简介

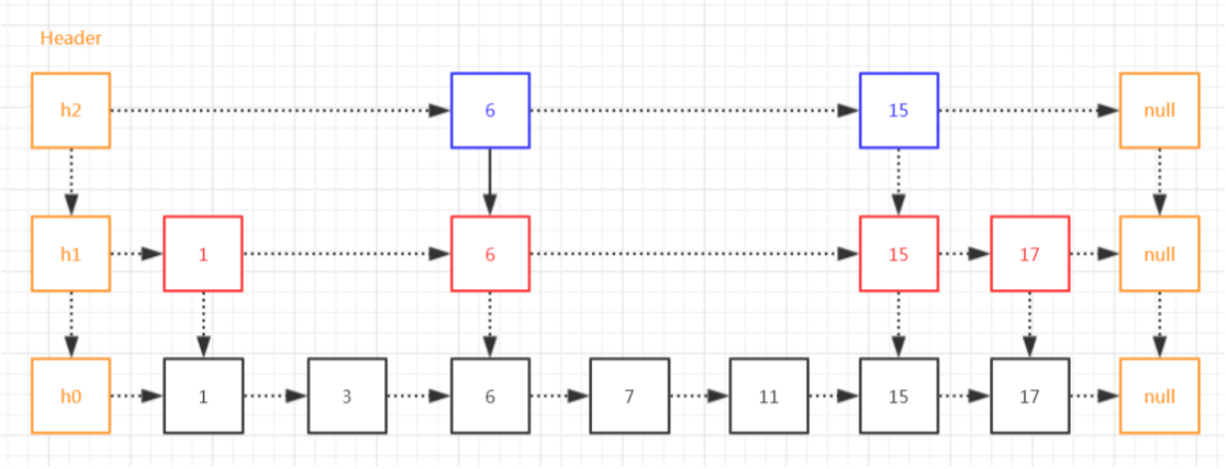
跳表是一个随机化的数据结构，实质就是一种可以进行**二分**查找的**有序链表**。

跳表在原有的有序链表上面增加了多级索引，通过索引来实现快速查找。

跳表不仅能提高搜索性能，同时也可以提高插入和删除操作的性能。

存储结构

跳表在原有的有序链表上面增加了多级索引，通过索引来实现快速查找。



源码分析

主要内部类

内部类跟存储结构结合着来看，大概能预测到代码的组织方式。

```
1.    // 数据节点，典型的单链表结构
2.    static final class Node<K,V> {
3.        final K key;
4.        // 注意：这里value的类型是Object，而不是V
5.        // 在删除元素的时候value会指向当前元素本身
6.        volatile Object value;
7.        volatile Node<K,V> next;
8.
9.        Node(K key, Object value, Node<K,V> next) {
10.           this.key = key;
11.           this.value = value;
12.           this.next = next;
13.       }
14.
15.       Node(Node<K,V> next) {
16.           this.key = null;
17.           this.value = this; // 当前元素本身(marker)
18.           this.next = next;
19.       }
20.   }
21.
```

```
    // 索引节点，存储着对应的node值，及向下和向右的索引指针
    static class Index<K,V> {
        final Node<K,V> node;
        final Index<K,V> down;
        volatile Index<K,V> right;

        Index(Node<K,V> node, Index<K,V> down, Index<K,V> right) {
            this.node = node;
            this.down = down;
            this.right = right;
        }
    }

    // 头索引节点，继承自Index，并扩展一个level字段，用于记录索引的层级
    static final class HeadIndex<K,V> extends Index<K,V> {
        final int level;

        HeadIndex(Node<K,V> node, Index<K,V> down, Index<K,V> right, int level) {
            super(node, down, right);
            this.level = level;
        }
    }

```

（1）Node，数据节点，存储数据的节点，典型的单链表结构；

（2）Index，索引节点，存储着对应的node值，及向下和向右的索引指针；

（3）HeadIndex，头索引节点，继承自Index，并扩展一个level字段，用于记录索引的层级；

构造方法

```

    public ConcurrentSkipListMap() {
        this.comparator = null;
        initialize();
    }

    public ConcurrentSkipListMap(Comparator<? super K> comparator) {
        this.comparator = comparator;
        initialize();
    }

    public ConcurrentSkipListMap(Map<? extends K, ? extends V> m) {
        this.comparator = null;
        initialize();
        putAll(m);
    }

    public ConcurrentSkipListMap(SortedMap<K, ? extends V> m) {
        this.comparator = m.comparator();
        initialize();
        buildFromSorted(m);
    }

```

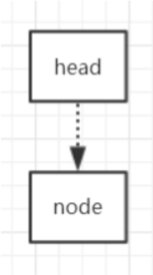
四个构造方法里面都调用了initialize()这个方法，那么，这个方法里面有什么呢?

```
    private static final Object BASE_HEADER = new Object();

    private void initialize() {
        keySet = null;
        entrySet = null;
```

```
        values = null;
        descendingMap = null;
        // Node(K key, Object value, Node<K,V> next)
        // HeadIndex(Node<K,V> node, Index<K,V> down, Index<K,V> right, int level)
        head = new HeadIndex<K,V>(new Node<K,V>(null, BASE_HEADER, null),
                                  null, null, 1);
    }

```

可以看到，这里初始化了一些属性，并创建了一个头索引节点，里面存储着一个数据节点，这个数据节点的值是空对象，且它的层级是1。

所以，初始化的时候，跳表中只有一个头索引节点，层级是1，数据节点是一个空对象，down和right都是null。



通过内部类的结构我们知道，一个头索引指针包含node, down, right三个指针，为了便于理解，我们把指向node的指针用虚线表示，其它两个用实线表示，也就是虚线不是表明方向的。

添加元素

通过【拜托，面试别再问我跳表了！】中的分析，我们知道跳表插入元素的时候会通过抛硬币的方式决定出它需要的层级，然后找到各层链中它所在的位置，最后通过单链表插入的方式把节点及索引插入进去来实现的。

那么，ConcurrentSkipList中是这么做的吗？让我们一起来探个究竟：

```
    public V put(K key, V value) {
        // 不能存储value为null的元素
        // 因为value为null标记该元素被删除（后面会看到）
        if (value == null)
            throw new NullPointerException();

        // 调用doPut()方法添加元素
        return doPut(key, value, false);
    }

    private V doPut(K key, V value, boolean onlyIfAbsent) {
        // 添加元素后存储在z中
        Node<K,V> z;             // added node
        // key也不能为null
        if (key == null)
            throw new NullPointerException();
        Comparator<? super K> cmp = comparator;

        // Part I：找到目标节点的位置并插入
        // 这里的目标节点是数据节点，也就是最底层的那条链
        // 自旋
        outer: for (;;) {
            // 寻找目标节点之前最近的一个索引对应的数据节点，存储在b中，b=before
            // 并把b的下一个数据节点存储在n中，n=next
            // 为了便于描述，我这里把b叫做当前节点，n叫做下一个节点
            for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {
                // 如果下一个节点不为空
                // 就拿其key与目标节点的key比较，找到目标节点应该插入的位置
                if (n != null) {
                    // v=value，存储节点value值
```

```
                    // c=compare，存储两个节点比较的大小
                    Object v; int c;
                    // n的下一个数据节点，也就是b的下一个节点的下一个节点（孙子节点）
                    Node<K,V> f = n.next;
                    // 如果n不为b的下一个节点
                    // 说明有其它线程修改了数据，则跳出内层循环
                    // 也就是回到了外层循环自旋的位置，从头来过
                    if (n != b.next)               // inconsistent read
                        break;
                    // 如果n的value值为空，说明该节点已删除，协助删除节点
                    if ((v = n.value) == null) {   // n is deleted
                        // todo 这里为啥会协助删除？后面讲
                        n.helpDelete(b, f);
                        break;
                    }
                    // 如果b的值为空或者v等于n，说明b已被删除
                    // 这时候n就是marker节点，那b就是被删除的那个
                    if (b.value == null || v == n) // b is deleted
                        break;
                    // 如果目标key与下一个节点的key大
                    // 说明目标元素所在的位置还在下一个节点的后面
                    if ((c = cpr(cmp, key, n.key)) > 0) {
                        // 就把当前节点往后移一位
                        // 同样的下一个节点也往后移一位
                        // 再重新检查新n是否为空，它与目标key的关系
                        b = n;
                        n = f;
                        continue;
                    }
                    // 如果比较时发现下一个节点的key与目标key相同
                    // 说明链表中本身就存在目标节点
                    if (c == 0) {
                        // 则用新值替换旧值，并返回旧值（onlyIfAbsent=false）
                        if (onlyIfAbsent || n.casValue(v, value)) {
                            @SuppressWarnings("unchecked") V vv = (V)v;
                            return vv;
                        }
                        // 如果替换旧值时失败，说明其它线程先一步修改了值，从头来过
                        break; // restart if lost race to replace value
                    }
                    // 如果c<0，就往下走，也就是找到了目标节点的位置
                    // else c < 0; fall through
                }

                // 有两种情况会到这里
                // 一是到链表尾部了，也就是n为null了
                // 二是找到了目标节点的位置，也就是上面的c<0

                // 新建目标节点，并赋值给z
                // 这里把n作为新节点的next
                // 如果到链表尾部了，n为null，这毫无疑问
                // 如果c<0，则n的key比目标key大，相妆于在b和n之间插入目标节点z
                z = new Node<K,V>(key, value, n);
                // 原子更新b的下一个节点为目标节点z
                if (!b.casNext(n, z))
                    // 如果更新失败，说明其它线程先一步修改了值，从头来过
                    break;          // restart if lost race to append to b
                // 如果更新成功，跳出自旋状态
                break outer;
            }
        }

        // 经过Part I，目标节点已经插入到有序链表中了

        // Part II：随机决定是否需要建立索引及其层次，如果需要则建立自上而下的索引
```

```
96.
97.         // 取个随机数
98.         int rnd = ThreadLocalRandom.nextSecondarySeed();
99.         // 0x80000001展开为二进制为10000000000000000000000000000001
100.        // 只有两头是1
101.        // 这里(rnd & 0x80000001) == 0
102.        // 相当于排除了负数（负数最高位是1），排除了奇数（奇数最低位是1）
103.        // 只有最高位最低位都不为1的数跟0x80000001做&操作才会为0
104.        // 也就是正偶数
105.        if ((rnd & 0x80000001) == 0) { // test highest and lowest bits
106.            // 默认level为1，也就是只要到这里了就会至少建立一层索引
107.            int level = 1, max;
108.            // 随机数从最低位的第二位开始，有几个连续的1则level就加几
109.            // 因为最低位肯定是0，正偶数嘛
110.            // 比如，1100110，level就加2
111.            while (((rnd >>>= 1) & 1) != 0)
112.                ++level;
113.
114.            // 用于记录目标节点建立的最高的那层索引节点
115.            Index<K,V> idx = null;
116.            // 取头索引节点（这是最高层的头索引节点）
117.            HeadIndex<K,V> h = head;
118.            // 如果生成的层数小于等于当前最高层的层级
119.            // 也就是跳表的高度不会超过现有高度
120.            if (level <= (max = h.level)) {
121.                // 从第一层开始建立一条竖直的索引链表
122.                // 这条链表使用down指针连接起来
123.                // 每个索引节点里面都存储着目标节点这个数据节点
124.                // 最后idx存储的是这条索引链表的最高层节点
125.                for (int i = 1; i <= level; ++i)
126.                    idx = new Index<K,V>(z, idx, null);
127.            }
```

```
128.            else { // try to grow by one level
129.                // 如果新的层数超过了现有跳表的高度
130.                // 则最多只增加一层
131.                // 比如现在只有一层索引，那下一次最多增加到两层索引，增加多了也没有意义
132.                level = max + 1; // hold in array and later pick the one to use
133.                // idxs用于存储目标节点建立的竖起索引的所有索引节点
134.                // 其实这里直接使用idx这个最高节点也是可以完成的
135.                // 只是用一个数组存储所有节点要方便一些
136.                // 注意，这里数组0号位是没有使用的
137.                @SuppressWarnings("unchecked")Index<K,V>[] idxs =
138.                        (Index<K,V>[])new Index<?,?>[level+1];
139.                // 从第一层开始建立一条竖的索引链表（跟上面一样，只是这里顺便把索引节点放到数组里面了）
140.                for (int i = 1; i <= level; ++i)
141.                    idxs[i] = idx = new Index<K,V>(z, idx, null);
142.
143.                // 自旋
144.                for (;;) {
145.                    // 旧的最高层头索引节点
146.                    h = head;
147.                    // 旧的最高层级
148.                    int oldLevel = h.level;
149.                    // 再次检查，如果旧的最高层级已经不比新层级矮了
150.                    // 说明有其它线程先一步修改了值，从头来过
151.                    if (level <= oldLevel) // lost race to add level
152.                        break;
153.                    // 新的最高层头索引节点
154.                    HeadIndex<K,V> newh = h;
155.                    // 头节点指向的数据节点
156.                    Node<K,V> oldbase = h.node;
157.                    // 超出的部分建立新的头索引节点
158.                    for (int j = oldLevel+1; j <= level; ++j)
159.                        newh = new HeadIndex<K,V>(oldbase, newh, idxs[j], j);
160.                    // 原子更新头索引节点
```

```
                    if (casHead(h, newh)) {
                        // h指向新的最高层头索引节点
                        h = newh;
                        // 把level赋值为旧的最高层级的
                        // idx指向的不是最高的索引节点了
                        // 而是与旧最高层平齐的索引节点
                        idx = idxs[level = oldLevel];
                        break;
                    }
                }
            }

            // 经过上面的步骤，有两种情况
            // 一是没有超出高度，新建一条目标节点的索引节点链
            // 二是超出了高度，新建一条目标节点的索引节点链，同时最高层头索引节点同样往上长

            // Part III：将新建的索引节点（包含头索引节点）与其它索引节点通过右指针连接在一起

            // 这时level是等于旧的最高层级的，自旋
            splice: for (int insertionLevel = level;;) {
                // h为最高头索引节点
                int j = h.level;

                // 从头索引节点开始遍历
                // 为了方便，这里叫q为当前节点，r为右节点，d为下节点，t为目标节点相应层级的索引
                for (Index<K,V> q = h, r = q.right, t = idx;;) {
                    // 如果遍历到了最右边，或者最下边，
                    // 也就是遍历到头了，则退出外层循环
                    if (q == null || t == null)
                        break splice;
                    // 如果右节点不为空
                    if (r != null) {
                        // n是右节点的数据节点，为了方便，这里直接叫右节点的值
                        Node<K,V> n = r.node;
                        // 比较目标key与右节点的值
                        int c = cpr(cmp, key, n.key);
                        // 如果右节点的值为空了，则表示此节点已删除
                        if (n.value == null) {
                            // 则把右节点删除
                            if (!q.unlink(r))
                                // 如果删除失败，说明有其它线程先一步修改了，从头来过
                                break;
                            // 删除成功后重新取右节点
                            r = q.right;
                            continue;
                        }
                        // 如果比较c>0，表示目标节点还要往右
                        if (c > 0) {
                            // 则把当前节点和右节点分别右移
                            q = r;
                            r = r.right;
                            continue;
                        }
                    }

                    // 到这里说明已经到当前层级的最右边了
                    // 这里实际是会先走第二个if

                    // 第一个if
                    // j与insertionLevel相等了
                    // 实际是先走的第二个if，j自减后应该与insertionLevel相等
                    if (j == insertionLevel) {
                        // 这里是真正连右指针的地方
                        if (!q.link(r, t))
                            // 连接失败，从头来过
```

```
                            break; // restart
                        // t节点的值为空，可能是其它线程删除了这个元素
                        if (t.node.value == null) {
                            // 这里会去协助删除元素
                            findNode(key);
                            break splice;
                        }
                        // 当前层级右指针连接完毕，向下移一层继续连接
                        // 如果移到了最下面一层，则说明都连接完成了，退出外层循环
                        if (--insertionLevel == 0)
                            break splice;
                    }

                    // 第二个if
                    // j先自减1，再与两个level比较
                    // j、insertionLevel和t(idx)三者是对应的，都是还未把右指针连好的那个层级
                    if (--j >= insertionLevel && j < level)
                        // t往下移
                        t = t.down;

                    // 当前层级到最右边了
                    // 那只能往下一层级去走了
                    // 当前节点下移
                    // 再取相应的右节点
                    q = q.down;
                    r = q.right;
                }
            }
        }
        return null;
    }
```

```
    // 寻找目标节点之前最近的一个索引对应的数据节点
    private Node<K,V> findPredecessor(Object key, Comparator<? super K> cmp) {
        // key不能为空
        if (key == null)
            throw new NullPointerException(); // don't postpone errors
        // 自旋
        for (;;) {
            // 从最高层头索引节点开始查找，先向右，再向下
            // 直到找到目标位置之前的那个索引
            for (Index<K,V> q = head, r = q.right, d;;) {
                // 如果右节点不为空
                if (r != null) {
                    // 右节点对应的数据节点，为了方便，我们叫右节点的值
                    Node<K,V> n = r.node;
                    K k = n.key;
                    // 如果右节点的value为空
                    // 说明其它线程把这个节点标记为删除了
                    // 则协助删除
                    if (n.value == null) {
                        if (!q.unlink(r))
                            // 如果删除失败
                            // 说明其它线程先删除了，从头来过
                            break;           // restart
                        // 删除之后重新读取右节点
                        r = q.right;         // reread r
                        continue;
                    }
                    // 如果目标key比右节点还大，继续向右寻找
                    if (cpr(cmp, key, k) > 0) {
                        // 往右移
                        q = r;
                        // 重新取右节点
```

```
                        r = r.right;
                        continue;
                    }
                    // 如果c<0，说明不能再往右了
                }
                // 到这里说明当前层级已经到最右了
                // 两种情况：一是r==null，二是c<0
                // 再从下一级开始找

                // 如果没有下一级了，就返回这个索引对应的数据节点
                if ((d = q.down) == null)
                    return q.node;

                // 往下移
                q = d;
                // 重新取右节点
                r = d.right;
            }
        }
    }

    // Node.class中的方法，协助删除元素
    void helpDelete(Node<K,V> b, Node<K,V> f) {
        /*
         * Rechecking links and then doing only one of the
         * help-out stages per call tends to minimize CAS
         * interference among helping threads.
         */
        // 这里的调用者this==n，三者关系是b->n->f
        if (f == next && this == b.next) {
            // 将n的值设置为null后，会先把n的下个节点设置为marker节点
            // 这个marker节点的值是它自己
            // 这里如果不是它自己说明marker失败了，重新marker
            if (f == null || f.value != f) // not already marked
                casNext(f, new Node<K,V>(f));
            else
                // marker过了，就把b的下个节点指向marker的下个节点
                b.casNext(this, f.next);
        }
    }

    // Index.class中的方法，删除succ节点
    final boolean unlink(Index<K,V> succ) {
        // 原子更新当前节点指向下一个节点的下一个节点
        // 也就是删除下一个节点
        return node.value != null && casRight(succ, succ.right);
    }

    // Index.class中的方法，在当前节点与succ之间插入newSucc节点
    final boolean link(Index<K,V> succ, Index<K,V> newSucc) {
        // 在当前节点与下一个节点中间插入一个节点
        Node<K,V> n = node;
        // 新节点指向当前节点的下一个节点
        newSucc.right = succ;
        // 原子更新当前节点的下一个节点指向新节点
        return n.value != null && casRight(succ, newSucc);
    }

```

我们这里把整个插入过程分成三个部分：

Part I：找到目标节点的位置并插入

（1）这里的目标节点是数据节点，也就是最底层的那条链；

（2）寻找目标节点之前最近的一个索引对应的数据节点（数据节点都是在最底层的链表上）；

（3）从这个数据节点开始往后遍历，直到找到目标节点应该插入的位置；

（4）如果这个位置有元素，就更新其值（onlyIfAbsent=false）；

（5）如果这个位置没有元素，就把目标节点插入；

（6）至此，目标节点已经插入到最底层的数据节点链表中了；

Part II：随机决定是否需要建立索引及其层次，如果需要则建立自上而下的索引

（1）取个随机数rnd，计算(rnd & 0x80000001)；

（2）如果不等于0，结束插入过程，也就是不需要创建索引，返回；

（3）如果等于0，才进入创建索引的过程（只要正偶数才会等于0）；

（4）计算 `while (((rnd >>>= 1) & 1) != 0)` ，决定层级数，level从1开始；

（5）如果算出来的层级不高于现有最高层级，则直接建立一条竖直的索引链表（只有down有值），并结束Part II；

（6）如果算出来的层级高于现有最高层级，则新的层级只能比现有最高层级多1；

（7）同样建立一条竖直的索引链表（只有down有值）；

（8）将头索引也向上增加到相应的高度，结束Part II；

（9）也就是说，如果层级不超过现有高度，只建立一条索引链，否则还要额外增加头索引链的高度（脑补一下，后面举例说明）；

Part III：将新建的索引节点（包含头索引节点）与其它索引节点通过右指针连接在一起（补上right指针）

（1）从最高层级的头索引节点开始，向右遍历，找到目标索引节点的位置；

（2）如果当前层有目标索引，则把目标索引插入到这个位置，并把目标索引前一个索引向下移一个层级；

（3）如果当前层没有目标索引，则把目标索引位置前一个索引向下移一个层级；

（4）同样地，再向右遍历，寻找新的层级中目标索引的位置，回到第（2）步；

（5）依次循环找到所有层级目标索引的位置并把它们插入到横向的索引链表中；

总结起来，一共就是三大步：

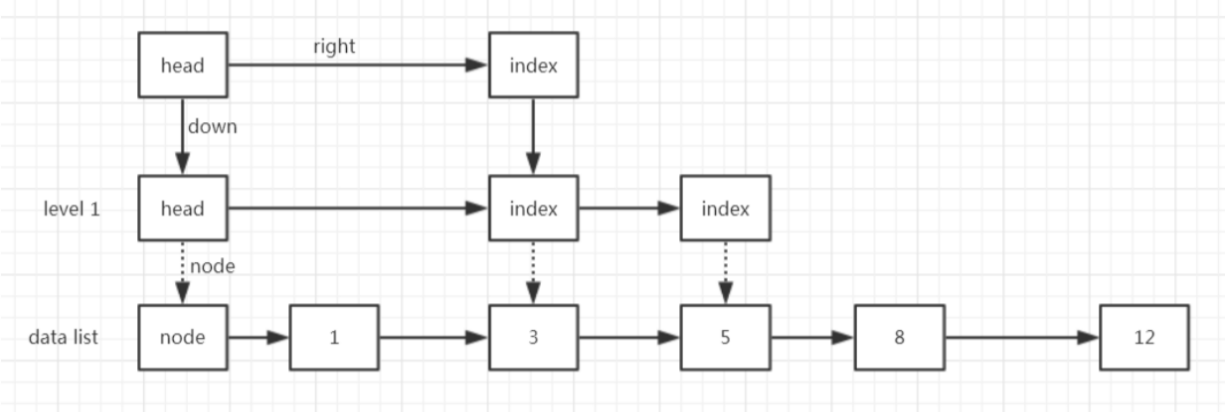
（1）插入目标节点到数据节点链表中；

（2）建立竖直的down链表；

（3）建立横向的right链表；

添加元素举例

假设初始链表是这样：

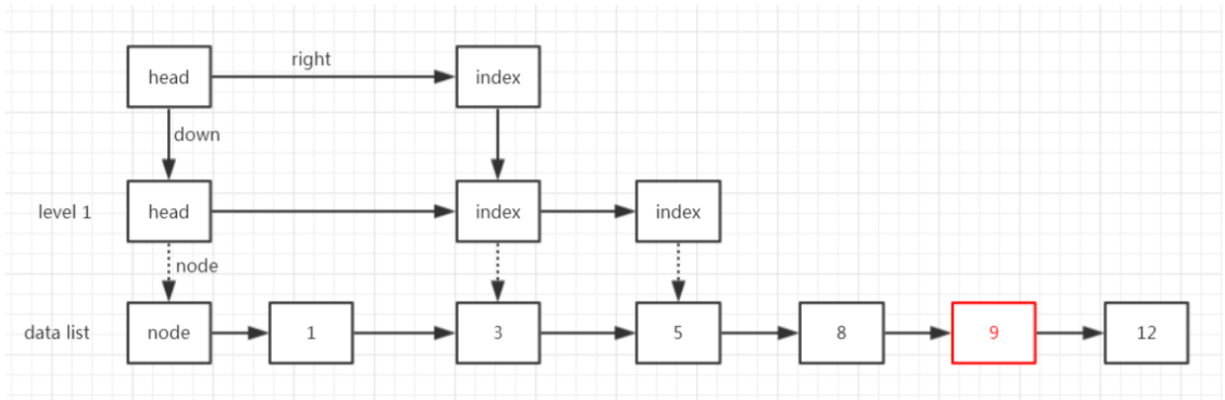


假如，我们现在要插入一个元素9。

（1）寻找目标节点之前最近的一个索引对应的数据节点，在这里也就是找到了5这个数据节点；

（2）从5开始向后遍历，找到目标节点的位置，也就是在8和12之间；

（3）插入9这个元素，Part I 结束；

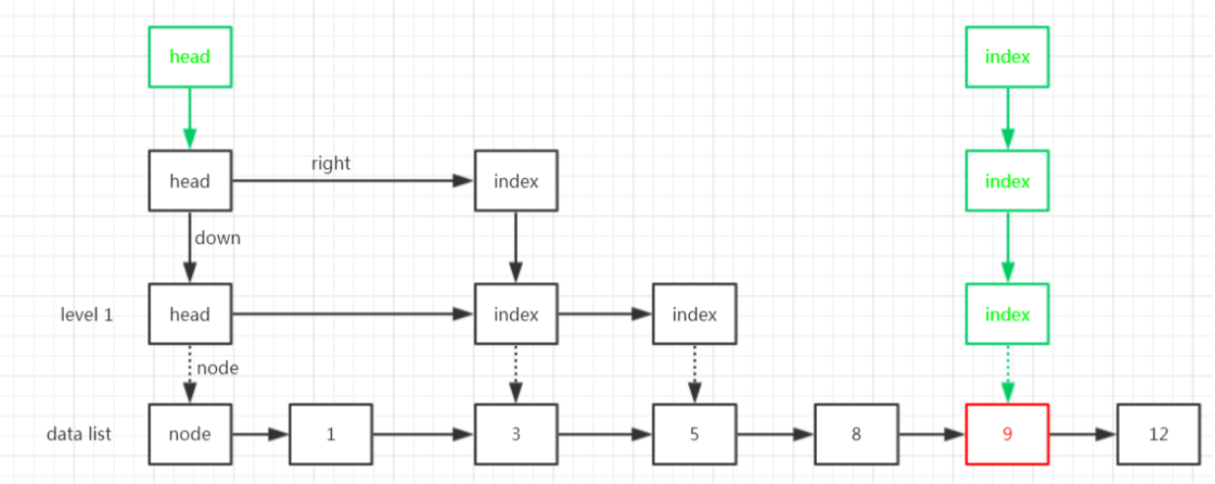


然后，计算其索引层级，假如是3，也就是level=3。

（1）建立竖直的down索引链表；

（2）超过了现有高度2，还要再增加head索引链的高度；

（3）至此，Part II 结束；



最后，把right指针补齐。

（1）从第3层的head往右找当前层级目标索引的位置；

（2）找到就把目标索引和它前面索引的right指针连上，这里前一个正好是head；

（3）然后前一个索引向下移，这里就是head下移；

（4）再往右找目标索引的位置；

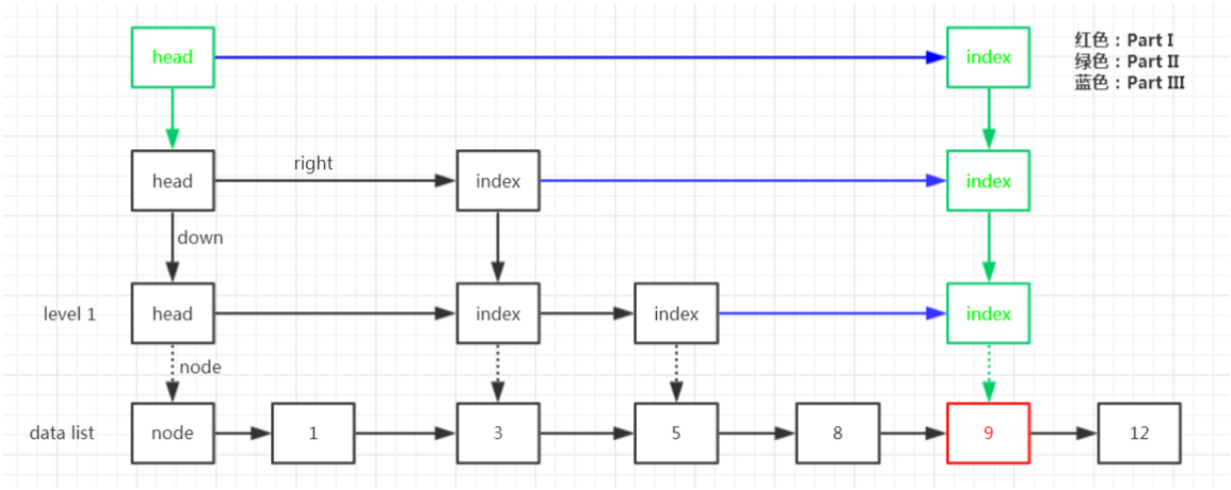
（5）找到了就把right指针连上，这里前一个是3的索引；

（6）然后3的索引下移；

（7）再往右找目标索引的位置；

（8）找到了就把right指针连上，这里前一个是5的索引；

（9）然后5下移，到底了，Part III 结束，整个插入过程结束；



是不是很简单^^

删除元素

删除元素，就是把各层级中对应的元素删除即可，真的这么简单吗？来让我们上代码：

```java
    public V remove(Object key) {
        return doRemove(key, null);
    }

    final V doRemove(Object key, Object value) {
        // key不为空
        if (key == null)
            throw new NullPointerException();
        Comparator<? super K> cmp = comparator;
        // 自旋
        outer: for (;;) {
            // 寻找目标节点之前的最近的索引节点对应的数据节点
            // 为了方便，这里叫b为当前节点，n为下一个节点，f为下下个节点
            for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {
                Object v; int c;
                // 整个链表都遍历完了也没找到目标节点，退出外层循环
                if (n == null)
                    break outer;
                // 下下个节点
                Node<K,V> f = n.next;
                // 再次检查
                // 如果n不是b的下一个节点了
                // 说明有其它线程先一步修改了，从头来过
                if (n != b.next)                    // inconsistent read
                    break;
                // 如果下个节点的值突为null了
                // 说明有其它线程标记该元素为删除状态了
                if ((v = n.value) == null) {        // n is deleted
                    // 协助删除
                    n.helpDelete(b, f);
                    break;
                }
                // 如果b的值为空或者v等于n，说明b已被删除
                // 这时候n就是marker节点，那b就是被删除的那个
                if (b.value == null || v == n)      // b is deleted
                    break;
                // 如果c<0，说明没找到元素，退出外层循环
                if ((c = cpr(cmp, key, n.key)) < 0)
                    break outer;
                // 如果c>0，说明还没找到，继续向右找
                if (c > 0) {
                    // 当前节点往后移
                    b = n;
                    // 下一个节点往后移
                    n = f;
                    continue;
                }
                // c=0，说明n就是要找的元素
                // 如果value不为空且不等于找到元素的value，不需要删除，退出外层循环
                if (value != null && !value.equals(v))
                    break outer;
                // 如果value为空，或者相等
                // 原子标记n的value值为空
                if (!n.casValue(v, null))
                    // 如果删除失败，说明其它线程先一步修改了，从头来过
                    break;

                // P.S.到了这里n的值肯定是设置成null了

                // 关键！！！！
                // 让n的下一个节点指向一个market节点
                // 这个market节点的key为null，value为marker自己，next为n的下个节点f
```

```
                // 或者让b的下一个节点指向下下个节点
                // 注意：这里是或者||，因为两个CAS不能保证都成功，只能一个一个去尝试
                // 这里有两层意思：
                // 一是如果标记market成功，再尝试将b的下个节点指向下下个节点，如果第二步失败了，进入条件，如果成功了就不用进入条件了
                // 二是如果标记market失败了，直接进入条件
                if (!n.appendMarker(f) || !b.casNext(n, f))
                    // 通过findNode()重试删除（里面有个helpDelete()方法）
                    findNode(key);                  // retry via findNode
                else {
                    // 上面两步操作都成功了，才会进入这里，不太好理解，上面两个条件都有非"!"操作
                    // 说明节点已经删除了，通过findPredecessor()方法删除索引节点
                    // findPredecessor()里面有unlink()操作
                    findPredecessor(key, cmp);      // clean index
                    // 如果最高层头索引节点没有右节点，则跳表的高度降级
                    if (head.right == null)
                        tryReduceLevel();
                }
                // 返回删除的元素值
                @SuppressWarnings("unchecked") V vv = (V)v;
                return vv;
            }
        }
        return null;
    }

```

(1) 寻找目标节点之前最近的一个索引对应的数据节点（数据节点都是在最底层的链表上）；

(2) 从这个数据节点开始往后遍历，直到找到目标节点的位置；

(3) 如果这个位置没有元素，直接返回null，表示没有要删除的元素；

(4) 如果这个位置有元素，先通过 `n.casValue(v, null)` 原子更新把其value设置为null；

(5) 通过 `n.appendMarker(f)` 在当前元素后面添加一个marker元素标记当前元素是要删除的元素；

(6) 通过 `b.casNext(n, f)` 尝试删除元素；

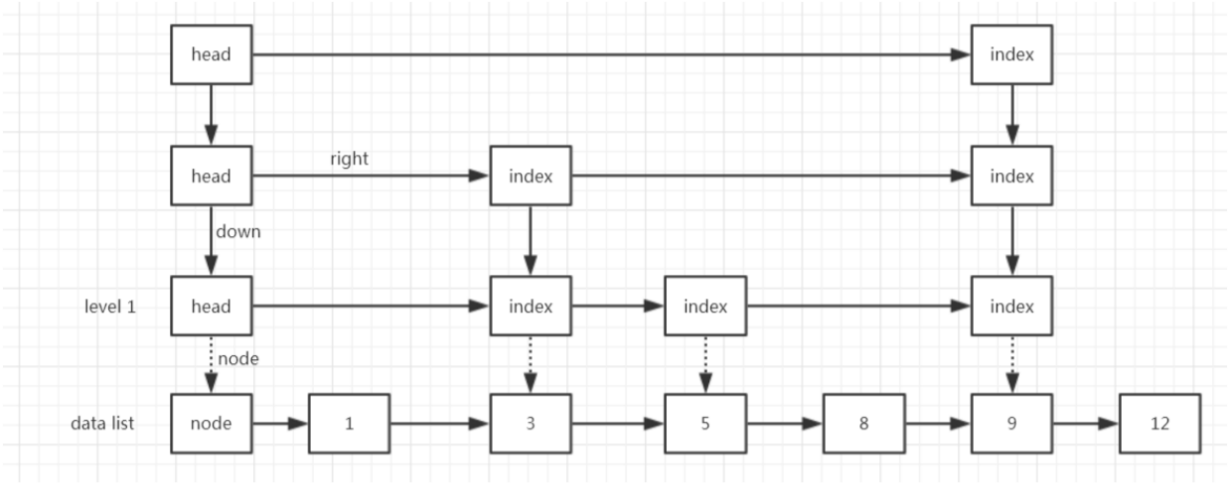
(7) 如果上面两步中的任意一步失败了都通过 `findNode(key)` 中的 `n.helpDelete(b, f)` 再去不断尝试删除；

(8) 如果上面两步都成功了，再通过 `findPredecessor(key, cmp)` 中的 `q.unlink(r)` 删除索引节点；

(9) 如果head的right指针指向了null，则跳表高度降级；

删除元素举例

假如初始跳表如下图所示，我们要删除9这个元素。



（1）找到9这个数据节点；

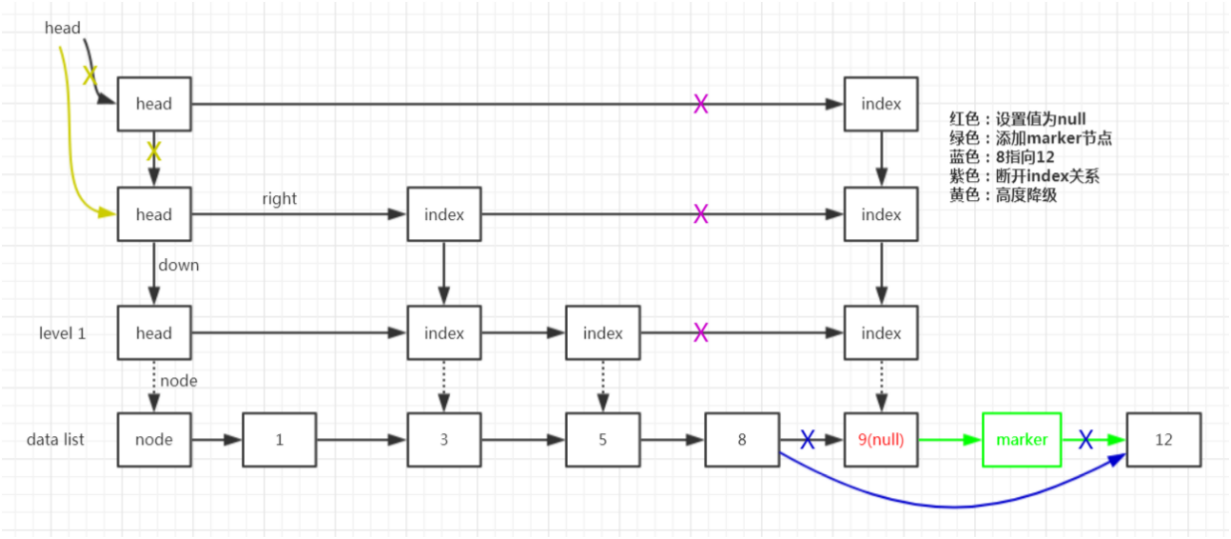
（2）把9这个节点的value值设置为null；

（3）在9后面添加一个marker节点，标记9已经删除了；

（4）让8指向12；

（5）把索引节点与它前一个索引的right断开联系；

（6）跳表高度降级；



至于，为什么要有（2）（3）（4）这么多步骤呢，因为多线程下如果直接让8指向12，可以其它线程先一步在9和12间插入了一个元素10呢，这时候就不对了。

所以这里搞了三步来保证多线程下操作的正确性。

如果第（2）步失败了，则直接重试；

如果第（3）或（4）步失败了，因为第（2）步是成功的，则通过helpDelete()不断重试去删除；

其实helpDelete()里面也是不断地重试（3）和（4）；

只有这三步都正确完成了，才能说明这个元素彻底被删除了。

这一块结合上面图中的红绿蓝色好好理解一下，一定要想在并发环境中会怎么样。

查找元素

经过上面的插入和删除，查找元素就比较简单了，直接上代码：

```
    public V get(Object key) {
        return doGet(key);
    }

    private V doGet(Object key) {
        // key不为空
        if (key == null)
            throw new NullPointerException();
        Comparator<? super K> cmp = comparator;
        // 自旋
        outer: for (;;) {
            // 寻找目标节点之前最近的索引对应的数据节点
            // 为了方便，这里叫b为当前节点，n为下个节点，f为下下个节点
            for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {
                Object v; int c;
                // 如果链表到头还没找到元素，则跳出外层循环
                if (n == null)
                    break outer;
                // 下下个节点
                Node<K,V> f = n.next;
                // 如果不一致读，从头来过
                if (n != b.next)                // inconsistent read
                    break;
                // 如果n的值为空，说明节点已被其它线程标记为删除
                if ((v = n.value) == null) {    // n is deleted
                    // 协助删除，再重试
                    n.helpDelete(b, f);
                    break;
                }
                // 如果b的值为空或者v等于n，说明b已被删除
                // 这时候n就是marker节点，那b就是被删除的那个
                if (b.value == null || v == n)  // b is deleted
                    break;
                // 如果c==0，说明找到了元素，就返回元素值
                if ((c = cpr(cmp, key, n.key)) == 0) {
                    @SuppressWarnings("unchecked") V vv = (V)v;
                    return vv;
                }
                // 如果c<0，说明没找到元素
                if (c < 0)
                    break outer;
                // 如果c>0，说明还没找到，继续寻找
                // 当前节点往后移
                b = n;
                // 下一个节点往后移
                n = f;
            }
        }
        return null;
    }

```

（1）寻找目标节点之前最近的一个索引对应的数据节点（数据节点都是在最底层的链表上）；

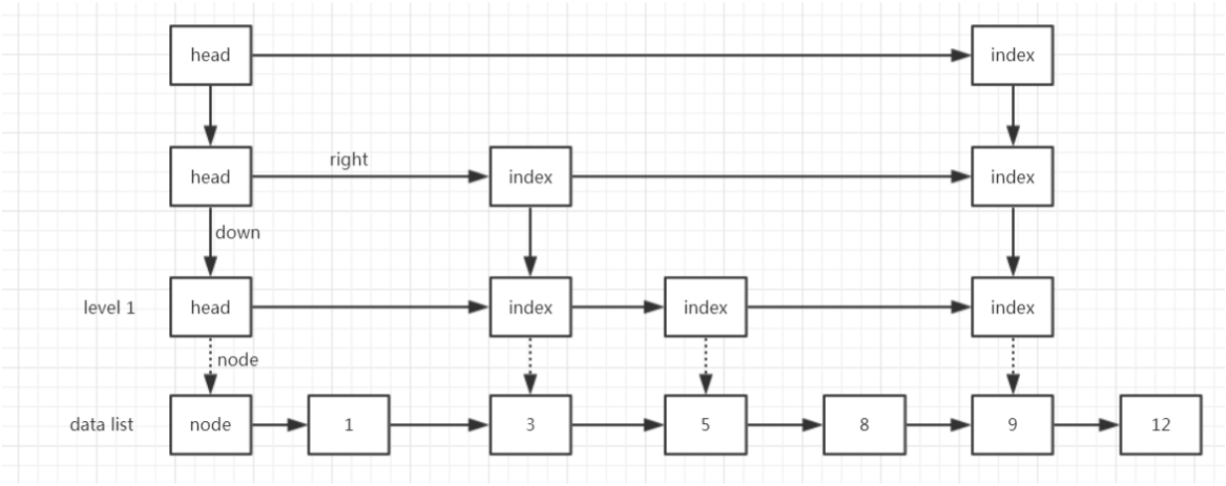
（2）从这个数据节点开始往后遍历，直到找到目标节点的位置；

（3）如果这个位置没有元素，直接返回null，表示没有找到元素；

（4）如果这个位置有元素，返回元素的value值；

查找元素举例

假如有如下图所示这个跳表，我们要查找9这个元素，它走过的路径是怎样的呢？可能跟你相像的不一样。。

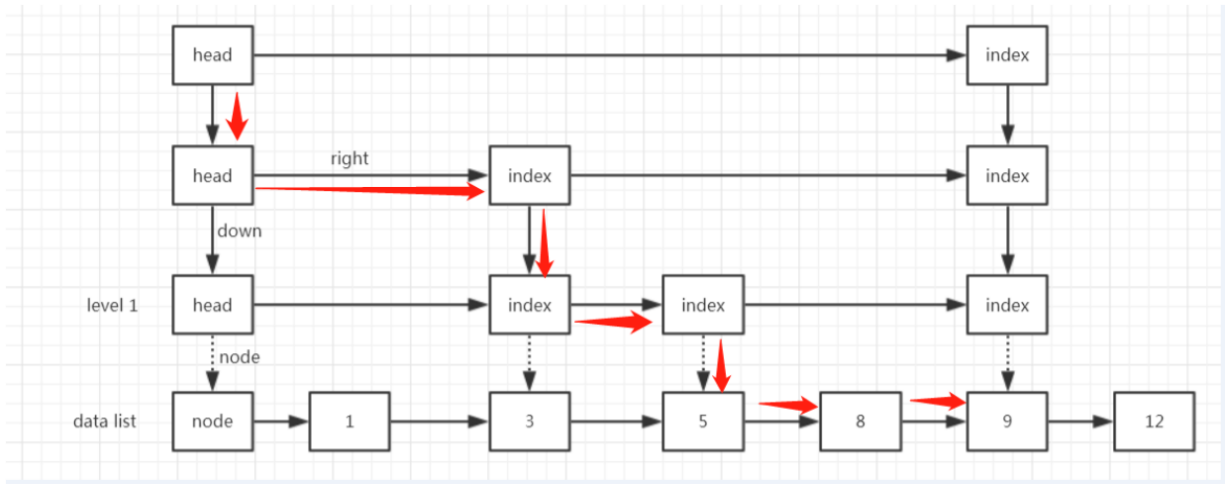


（1）寻找目标节点之前最近的一个索引对应的数据节点，这里就是5；

（2）从5开始往后遍历，经过8，到9；

（3）找到了返回；

整个路径如下图所示：



是不是很操蛋？

为啥不从9的索引直接过来呢？

从我实际打断点调试来看确实是按照上图的路径来走的。

我猜测可能是因为findPredecessor()这个方法是插入、删除、查找元素多个方法共用的，在单链表中插入和删除元素是需要记录前一个元素的，而查找并不需要，这里为了兼容三者使得编码相对简单一点，所以就使用了同样的逻辑，而没有单独对查找元素进行优化。

不过也可能是Doug Lea大神不小心写了个bug，如果有人知道原因请告诉我。（公众号后台留言，新公众号的文章下面不支持留言了，蛋疼）

彩蛋

为什么Redis选择使用跳表而不是红黑树来实现有序集合？

请查看【拜托，面试别再问我跳表了！】这篇文章。