K,V> e = entryAt(tab, index);
        HashEntry<K,V> pred = null;
        while (e != null) {
            K k;
            HashEntry<K,V> next = e.next;
            if ((k = e.key) == key ||
                (e.hash == hash && key.equals(k))) {
                V v = e.value;
                if (value == null || value == v || value.equals(v)) {
                    if (pred == null)
                        setEntryAt(tab, index, next);
                    else
                        pred.setNext(next);
                    ++modCount;
                    --count;
                    oldValue = v;
                }
                break;
            }
            pred = e;
            e = next;
        }
    } finally {
        unlock();
    }
    return oldValue;
}
```

remove方法，同样是代理到Segment的remove，在这里调用了scanAndLock方法，这个在前面已经说过了。这里的remove逻辑是比较简单的就不赘述了。

size方法：

接下来看最后一个方法，也是一个跨Segment的方法：

```
/**
     * Returns the number of key-value mappings in this map.  If the
     * map contains more than <tt>Integer.MAX_VALUE</tt> elements, returns
     * <tt>Integer.MAX_VALUE</tt>.
     *
     * @return the number of key-value mappings in this map
     */
    public int size() {
        // Try a few times to get accurate count. On failure due to
        // continuous async changes in table, resort to locking.
        final Segment<K,V>[] segments = this.segments;
        int size;
        boolean overflow; // true if size overflows 32 bits
        long sum;         // sum of modCounts
        long last = 0L;   // previous sum
        int retries = -1; // first iteration isn't retry
        try {
            for (;;) {
                if (retries++ == RETRIES_BEFORE_LOCK) {
                    for (int j = 0; j < segments.length; ++j)
                        ensureSegment(j).lock(); // force creation
                }
                sum = 0L;
                size = 0;
                overflow = false;
                for (int j = 0; j < segments.length; ++j) {
                    Segment<K,V> seg = segmentAt(segments, j);
                    if (seg != null) {
                        sum += seg.modCount;
                        int c = seg.count;
                        if (c < 0 || (size += c) < 0)
                            overflow = true;
                    }
                }
                if (sum == last)
                    break;
                last = sum;
            }
        } finally {
            if (retries > RETRIES_BEFORE_LOCK) {
                for (int j = 0; j < segments.length; ++j)
                    segmentAt(segments, j).unlock();
            }
        }
        return overflow ? Integer.MAX_VALUE : size;
    }
```

size是一个跨Segment的操作，所以避免不了多个锁的获取，这里主要是通过如下方法进行所有锁的获取：

```
 if (retries++ == RETRIES_BEFORE_LOCK) {
                    for (int j = 0; j < segments.length; ++j)
                        ensureSegment(j).lock(); // force creation
                }
```

获取所有锁之后，对每一个Segment的size获取，最后相加返回。

参考链接：

为什么ConcurrentHashMap是弱一致的 (http://ifeve.com/concurrenthashmap-weakly-consistent/)
Under The Covers Of Concurrent Hash Map (https://bansihaudakari.wordpress.com/interview-zone/core-java/concurrency/under-the-covers-of-concurrent-hash-map/)
Java集合---ConcurrentHashMap原理分析 (http://www.cnblogs.com/ITtangtang/p/3948786.html)
Java Core系列之ConcurrentHashMap实现(JDK 1.7) (http://www.blogjava.net/DLevin/archive/2013/10/18/405030.html)
探索 ConcurrentHashMap 高并发性的实现机制 (http://www.ibm.com/developerworks/cn/java/java-lo-concurrenthashmap/)

java基础技术 (/nb/4489989)

举报文章

codertom (/u/cd1316ad27cf)
写了 87837 字，被 96 人关注，获得了 207 个喜欢
(/u/cd1316ad27cf)

+关注

帝都小码农，目前就职帝都一个二线互联网公司，比较关注账户系统和分布式存储计算以及数据处理一些相...

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

赞赏支持

喜欢 (/sign_in) 4

更多分享
(http://cwb.assets.jianshu.io/notes/images/4933572

登录 (/sign_in)后发表评论

评论

智慧如你，不想发表一点想法 (/sign_in)咩~

被以下专题收入，发现更多相似内容

互联网科技
首页投稿
@IT·互联网
程序员