1.介绍

本章介绍使用的最频繁的并发集合类之一ConcurrentHashMap，之前介绍过HashMap和HashTable，指出了HashTable的问题。虽然可以使用Collections.synchronizedMap方法包装HashMap完成线程安全的Map，但是这样做是远远无法满足我们对性能的需求。因为Map使用频率十分高，键值对在程序中是十分常见的，每一次put或get操作锁住整个Map，无疑是十分糟糕的。值得一提的是ConcurrentHashMap在早起的版本中好像是采取了分段式锁的方法进行控制的，但是至少JDK7之后（JDK8没有使用segment的方式，只装了7,8）并没有采取这种方式，而是使用CAS操作与synchronized锁更快的方法实现的，Segment的存在意义也只是为了兼容之前的版本，但是实际还是一种分段锁的思路，锁的是hash桶。

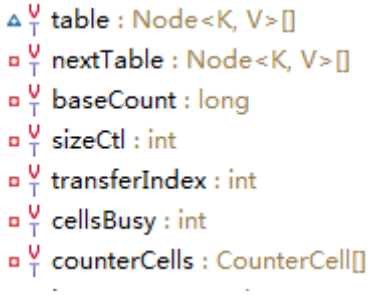
通常检索操作（包括get）都是非阻塞的，所以可能与更新操作重叠（put,remove)。检索结果受到最近的更新操作的影响，简单说就是一个键的更新操作对于任何检索操作而已表现成happens-before的关系，所以键会展示出最近更新的值。对于总的操作（putAll,clear）。检索可能受到部分键值对的插入或移除影响。同样的，迭代器受到在创建的时候某个时刻hash表的状态影响，并不会抛出并发异常，但是迭代器的设计只考虑了单线程的使用。

冲突严重的时候Hash表会自动扩容，扩容的遍历操作被视为是一个缓慢的操作，如果可能通过初始容量构造参数设置大小，loadFactor和HashMap一样默认是0.75。

注意ConcurrentHashMap是不允许键或值为null，HashMap可以。

2.ConcurrentHashMap

2.1 数据结构



ConcurrentHashMap有许多的参数，上图是部分变量参数。还有很多常量参数：

MAXIMUM_CAPACITY：最大容量230。

DEFAULT_CAPACITY：默认容量16

MAX_ARRAY_SIZE：最大数组大小（toArray等方法使用）Integer.MAX_VALUE - 8

DEFAULT_CONCURRENCY_LEVEL：默认并发级别16（无用，兼容前版本）

LOAD_FACTOR：载入因子0.75

TREEIFY_THRESHOLD：转变红黑树的阈值8

UNTREEIFY_THRESHOLD：普通链表的阈值6

MIN_TREEIFY_CAPACITY：最小的表容量，只有超过这个容量链表才可能转换成红黑树。至少要4倍的TREEIFY_THRESHOLD，默认64

MIN_TRANSFER_STRIDE：每次转移步骤最小的重建桶数量

其它的参数：

table：基本的hash表，大小是2n

nextTable：resize的时候使用

baseCount：基础的统计值，通常在无竞争的时候使用

sizeCtl：表初始化和resize的控制

transferIndex：resize时下一个表分割的索引+1

cellsBusy：resize或创建CounterCells的自旋锁

counterCells：counter cells表，非空时大小是2n

2.2 基本操作

获取一个元素，无锁：

```
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;
}
```

步骤很简单：

1.通过键的hashCode计算出该键在Hash表上散列的位置h

2.如果hash表不存在或者表为空或者hash表指向h的位置（当然是通过h进一步计算定位该位置）为空，意味着没有该键的值，返回null

3.步骤2不成立，就查找该位置：

第一个的元素的hash值就相同，再比较键的值是否相等，相等就返回该值，不等就继续最后的while循环查找hash值相等，键相等的，返回其值。这里键为null就会出现空指针异常了。

第一个元素的hash值小于0，也是通过Node类自身的方法遍历链表，和后面while步骤区别不大（不明白为什么要多写这步）

存入一个元素：

存入的步骤就相对比较复杂，会不断的循环尝试。步骤如下：

1.表不存在创建表：

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            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;
}
```

创建表的步骤也是一个死循环，通过sizeCtl来进行多线程初始化表控制，当有线程优先获取修改权时就会将这个值改成-1，其它线程就不能修改让出CPU执行权，继续循环尝试获取修改权（前提是还满足while循环的条件，表为null或容量为0）。获取修改权的表会使用默认的容量16创建hash表，sc会设置成当前容量的3/4，最后将其赋值给sizeCtl。

2.如果表存在就会查找该键所在表的位置，该位置还没有值意味着无冲突，首次设置，就通过CAS尝试设置，设置成功跳出循环，失败了也不要紧，继续循环。

3.如果指定位置第一个元素的hash是被标记成moved状态，进行transfer操作调整后再循环，transfer具体操作如下：

```
/**
 * Helps transfer if a resize is in progress.
 */
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) {
        int rs = resizeStamp(tab.length);
        while (nextTab == nextTable && table == tab &&
               (sc = sizeCtl) < 0) {
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                sc == rs + MAX_RESIZERS || transferIndex <= 0)
                break;
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                transfer(tab, nextTab);
                break;
            }
        }
        return nextTab;
    }
    return table;
}
```

看方法上的注释也能够明白moved状态意味着该位置的所有节点处于resize的移动状态，这里处理的就是扩容和put之间的冲突了。表不为null且节点是ForwardingNode且nextTable不为空，判断是在resize，后面的计算原理不太清楚，但还是能明白这段代码所起到的作用。先计算出rs，resize的时候nextTable,table和sizeCtl都是小于0的，才是未结束，进入循环在判断一下sc和rs的关系和transferIndex下标判断有没有处理完，处理完了就跳出循环，没处理完就会抢占处理，最终都是返回转移的nextTab，put方法就在resize阶段无缝衔接到新的表上了。这里逻辑很清楚，并发put的时候helpTransfer也可能并发调用，我们预期的应该是返回一个新表，但是新表的while循环是根据sizeCtl来控制的，意味着sizeCtl<0的时候移动已经完成，返回新表是没什么问题的，所以这个值是最后才会改变，其它的情况所有的线程都会获取新表，而更上一层的moved状态应该是更后面改变，这样就不会产生问题，所有新加的元素都进入更新后的表。而循环中满足跳出条件就意味着转换完成，这些变化发生的应该更早。最大的疑问在于抢夺transfer权限之后sc的值改变了，后续会直接返回nextTab,这个时间nextTable真的准备好了吗，也就是transfer的操作可能慢了，导致实际nextTab还没有准备好。否则sc+1还是小于0的，依旧进入了循环，但是多个线程可能同时操作tranfer方法，可能会造成处理冲突。所有的关键在于transfer方法的实现了。

transfer的代码过长，这里简述一下其步骤：

1.计算stride的值，单核就是表长度，多核是表长度/8/核数，最小16，这个字段的含义是一次处理多少个hash桶。

2.如果nextTab==null，创建nextTab，原长度的2倍

3.构建ForwardingNode节点，其hash就是模式moved，里面包含的就是新表

4.死循环开始转移节点到新表：

先循环找到下一个处理的hash表位置。过程是倒着处理的，先处理数组的后面部分，transferIndex指示下一次处理的下标开始位置。stride是一次处理多少个hash桶。跳过这个查找位置的方法就是所有节点都已经被分配处理了，通过CAS线程抢夺处理权，这里就巧妙的进行了多线程分段处理，各个线程互不干扰。只要其他线程要等所有处理完才结束，那么就不会产生线程安全问题。

如果i<0等条件成立，意味着当前线程没有可以处理的，判断sc的大小，如果符合结束的判断改变并设置finish，在下一个循环完成表的更新。这里可以看出结束判断都是交给sc的变化，这个对所有线程都是一样处理判断的，判断不通过会直接返回，任何线程进入resize的transfer环节都会使得sc+1,所有线程离开都会-1，问题是只有一个线程才会知道所有的都处理完成了，改变table。（**这导致了上述问题，put在resize环节，返回的table可能是没有准备好的newTable，但是并不要紧，因为转移的过程中锁住了表的hash桶第一个元素，put操作锁的是同一个，所以依旧要等转换完成**）

i是当前线程处理的hash表下标，如果为null，设置成ForwardingNode，即Moved模式

不为null，但是hash是moved状态，表明正在处理，需要重新选择处理的位置。

锁住hash表下标对象f，如果hash表下标f没有改变才意味着没有变化，能进行转移。转移的部分就不再进行描述，上面选择处理端，锁住处理的下标就足够保证线程安全了。顺便一提，其在处理的时候就会将第一个元素改成ForwardingNode，put的时候就知道这个正在处理，会进入helpTransfer。

4.锁住当前表下标第一个元素，进行插入操作。这里有个问题，虽然和resize环节一样锁的是表下标的第一个元素，但是会不会出现不一致的情况？transfer的f计算是通过旧表的得到的第一个，put环节第一回合确实是旧表，但是helpTransfer之后就会变成新表，再循环的时候不是出现了不一致了吗，此时新表还是未准备好的表。实际上不会，因为只有该下标处理完了，才会设置成moved状态，put环节才会切换成新表，否则就会进入锁的环节，此时如果正在转移这个区间的内容，就不会进入，没转移也不要紧，直接放入旧表就好了，之后处理的时候会一并处理。而synchronize在put等待transfer完成之后，做了再次判断第一个节点是否是一致了，就不用担心会插入旧表，因为新表完成后旧表该节点就是ForwardingNode节点，不会相等，下次循环就进入切换到新表了。

5.**计数，不进行描述，通过CounterCell。**

3.面试题

1. ConcurrentHashMap中变量使用final和volatile修饰有什么用呢?
Final域使得确保初始化安全性（initialization safety）成为可能，初始化安全性让不可变形对象不需要同步就能自由地被访问和共享。
使用volatile来保证某个变量内存的改变对其他线程即时可见，在配合CAS可以实现不加锁对并发操作的支持。get操作可以无锁是由于Node的元素val和指针next是用volatile修饰的，在多线程环境下线程A修改结点的val或者新增节点的时候是对线程B可见的。

2.我们可以使用CocurrentHashMap来代替Hashtable吗?
我们知道Hashtable是synchronized的，但是ConcurrentHashMap同步性能更好，因为它仅仅根据同步级别对map的一部分进行上锁。ConcurrentHashMap当然可以代替HashTable，但是HashTable提供更强的线程安全性。它们都可以用于多线程的环境，但是当Hashtable的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。因为ConcurrentHashMap引入了分割(segmentation)，不论它变得多么大，仅仅需要锁定map的某个部分，而其它的线程不需要等到迭代完成才能访问map。简而言之，在迭代的过程中，ConcurrentHashMap仅仅锁定map的某个部分，而Hashtable则会锁定整个map。

3. ConcurrentHashMap有什么缺陷吗?
ConcurrentHashMap 是设计为非阻塞的。在更新时会局部锁住某部分数据，但不会把整个表都锁住。同步读取操作则是完全非阻塞的。好处是在保证合理的同步前提下，效率很高。**坏处是严格来说读取操作不能保证反映最近的更新。**例如线程A调用putAll写入大量数据，期间线程B调用get，则只能get到目前为止已经顺利插入的部分数据。

4. ConcurrentHashMap在JDK 7和8之间的区别

- JDK1.8的实现降低锁的粒度，JDK1.7版本锁的粒度是基于Segment的，包含多个HashEntry，而JDK1.8锁的粒度就是HashEntry（首节点）
- JDK1.8版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用synchronized来进行同步，所以不需要分段锁的概念，也就不需要Segment这种数据结构了，由于粒度的降低，实现的复杂度也增加了
- JDK1.8使用红黑树来优化链表，基于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表，这样形成一个最佳拍档