

## 为并发而生的 ConcurrentHashMap (Java 8)

HashMap 是我们日常最常见的一种容器，它以键值对的形式完成对数据的存储，但众所周知，它在高并发的情境下是不安全的。尤其是在 jdk 1.8 之前，rehash 的过程中采用头插法转移结点，高并发下，多个线程同时操作一条链表将直接导致闭链，死循环并占满 CPU。

当然，jdk 1.8 以来，对 HashMap 的内部进行了很大的改进，采用数组+链表+红黑树来进行数据的存储。rehash 的过程也进行了改动，基于复制的算法思想，不直接操作原链，而是定义了两条链表分别完成对原链的结点分离操作，即使是多线程的情况下也是安全的。当然，它毕竟不是并发容器，除非大改，否则依然是不能应对高并发场景的，或者说即使没有因多线程访问而造成什么问题，但是效率必然是受到影响的。例如在多线程同时添加元素的时候可能会丢失数据，迭代 map 的时候发生 fail-fast 等。

本篇文章将要介绍的 ConcurrentHashMap 是 HashMap 的并发版本，它是线程安全的，并且在高并发的情境下，性能优于 HashMap 很多。我们主要从以下几个方面对其进行学习：

- 历史版本的实现演变
- 重要成员属性的介绍
- put 方法实现并发添加
- remove 方法实现并发删除
- 其他的一些方法的简单介绍

### 一、历史版本的实现演变

jdk 1.7 采用分段锁技术，整个 Hash 表被分成多个段，每个段中会对应一个 Segment 段锁，段与段之间可以并发访问，但是多线程想要操作同一个段是需要获取锁的。所有的 put, get, remove 等方法都是根据键的 hash 值对应到相应的段中，然后尝试获取锁进行访问。



jdk 1.8 取消了基于 Segment 的分段锁思想，改用 CAS + synchronized 控制并发操作，在某些方面提升了性能。并且追随 1.8 版本的 HashMap 底层实现，使用数组+链表+红黑树进行数据存储。本篇主要介绍 1.8 版本的 ConcurrentHashMap 的具体实现，有关其之前版本的实现情况，这里推荐几篇文章：

[谈谈ConcurrentHashMap1.7和1.8的不同实现](#)  
[ConcurrentHashMap在jdk1.8中的改进](#)  
[ConcurrentHashMap原理分析（1.7与1.8）](#)

### 二、重要成员属性的介绍

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

和 HashMap 中的语义一样，代表整个哈希表。

```
/**
 * The next table to use; non-null only while resizing.
 */
private transient volatile Node<K,V>[] nextTable;
```

这是一个连接表，用于哈希表扩容，扩容完成后会被重置为 null。

```
private transient volatile long baseCount;
```

该属性保存着整个哈希表中存储的所有的结点的个数总和，有点类似于 HashMap 的 size 属性。

```
private transient volatile int sizeCtl;
```

这是一个重要的属性，无论是初始化哈希表，还是扩容 rehash 的过程，都是需要依赖这个关键属性的。该属性有以下几种取值：

#### 公告

昵称：Single\_Yam  
园龄：3年  
粉丝：230  
关注：23  
[+加关注](#)

< 2019年6			
日	一	二	三
26	27	28	29
2	3	4	5
9	10	11	12
16	17	18	19
23	24	25	26
30	1	2	3

#### 搜索

#### 随笔分类

Git版本控制(2)

Hibernate(7)

java web(10)

Java并发(9)

java高级(17)

java基础(11)

OverView(47)

Spring(5)

- 0：默认值
- -1：代表哈希表正在进行初始化
- 大于0：相当于 HashMap 中的 threshold，表示阈值
- 小于-1：代表有多个线程正在进行扩容

该属性的使用还是有点复杂的，在我们分析扩容源码的时候再给予更加详尽的描述，此处了解其可取的几个值都分别代表着什么样的含义即可。

构造函数的实现也和我们上篇介绍的 HashMap 的实现类似，此处不再赘述，贴出源码供大家比较。

```
public ConcurrentHashMap(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
        MAXIMUM_CAPACITY :
        tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap;
}
```

其他常用的方法我们将在文末进行简单介绍，下面我们主要来分析下 ConcurrentHashMap 的一个核心方法 put，我们也会一并解决掉该方法中涉及到的扩容、辅助扩容，初始化哈希表等方法。

三、put 方法实现并发添加

对于 HashMap 来说，多线程并发添加元素会导致数据丢失等并发问题，那么 ConcurrentHashMap 又是如何做到并发添加的呢？

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

putVal 的方法比较多，我们分两个部分进行分析。

```
//第一部分
final V putVal(K key, V value, boolean onlyIfAbsent) {
    //对传入的参数进行合法性判断
    if (key == null || value == null) throw new NullPointerException();
    //计算键所对应的 hash 值
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        //如果哈希表还未初始化，那么初始化它
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        //根据键的 hash 值找到哈希数组相应的索引位置
        //如果为空，那么以CAS无锁式向该位置添加一个节点
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null,
                new Node<K,V>(hash, key, value, null)))
                break;
        }
    }
```

这里需要详细说明的只有 initTable 方法，这是一个初始化哈希表的操作，它同时只允许一个线程进行初始化操作。

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    //如果表为空才进行初始化操作
    while ((tab = table) == null || tab.length == 0) {
        //sizeCtl 小于零说明已经有线程正在进行初始化操作
        //当前线程应该放弃 CPU 的使用
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        //否则说明还未有线程对表进行初始化，那么本线程就来做这个工作
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            //保险起见，再次判断下表是否为空
            try {
                if ((tab = table) == null || tab.length == 0) {
                    //sc 大于零说明容量已经初始化了，否则使用默认容量
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    //根据容量构建数组
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    //计算阈值，等效于 n*0.75
                    sc = n - (n >>> 2);
                }
            } finally {
                //设置阈值
                sizeCtl = sc;
            }
            break;
        }
    }
}
```

Struts2(9)

计算机基础(1)

数据结构和算法(1)

译文集(1)

随笔档案

2019年3月 (1)

2019年2月 (5)

2019年1月 (7)

2018年10月 (1)

2018年9月 (3)

2018年8月 (2)

2018年7月 (4)

2018年6月 (6)

2018年5月 (4)

2018年4月 (9)

2018年3月 (7)

2017年12月 (5)

2017年11月 (7)

2017年10月 (7)

2017年9月 (5)

2017年8月 (2)

2017年7月 (2)

2017年6月 (3)

2017年5月 (12)

2017年4月 (13)

2017年3月 (10)

```
        return tab;
    }
}
```

关于 initTable 方法的每一步实现都已经给出注释，该方法的核心思想就是，只允许一个线程对表进行初始化，如果不巧有其他线程进来了，那么会让其他线程交出 CPU 等待下次系统调度。这样，保证了表同时只会被一个线程初始化。

接着，我们回到 putVal 方法，这样的话，我们第一部分的 putVal 源码就分析结束了，下面我们看后一部分的源码：

```
//检测到桶结点是 ForwardingNode 类型，协助扩容
else if ((fh = f.hash) == MOVED)
    tab = helpTransfer(tab, f);
//桶结点是普通的结点，锁住该桶头结点并试图在该链表的尾部添加一个节点
else {
    V oldVal = null;
    synchronized (f) {
        if (tabAt(tab, i) == f) {
            //向普通的链表中添加元素，无需赘述
            if (fh >= 0) {
                binCount = 1;
                for (Node<K,V> e = f;; ++binCount) {
                    K ek;
                    if (e.hash == hash && ((ek = e.key) == key || (ek != null && key.equals(ek)))) {
                        oldVal = e.val;
                        if (!onlyIfAbsent)
                            e.val = value;
                        break;
                    }
                    Node<K,V> pred = e;
                    if ((e = e.next) == null) {
                        pred.next = new Node<K,V>(hash, key, value, null);
                        break;
                    }
                }
            }
            //向红黑树中添加元素，TreeBin 结点的hash值为TREEBIN (-2)
            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;
                }
            }
        }
    }
    //binCount != 0 说明向链表或者红黑树中添加或修改一个节点成功
    //binCount == 0 说明 put 操作将一个新节点添加成为某个桶的首节点
    if (binCount != 0) {
        //链表深度超过 8 转换为红黑树
        if (binCount >= TREEIFY_THRESHOLD)
            treeifyBin(tab, i);
        //oldVal != null 说明此次操作是修改操作
        //直接返回旧值即可，无需做下面的扩容边界检查
        if (oldVal != null)
            return oldVal;
        break;
    }
}
//CAS 式更新baseCount，并判断是否需要扩容
addCount(1L, binCount);
//程序走到这一步说明此次 put 操作是一个添加操作，否则早就 return 返回了
return null;
}
```

这一部分的源码大体上已如注释所描述，至此整个 putVal 方法的大体逻辑实现相信你也已经清晰了，好好回味一下。下面我们对这部分中的某些方法的实现细节再做一些深入学习。

首先需要介绍一下，ForwardingNode 这个节点类型，

```
static final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        //注意这里
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
    //省略其 find 方法
}
```

这个节点内部保存了一 nextTable 引用，它指向一张 hash 表。在扩容操作中，我们需要对每个桶中的结点进行分离和转移，如果某个桶结点中所有结点都已经迁移完成了（已经被转移到新表 nextTable 中了），那么会在原 table 表的该位置挂上一个 ForwardingNode 结点，说明此桶已经完成迁移。

2017年2月 (7)

最新评论

1. Re:为并发而生的 ConcurrentHashMap (Java 8)

图片挂了

2. Re:Spring框架学习位置 (一)

赞赞赞赞赞赞赞赞赞赞

3. Re:JAVA 注解的基本原理

图片挂了。请更新。楼主的

4. Re:线程间的协作机制

非常NICE，对我很有帮助

5. Re:弄懂 JRE、JDK、JVM 的区别与联系

多谢分享

阅读排行榜

1. JAVA 注解的基本原理

2. Java 字节流操作(1:OutputStream)

3. Java 的字节流文件读写操作(1:FileInputStream)

4. 弄懂 JRE、JDK、JVM 的联系(11074)

5. Java并发之线程中断

评论排行榜

1. 全面理解java异常机制

2. Maven 整合 SSH 框架

ForwardingNode 继承自 Node 结点，并且它唯一的构造函数将构建一个键，值，next 都为 null 的结点，反正它就是个标识，无需那些属性。但是 hash 值却为 MOVED。

所以，我们在 putVal 方法中遍历整个 hash 表的桶结点，如果遇到 hash 值等于 MOVED，说明已经有线程正在扩容 rehash 操作，整体上还未完成，不过我们要插入的桶的位置已经完成了所有节点的迁移。

由于检测到当前哈希表正在扩容，于是让当前线程去协助扩容。

```
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) {
        //返回一个 16 位长度的扩容校验标识
        int rs = resizeStamp(tab.length);
        while (nextTab == nextTable && table == tab &&
            (sc = sizeCtl) < 0) {
            //sizeCtl 如果处于扩容状态的话
            //前 16 位是数据校验标识，后 16 位是当前正在扩容的线程总数
            //这里判断校验标识是否相等，如果校验符不等或者扩容操作已经完成了，直接退出循环，不用协助它们扩容了
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                sc == rs + MAX_RESIZERS || transferIndex <= 0)
                break;
            //否则调用 transfer 帮助它们进行扩容
            //sc + 1 标识增加了一个线程进行扩容
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                transfer(tab, nextTab);
                break;
            }
        }
        return nextTab;
    }
    return table;
}
```

下面我们看这个稍显复杂的 transfer 方法，我们依然分几个部分来说。

```
//第一部分
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    int n = tab.length, stride;
    //计算单个线程允许处理的最少table桶首节点个数，不能小于 16
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE;
    //刚开始扩容，初始化 nextTab
    if (nextTab == null) {
        try {
            @SuppressWarnings("unchecked")
            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
            nextTab = nt;
        } catch (Throwable ex) {
            sizeCtl = Integer.MAX_VALUE;
            return;
        }
        nextTable = nextTab;
        //transferIndex 指向最后一个桶，方便从后向前遍历
        transferIndex = n;
    }
    int nextn = nextTab.length;
    //定义 ForwardingNode 用于标记迁移完成的桶
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
```

这部分代码还是比较简单的，主要完成的是对单个线程能处理的最少桶结点个数的计算和一些属性的初始化操作。

```
//第二部分，并发扩容控制的核心
boolean advance = true;
boolean finishing = false;
//i 指向当前桶，bound 指向当前线程需要处理的桶结点的区间下限
for (int i = 0, bound = 0;;) {
    Node<K,V> f; int fh;
    //这个 while 循环的目的就是通过 --i 遍历当前线程所分配到的桶结点
    //一个桶一个桶的处理
    while (advance) {
        int nextIndex, nextBound;
        if (--i >= bound || finishing)
            advance = false;
        //transferIndex <= 0 说明已经没有需要迁移的桶了
        else if ((nextIndex = transferIndex) <= 0) {
            i = -1;
            advance = false;
        }
        //更新 transferIndex
        //为当前线程分配任务，处理的桶结点区间为 (nextBound,nextIndex)
        else if (U.compareAndSwapInt(this, TRANSFERINDEX, nextIndex,nextBound = (nextIndex > stride ? nextIndex -
            stride : 0))) {
            bound = nextBound;
            i = nextIndex - 1;
        }
    }
}
```

3. 初识Hibernate之环

4. Java并发之线程间的

5. Hibernate框架学习  
(4)

推荐排行榜

1. 完整的一次 HTTP 请  
(一) (13)

2. 揭秘 HashMap 实现  
(12)

3. JAVA 注解的基本原

4. 为并发而生的 Conci  
(Java 8) (7)

5. 面试中常用排序算法

```
        advance = false;
    }
}
//当前线程所有任务完成
if (i < 0 || i >= n || i + n >= nextn) {
    int sc;
    if (finishing) {
        nextTable = null;
        table = nextTab;
        sizeCtl = (n << 1) - (n >>> 1);
        return;
    }
    if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
        if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
            return;
        finishing = advance = true;
        i = n;
    }
}
//待迁移桶为空,那么在此位置 CAS 添加 ForwardingNode 结点标识该桶已经被处理过了
else if ((f = tabAt(tab, i)) == null)
    advance = casTabAt(tab, i, null, fwd);
//如果扫描到 ForwardingNode,说明此桶已经被处理过了,跳过即可
else if ((fh = f.hash) == MOVED)
    advance = true;
```

每个新参加进来扩容的线程必然先进 while 循环的最后一个判断条件中去领取自己需要迁移的桶的区间。然后 i 指向区间的最后一个位置,表示迁移操作从后往前的做。接下来的几个判断就是实际的迁移结点操作了。等我们大致介绍完成第三部分的源码再回来对各个判断条件下的迁移过程进行详细的叙述。


```
//第三部分
else {
    //
    synchronized (f){
        if (tabAt(tab, i) == f) {
            Node<K,V> ln, hn;
            //链表的迁移操作
            if (fh >= 0) {
                int runBit = fh & n;
                Node<K,V> lastRun = f;
                //整个 for 循环为了找到整个桶中最后连续的 fh & n 不变的结点
                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;
                }
                //如果fh&n不变的链表的runbit都是0,则nextTab[i]内元素ln前逆序,ln及其之后顺序
                //否则,nextTab[i+n]内元素全部相对原table逆序
                //这是通过一个节点一个节点的往nextTab添加
                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);
                }
                //把两条链表整体迁移到nextTab中
                setTabAt(nextTab, i, ln);
                setTabAt(nextTab, i + n, hn);
                //将原桶标识位已经处理
                setTabAt(tab, i, fwd);
                advance = true;
            }
            //红黑树的复制算法,不再赘述
            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 = p;
                    }
                    else {
                        if ((p.next = hiTail) == null)
                            hi = p;
                        else
                            hiTail = p;
                    }
                    ++hc;
                }
                //把两条链表整体迁移到nextTab中
                setTabAt(nextTab, i, lo);
                setTabAt(nextTab, i + n, hi);
                //将原桶标识位已经处理
                setTabAt(tab, i, fwd);
                advance = true;
            }
        }
    }
}
```

```
        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;
}
```

那么至此，有关迁移的几种情况已经介绍完成了，下面我们整体上把控一下整个扩容和迁移过程。

首先，每个线程进来会先领取自己的任务区间，然后开始 `--i` 来遍历自己的任务区间，对每个桶进行处理。如果遇到桶的头结点是空的，那么使用 `ForwardingNode` 标识该桶已经被处理完成了。如果遇到已经处理完成的桶，直接跳过进行下一个桶的处理。如果是正常的桶，对桶首节点加锁，正常的迁移即可，迁移结束后依然会将原表的该位置标识位已经处理。

当 `i < 0`，说明本线程处理速度够快的，整张表的最后一部分已经被它处理完了，现在需要看看是否还有其他线程在自己的区间段还在迁移中。这是退出的逻辑判断部分：

这里写图片描述

`finnish` 是一个标志，如果为 `true` 则说明整张表的迁移操作已经全部完成了，我们只需要重置 `table` 的引用并将 `nextTable` 赋为空即可。否则，CAS 式的将 `sizeCtl` 减一，表示当前线程已经完成了任务，退出扩容操作。

如果退出成功，那么需要进一步判断是否还有其他线程仍然在执行任务。

```
if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
    return;
```

我们说过 `resizeStamp(n)` 返回的是对 `n` 的一个数据校验标识，占 16 位。而 `RESIZE_STAMP_SHIFT` 的值为 16，那么位运算后，整个表达式必然在右边空出 16 个零。也正如我们所说的，`sizeCtl` 的高 16 位为数据校验标识，低 16 位为表示正在进行扩容的线程数量。

`(resizeStamp(n) << RESIZE_STAMP_SHIFT) + 2` 表示当前只有一个线程正在工作，相对应的，如果 `(sc - 2) == resizeStamp(n) << RESIZE_STAMP_SHIFT`，说明当前线程就是最后一个还在扩容的线程，那么会将 `finishing` 标识为 `true`，并在下一次循环中退出扩容方法。

这一块的难点在于对 `sizeCtl` 的各个值的理解，关于它的深入理解，这里推荐一篇文章。

#### [着重理解位操作](#)

看到这里，真的为 Doug Lea 精妙的设计而折服，针对于多线程访问问题，不但没有拒绝式得将他们阻塞在门外，反而邀请他们来帮忙一起工作。

好了，我们一路往回走，回到我们最初分析的 `putVal` 方法。接着前文的分析，当我们根据 `hash` 值，找到对应的桶结点，如果发现该结点为 `ForwardingNode` 结点，表明当前的哈希表正在扩容和 `rehash`，于是将本线程送进去帮忙扩容。否则如果是普通的桶结点，于是锁住该桶，分链表和红黑树的插入一个节点，具体插入过程类似 `HashMap`，此处不再赘述。

当我们成功的添加完成一个结点，最后是需要判断添加操作后是否会导致哈希表达到它的阈值，并针对不同情况决定是否需要进行扩容，还有 CAS 式更新哈希表实际存储的键值对数量。这些操作都封装在 `addCount` 这个方法中，当然 `putVal` 方法的最后必然会调用该方法进行处理。下面我们看看该方法的具体实现，该方法主要做两件事情。一是更新 `baseCount`，二是判断是否需要扩容。

```
//第一部分，更新 baseCount
private final void addCount(long x, int check) {
    CounterCell[] as; long b, s;
    //如果更新失败才会进入的 if 的主体代码中
    //s = b + x 其中 x 等于 1
    if ((as = counterCells) != null ||
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
        //高并发下 CAS 失败会执行 fullAddCount 方法
        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();
    }
}
```

这一部分主要完成的是对 baseCount 的 CAS 更新。

```
//第二部分,判断是否需要扩容
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();
    }
}
```

这部分代码也是比较简单的,不再赘述。

至此,对于 put 方法的源码分析已经完全结束了,很复杂但也很让人钦佩。下面我们简单看看 remove 方法的实现。

#### 四、remove 方法实现并发删除

在我们分析完 put 方法的源码之后,相信 remove 方法对你而言就比较轻松了,无非就是先定位再删除的复合。

限于篇幅,我们这里简单的描述下 remove 方法的并发删除过程。

首先遍历整张表的桶结点,如果表还未初始化或者无法根据参数的 hash 值定位到桶结点,那么将返回 null。

如果定位到的桶结点类型是 ForwardingNode 结点,调用 helpTransfer 协助扩容。

否则就老老实实的给桶加锁,删除一个节点。

最后会调用 addCount 方法 CAS 更新 baseCount 的值。

#### 五、其他的一些常用方法的基本介绍

最后我们在补充一些 ConcurrentHashMap 中的小而常用的方法的介绍。

##### 1、size

size 方法的作用是为我们返回哈希表中实际存在的键值对的总数。

```
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 : (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE : (int)n);
}
```

```
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;
}
```

可能你会有所疑问,ConcurrentHashMap 中的 baseCount 属性不就是记录的所有键值对的总数吗?直接返回它不就行了吗?

之所以没有这么做,是因为我们的 addCount 方法用于 CAS 更新 baseCount,但很有可能在高并发的情况下,更新失败,那么这些节点虽然已经被添加到哈希表中了,但是数量却没有被统计。

还好,addCount 方法在更新 baseCount 失败的时候,会调用 fullAddCount 将这些失败的结点包装成一个 CounterCell 对象,保存在 CounterCell 数组中。那么整张表实际的 size 其实是 baseCount 加上 CounterCell 数组中元素的个数。

##### 2、get

get 方法可以根据指定的键,返回对应的键值对,由于是读操作,所以不涉及到并发问题。源码也是比较简单的。

```
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;
}
```

3. clear

clear 方法将删除整张哈希表中所有的键值对，删除操作也是一个桶一个桶的进行删除。

```
public void clear() {
    long delta = 0L; // negative number of deletions
    int i = 0;
    Node<K,V>[] tab = table;
    while (tab != null && i < tab.length) {
        int fh;
        Node<K,V> f = tabAt(tab, i);
        if (f == null)
            ++i;
        else if ((fh = f.hash) == MOVED) {
            tab = helpTransfer(tab, f);
            i = 0; // restart
        }
        else {
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    Node<K,V> p = (fh >= 0 ? f : (f instanceof TreeBin) ? ((TreeBin<K,V>)f).first : null);
                    //循环到链表或者红黑树的尾部
                    while (p != null) {
                        --delta;
                        p = p.next;
                    }
                    //首先删除链、树的末尾元素，避免产生大量垃圾
                    //利用CAS无锁置null
                    setTabAt(tab, i++, null);
                }
            }
        }
        ++i;
    }
    if (delta != 0L)
        addCount(delta, -1);
}
```

到此为止，有关这个为并发而生的 ConcurrentHashMap 内部的核心的部分，我们已经通过源码进行了分析。确实挺费脑，再次膜拜下jdk大神们的智慧。总结不到之处，望指出！

分类： Java并发

好文要顶

关注我

收藏该文

Single\_Yam

关注 - 23

粉丝 - 230

+加关注

7

0

« 上一篇：本博客申明

» 下一篇：基于跳跃表的 ConcurrentSkipListMap 内部实现 (Java 8)

posted @ 2017-12-13 09:20 Single\_Yam 阅读(7365) 评论(4) 编辑 收藏

评论列表

# 1楼 2017-12-13 14:35 javaEE小菜鸟

讲解很仔细，不容易

支持(0) 反对(0)

# 2楼 2018-02-23 16:06 Shadowdsp

```Java

//如果fh&n不变的链表的runbit都是0，则nextTab[i]内元素In前逆序，In及其之后顺序

//否则，nextTab[i+n]内元素全部相对原table逆序

```

请问transfer()里面，我怎么感觉如果Runbit是1的话，那么hn也是和In一样，在lastRun之后顺序，在lastRun之前逆序？



支持(0) 反对(0)

# 3楼 2018-08-18 09:40 雨山木工

写的不错呢，自己读helpTransfer有点懵逼，这个类东西太多，看了楼主的博客，清晰了不少

支持(0) 反对(0)

# 4楼 2019-05-31 20:28 Only雪里梅

图片挂了

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问](#) [网站首页](#)。

- 【推荐】超50万C++/C#源码: 大型实时仿真组态图形源码
- 【前端】SpreadJS表格控件，可嵌入系统开发的在线Excel
- 【推荐】码云企业版，高效的企业级软件协作开发管理平台
- 【推荐】程序员问答平台，解决您开发中遇到的技术难题

相关博文：

- Java容器：HashTable,synchronizedMap与ConcurrentHashMap
- ConcurrentHashMap实现解析
- 高并发下的HashMap，ConcurrentHashMap
- Java并发容器——ConcurrentSkipListMap和ConcurrentHashMap
- Java ConcurrentHashMap

最新新闻：

- 深空原子钟：让航天器自主导航
- 一线丨中兴发布下一代8K大视频智能机顶盒 可语音遥控电视
- 美宇航局发布谷神星新图像：神秘高山成因成谜
- 马斯克身家从哪里来？SpaceX占2/3而特斯拉仅占1/3
- “电子足球”分子帮助解开一个巨大的星际之谜
- » 更多新闻...