wuzhaoyang的个人博客

java8集合框架(三) - Map的实现类（ConcurrentHashMap）

2016-09-05 | java |

java8集合框架(三) - Map的实现类（ConcurrentHashMap）

不管是 HashMap 还是 LinkedHashMap 都是非线程安全的。在多线程的情况下，会出现各种问题。接着分析一下它们的线程安全版本是如何实现的。

> Note:本篇源码分析基于jdk1.8版本

ConcurrentHashMap

HashMap 的线程安全版本，可以用来替换 HashTable 。在hash碰撞过多的情况下会将链表转化成红黑树。1.8版本的 ConcurrentHashMap 的实现与1.7版本有很大的差别，放弃了*段锁*的概念，借鉴了 HashMap 的数据结构：数组 + 链表 + 红黑树。分析下来，太精妙了！！！ ConcurrentHashMap 不接受 null key和 null value。

数据结构

ConcurrentHashMap 的数据结构较复杂，因为多线程控制和协作，引入了很多辅助类。

```
1  //很重要的一个字段。
2  //负数：table正在初始化或扩容， -1初始化， -(1 + 参与扩容的线程数)
3  //当table为null，保存要初始化table的size。
4  //初始化过后，保存下一次扩容的阈值
5  private transient volatile int sizeCtl;
6
7  //bucket，hash桶
```

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

//只会在扩容时有用的临时表
private transient volatile Node<K,V>[] nextTable;

//ConcurrentHashMap的元素个数 = baseCount + SUM(counterCells)!!
//元素基础个数，通过CAS更新，当CAS失败则将要加的值加到counterCells数组
private transient volatile long baseCount;

//下一个线程领扩容任务时，分配的hash桶起始索引
private transient volatile int transferIndex;

//用数组来处理当CAS失败时，元素统计提高效率的方案。参见java.util.concurrent.atomic.LongAdder
private transient volatile CounterCell[] counterCells;

//关注定义与HashMap节点差异的三个点
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    //1.关注这里的2个volatile，在多线程中保证内存可见性
    volatile V val;
    volatile Node<K,V> next;

    Node(int hash, K key, V val, Node<K,V> next) {...}
    public final K getKey()       { ... }
    public final V getValue()     {... }
    public final int hashCode()   { ...}
    public final String toString(){ ... }
    public final V setValue(V value) {
        //2.注意这里不支持直接setValue
        throw new UnsupportedOperationException();
    }
    public final boolean equals(Object o) {...}
    //3.注意这里的find方法
    Node<K,V> find(int h, Object k) {...}
}

//Forward类型的节点，表示桶的扩容处理完成
static final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
    Node<K,V> find(int h, Object k) {...}
}
```

关键操作 - 初始化

```
private final Node<K,V>[] initTable() {
```

```
    Node<K,V>[] tab; int sc;
    //未初始化
    while ((tab = table) == null || tab.length == 0) {
        //利用sizeCtl<0等待。保证了初始化由单线程完成
        if ((sc = sizeCtl) < 0)
                  //告诉线程调度，如果有人需要可以礼让别人先执行，没有的话我就继续执行。
            Thread.yield(); // lost initialization race; just spin
        //第一个线程进来，将sizeCtl设置为 - 1，代表将hash表的状态置为初始化中
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                        //第一个线程初始化好后，还在loop的其它线程进来执行，不用再初始化
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    //初始化Node数组
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    //这种0.75是不是很精妙？
                    sc = n - (n >>> 2);
                }
            } finally {

                sizeCtl = sc;
            }
            //死循环唯一出口
            break;
        }
    }
    return tab;
}
```

说明：初始化是由单个线程完成的。为了确保单线程，第一个线程CAS了SizeCtl = - 1。其他线程的CAS失败，而后SizeCtl<0，继续执行死循环，直到初始化完成。SizeCtl初始化为下次扩容阈值(大于0，容量的0.75倍)，其他线程CAS成功，进入初始化代码段，因初始化已完成，break结束循环，返回tab。

> 初始化代码段的break是循环唯一出口

关键操作 - put

```
/** Implementation for put and putIfAbsent */
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    //分散Hash
    int hash = spread(key.hashCode());
    int binCount = 0;
    //这里是一个死循环，可能的出口如下
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            //上面已经分析了初始化过程，初始化完成后继续执行死循环
```

```
11          //上面已经分析了初始化过程，初始化完成后继续执行死循环
12              tab = initTable();
13          //数组的第一个元素为空，则赋值
14          else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
15          //这里使用了CAS，避免使用锁。如果CAS失败，说明该节点已经发生改变，
16          //可能被其他线程插入了，那么继续执行死循环，在链尾插入。
17              if (casTabAt(tab, i, null,
18                           new Node<K,V>(hash, key, value, null)))
19                  //可能的出口一
20                  break;                   // no lock when adding to empty bin
21          }
22          //如果tab正在resize，则帮忙一起执行resize
23          //这里监测到的的条件是目标桶被设置成了FORWORD。如果桶没有设置为
24          //FORWORD节点，即使正在扩容，该线程也无感知。
25          else if ((fh = f.hash) == MOVED)
26              tab = helpTransfer(tab, f);
27          //执行put操作
28          else {
29              V oldVal = null;
30              //这里请求了synchronized锁。这里要注意，不会出现
31              //桶正在resize的过程中执行插入，因为桶resize的时候
32              //也请求了synchronized锁。即如果该桶正在resize，这里会发生锁等待
33              synchronized (f) {
34                          //如果是链表的首个节点
35                  if (tabAt(tab, i) == f) {
36                                  //并且是一个用户节点，非Forwarding等节点
37                      if (fh >= 0) {
38                          binCount = 1;
39                          for (Node<K,V> e = f;; ++binCount) {
40                              K ek;
41                              //找到相等的元素更新其value
42                              if (e.hash == hash &&
43                                  ((ek = e.key) == key ||
44                                   (ek != null && key.equals(ek)))) {
45                                  oldVal = e.val;
46                                  if (!onlyIfAbsent)
47                                      e.val = value;
48                                  //可能的出口二
49                                  break;
50                              }
51                              //否则添加到链表尾部
52                              Node<K,V> pred = e;
53                              if ((e = e.next) == null) {
54                                  pred.next = new Node<K,V>(hash, key,
55                                                            value, null);
56                                  //可能的出口三
57                                  break;
58                              }
59                          }
60                      }
61                      else if (f instanceof TreeBin) {
62                          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) {
            //如果链表长度（碰撞次数）超过8，将链表转化为红黑树
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    //见下面的分析
    addCount(1L, binCount);
    return null;
}
```

说明：同样是一个死循环，循环的出口包括一下几个

1. Hash桶上的元素为空，CAS更新
2. Hash桶上链表有相同的key，更新
3. Hash桶上链表无相同key，插入链尾

其中2，3在synchronized中完成，无需CAS。

执行过程：

- tab为空则初始化
- hash桶上元素为空，则CAS更新
- 如果桶上的元素是 Forward 类型，帮助扩容
- 锁住桶上元素（即链表头节点），更新 / 插入节点
- 检测hash碰撞次数，判断是否需要将链表转化为红黑树
- 如果是插入节点，元素个数 + 1，判断是否需要扩容

关键操作 - 计数

```
//如果数组太小并且没有扩容，那么启动扩容。如果正在扩容，帮忙一起扩容。
//每次扩容后检查占用率是否需要进行再一次扩容，因为扩容滞后于添加元素。
```

```
3   private final void addCount(long x, int check) {
4       CounterCell[] as; long b, s;
5       //baseCount更新失败，则使用counterCells
6       if ((as = counterCells) != null ||
7           !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
8           CounterCell a; long v; int m;
9           //baseCount更新失败,CAS更新CounterCell数组的元素值+x，uncontended表示更新CounterCel
10          boolean uncontended = true;
11          if (as == null || (m = as.length - 1) < 0 ||
12              (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
13              !(uncontended =
14                U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
15              //如果这也失败，说明这个数组元素也被争用了。必须要动粗了
16              //fullAddCount实现思想同LongAdder，这个类也是1.8加入的。
17              //作用就是将x加到counterCells数组中或baseCount中
18              fullAddCount(x, uncontended);
19              return;
20          }
21          if (check <= 1)
22              return;
23          //统计元素的个数
24          s = sumCount();
25      }
26      if (check >= 0) {
27          Node<K,V>[] tab, nt; int n, sc;
28          //元素个数>扩容阈值，并且tab不为空
29          while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
30                 (n = tab.length) < MAXIMUM_CAPACITY) {
31              int rs = resizeStamp(n);
32              //如果正在扩容
33              if (sc < 0) {
34                                  //本轮扩容结束或没有桶可分配，线程离开
35                  if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
36                      sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
37                      transferIndex <= 0)
38                      break;
39                  // sc + 1表示扩容线程+1
40                  if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
41                      transfer(tab, nt);
42              }
43              //第一个监测到要扩容的线程进来，设置 (rs << RESIZE_STAMP_SHIFT) + 2
44              //表示现在只有一个线程在扩容，也就是当前进来的线程
45              else if (U.compareAndSwapInt(this, SIZECTL, sc,
46                                           (rs << RESIZE_STAMP_SHIFT) + 2))
47                  transfer(tab, null);
48              //统计个数，继续循环检测
49              s = sumCount();
50          }
51      }
52  }
```

说明：计数的逻辑参考了 java.util.concurrent.atomic.LongAdder 。它的作者也是**Doug Lea**。核心思想是在并发较低时，只需更新base值。在高并发的情况下，将对单一值的更新转化为数组上元素的更新，以降低并发争用。总的映射个数为base + CounterCell各个元素的和。如果总数大于阈值，扩容。

关键操作 - 扩容

```
final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
    Node<K,V>[] nextTab; int sc;
    //nextTab为空时，则说明扩容已经完成
    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;
}

//复制元素到nextTab
transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    int n = tab.length, stride;
    //NCPU为CPU核心数，每个核心均分复制任务，如果均分小于16个
    //那么以16为步长分给处理器：例如0-15号给处理器1，16-32号分给处理器2。处理器3就不用接任务了
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE; // subdivide range
     //如果nextTab为空则初始化为原tab的两倍，这里只会时单线程进得来，因为这初始化了nextTab，
     //addcount里面判断了nextTab为空则不执行扩容任务
    if (nextTab == null) {            // initiating
        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;
    //构造一个forword节点
```

```
45          ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
46          boolean advance = true;
47          boolean finishing = false; // to ensure sweep before committing nextTab
48          for (int i = 0, bound = 0;;) {
49              Node<K,V> f; int fh;
50              while (advance) {
51                  int nextIndex, nextBound;
52                  if (--i >= bound || finishing)
53                      advance = false;
54                  else if ((nextIndex = transferIndex) <= 0) {
55                      i = -1;
56                      advance = false;
57                  }
58                  else if (U.compareAndSwapInt
59                           (this, TRANSFERINDEX, nextIndex,
60                            nextBound = (nextIndex > stride ?
61                                         nextIndex - stride : 0))) {
62                      bound = nextBound;
63                      i = nextIndex - 1;
64                      advance = false;
65                  }
66              }
67              if (i < 0 || i >= n || i + n >= nextn) {
68                  int sc;
69                  if (finishing) {
70                      nextTable = null;
71                      table = nextTab;
72                      // sizeCtl = nextTab.length*0.75=2*tab.length*0.75=tab.length*1.5!!!
73                      sizeCtl = (n << 1) - (n >>> 1);
74                      return;
75                  }
76                  //sc - 1表示当前线程完成了扩容任务，sizeCtl的线程数要 - 1
77                  if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
78                              //还有线程在扩容,就不能设置finish为true
79                      if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
80                          return;
81                      finishing = advance = true;
82                      i = n; // recheck before commit
83                  }
84              }
85              else if ((f = tabAt(tab, i)) == null)
86                  advance = casTabAt(tab, i, null, fwd);
87              else if ((fh = f.hash) == MOVED)
88                  advance = true; // already processed
89              else {
90              //这保证了不会出现该桶正在resize又执行put操作的情况
91                  synchronized (f) {
92                      if (tabAt(tab, i) == f) {
93                          Node<K,V> ln, hn;
94                          if (fh >= 0) {
95                              int runBit = fh & n;
96                              Node<K,V> lastRun = f;
```

```
97                      for (Node<K,V> p = f.next; p != null; p = p.next) {
98                          int b = p.hash & n;
99                          //这里尽量少的复制链表节点，从lastrun到链尾的这段链表段，无需复制
100                         if (b != runBit) {
101                             runBit = b;
102                             lastRun = p;
103                         }
104                     }
105                     if (runBit == 0) {
106                         ln = lastRun;
107                         hn = null;
108                     }
109                     else {
110                         hn = lastRun;
111                         ln = null;
112                     }
113                     //其他节点执行复制
114                     for (Node<K,V> p = f; p != lastRun; p = p.next) {
115                         int ph = p.hash; K pk = p.key; V pv = p.val;
116                         if ((ph & n) == 0)
117                             ln = new Node<K,V>(ph, pk, pv, ln);
118                         else
119                             hn = new Node<K,V>(ph, pk, pv, hn);
120                     }
121                     setTabAt(nextTab, i, ln);
122                     setTabAt(nextTab, i + n, hn);
123                     setTabAt(tab, i, fwd);
124                     advance = true;
125                 }
126                 else if (f instanceof TreeBin) {
127                     TreeBin<K,V> t = (TreeBin<K,V>)f;
128                     TreeNode<K,V> lo = null, loTail = null;
129                     TreeNode<K,V> hi = null, hiTail = null;
130                     int lc = 0, hc = 0;
131                     for (Node<K,V> e = t.first; e != null; e = e.next) {
132                         int h = e.hash;
133                         TreeNode<K,V> p = new TreeNode<K,V>
134                             (h, e.key, e.val, null, null);
135                         if ((h & n) == 0) {
136                             if ((p.prev = loTail) == null)
137                                 lo = p;
138                             else
139                                 loTail.next = p;
140                             loTail = p;
141                             ++lc;
142                         }
143                         else {
144                             if ((p.prev = hiTail) == null)
145                                 hi = p;
146                             else
147                                 hiTail.next = p;
148                             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;
                    }
                }
            }
        }
    }
```

说明：扩容的操作相交1.7版本有了很大的性能提升。不仅表现在CAS无锁算法的应用，而且支持多线程处理扩容过程中元素复制。

扩容的过程：

- 确定步长，多线程复制过程中防止出现混乱。每个线程分配步长长度的hash桶长度。最低不少于16。
- 初始化nexttab。保证单线程执行，nexttab只存在于resize阶段，可以看作是临时表。
- 构造Forword节点，以标志扩容完成的Hash桶。
- 执行死循环
    - 分配线程处理hash桶的bound
    - 从n - 1到bound，倒序遍历hash桶
    - 如果桶节点为空，CAS为Forword节点，表明处理完成
    - 如果桶节点为Forword，则跳过
    - 锁定桶节点，执行复制操作。在复制到nexttab的过程中，未破坏原tab的链表顺序和结构，所以不影响原tab的检索。
    - 复制完成，设置桶节点为Forword
    - 所有线程完成任务，则扩容结束，nexttab赋值给tab，nexttab置为空，sizeCtl置为原tab长度的1.5倍（见注释）

如何保证nextTab的初始化由单线程执行？
所有调用 `transfer` 的方法（例如 `helperTransfer` 、 `addCount`)几乎都预先判断了 `nextTab!=null` ,而nextTab只会在 `transfer` 方法中初始化，保证了第一个进来的线程初始化之后其他线程才能进入。

关键操作 - get

```
//不用担心get的过程中发生resize，get可能遇到两种情况
//1.桶未resize（无论是没达到阈值还是resize已经开始但是还未处理该桶），遍历链表
//2.在桶的链表遍历的过程中resize，上面的resize分析可以看出并未破坏原tab的桶的节点关系，遍历仍可
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;
}
```

说明：有了上面的基础， `get` 方法看起来就很简单了。

1. 在没有遇到forword节点时，遍历原tab。上面也说了，即使正在扩容也不影响没有处理或者正在处理的桶链表遍历，因为它没有破坏原tab的链表关系。
2. 遇到forword节点，遍历nextTab（通过调用forword节点的 `find` 方法）

着重理解位操作

在阅读源码的时候，对里面的几个位操作的理解花了一些时间。

```
static final int resizeStamp(int n) {
    return Integer.numberOfLeadingZeros(n) | (1 << (RESIZE_STAMP_BITS - 1));
}
```

这里通过位操作返回了一个标志。意义是**标志对长度为n的表扩容**。

> n=16
>
> Integer.numberOfLeadingZeros(n) = 28
>
> resizeStamp(16) = 0001 1100 | 1000 0000 0000 0000 = 1000 0000 0001 1100
>
> n=64
>
> Integer.numberOfLeadingZeros(n) = 26

> resizeStamp(64) = 0001 1010 | 1000 0000 0000 0000 = 1000 0000 0001 1010

```
1   U.compareAndSwapInt(this, SIZECTL, sc,(rs << RESIZE_STAMP_SHIFT) + 2)
```

第一个线程进入扩容操作的CAS，置换了sc。当sc<0,其高16位为对长度为n的表扩容的标志，低16位表示参与到扩容的线程个数+1。这里为什么要加2，因为1表示的是初始化，2表示一个线程在执行扩容，3表示2个线程在执行扩容，以此类推（见sizeCtl的说明）

> rs就是 `resizeStamp` 计算的结果。假设n=64
> rs << RESIZE-STAMP-SHIFT = 1000 0000 0001 1010 << 16 = 1000 0000 0001 1010 0000 0000 0000 0000
> (rs << RESIZE-STAMP-SHIFT) + 2 = 1000 0000 0001 1010 0000 0000 0000 0000 + 10 = 1000 0000 0001 1010 0000 0000 0000 0010

```
1   if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
2       if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
3           return;
```

这段出现在扩容操作中，根据上面对 `sizeCtl` 的描述， `sc-1` 代表该线程完成了扩容任务，激活的扩容线程数减少一个。重点理解 `(sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT`

> 假设n=64
> resizeStamp(n) << RESIZE_STAMP_SHIFT = 1000 0000 0001 1010 0000 0000 0000 0000
> 低16位为0，已经不是扩容状态了，没有线程在执行扩容操作了。如果 `sc - 2` 与之相等， `sc` 为1000 0000 0001 1010 0000 0000 0000 0010。也就是说当前线程已经是整个扩容操作的最后一个线程了，随着它的结束，整个扩容就都完成了。

总结

在分析 `ConcurrentHashMap` 的过程中，我被Doug Lea老爷子精妙的设计所折服。这个类很复杂，包含了很多其他的概念。例如 `LongAdder` 、 `ThreadLocalRandom` 、大量的CAS操作。理解起来着实费力，所以用了比较长的时间去吃透作者的意图。但是对自己的提升是巨大的，分析完了这个类，对于其他的无锁多线程实现类的理解就变得较为简单了。就像你练了九阳神功，在练其他武功就快的多了。

参考

- ConcurrentHashMap源码分析（JDK8版本）
- 探索jdk8之ConcurrentHashMap 的实现机制
- 从LongAdder 看更高效的无锁实现
- 使用ThreadLocalRandom产生并发随机数

- Java位操作全面总结

作者： wuzhaoyang(John)
出处： http://wuzhaoyang.me/
因为作者水平有限，无法保证每句话都是对的，但能保证不复制粘贴，每句话经过推敲。希望能表达自己对于技术的态度，做一名优秀的软件工程师。

Map # ConcurrentHashMap

‹ java8集合框架(二) - Map的实现类 (HashMap，LinkedHashMap)

java8集合框架(四) - Map的实现类 （CurrentSkipListMap） ›