并发容器类Map

JDK源码学习方法

逻辑思维能力是梳理学习方法的基础。养成线性思维：两个或者多个概念，像一条线串起来。

1. 演绎推导

   因果推理。因为JAVA中网络编程只提供了BIO和NIO两种方式，所以一切框架中，涉及到网络处理的，都可以用着两个知识点去探究原理。

2. 归纳总结

   可能正确的猜想。线上10台服务器，有三台总是每天会自动重启，收集相关信息后，发现是运维在修改监控系统配置的时候，漏掉了提高这三台机器的重启阈值。

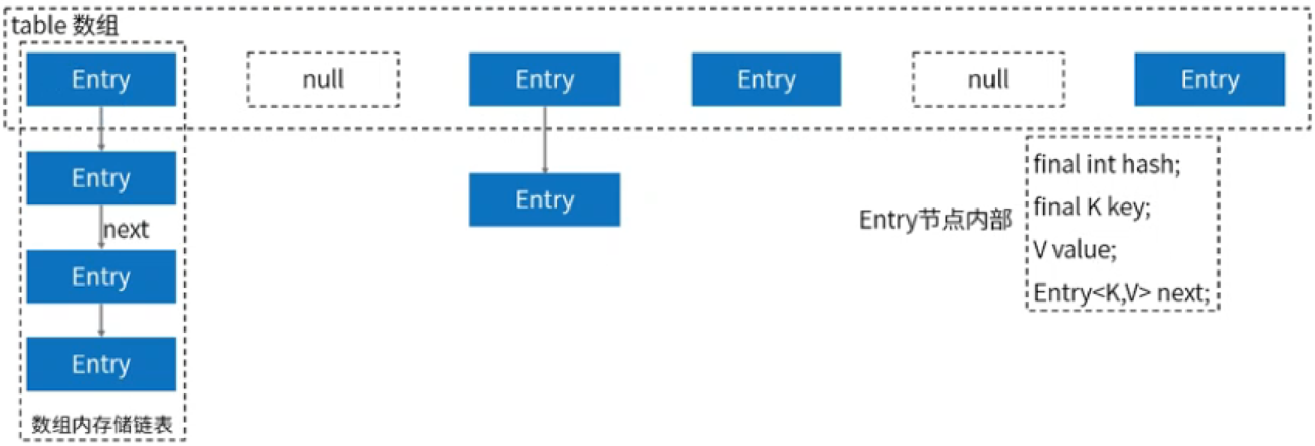
3. 类比法

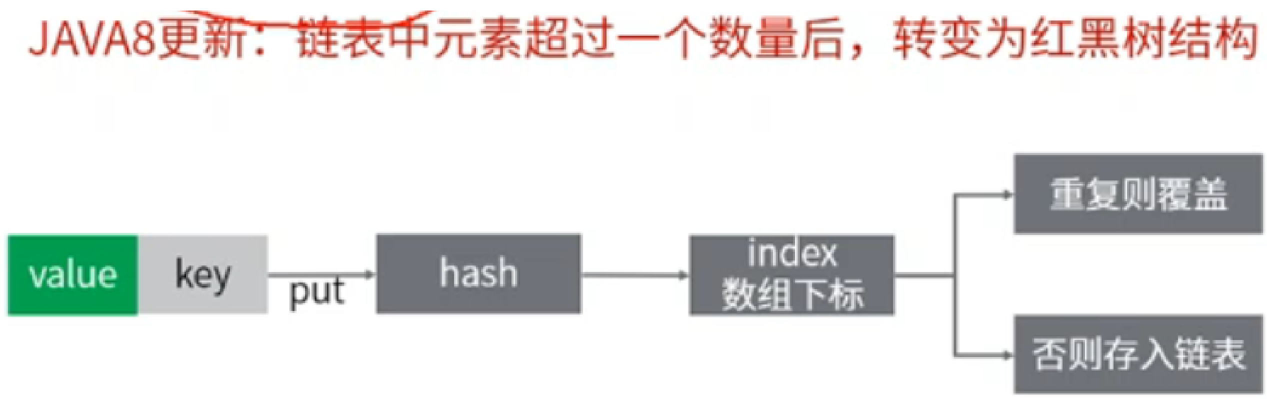
   集群就好像是马在拉车，一匹马拉不动的时候，就使用多匹马去拉。分布式的概念，就像是理发的过程中，洗头发和剪头发是不同的人负责的。

HashMap-非线程安全-JDK1.7

1. 存储结构

   如下图所示：HashMap内部的存储结构是数组加链表，Hash不冲突的元素放在数组不同的位置，冲突的元素组织成链表，采用的是头插法。





```
transient Entry<K,V>[] table;

Entry中的主要属性：

final K key;
V value;
Entry<K,V> next;
int hash;
```

2. 初始化

```
static final int MAXIMUM_CAPACITY = 1 << 30;
static final int DEFAULT_INITIAL_CAPACITY = 16;
static final float DEFAULT_LOAD_FACTOR = 0.75f;
// 元素数量
transient int size;
// 扩容的阈值，等于capacity*loadFactor
int threshold;
// 负载因子
final float loadFactor;

public HashMap() {
        this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR);
}

public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                               loadFactor);

        // 找到小于给定初始容量最接近2的次方的值
        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <<= 1;

        this.loadFactor = loadFactor;
        threshold = (int)Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
        table = new Entry[capacity];
        useAltHashing = sun.misc.VM.isBooted() &&
                (capacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);
        init();
    }
```

3. put方法

```
public V put(K key, V value) {
        if (key == null)
```

```java
            return putForNullKey(value);
        // 生成key的hash值
        int hash = hash(key);
        // 根据hash值定位在数组中的位置
        int i = indexFor(hash, table.length);
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            // key完全一样则覆盖,返回旧值
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }

        modCount++;
        // 生成新的Entry
        addEntry(hash, key, value, i);
        return null;
    }

final int hash(Object k) {
        int h = 0;
        if (useAltHashing) {
            if (k instanceof String) {
                return sun.misc.Hashing.stringHash32((String) k);
            }
            h = hashSeed;
        }

        h ^= k.hashCode();

        h ^= (h >>> 20) ^ (h >>> 12);
        return h ^ (h >>> 7) ^ (h >>> 4);
    }

static int indexFor(int h, int length) {
        return h & (length-1);
    }

void addEntry(int hash, K key, V value, int bucketIndex) {
        // 当map中元素的数量大于阈值并且下一次放入发生冲突时会resize
        // 即HashMap最多在有27个元素时会发生扩容11 + 16
        if ((size >= threshold) && (null != table[bucketIndex])) {
            resize(2 * table.length);
            hash = (null != key) ? hash(key) : 0;
            bucketIndex = indexFor(hash, table.length);
        }

        createEntry(hash, key, value, bucketIndex);
    }
```

```java
void createEntry(int hash, K key, V value, int bucketIndex) {
        Entry<K,V> e = table[bucketIndex];
        // 头插法
        table[bucketIndex] = new Entry<>(hash, key, value, e);
        size++;
    }
```

4. get方法

```java
public V get(Object key) {
        if (key == null)
            return getForNullKey();
        Entry<K,V> entry = getEntry(key);

        return null == entry ? null : entry.getValue();
    }

final Entry<K,V> getEntry(Object key) {
        int hash = (key == null) ? 0 : hash(key);
        // 根据hash值确定在数组中的下标
        for (Entry<K,V> e = table[indexFor(hash, table.length)];
             e != null;
             e = e.next) {
            Object k;
            // 如果在链表中的第一个节点找到就返回
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                return e;
        }
        return null;
    }
```

HashMap-非线程安全-JDK1.8

1. 存储结构

   1.8中在冲突不大的情况下还是数组加链表结构，在冲突严重时会链表会转换成红黑树。

```java
transient Node<K,V>[] table;

// Node的结构
final int hash;
final K key;
V value;
Node<K,V> next;
```

2. 初始化

```java
// 在1.7的基础上增加了如下几个值

// 链表要转换成红黑树的大小阈值
static final int TREEIFY_THRESHOLD = 8;
```

```
// 红黑树转换回链表的大小阈值
static final int UNTREEIFY_THRESHOLD = 6;
// 链表转红黑树的数组大小阈值
static final int MIN_TREEIFY_CAPACITY = 64;

 public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }

public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                               loadFactor);
        this.loadFactor = loadFactor;
        this.threshold = tableSizeFor(initialCapacity);
    }

static final int tableSizeFor(int cap) {
        int n = cap - 1;
        n |= n >>> 1;
        n |= n >>> 2;
        n |= n >>> 4;
        n |= n >>> 8;
        n |= n >>> 16;
        return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
    }
```

3. put方法

```
public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        // 初始化
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        // 不冲突，直接放
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            // key值完全一样则覆盖
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
```

```java
                else if (p instanceof TreeNode)
                    e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
                else {
                    for (int binCount = 0; ; ++binCount) {
                        // 找到链表尾部
                        if ((e = p.next) == null) {
                            // hash冲突插入链表，尾插法
                            p.next = newNode(hash, key, value, null);
                            // 如果链表中元素多余8个转红黑树
                            if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                                treeifyBin(tab, hash);
                            break;
                        }
                        if (e.hash == hash &&
                            ((k = e.key) == key || (key != null && key.equals(k))))
                            break;
                        // 更新循环变量
                        p = e;
                    }
                }
                if (e != null) { // existing mapping for key
                    V oldValue = e.value;
                    if (!onlyIfAbsent || oldValue == null)
                        e.value = value;
                    afterNodeAccess(e);
                    return oldValue;
                }
            }
            ++modCount;
            // 大于阈值直接扩容
            if (++size > threshold)
                resize();
            afterNodeInsertion(evict);
            return null;
        }

    final void treeifyBin(Node<K,V>[] tab, int hash) {
            int n, index; Node<K,V> e;
            // 如果转换红黑树时，数组小组64则扩容
            // 此时发生冲突的主要原因是数组太小
            if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
                resize();
            else if ((e = tab[index = (n - 1) & hash]) != null) {
                TreeNode<K,V> hd = null, tl = null;
                do {
                    TreeNode<K,V> p = replacementTreeNode(e, null);
                    if (tl == null)
                        hd = p;
                    else {
                        p.prev = tl;
                        tl.next = p;
                    }
                    tl = p;
```

```
                } while ((e = e.next) != null);
                if ((tab[index] = hd) != null)
                    hd.treeify(tab);
            }
        }
```

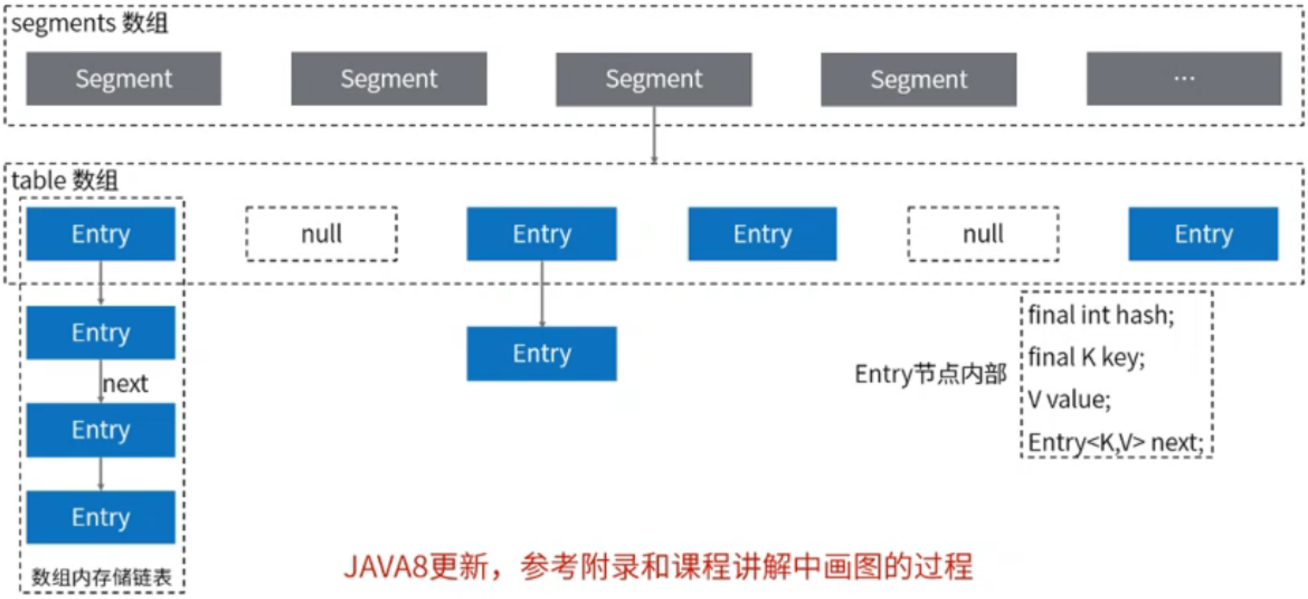
4. get方法

```
public V get(Object key) {
        Node<K,V> e;
        return (e = getNode(hash(key), key)) == null ? null : e.value;
    }

final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            // 链表第一个位置即为要找元素
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
            if ((e = first.next) != null) {
                if (first instanceof TreeNode)
                    // 在红黑树中查找
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do {
                    // 在链表中查找
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

ConcurrentHashMap-线程安全-JDK1.7

HashTable在每个put和get方法加同步锁，效率低，推荐使用ConcurrentHashMap，包含一层segment确保线程安全。



1. 存储结构

   ConcurrentHashMap采用了分段加锁的机制，默认16个段，每个段之间的读写互不影响。

```
final Segment<K,V>[] segments;

// segment其实就是一个锁
static final class Segment<K,V> extends ReentrantLock implements Serializable
// 每个segment里又有一个HashEntry数组
transient volatile HashEntry<K,V>[] table;

// HashEntry结构
final int hash;
final K key;
volatile V value;
volatile HashEntry<K,V> next;
```

2. 初始化

```
static final int DEFAULT_INITIAL_CAPACITY = 16;
static final float DEFAULT_LOAD_FACTOR = 0.75f;
static final int DEFAULT_CONCURRENCY_LEVEL = 16;
static final int MAXIMUM_CAPACITY = 1 << 30;
static final int MIN_SEGMENT_TABLE_CAPACITY = 2;
static final int MAX_SEGMENTS = 1 << 16;

public ConcurrentHashMap() {
        this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR, DEFAULT_CONCURRENCY_LEVEL);
    }

public ConcurrentHashMap(int initialCapacity, float loadFactor) {
        this(initialCapacity, loadFactor, DEFAULT_CONCURRENCY_LEVEL);
    }

public ConcurrentHashMap(int initialCapacity,
```

```
                         float loadFactor, int concurrencyLevel) {
        if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
            throw new IllegalArgumentException();
        if (concurrencyLevel > MAX_SEGMENTS)
            concurrencyLevel = MAX_SEGMENTS;
        // 适合参数的2的次方的值
        int sshift = 0;
        int ssize = 1;
        while (ssize < concurrencyLevel) {
            ++sshift;
            ssize <<= 1;
        }
        this.segmentShift = 32 - sshift;
        this.segmentMask = ssize - 1;
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        int c = initialCapacity / ssize;
        if (c * ssize < initialCapacity)
            ++c;
        int cap = MIN_SEGMENT_TABLE_CAPACITY;
        while (cap < c)
            cap <<= 1;
        // create segments and segments[0]
        Segment<K,V> s0 =
            new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
                             (HashEntry<K,V>[])new HashEntry[cap]);
        Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
        UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]
        this.segments = ss;
    }
```

3. put方法

```
public V put(K key, V value) {
        Segment<K,V> s;
        if (value == null)
            throw new NullPointerException();
        int hash = hash(key);
        // 定位在那个segment
        int j = (hash >>> segmentShift) & segmentMask;
        if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
             (segments, (j << SSHIFT) + SBASE)) == null) //  in ensureSegment
            s = ensureSegment(j);
        return s.put(key, hash, value, false);
    }

final V put(K key, int hash, V value, boolean onlyIfAbsent) {
            // put时给每个段加锁
            HashEntry<K,V> node = tryLock() ? null :
                scanAndLockForPut(key, hash, value);
            V oldValue;
            try {
                HashEntry<K,V>[] tab = table;
```

```
            int index = (tab.length - 1) & hash;
            // 链表第一个元素
            HashEntry<K,V> first = entryAt(tab, index);
            for (HashEntry<K,V> e = first;;) {
                // 冲突
                if (e != null) {
                    K k;
                    // key完全相同覆盖
                    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)
                        // 尾插法
                        node.setNext(first);
                    else
                        // 第一个元素
                        node = new HashEntry<K,V>(hash, key, value, first);
                    int c = count + 1;
                    // 超过阈值并且数组大小小于最大值则扩容
                    if (c > threshold && tab.length < MAXIMUM_CAPACITY)
                        rehash(node);
                    else
                        // 更新segment
                        setEntryAt(tab, index, node);
                    ++modCount;
                    count = c;
                    oldValue = null;
                    break;
                }
            }
        } finally {
            unlock();
        }
        return oldValue;
    }
```

4. get方法

```
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;
        // 定位segment
```

```
        if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
            (tab = s.table) != null) {
            // 判断链表中元素
            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;
    }
```

ConcurrentHashMap-线程安全-JDK1.8

1. 存储结构

   和JDK1.8中的HashMap的存储结构一致，兼容1.7的segment，在冲突严重时会转成红黑树。对线程安全处理方式进行了更新，在没有冲突时采用CAS无锁机制，在有冲突时进行加锁。

2. 初始化

```
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;
    }
```

3. put方法

```
public V put(K key, V value) {
        return putVal(key, value, false);
    }

    final V putVal(K key, V value, boolean onlyIfAbsent) {
        if (key == null || value == null) throw new NullPointerException();
        int hash = spread(key.hashCode());
        int binCount = 0;
        for (Node<K,V>[] tab = table;;) {
            Node<K,V> f; int n, i, fh;
            // 初始化
            if (tab == null || (n = tab.length) == 0)
                tab = initTable();
            // 没有冲突采用cas机制放值
            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
        }
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            // key完全相同覆盖
                            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;
                            // 插入链表尾部
                            if ((e = e.next) == null) {
                                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) {
                // 转换为红黑树
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
```

```
        addCount(1L, binCount);
        return null;
    }
```

4. get方法

```
public V get(Object key) {
        Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
        int h = spread(key.hashCode());
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (e = tabAt(tab, (n - 1) & h)) != null) {
            if ((eh = e.hash) == h) {
                if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                    return e.val;
            }
            else if (eh < 0)
                return (p = e.find(h, key)) != null ? p.val : null;
            while ((e = e.next) != null) {
                if (e.hash == h &&
                    ((ek = e.key) == key || (ek != null && key.equals(ek))))
                    return e.val;
            }
        }
        return null;
    }
```