

ConcurrentHashMap源码阅读

是针对HashMap进行对比，所以源码涉及的不多

概述

1. ConcurrentHashMap使用**分段锁**，能够提高并发度；



1.8之前是用segment进行分段，先定位住segment然后计算；**1.8开始直接是对每个index中的链表进行锁定，进一步减小粒度。**桶HashMap，也是防止链表过长，会进行红黑树的转换。

2. JDK1.6用segment来充当锁，而segment是继承了ReentrantLock，segment维护了哈希表的若干个桶

1.8 版本舍弃了 segment，并且大量使用了 synchronized，以及 CAS 无锁操作以保证

ConcurrentHashMap 操作的线程安全性（synchronized有了较大的改进，用了锁升级，所以性能有了很大的提高）



- 3.

1. 实例变量 & 静态变量

静态的参数很大一部分和HashMap一样。

```
transient volatile Node<K,V>[] table;           // 存储键值对的数组——桶数组

private transient volatile long baseCount;      // 记录结点数，在非并发场景下使用，CAS更新，但是求size时并不会使用
```

被volatile修饰的，保证可见性。



采用懒加载的方式，直到第一次插入数据的时候才会进行初始化操作，数组的大小总是为 2 的幂次方。

```
private transient volatile Node<K,V>[] nextTable; // 扩容时使用，平时为 null，只有在扩容的时候才为非 null
```

```
private transient volatile int sizeCtl;           // 存在并发情况，所以用volatile修饰
```

该属性用来控制 table 数组的大小，根据是否初始化和是否正在扩容有几种情况：

- 当值为负数时：如果为-1 表示正在初始化，如果为-N 则表示当前正有 N-1 个线程进行扩容操作；
- 当值为正数时：如果当前数组为 null 的话表示 table 在初始化过程中，sizeCtl 表示为需要新建数组的长度；若已经初始化了，表示当前数据容器（table 数组）可用容量也可以理解成临界值（插入节点数超过了该临界值就需要扩容），具体指为数组的长度 n 乘以 加载因子 loadFactor；当值为 0 时，即数组长度为默认初始值。

2. Node类

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;           // 因为哈希值是固定的，所以就是普通的变量
    final K key;              // key也是固定的，所以普通变量
    volatile V val;           // 可能存在并发修改值，所以需要volatile修饰
    volatile Node<K,V> next;
    ....
}
```

理解：可以看到val、next都是用了volatile修饰的。

3. 构造方法

和HashMap类似，这边不做研究

4. 实例方法

选择重要的方法进行分析：初始化数组、插入节点、删除节点

1. initTable



```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0) // 如果存在多个线程同时进行哈希表初始化，那么放弃当前时间片
            Thread.yield();
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) { // 将当前状态设置为初始化状态，用CAS
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

这边和HashMap不同在于：存在并发初始化的问题，所以需要保证只有一个线程来进行初始化，其他线程均需要放弃，然后进行初始化的线程将当前状态**设置为初始化状态。**

如果没有设定长度，那么给的默认长度就是16

2. put

整体技术上：使用了 **synchronized** 和 **CAS** 的方式

如果出现冲突，用的是拉链法，就是在节点后面链接

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode()); // 计算key的哈希值，即在原来的哈希值的基础上再将高16位和低16位进行异或
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            tab = initTable(); // 数组没有初始化，那就去初始化
        // 如果插入的index不存在节点，那么用CAS直接插入（就是链表头）
        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
        }
        /*如果当前节点不为 null，且该节点为特殊节点（forwardingNode）的话，就说明当前正在进行扩容操作
        通过判断该节点的 hash 值是不是等于-1，就能确定节点是否在扩容
        */
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        /* table[i]不为 null 并且不为 forwardingNode 时，并且当前 Node f 的 hash 值大于 0 (fh >= 0) 的话
        说明当前节点 f 为当前桶的所有的节点组成的链表的头结点
        */
        else {
            V oldVal = null;
            synchronized (f) { // synchronized修饰
                if (tabAt(tab, i) == f) { // 在链表重插入新的键值对——同普通的
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) { // 如果遇到相同的就
                            K ek;
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                        }
                        Node<K,V> pred = e;
                        if ((e = e.next) == null) {
                            pred.next = new Node<K,V>(hash, key,
                                value, null);
                            break;
                        }
                    }
                }
            }
        }
    }
}
```

```

        else if (f instanceof TreeBin) { // 按照红黑树的结构

            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) {
        if (binCount >= TREEIFY_THRESHOLD)
            treeifyBin(tab, i);
        if (oldVal != null)
            return oldVal;
        break;
    }
}
addCount(1L, binCount); // 如果增加之后的容量超过阈值，那么需要进行扩容
return null;
}

```



```

static final int spread(int h) { // 重新进行哈希，以减少冲突
    return (h ^ (h >>> 16)) & HASH_BITS;
}

static final int HASH_BITS = 0x7fffffff; // 除去符号位

```

理解：

1. 可以看到在原来哈希值的基础上再次进行哈希，就是将高16位和低16位进行异或，防止对象的哈希算法只涉及到了低位。通过高低位共同参与，那么能够减少哈希的冲突数 
 2. 整体流程：
 1. 首先，根据原来的哈希值，再次进行哈希，减少冲突
 2. 先判断table是否还未初始化，如果未初始化，那么就去初始化数组，默认长度为16 
 3. 初始化之后，如果哈希值对应的index内容是null，那么直接用CAS赋值即可
 4. 如果index存在内容，且该节点 fh==MOVED(代表 forwardingNode,数组正在进行扩容)，那么让他先扩容
 5. 如果不在扩容，那么获得synchronized，后获得index对应的头结点，从头结点开始遍历，如果找到重复的key，那么覆盖之前的value；如果到尾巴也没有找到key，那么将结点插入到尾巴；如果是红黑树，那么按照红黑树的方式插入
- 如果链表长度超过指定长度之后 (>8)，那么将链表转换为红黑树；
- 如果哈希表的容量超过阈值之后，进行扩容

3. get

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    int h = spread(key.hashCode()); // 重新计算哈希，否则找不到
    if ((tab = table) != null && (n = tab.length) > 0 && //已经初始化了，且存在结
        点，那么定位到指定的index
        (e = tabAt(tab, (n - 1) & h)) != null) {
        if ((eh = e.hash) == h) { // 从头开始查找，如果头结点就是查找的，那么直接
            返回
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        else if (eh < 0) // 哈希值小于0，表示是红黑树结构，按照红黑树的方式去查找
            return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) { // 从head开始查找，直到找到key对象 or 满
            足key的equal的
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

由于不需要写，所以和普通的HashMap一样

4. 扩容：transfer

比HashMap差别最大的就是扩容操作。

总体思路都是HashMap一致。

```
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) { // 辅助数组，创建一个新的node数组，是之前容量的2倍
        (2倍的index)
        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); // 新建
    forwardingNode引用，在之后会用到，哈希值就是MOVED
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab
    for (int i = 0, bound = 0;;) {
```

```

Node<K,V> f; int fh;
while (advance) {
    int nextIndex, nextBound;
    if (--i >= bound || finishing)
        advance = false;
    else if ((nextIndex = transferIndex) <= 0) {
        i = -1;
        advance = false;
    }
    else if (U.compareAndSwapInt
        (this, TRANSFERINDEX, nextIndex,
         nextBound = (nextIndex > stride ?
                     nextIndex - stride : 0))) {
        bound = nextBound;
        i = nextIndex - 1;
        advance = false;
    }
}
// 将原数组的元素复制到新的数组中
if (i < 0 || i >= n || i + n >= nextn) {
    int sc;
    if (finishing) { // 当扩容结束之后, 设置sizeCtl属性
        nextTable = null;
        table = nextTab; // 将完成扩容的辅助数组赋值给table, 那么就
完成了更新

        sizeCtl = (n << 1) - (n >>> 1);
        return;
    }
    if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
        if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
            return;
        finishing = advance = true;
        i = n; // recheck before commit
    }
}

else if ((f = tabAt(tab, i)) == null) // 当前index为null, 不存在结点, 用
CAS设置成特殊节点forwardingNode
    advance = casTabAt(tab, i, null, fwd);
else if ((fh = f.hash) == MOVED) // 如果遍历到ForwardingNode节点 说明
这个点已经被处理过了 直接跳过——这个就是针对并发的情况
    advance = true; // already processed
else {
    synchronized (f) { // 互斥执行, 这个就是同HashMap中的, 将链表的结点分为两个
链表, 然后一个插入高位index
        if (tabAt(tab, i) == f) {
            Node<K,V> ln, hn;
            if (fh >= 0) {
                int runBit = fh & n;
                Node<K,V> lastRun = f;
                for (Node<K,V> p = f.next; p != null; p = p.next) {
                    int b = p.hash & n;
                    if (b != runBit) {
                        runBit = b;
                        lastRun = p;
                    }
                }
            }
            if (runBit == 0) {
                ln = lastRun;

```

fwd, 表示已经访问过了

```
        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); // 将原数组的index设置为
    advance = true;
}
else if (f instanceof TreeBin) { // 红黑树的操作
    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;
}
}
}
}
}
```

```
}
```

基本思想还是同HashMap，只不过要考虑并发，在遍历链表的时候用到了synchronized，其余用到了一个关键的fwd，对已经处理过的index（可能结点为空，也可能index对应的链表/红黑树已经处理完成），那么都会用fwd来占位，一旦出现这个表示已经处理过了，需要跳过——从而实现并发扩容。

5. 计算结点个数

由于存在并发，所以不能准确计算出结点个数，只能估个大概。

addCount

主要进行2个操作：增加baseCount的值；检测是否需要扩容。

```
private final void addCount(long x, int check) {
    CounterCell[] as; long b, s;
    if ((as = counterCells) != null || // CAS的方法更新baseCount值
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            !(uncontended =
                U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        s = sumCount();
    }
    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) {
                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);
            s = sumCount();
        }
    }
}
```

参考：

1. <https://juejin.cn/post/6844903602423595015#comment>

ps：附加知识

1. HashMap多线程安全的表现

注意，由于JDK8之后，插入方式都采用尾插法，扩容也是分为两个链表进行尾插法，所以不会存在死循环的情况了

但是hashMap仍是多线程安全的

1.1 多线程put会导致元素丢失

如果多个线程并发插入可能会存在问题

举例场景：如果存在链表，然后两个线程均需要插入1个结点到尾巴上

```
if ((e = p.next) == null) { // 关键1，此时p就是tail结点
    p.next = newNode(hash, key, value, null); // 关键2
    if (binCount >= TREEIFY_THRESHOLD - 1) // 变成红黑树（无关）
        treeifyBin(tab, hash);
    break;
}
```

描述：如果线程A执行到1，此时已经获得tail了，为p；线程A被切出，然后线程B开始执行，B也执行到1，然后执行到2；即在tail后面添加了一个结点，即tail.next=线程Bnew的结点；后线程B被切出，线程A开始执行，而A的tail还是原来的值，那么tail.next=线程Anew的结点

可以发现B创建的结点被A覆盖了

1.2 put和get并发，可能会导致get为null

在put的时候超过阈值，需要扩容。

```
// hash表
transient Node<K,V>[] table;

final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    ....
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap]; // #1
    table = newTab; // #2
    ....
    return table;
}
```

在返回table之前，如果进行get查询，那么得到值都是null

1.3 JDK7中的死循环

具体见<https://coolshell.cn/articles/9606.html>

2. 不采用HashTable的原因

HashTable是最早的并发API，但是它是直接对put、get方法加上**synchronized**来**保证线程安全**。安全性保证，但是性能很差——加上一个大锁，那么线程不能同时访问该对象，性能就差很多。