死磕 java集合之ConcurrentHashMap源码分析（一）

原创 唐彤 彤哥读源码 2019-04-09

收录于话题
#JDK源码

71个

欢迎关注我的公众号“彤哥读源码”，查看更多源码系列文章，与彤哥一起畅游源码的海洋。

前记，从这篇文章开始我们换一种学习的方式，彤哥先抛出问题，大家尝试着在脑海中回答这些问题，然后再进入我们的源码分析过程，最后彤哥再挑几个问题回答。

开篇问题

（1）ConcurrentHashMap与HashMap的数据结构是否一样？

（2）HashMap在多线程环境下何时会出现并发安全问题？

（3）ConcurrentHashMap是怎么解决并发安全问题的？

（4）ConcurrentHashMap使用了哪些锁？

（5）ConcurrentHashMap的扩容是怎么进行的？

（6）ConcurrentHashMap是否是强一致性的？

（7）ConcurrentHashMap不能解决哪些问题？

（8）ConcurrentHashMap中有哪些不常见的技术值得学习？

简介

ConcurrentHashMap是HashMap的线程安全版本，内部也是使用（数组 + 链表 + 红黑树）的结构来存储元素。

相比于同样线程安全的HashTable来说，效率等各方面都有极大地提高。

各种锁简介

这里先简单介绍一下各种锁，以便下文讲到相关概念时能有个印象。

（1）synchronized

java中的关键字，内部实现为监视器锁，主要是通过对象监视器在对象头中的字段来表明的。

synchronized从旧版本到现在已经做了很多优化了，在运行时会有三种存在方式：偏向锁，轻量级锁，重量级锁。

偏向锁，是指一段同步代码一直被一个线程访问，那么这个线程会自动获取锁，降低获取锁的代价。

轻量级锁，是指当锁是偏向锁时，被另一个线程所访问，偏向锁会升级为轻量级锁，这个线程会通过自旋的方式尝试获取锁，不会阻塞，提高性能。

重量级锁，是指当锁是轻量级锁时，当自旋的线程自旋了一定的次数后，还没有获取到锁，就会进入阻塞状态，该锁升级为重量级锁，重量级锁会使其他线程阻塞，性能降低。

（2）CAS

CAS，Compare And Swap，它是一种乐观锁，认为对于同一个数据的并发操作不一定会发生修改，在更新数据的时候，尝试去更新数据，如果失败就不断尝试。

（3）volatile（非锁）

java中的关键字，当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。（这里牵涉到java内存模型的知识，感兴趣的同学可以自己查查相关资料）

volatile只保证可见性，不保证原子性，比如 volatile修改的变量 i，针对i++操作，不保证每次结果都正确，因为i++操作是两步操作，相当于 i = i +1，先读取，再加1，这种情况volatile是无法保证的。

（4）自旋锁

自旋锁，是指尝试获取锁的线程不会阻塞，而是循环的方式不断尝试，这样的好处是减少线程的上下文切换带来的开锁，提高性能，缺点是循环会消耗CPU。

（5）分段锁

分段锁，是一种锁的设计思路，它细化了锁的粒度，主要运用在ConcurrentHashMap中，实现高效的并发操作，当操作不需要更新整个数组时，就只锁数组中的一项就可以了。

（5）ReentrantLock

可重入锁，是指一个线程获取锁之后再尝试获取锁时会自动获取锁，可重入锁的优点是避免死锁。

其实，synchronized也是可重入锁。

源码分析

构造方法

```
public ConcurrentHashMap() {
}

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;
}
```

```
public ConcurrentHashMap(Map<? extends K, ? extends V> m) {
    this.sizeCtl = DEFAULT_CAPACITY;
    putAll(m);
}

public ConcurrentHashMap(int initialCapacity, float loadFactor) {
    this(initialCapacity, loadFactor, 1);
}

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;
}
```

构造方法与HashMap对比可以发现，没有了HashMap中的threshold和loadFactor，而是改用了sizeCtl来控制，而且只存储了容量在里面，那么它是怎么用的呢？官方给出的解释如下：

（1）-1，表示有线程正在进行初始化操作

（2）-(1 + nThreads)，表示有n个线程正在一起扩容

（3）0，默认值，后续在真正初始化的时候使用默认容量

（4）> 0，初始化或扩容完成后下一次的扩容门槛

至于，官方这个解释对不对我们后面再讨论。

添加元素

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

final V putVal(K key, V value, boolean onlyIfAbsent) {
    // key和value都不能为null
    if (key == null || value == null) throw new NullPointerException();
    // 计算hash值
    int hash = spread(key.hashCode());
    // 要插入的元素所在桶的元素个数
    int binCount = 0;
    // 死循环，结合CAS使用（如果CAS失败，则会重新取整个桶进行下面的流程）
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            // 如果桶未初始化或者桶个数为0，则初始化桶
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 如果要插入的元素所在的桶还没有元素，则把这个元素插入到这个桶中
```

```
            if (casTabAt(tab, i, null,
                    new Node<K,V>(hash, key, value, null)))
                // 如果使用CAS插入元素时，发现已经有元素了，则进入下一次循环，重新操作
                // 如果使用CAS插入元素成功，则break跳出循环，流程结束
                break;                   // no lock when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED)
            // 如果要插入的元素所在的桶的第一个元素的hash是MOVED，则当前线程帮忙一起迁移元素
            tab = helpTransfer(tab, f);
        else {
            // 如果这个桶不为空且不在迁移元素，则锁住这个桶（分段锁）
            // 并查找要插入的元素是否在这个桶中
            // 存在，则替换值（onlyIfAbsent=false）
            // 不存在，则插入到链表结尾或插入树中
            V oldVal = null;
            synchronized (f) {
                // 再次检测第一个元素是否有变化，如果有变化则进入下一次循环，从头来过
                if (tabAt(tab, i) == f) {
                    // 如果第一个元素的hash值大于等于0（说明不是在迁移，也不是树）
                    // 那就是桶中的元素使用的是链表方式存储
                    if (fh >= 0) {
                        // 桶中元素个数赋值为1
                        binCount = 1;
                        // 遍历整个桶，每次结束binCount加1
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            if (e.hash == hash &&
                                    ((ek = e.key) == key ||
                                            (ek != null && key.equals(ek)))) {
                                // 如果找到了这个元素，则赋值了新值（onlyIfAbsent=false）
                                // 并退出循环
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                // 如果到链表尾部还没有找到元素
                                // 就把它插入到链表结尾并退出循环
                                pred.next = new Node<K,V>(hash, key,
                                        value, null);
                                break;
                            }
                        }
                    }
                    else if (f instanceof TreeBin) {
                        // 如果第一个元素是树节点
                        Node<K,V> p;
                        // 桶中元素个数赋值为2
                        binCount = 2;
                        // 调用红黑树的插入方法插入元素
                        // 如果成功插入则返回null
                        // 否则返回寻找到的节点
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
```

```
                                value)) != null) {
                            // 如果找到了这个元素，则赋值了新值（onlyIfAbsent=false）
                            // 并退出循环
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                }
            }
            // 如果binCount不为0，说明成功插入了元素或者寻找到了元素
            if (binCount != 0) {
                // 如果链表元素个数达到了8，则尝试树化
                // 因为上面把元素插入到树中时，binCount只赋值了2，并没有计算整个树中元素的个数
                // 所以不会重复树化
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                // 如果要插入的元素已经存在，则返回旧值
                if (oldVal != null)
                    return oldVal;
                // 退出外层大循环，流程结束
                break;
            }
        }
        }
        // 成功插入元素，元素个数加1（是否要扩容在这个里面）
        addCount(1L, binCount);
        // 成功插入元素返回null
        return null;
    }
```

整体流程跟HashMap比较类似，大致是以下几步：

（1）如果桶数组未初始化，则初始化；

（2）如果待插入的元素所在的桶为空，则尝试把此元素直接插入到桶的第一个位置；

（3）如果正在扩容，则当前线程一起加入到扩容的过程中；

（4）如果待插入的元素所在的桶不为空且不在迁移元素，则锁住这个桶（分段锁）；

（5）如果当前桶中元素以链表方式存储，则在链表中寻找该元素或者插入元素；

（6）如果当前桶中元素以红黑树方式存储，则在红黑树中寻找该元素或者插入元素；

（7）如果元素存在，则返回旧值；

（8）如果元素不存在，整个Map的元素个数加1，并检查是否需要扩容；

添加元素操作中使用的锁主要有（自旋锁 + CAS + synchronized + 分段锁）。

为什么使用synchronized而不是ReentrantLock?

因为synchronized已经得到了极大地优化，在特定情况下并不比ReentrantLock差。

未完待续~~

现在文章没办法留言了，大家如果有任何建议或者意见，欢迎大家在公众号后台给我留言，谢谢~

欢迎关注我的公众号“彤哥读源码”，查看更多源码系列文章，与彤哥一起畅游源码的海洋。

收录于话题 #JDK源码·71个

上一篇
死磕 java集合之ConcurrentHashMap源码分析（二）

下一篇
死磕 java集合之TreeMap源码分析（一）- 内含红黑树分析全过程

阅读原文

死磕 java集合之ConcurrentHashMap源码分析（二）

原创 唐彤 彤哥读源码 2019-04-10

收录于话题
#JDK源码

71个

欢迎关注我的公众号“彤哥读源码”，查看更多源码系列文章，与彤哥一起畅游源码的海洋。

本章接着上一章，链接直达 死磕 java集合之ConcurrentHashMap源码分析（一）

初始化桶数组

第一次放元素时，初始化桶数组。

```
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sizeCtl<0说明正在初始化或者扩容，让出CPU
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            // 如果把sizeCtl原子更新为-1成功，则当前线程进入初始化
            // 如果原子更新失败则说明有其它线程先一步进入初始化了，则进入下一次循环
            // 如果下一次循环时还没初始化完毕，则sizeCtl<0进入上面if的逻辑让出CPU
            // 如果下一次循环更新完毕了，则table.length!=0，退出循环
            try {
                // 再次检查table是否为空，防止ABA问题
                if ((tab = table) == null || tab.length == 0) {
                    // 如果sc为0则使用默认值16
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    // 新建数组
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    // 赋值给table桶数组
                    table = tab = nt;
                    // 设置sc为数组长度的0.75倍
                    // n - (n >>> 2) = n - n/4 = 0.75n
                    // 可见这里装载因子和扩容门槛都是写死了的
                    // 这也正是没有threshold和loadFactor属性的原因
                    sc = n - (n >>> 2);
                }
            } finally {
                // 把sc赋值给sizeCtl，这时存储的是扩容门槛
                sizeCtl = sc;
            }
            break;
        }
```

```
    }
    return tab;
}
```

（1）使用CAS锁控制只有一个线程初始化桶数组；

（2）sizeCtl在初始化后存储的是扩容门槛；

（3）扩容门槛写死的是桶数组大小的0.75倍，桶数组大小即map的容量，也就是最多存储多少个元素。

判断是否需要扩容

每次添加元素后，元素数量加1，并判断是否达到扩容门槛，达到了则进行扩容或协助扩容。

```
private final void addCount(long x, int check) {
    CounterCell[] as; long b, s;
    // 这里使用的思想跟LongAdder类是一模一样的（后面会讲）
    // 把数组的大小存储根据不同的线程存储到不同的段上（也是分段锁的思想）
    // 并且有一个baseCount，优先更新baseCount，如果失败了再更新不同线程对应的段
    // 这样可以保证尽量小的减少冲突

    // 先尝试把数量加到baseCount上，如果失败再加到分段的CounterCell上
    if ((as = counterCells) != null ||
            !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
        // 如果as为空
        // 或者长度为0
        // 或者当前线程所在的段为null
        // 或者在当前线程的段上加数量失败
        if (as == null || (m = as.length - 1) < 0 ||
                (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
                !(uncontended =
                        U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            // 强制增加数量（无论如何数量是一定要加上的，并不是简单地自旋）
            // 不同线程对应不同的段都更新失败了
            // 说明已经发生冲突了，那么就对counterCells进行扩容
            // 以减少多个线程hash到同一个段的概率
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        // 计算元素个数
        s = sumCount();
    }
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        // 如果元素个数达到了扩容门槛，则进行扩容
        // 注意，正常情况下sizeCtl存储的是扩容门槛，即容量的0.75倍
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
                (n = tab.length) < MAXIMUM_CAPACITY) {
            // rs是扩容时的一个邮戳标识
            int rs = resizeStamp(n);
```

```
            if (sc < 0) {
                // sc<0说明正在扩容中
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                        sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                        transferIndex <= 0)
                    // 扩容已经完成了，退出循环
                    // 正常应该只会触发nextTable==null这个条件，其它条件没看出来何时触发
                    break;

                // 扩容未完成，则当前线程加入迁移元素中
                // 并把扩容线程数加1
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                    (rs << RESIZE_STAMP_SHIFT) + 2))
                // 这里是触发扩容的那个线程进入的地方
                // sizeCtl的高16位存储着rs这个扩容邮戳
                // sizeCtl的低16位存储着扩容线程数加1，即(1+nThreads)
                // 所以官方说的扩容时sizeCtl的值为 -(1+nThreads)是错误的

                // 进入迁移元素
                transfer(tab, null);
            // 重新计算元素个数
            s = sumCount();
        }
    }
}
```

（1）元素个数的存储方式类似于LongAdder类，存储在不同的段上，减少不同线程同时更新size时的冲突；

（2）计算元素个数时把这些段的值及baseCount相加算出总的元素个数；

（3）正常情况下sizeCtl存储着扩容门槛，扩容门槛为容量的0.75倍；

（4）扩容时sizeCtl高位存储扩容邮戳(resizeStamp)，低位存储扩容线程数加1（1+nThreads）；

（5）其它线程添加元素后如果发现存在扩容，也会加入的扩容行列中来；

协助扩容（迁移元素）

线程添加元素时发现正在扩容且当前元素所在的桶元素已经迁移完成了，则协助迁移其它桶的元素。

```
final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
    Node<K,V>[] nextTab; int sc;
    // 如果桶数组不为空，并且当前桶第一个元素为ForwardingNode类型，并且nextTab不为空
    // 说明当前桶已经迁移完毕了，才去帮忙迁移其它桶的元素
    // 扩容时会把旧桶的第一个元素置为ForwardingNode，并让其nextTab指向新桶数组
    if (tab != null && (f instanceof ForwardingNode) &&
            (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
        int rs = resizeStamp(tab.length);
        // sizeCtl<0，说明正在扩容
        while (nextTab == nextTable && table == tab &&
```

```
                (sc = sizeCtl) < 0) {
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || transferIndex <= 0)
                break;
            // 扩容线程数加1
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                // 当前线程帮忙迁移元素
                transfer(tab, nextTab);
                break;
            }
        }
        return nextTab;
    }
    return table;
}
```

当前桶元素迁移完成了才去协助迁移其它桶元素；

迁移元素

扩容时容量变为两倍，并把部分元素迁移到其它桶中。

```
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
        // 如果nextTab为空，说明还没开始迁移
        // 就新建一个新桶数组
        try {
            // 新桶数组是原桶的两倍
            @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;
    }
    // 新桶数组大小
    int nextn = nextTab.length;
    // 新建一个ForwardingNode类型的节点，并把新桶数组存储在里面
    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;
        // 整个while循环就是在算i的值，过程太复杂，不用太关心
        // i的值会从n-1依次递减，感兴趣的可以打下断点就知道了
        // 其中n是旧桶数组的大小，也就是说i从15开始一直减到1这样去迁移元素
        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;
            }
        }
        if (i < 0 || i >= n || i + n >= nextn) {
            // 如果一次遍历完成了
            // 也就是整个map所有桶中的元素都迁移完成了
            int sc;
            if (finishing) {
                // 如果全部迁移完成了，则替换旧桶数组
                // 并设置下一次扩容门槛为新桶数组容量的0.75倍
                nextTable = null;
                table = nextTab;
                sizeCtl = (n << 1) - (n >>> 1);
                return;
            }
            if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                // 当前线程扩容完成，把扩容线程数-1
                if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                    // 扩容完成两边肯定相等
                    return;
                // 把finishing设置为true
                // finishing为true才会走到上面的if条件
                finishing = advance = true;
                // i重新赋值为n
                // 这样会再重新遍历一次桶数组，看看是不是都迁移完成了
                // 也就是第二次遍历都会走到下面的(fh = f.hash) == MOVED这个条件
                i = n; // recheck before commit
            }
        }
        else if ((f = tabAt(tab, i)) == null)
            // 如果桶中无数据，直接放入ForwardingNode标记该桶已迁移
            advance = casTabAt(tab, i, null, fwd);
        else if ((fh = f.hash) == MOVED)
            // 如果桶中第一个元素的hash值为MOVED
            // 说明它是ForwardingNode节点
            // 也就是该桶已迁移
            advance = true; // already processed
        else {
            // 锁定该桶并迁移元素
            synchronized (f) {
                // 再次判断当前桶第一个元素是否有修改
                // 也就是可能其它线程先一步迁移了元素
                if (tabAt(tab, i) == f) {
```

```
// 把一个链表分化成两个链表
// 规则是桶中各元素的hash与桶大小n进行与操作
// 等于0的放到低位链表(low)中，不等于0的放到高位链表(high)中
// 其中低位链表迁移到新桶中的位置相对旧桶不变
// 高位链表迁移到新桶中位置正好是其在旧桶的位置加n
// 这也正是为什么扩容时容量在变成两倍的原因
Node<K,V> ln, hn;
if (fh >= 0) {
    // 第一个元素的hash值大于等于0
    // 说明该桶中元素是以链表形式存储的
    // 这里与HashMap迁移算法基本类似
    // 唯一不同的是多了一步寻找lastRun
    // 这里的lastRun是提取出链表后面不用处理再特殊处理的子链表
    // 比如所有元素的hash值与桶大小n与操作后的值分别为 0 0 4 4 0 0 0
    // 则最后后面三个0对应的元素肯定还是在同一个桶中
    // 这时lastRun对应的就是倒数第三个节点
    // 至于为啥要这样处理，我也没太搞明白
    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;
        }
    }
    // 看看最后这几个元素归属于低位链表还是高位链表
    if (runBit == 0) {
        ln = lastRun;
        hn = null;
    }
    else {
        hn = lastRun;
        ln = null;
    }
    // 遍历链表，把hash&n为0的放在低位链表中
    // 不为0的放在高位链表中
    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);
    }
    // 低位链表的位置不变
    setTabAt(nextTab, i, ln);
    // 高位链表的位置是原位置加n
    setTabAt(nextTab, i + n, hn);
    // 标记当前桶已迁移
    setTabAt(tab, i, fwd);
    // advance为true，返回上面进行--i操作
    advance = true;
}
else if (f instanceof TreeBin) {
    // 如果第一个元素是树节点
```

```
                        // 也是一样，分化成两颗树
                        // 也是根据hash&n为0放在低位树中
                        // 不为0放在高位树中
                        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;
                        // 遍历整颗树，根据hash&n是否为0分化成两颗树
                        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;
                            }
                        }
                        // 如果分化的树中元素个数小于等于6，则退化成链表
                        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;
                        // 低位树的位置不变
                        setTabAt(nextTab, i, ln);
                        // 高位树的位置是原位置加n
                        setTabAt(nextTab, i + n, hn);
                        // 标记该桶已迁移
                        setTabAt(tab, i, fwd);
                        // advance为true，返回上面进行--i操作
                        advance = true;
                    }
                }
            }
        }
    }
}
```

（1）新桶数组大小是旧桶数组的两倍；

（2）迁移元素先从靠后的桶开始；

（3）迁移完成的桶在里面放置一ForwardingNode类型的元素，标记该桶迁移完成；

（4）迁移时根据hash&n是否等于0把桶中元素分化成两个链表或树；

（5）低位链表（树）存储在原来的位置；

（6）高们链表（树）存储在原来的位置加n的位置；

（7）迁移元素时会锁住当前桶，也是分段锁的思想；

未完待续~~

现在文章没办法留言了，如果有任何建议或者意见，欢迎大家在公众号后台给我留言，谢谢~

欢迎关注我的公众号“彤哥读源码”，查看更多源码系列文章，与彤哥一起畅游源码的海洋。

死磕 java集合之ConcurrentHashMap源码分析（三）

原创 唐彤 彤哥读源码 2019-04-11

收录于话题
#JDK源码

71个

欢迎关注我的公众号“彤哥读源码”，查看更多源码系列文章，与彤哥一起畅游源码的海洋。

本章接着上两章，链接直达：

死磕 java集合之ConcurrentHashMap源码分析（一）

死磕 java集合之ConcurrentHashMap源码分析（二）

注：源码部分阅读不方便的，可以把手机横屏阅读，很清晰。

删除元素

删除元素跟添加元素一样，都是先找到元素所在的桶，然后采用分段锁的思想锁住整个桶，再进行操作。

```
public V remove(Object key) {
    // 调用替换节点方法
    return replaceNode(key, null, null);
}

final V replaceNode(Object key, V value, Object cv) {
    // 计算hash
    int hash = spread(key.hashCode());
    // 自旋
    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)
            // 如果目标key所在的桶不存在，跳出循环返回null
            break;
        else if ((fh = f.hash) == MOVED)
            // 如果正在扩容中，协助扩容
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            // 标记是否处理过
            boolean validated = false;
            synchronized (f) {
                // 再次验证当前桶第一个元素是否被修改过
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        // fh>=0表示是链表节点
```

```
                validated = true;
                // 遍历链表寻找目标节点
                for (Node<K,V> e = f, pred = null;;) {
                    K ek;
                    if (e.hash == hash &&
                            ((ek = e.key) == key ||
                                    (ek != null && key.equals(ek)))) {
                        // 找到了目标节点
                        V ev = e.val;
                        // 检查目标节点旧value是否等于cv
                        if (cv == null || cv == ev ||
                                (ev != null && cv.equals(ev))) {
                            oldVal = ev;
                            if (value != null)
                                // 如果value不为空则替换旧值
                                e.val = value;
                            else if (pred != null)
                                // 如果前置节点不为空
                                // 删除当前节点
                                pred.next = e.next;
                            else
                                // 如果前置节点为空
                                // 说明是桶中第一个元素，删除之
                                setTabAt(tab, i, e.next);
                        }
                        break;
                    }
                    pred = e;
                    // 遍历到链表尾部还没找到元素，跳出循环
                    if ((e = e.next) == null)
                        break;
                }
            }
            else if (f instanceof TreeBin) {
                // 如果是树节点
                validated = true;
                TreeBin<K,V> t = (TreeBin<K,V>)f;
                TreeNode<K,V> r, p;
                // 遍历树找到了目标节点
                if ((r = t.root) != null &&
                        (p = r.findTreeNode(hash, key, null)) != null) {
                    V pv = p.val;
                    // 检查目标节点旧value是否等于cv
                    if (cv == null || cv == pv ||
                            (pv != null && cv.equals(pv))) {
                        oldVal = pv;
                        if (value != null)
                            // 如果value不为空则替换旧值
                            p.val = value;
                        else if (t.removeTreeNode(p))
                            // 如果value为空则删除元素
                            // 如果删除后树的元素个数较少则退化成链表
                            // t.removeTreeNode(p)这个方法返回true表示删除节点后树的元
                            setTabAt(tab, i, untreeify(t.first));
                    }
```

```
                        }
                    }
                }
            }
            // 如果处理过，不管有没有找到元素都返回
            if (validated) {
                // 如果找到了元素，返回其旧值
                if (oldVal != null) {
                    // 如果要替换的值为空，元素个数减1
                    if (value == null)
                        addCount(-1L, -1);
                    return oldVal;
                }
                break;
            }
        }
    }
    // 没找到元素返回空
    return null;
}
```

（1）计算hash；

（2）如果所在的桶不存在，表示没有找到目标元素，返回；

（3）如果正在扩容，则协助扩容完成后再进行删除操作；

（4）如果是以链表形式存储的，则遍历整个链表查找元素，找到之后再删除；

（5）如果是以树形式存储的，则遍历树查找元素，找到之后再删除；

（6）如果是以树形式存储的，删除元素之后树较小，则退化成链表；

（7）如果确实删除了元素，则整个map元素个数减1，并返回旧值；

（8）如果没有删除元素，则返回null；

获取元素

获取元素，根据目标key所在桶的第一个元素的不同采用不同的方式获取元素，关键点在于find()方法的重写。

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    // 计算hash
    int h = spread(key.hashCode());
    // 如果元素所在的桶存在且里面有元素
    if ((tab = table) != null && (n = tab.length) > 0 &&
            (e = tabAt(tab, (n - 1) & h)) != null) {
        // 如果第一个元素就是要找的元素，直接返回
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
```

```
        }
        else if (eh < 0)
            // hash小于0，说明是树或者正在扩容
            // 使用find寻找元素，find的寻找方式依据Node的不同子类有不同的实现方式
            return (p = e.find(h, key)) != null ? p.val : null;

        // 遍历整个链表寻找元素
        while ((e = e.next) != null) {
            if (e.hash == h &&
                    ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

（1）hash到元素所在的桶；

（2）如果桶中第一个元素就是该找的元素，直接返回；

（3）如果是树或者正在迁移元素，则调用各自Node子类的find()方法寻找元素；

（4）如果是链表，遍历整个链表寻找元素；

（5）获取元素没有加锁；

获取元素个数

元素个数的存储也是采用分段的思想，获取元素个数时需要把所有段加起来。

```
public int size() {
    // 调用sumCount()计算元素个数
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
                    (int)n);
}

final long sumCount() {
    // 计算CounterCell所有段及baseCount的数量之和
    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;
}
```

（1）元素的个数依据不同的线程存在在不同的段里；（见addCounter()分析）

（2）计算CounterCell所有段及baseCount的数量之和；

（3）获取元素个数没有加锁；

总结

（1）ConcurrentHashMap是HashMap的线程安全版本；

（2）ConcurrentHashMap采用（数组 + 链表 + 红黑树）的结构存储元素；

（3）ConcurrentHashMap相比于同样线程安全的HashTable，效率要高很多；

（4）ConcurrentHashMap采用的锁有 synchronized，CAS，自旋锁，分段锁，volatile等；

（5）ConcurrentHashMap中没有threshold和loadFactor这两个字段，而是采用sizeCtl来控制；

（6）sizeCtl = -1，表示正在进行初始化；

（7）sizeCtl = 0，默认值，表示后续在真正初始化的时候使用默认容量；

（8）sizeCtl > 0，在初始化之前存储的是传入的容量，在初始化或扩容后存储的是下一次的扩容门槛；

（9）sizeCtl = (resizeStamp << 16) + (1 + nThreads)，表示正在进行扩容，高位存储扩容邮戳，低位存储扩容线程数加1；

（10）更新操作时如果正在进行扩容，当前线程协助扩容；

（11）更新操作会采用synchronized锁住当前桶的第一个元素，这是分段锁的思想；

（12）整个扩容过程都是通过CAS控制sizeCtl这个字段来进行的，这很关键；

（13）迁移完元素的桶会放置一个ForwardingNode节点，以标识该桶迁移完毕；

（14）元素个数的存储也是采用的分段思想，类似于LongAdder的实现；

（15）元素个数的更新会把不同的线程hash到不同的段上，减少资源争用；

（16）元素个数的更新如果还是出现多个线程同时更新一个段，则会扩容段（CounterCell）；

（17）获取元素个数是把所有的段（包括baseCount和CounterCell）相加起来得到的；

（18）查询操作是不会加锁的，所以ConcurrentHashMap不是强一致性的；

（19）ConcurrentHashMap中不能存储key或value为null的元素；

彩蛋——值得学习的技术

ConcurrentHashMap中有哪些值得学习的技术呢？

我认为有以下几点：

（1）CAS + 自旋，乐观锁的思想，减少线程上下文切换的时间；

（2）分段锁的思想，减少同一把锁争用带来的低效问题；

（3）CounterCell，分段存储元素个数，减少多线程同时更新一个字段带来的低效；

（4）@sun.misc.Contended（CounterCell上的注解），避免伪共享；（p.s.伪共享我们后面也会讲的^^）

（5）多线程协同进行扩容；

（6）你还学到了哪些呢？

彩蛋——不能解决的问题

ConcurrentHashMap不能解决什么问题呢？

请看下面的例子：

```
private static final Map<Integer, Integer> map = new ConcurrentHashMap<>();

public void unsafeUpdate(Integer key, Integer value) {
    Integer oldValue = map.get(key);
    if (oldValue == null) {
        map.put(key, value);
    }
}
```

这里如果有多个线程同时调用unsafeUpdate()这个方法，ConcurrentHashMap还能保证线程安全吗？

答案是不能。因为get()之后if之前可能有其它线程已经put()了这个元素，这时候再put()就把那个线程put()的元素覆盖了。

那怎么修改呢？

答案也很简单，使用putIfAbsent()方法，它会保证元素不存在时才插入元素，如下：

```
public void safeUpdate(Integer key, Integer value) {
    map.putIfAbsent(key, value);
}
```

那么，如果上面oldValue不是跟null比较，而是跟一个特定的值比如1进行比较怎么办？也就是下面这样：

```
public void unsafeUpdate(Integer key, Integer value) {
    Integer oldValue = map.get(key);
    if (oldValue == 1) {
        map.put(key, value);
    }
}
```

这样的话就没办法使用putIfAbsent()方法了。

其实，ConcurrentHashMap还提供了另一个方法叫replace(K key, V oldValue, V newValue)可以解决这个问题。

replace(K key, V oldValue, V newValue)这个方法可不能乱用，如果传入的newValue是null，则会删除元素。

```
public void safeUpdate(Integer key, Integer value) {
    map.replace(key, 1, value);
}
```

那么，如果if之后不是简单的put()操作，而是还有其它业务操作，之后才是put()，比如下面这样，这该怎么办呢？

```
public void unsafeUpdate(Integer key, Integer value) {
    Integer oldValue = map.get(key);
    if (oldValue == 1) {
        System.out.println(System.currentTimeMillis());
        /**
         * 其它业务操作
         */
        System.out.println(System.currentTimeMillis());

        map.put(key, value);
    }
}
```

这时候就没办法使用ConcurrentHashMap提供的方法了，只能业务自己来保证线程安全了，比如下面这样：

```
public void safeUpdate(Integer key, Integer value) {
    synchronized (map) {
        Integer oldValue = map.get(key);
        if (oldValue == null) {
            System.out.println(System.currentTimeMillis());
            /**
             * 其它业务操作
             */
            System.out.println(System.currentTimeMillis());

            map.put(key, value);
        }
    }
}
```

这样虽然不太友好，但是最起码能保证业务逻辑是正确的。

当然，这里使用ConcurrentHashMap的意义也就不大了，可以换成普通的HashMap了。

上面只是举一个简单的例子，我们不能听说ConcurrentHashMap是线程安全的，就认为它无论什么情况下都是线程安全的，还是那句话尽信书不如无书。

这也正是我们读源码的目的之一，了解其本质，才能在我们的实际工作中少挖坑，不论是挖给别人还是挖给自己^^。

好了，整个ConcurrentHashMap就讲完了。

文章暂不支持留言功能，如果您有任何建议或意见请在公众号后台给我留言，留言必回复。