# 1、hashMap

## 1.1 treeify红黑树化

/\*\*

\* 红黑树化

\* @return 树的根节点

\*/

final void treeify(Node<K,V>[] tab) {

TreeNode<K,V> root = null;

//循环整理

for (TreeNode<K,V> x = this, next; x != null; x = next) {

//取出下一个链表节点

next = (TreeNode<K,V>)x.next;

//将x节点的左右节点设置为null

x.left = x.right = null;

//判断当前红黑树是否有根节点

if (root == null) {

//如果没有根节点

//当前节点的父节点设置为null

x.parent = null;

//设置颜色为黑色（根节点为黑色）

x.red = false;

//将x节点设置为根节点

root = x;

}

//当前红黑树存在根节点

else {

//获取x节点的key

K k = x.key;

//获取x节点的hash

int h = x.hash;

//key的class

Class<?> kc = null;

//从根节点遍历，将x节点插入到红黑树中

for (TreeNode<K,V> p = root;;) {

//定义dir(方向),ph(节点hash)

int dir, ph;

//取出p节点的key

K pk = p.key;

//当p节点的hash大于x节点的hash时

if ((ph = p.hash) > h)

//左侧

dir = -1;

else if (ph < h)

//右侧

dir = 1;

//如果上面的if分支没走，则证明两个节点key的hash值相等，需要通过其他方式进行比较

//如果当前节点(x)的key的类实现了comparable接口，且当前循环节点(p)是相同Class的实例

//那么就通过comparable进行比较

else if ((kc == null &&

(kc = comparableClassFor(k)) == null) ||

(dir = compareComparables(kc, k, pk)) == 0)

//若还是相等，就通过tieBreakOrder比较

dir = tieBreakOrder(k, pk);

//先缓存p节点

TreeNode<K,V> xp = p;

//根据dir方向，来选择在左侧还是右侧插入

//并判断是否为null

if ((p = (dir <= 0) ? p.left : p.right) == null) {

//选择的左/右子树为null

//将原来的p节点(现xp)设置为x的父节点

x.parent = xp;

//如果dir 小于等于0

//将x节点放置在原p(现xp)节点的左侧

if (dir <= 0)

xp.left = x;

//如果dir 大于0

//将x节点放置在原p(现xp)节点的右侧

xp.right = x;

//调用balanceInsertion进行插入平衡

root = balanceInsertion(root, x);

break;

}

}

}

}

//确保哈希桶指定位置存储的节点是红黑树的根节点

moveRootToFront(tab, root);

}

### 1.1.1 moveRootToFront确保哈希桶指定位置存储的节点是红黑树的根节点

/\*\*

\* 确保哈希桶指定位置存储的节点是红黑树的根节点

\*/

static <K,V> void moveRootToFront(Node<K,V>[] tab, TreeNode<K,V> root) {

int n;

if (root != null && tab != null && (n = tab.length) > 0) {

//索引位置

int index = (n - 1) & root.hash;

TreeNode<K,V> first = (TreeNode<K,V>)tab[index];

//如果不是红黑树的根节点

if (root != first) {

Node<K,V> rn;

//指向红黑树的根节点

tab[index] = root;

TreeNode<K,V> rp = root.prev;

//整理节点顺序

if ((rn = root.next) != null)

((TreeNode<K,V>)rn).prev = rp;

if (rp != null)

rp.next = rn;

if (first != null)

first.prev = root;

root.next = first;

root.prev = null;

}

//递归做一个恒定校验

assert checkInvariants(root);

}

}

### 1.1.2 tieBreakOrder方法

/\*\*

\* 用这个方法来比较两个对象，返回值要么大于0，要么小于0，不会为0

\* 也就是说这一步一定能确定要插入的节点要么是树的左节点，要么是右节点，不然就无继续满足二叉树结构了

\* 先比较两个对象的类名，类名是字符串对象，就按字符串的比较规则

\* 如果两个对象是同一个类型，那么调用本地方法为两个对象生成hashCode值，再进行比较，hashCode相等的话返回-1

\*/

static int tieBreakOrder(Object a, Object b) {

int d;

if (a == null || b == null ||

(d = a.getClass().getName().

compareTo(b.getClass().getName())) == 0)

d = (System.identityHashCode(a) <= System.identityHashCode(b) ?

-1 : 1);

return d;

}

### 1.1.3 balanceInsertion 进行插入后的平衡操作

/\*\*

\* 插入平衡

\*/

static <K,V> TreeNode<K,V> balanceInsertion(TreeNode<K,V> root,

TreeNode<K,V> x) {

//将x节点设为红色（新插入节点一开始为红色）

x.red = true;

//一个没有边界的循环(需要内部跳出)

for (TreeNode<K,V> xp, xpp, xppl, xppr;;) {

//取出x的父节点并判断是否为null

if ((xp = x.parent) == null) {

//x没有父节点

x.red = false;//变色(黑色)

return x;//x为根节点发那会

}

//如果x存在父节点且x的父节点为黑色或x的父父节点不存在

else if (!xp.red || (xpp = xp.parent) == null)

//返回root

return root;

//如果x的父节点是父父节点的左孩子

if (xp == (xppl = xpp.left)) {

//父父节点的右孩子不为null且为红色

if ((xppr = xpp.right) != null && xppr.red) {

xppr.red = false;//变色(黑)

xp.red = false;//变色(黑)

xpp.red = true;//变色(红)

x = xpp;

}

else {

//x是父节点的右孩子

if (x == xp.right) {

//左旋

root = rotateLeft(root, x = xp);

//处理x的父父节点

xpp = (xp = x.parent) == null ? null : xp.parent;

}

//x的父节点存在

if (xp != null) {

xp.red = false;//变色

//x的父父节点存在

if (xpp != null) {

xpp.red = true;//变色

//右旋

root = rotateRight(root, xpp);

}

}

}

}

//如果x的父节点是父父节点的右孩子

else {

//x的父父节点的左孩子存在且为红色

if (xppl != null && xppl.red) {

xppl.red = false;//变色(黑)

xp.red = false;//变色(黑)

xpp.red = true;//变色(红)

x = xpp;

}

else {

//如果x是父节点的左孩子

if (x == xp.left) {

//右旋

root = rotateRight(root, x = xp);

//处理x的父父节点

xpp = (xp = x.parent) == null ? null : xp.parent;

}

//如果x的父节点存在

if (xp != null) {

xp.red = false;//变色(黑)

//如果x的父父节点存在

if (xpp != null) {

xpp.red = true;//变色(红)

//左旋

root = rotateLeft(root, xpp);

}

}

}

}

}

}

## 2、面试题

1、hashmap的死循环

Jdk1.7hashMap在高并发场景下扩容会出现死循环，一个线程扩容以后第二个线程重复扩容，导致链表成环从而导致死锁，详细来说扩容时会直接把node直接赋值给数组，且采用头插法，两个线程头尾形成死环，jdk1.8的优化是扩容时某个节点上的链表拆成两个链表，一个还在原来的位置上，另一个在原位置加原数组长度的位置上，而不是一个节点一个节点的移动，先将链表转移完成才赋值给数组，且不会对原链表做改动

2、为什么链表长度等于8的时候转换为红黑树

数组容量必须大于64的时候才能转换红黑树，之所以选择数字8是因为根据**泊松分布**，链表长度达到8个元素的概率为0.00000006，是第一个小于百万分之一的，所以使用8作为临界值，链表转换为树之后空间消耗高了一倍

1. hashmap存在的问题

高并发场景下进行put操作，如果发生hash碰撞value值被覆盖，则会丢失数据

# ConcurrentHashMap

Jdk1.7 使用的分段锁segment，其实现ReentrantLock，每个分段锁下面有一个小的数组存放具体的数据

Jdk1.8 使用cas加synchronized实现线程安全，当数组下标位置为空时使用cas实现添加节点安全，当数组下标不为空时，则使用synchronized锁住方法内的node对象，扩容时如果有线程来存放数据，则以其扩容

# CopyOnWriteArrayList

并发条件下安全的list集合

核心思想：读写分离，空间换时间，避免为保证并发安全导致的激烈的锁竞争。

划关键点：

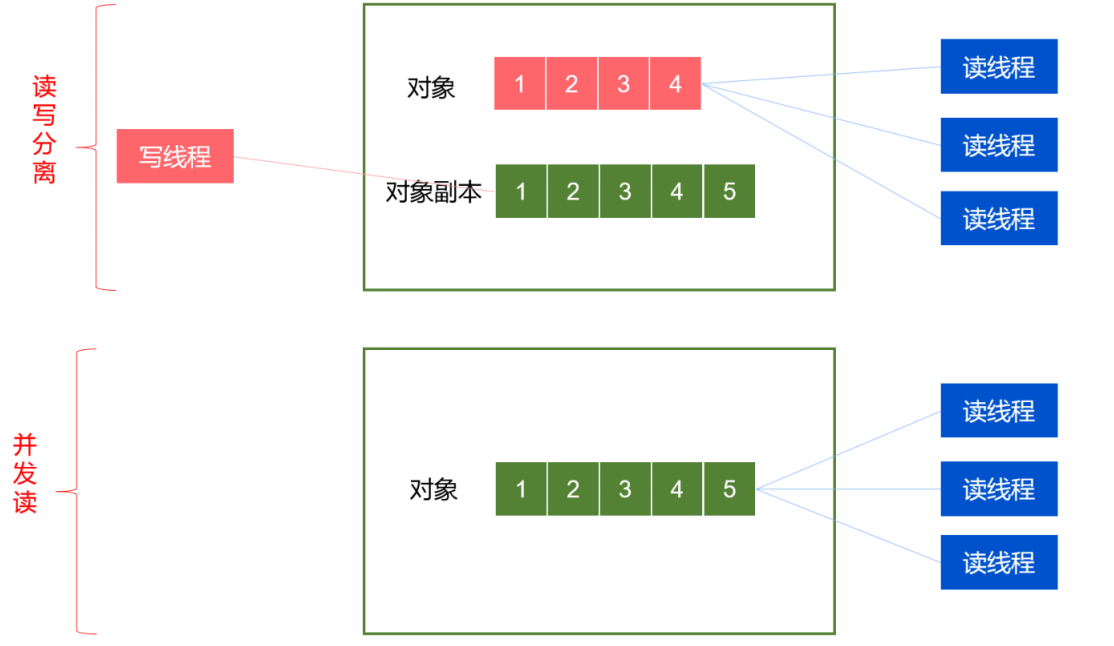
1、CopyOnWrite适用于读多写少的情况，最大程度的提高读的效率；

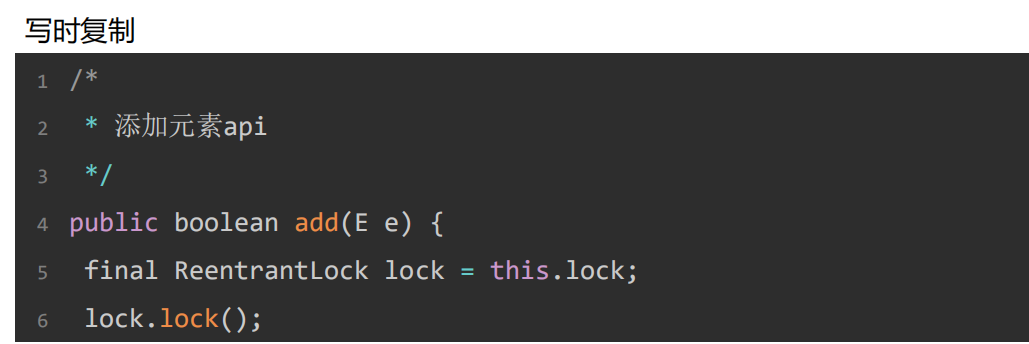
2、CopyOnWrite是最终一致性，在写的过程中，原有的读的数据是不会发生更新的，只有新的读

才能读到最新数据；

3、如何使其他线程能够及时读到新的数据，需要使用volatile变量；

4、写的时候不能并发写，需要对写操作进行加锁；







# ConcurrentSkipListMap基于调表skipList

并发条件下有序的map，在理论上能够O(log(n))时间内完成查找、插入、删除操作,skipList的时间复杂度为log(n)。

1、最底层的数据节点按照关键字升序排列。

2、包含多级索引，每个级别的索引节点按照其关联数据节点的关键字升序排列。

3、高级别索引是其低级别索引的子集。

4、如果关键字key在级别level=i的索引中出现，则级别level<=i的所有索引中都包含key。