集合

1、hashcode相等两个类一定相等吗?equals呢?相反呢?

比方说HashMap的key是要根据hashcode去寻找索引位置的，然后再判断equals，如果hashcode不重写而equals重写了，就会出现一些问题。

1.两个对象相等，hashcode一定相等
2.两个对象不等，hashcode不一定不等
3.hashcode相等，两个对象不一定相等
4.hashcode不等，两个对象一定不等

2、HashMap多线程并发会有什么问题

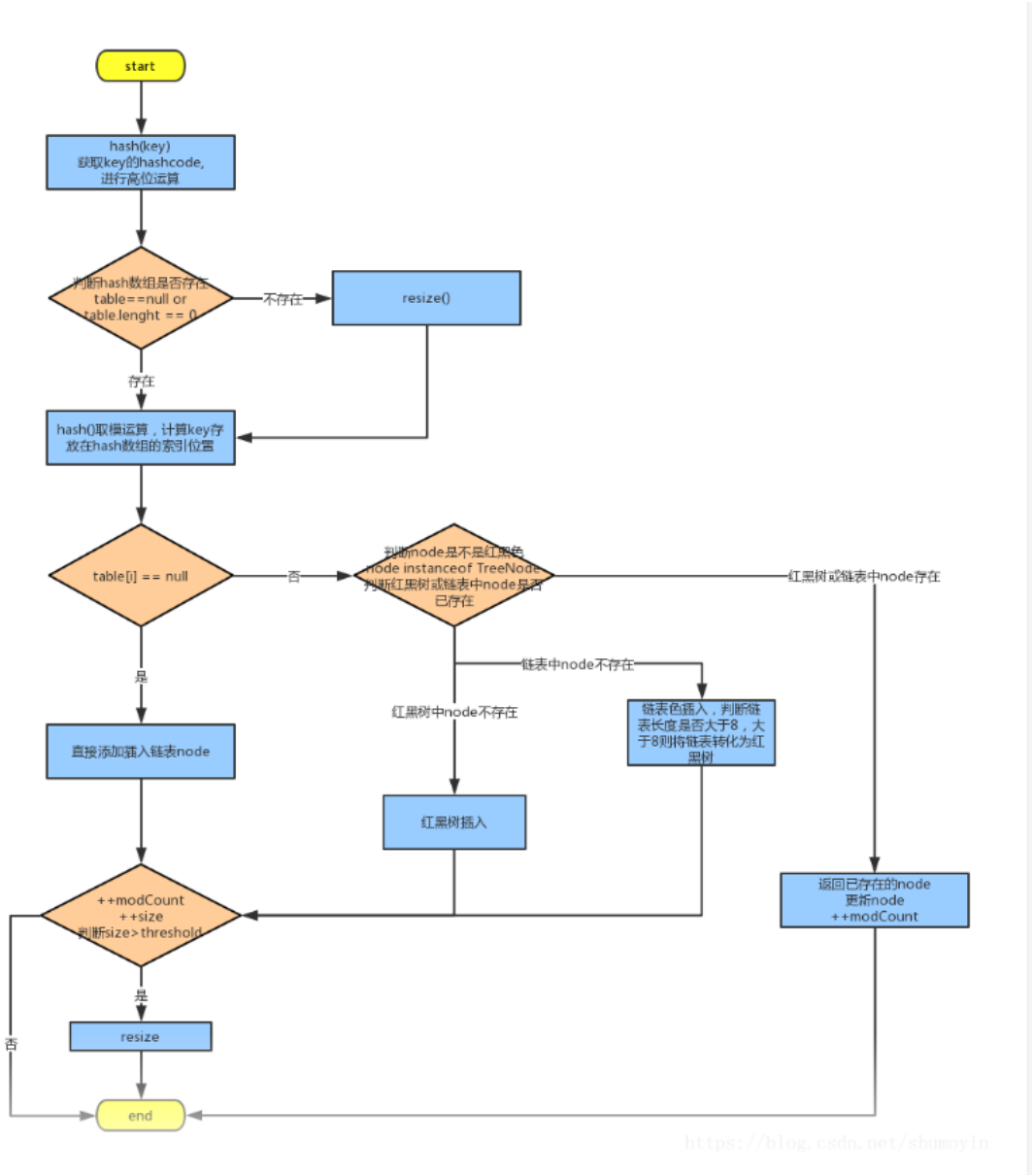
> 多线程put后可能导致get死循环

产生这个死循环的根源在于对一个未保护的共享变量,一个"HashMap"数据结构的操作,在所有操作的方法上加了"synchronized"后，一切恢复了正常

> 多线程put的时候可能导致元素丢失

主要问题出在addEntry方法的new Entry (hash, key, value, e)，如果两个线程都同时取得了e,则他们下一个元素都是e，然后赋值给table元素的时候有一个成功有一个丢失。

3、Hashmap的put操作过程



4、HashMap如何处理Hash冲突

HashMap中调用hashCode()方法来计算hashCode。由于在Java中两个不同的对象可能有一样的hashCode,所以不同的键可能有一样hashCode，从而导致冲突的产生。

在Jdk1.8之前，HashMap和其他基于Map类都是通过链地址发解决冲突，他们使用单项链表存储相同索引的元素。在最坏的情况下，这种方式将HashMap的get方法性能从O(1)降低到O(n)。

为了解决在频繁HashCode冲突性能降低问题，在jdk1.8中用平衡术（红黑树）存储冲突的元素。这意味着我们可以将最坏情况下的性能从O(n)提高到O(logn)。

在Java 8中使用常量TREEIFY_THRESHOLD来控制是否切换到平衡树来存储。目前，这个常量值是8，这意味着当有超过8个元素的索引一样时，HashMap会使用树来存储它们。

5、LinkedList与ArrayList区别适用场景Array list是如何扩容的

> ArrayList是基于数组实现的,ArryList初始化时，elementData数组大小默认为10,ArrayList线程不安全，Vector方法是同步的，线程安全；

> LinkedList是基于双链表实现的,初始化时，有个header Entry，值为null；使用header的优点是：在任何一个条目（包括第一个和最后一个）都有一个前置条目和一个后置条目，因此在LinkedList对象的开始或者末尾进行插入操作没有特殊的地方；

使用场景：从数据结构出发

```
1、如果应用程序对各个索引位置的元素进行大量的存取或删除操作，ArrayList对象要远优于LinkedList对象；
2、如果应用程序主要是对列表进行循环，并且循环时候进行插入或者删除操作，LinkedList对象要远优于ArrayList对象；
```

6、Treemap、HashMap、TreeSet、HashSet、HashTable、LinkedHashMap、concurrenthashtable、ConcurrentHashMap底层数据结构以及他们之间的相同点和不同点？

> Treemap

TreeMap是基于红黑树结构实现的一种Map，他默认是根据键值的自然顺序进行排序的，还可以根据创建映射时候提供的Comparator进行排序；

当未实现Comparator接口时候，key不可以为null，因为源码中如果为null，会抛出NullPointException； 当实现Comparator接口时候，如果比较其中没对Key进行判断，有可能抛出NullPointException；

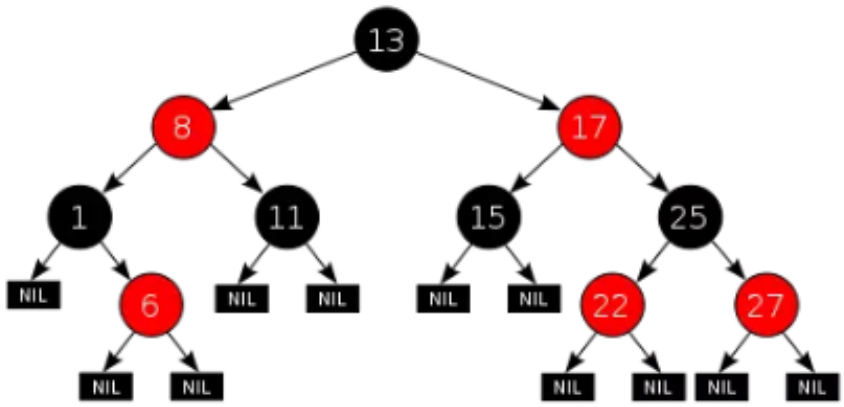
红黑树又称红-黑二叉树，它首先是一颗二叉树，它具体二叉树所有的特性。同时红黑树更是一颗自平衡的排序二叉树。

平衡二叉树必须具备如下特性：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。也就是说该二叉树的任何一个等等子节点，其左右子树的高度都相近。

红黑树顾名思义就是节点是红色或者黑色的平衡二叉树，它通过颜色的约束来维持着二叉树的平衡。对于一棵有效的红黑树二叉树而言我们必须增加如下规则：

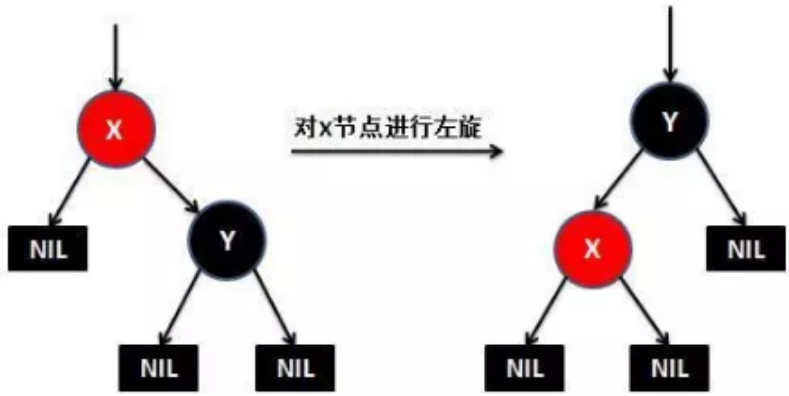
- 每个节点都只能是红色或者黑色
- 根节点是黑色
- 每个叶节点（NIL节点，空节点）是黑色的。
- 如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。
- 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

红黑树示意图如下：

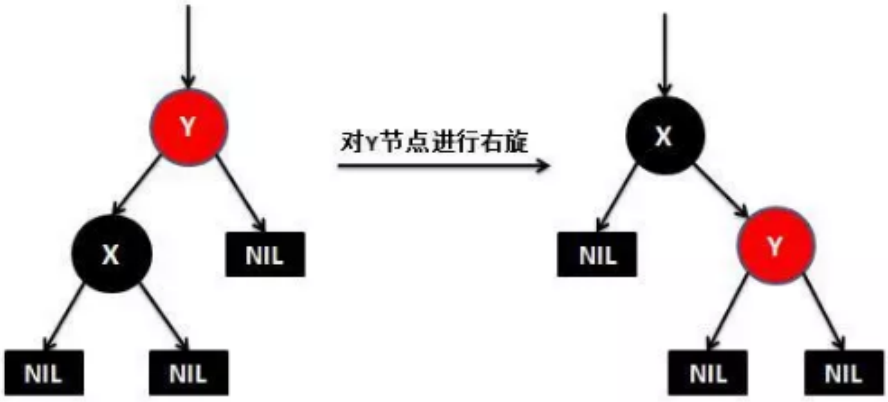


上面的规则前4条都好理解，第5条规则到底是什么情况，下面简单解释下，比如图中红8到1左边的叶子节点的路径包含两个黑色节点，到6下面的叶子节点的路径也包含两个黑色节点。 但是在在添加或删除节点后，红黑树就发生了变化，可能不再满足上面的5个特性，为了保持红黑树的以上特性，就有了三个动作：左旋、右旋、着色。

下面来看下什么是红黑树的左旋和右旋：



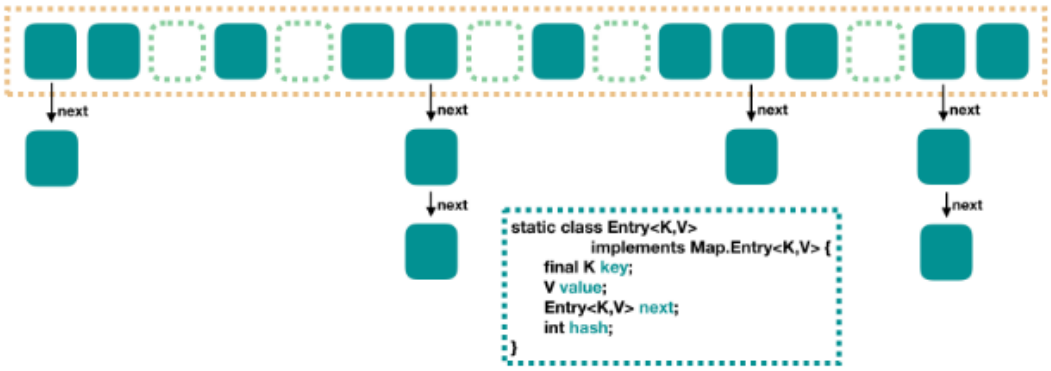
对x进行左旋，意味着"将x变成一个左节点"。



> HashMap

- jdk1.7： HashMap 里面是一个数组，然后数组中每个元素是一个单向链表,每个个的实体是嵌套类 Entry 的实例，Entry 包含四个属性：key, value, hash 值和用于单向链表的 next。

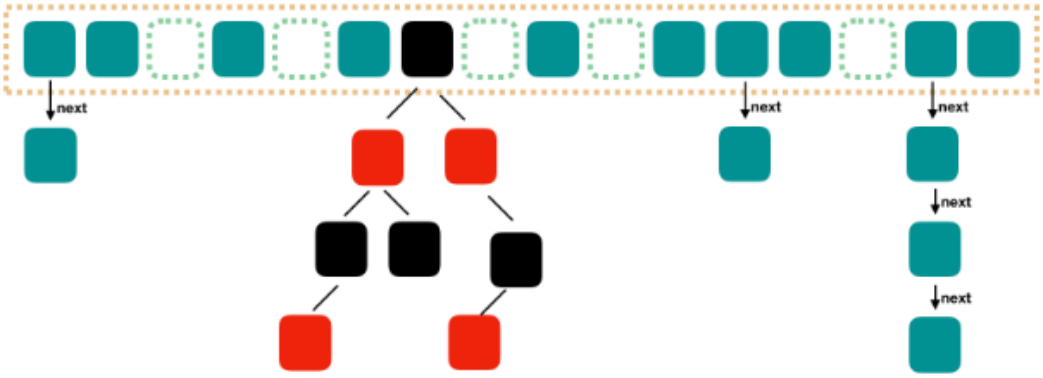
Java7 HashMap 结构



查找的时候，根据 hash 值我们能够快速定位到数组的具体下标，但是之后的话，需要顺着链表一个个比较下去才能找到我们需要的，时间复杂度取决于链表的长度，为 O(n)。

- jdk1.8： 最大的不同就是利用了红黑树，所以其由 数组+链表+红黑树 组成。

Java8 HashMap 结构



Java8 中使用 Node，基本没有区别，都是 key，value，hash 和 next 这四个属性，不过，Node 只能用于链表的情况，红黑树的情况需要使用 TreeNode。

- hashMap的[key]和[value]均可以为null:null

> HashSet vs. TreeSet vs. LinkedHashSet

- HashSet是采用hash表来实现的。其中的元素没有按顺序排列，add()、remove()以及contains()等方法都是复杂度为O(1)的方法。
- TreeSet是采用树结构实现(红黑树算法)。元素是按顺序进行排列，但是add()、remove()以及contains()等方法都是复杂度为O(log (n))的方法。它还提供了一些方法来处理排序的set，如first(), last(), headSet(), tailSet()等等。
- LinkedHashSet介于HashSet和TreeSet之间。它也是一个hash表，但是同时维护了一个双链表来记录插入的顺序。基本方法的复杂度为O(1)。
- TreeSet是SortedSet接口的唯一实现类，TreeSet可以确保集合元素处于排序状态。TreeSet支持两种排序方式，自然排序 和定制排序，其中自然排序为默认的排序方式。向TreeSet中加入的应该是同一个类的对象。
- TreeSet 是二差树实现的,Treeset中的数据是自动排好序的，不允许放入null值。
- HashSet 是哈希表实现的,HashSet中的数据是无序的，可以放入null，但只能放入一个null，两者中的值都不能重复，就如数据库中唯一约束。
- HashSet要求放入的对象必须实现HashCode()方法，放入的对象，是以hashcode码作为标识的，而具有相同内容的 String对象，hashcode是一样，所以放入的内容不能重复。但是同一个类的对象可以放入不同的实例 。
- inkedHashSet 底层是 数组 + 单链表 + 红黑树 + 双向链表的数据结构
- LinkedHashSet存储元素是无序的，但是由于双向链表的存在，迭代时获取元素的顺序等于元素的添加顺序，注意这里不是访问顺序

> HashTable

- 和HashMap一样，Hashtable 也是一个散列表，它存储的内容是键值对(key-value)映射。
- Hashtable 的函数都是同步的，这意味着它是线程安全的。它的key、value都不可以为null。此外，Hashtable中的映射不是有序的。
- Hashtable 的实例有两个参数影响其性能：初始容量 和 加载因子。容量 是哈希表中桶 的数量，初始容量 就是哈希表创建时的容量。注意，哈希表的状态为 open：在发生“哈希冲突”的情况下，单个桶会存储多个条目，这些条目必须按顺序搜索。加载因子 是对哈希表在其容量自动增加之前可以达到多满的一个尺度。初始容量和加载因子这两个参数只是对该实现的提示。关于何时以及是否调用 rehash 方法的具体细节则依赖于该实现。

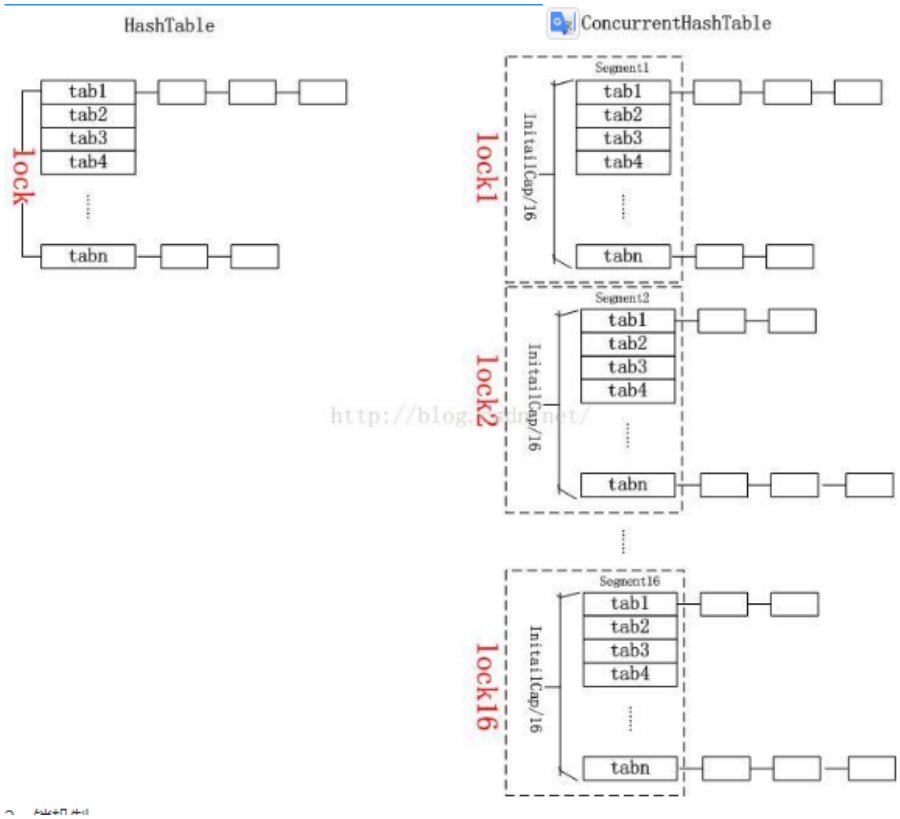
> LinkedHashMap

LinkedHashMap可以认为是HashMap+LinkedList，即它既使用HashMap操作数据结构，又使用LinkedList维护插入元素的先后顺序。Key和Value都允许空，Key重复会覆盖、Value允许重复

LinkedHashMap并未重写父类HashMap的put方法，而是重写了父类HashMap的put方法调用的子方法void recordAccess(HashMap m) ，void addEntry(int hash, K key, V value, int bucketIndex) 和void createEntry(int hash, K key, V value, int bucketIndex)，提供了自己特有的双向链接列表的实现。

> concurrenthashtable

ConcurrentHashTable可以看作是多个HashTable的组合，每个"HashTable"单元被成为一个段,一个段的大小为“HashTable”数组的长度，默认是InitailCapacity/16，在ConcurrentHashTable中InitialCapacity是用户创建时传进去的，容量和大小是不一样的，大小指元素的总个数，容量指的所有段中slot的总个数（小于InitialCapacity的最大的2的n次幂）。



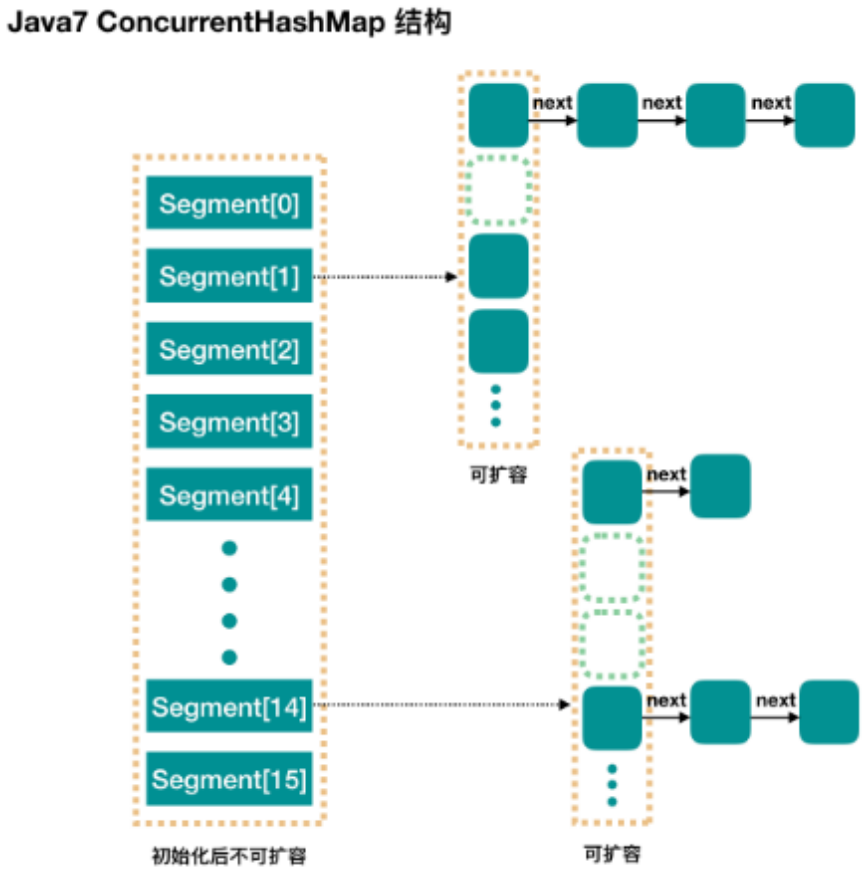
HashTable的线程安全使用的是一个单独的全部Map范围的锁，这个锁在所有的插入、删除、查询操作中都会持有，甚至在使用Iterator遍历整个Map时也会持有这个单独的锁。当锁被一个线程持有时，就能够防止其他线程访问该Map，即便其他线程都处于闲置状态。这种单个锁机制极大的限制了并发的性能。

> ConcurrentHashMap

- jdk1.7

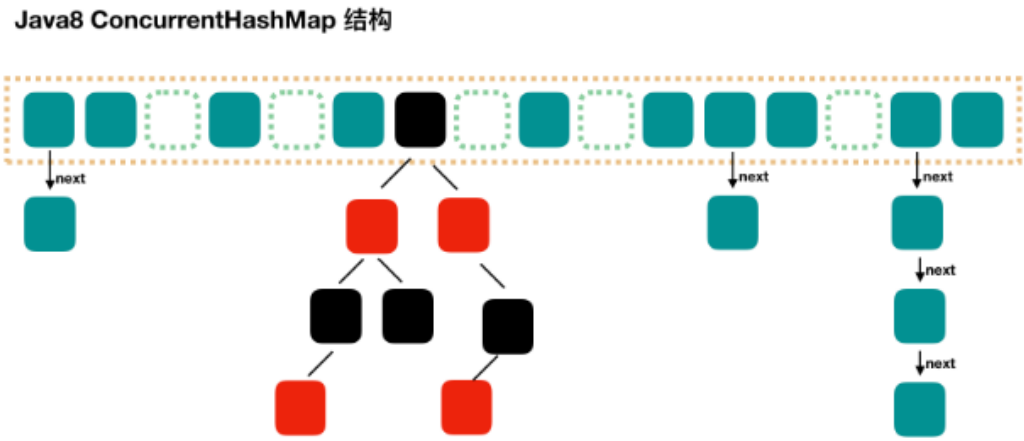
整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的意思，所以很多地方都会将其描述为分段锁。注意，行文中，我很多地方用了“槽”来代表一个 segment。

简单理解就是，ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。

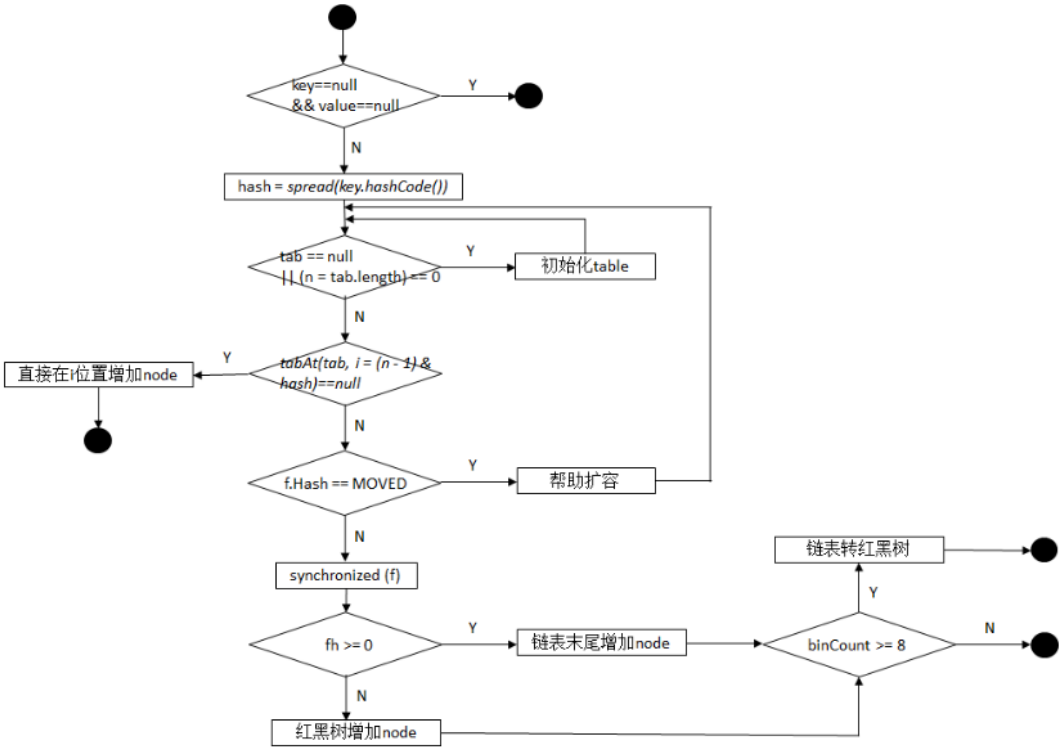


- jdk1.8

JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作



JDK8中的实现也是锁分离思想，只是锁住的是一个node，而不是JDK7中的Segment；锁住Node之前的操作是基于在volatile和CAS之上无锁并且线程安全的。



7、HashMap jdk1.8中对HashMap的优化

> 取消segments字段，直接采用transient volatile HashEntry<K,V>[] table保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。

> 将原先table数组＋单向链表的数据结构，变更为table数组＋单向链表＋红黑树的结构,对于hash表来说，最核心的能力在于将key hash之后能均匀的分布在数组中。如果hash之后散列的很均匀，那么table数组中的每个队列长度主要为0或者1。但实际情况并非总是如此理想，虽然ConcurrentHashMap类默认的加载因子为0.75，但是在数据量过大或者运气不佳的情况下，还是会存在一些队列长度过长的情况，如果还是采用单向列表方式，那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能

8、ThreadLocal是什么底层如何实现?写一个例子呗

```
public class ThreadLocalDemo {

    private static ThreadLocal<SimpleDateFormat> sdf = new
ThreadLocal<SimpleDateFormat>();

    public static void main(String[] args) {
        ExecutorService executorService =
Executors.newFixedThreadPool(10);
        for (int i = 0; i < 100; i++) {
            executorService.submit(new DateUtil("2019-11-25 09:00:" + i %
60));
        }
    }

    static class DateUtil implements Runnable{
        private String date;

        public DateUtil(String date) {
            this.date = date;
        }

        public void run() {
            if (sdf.get() == null) {
                sdf.set(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
            } else {
                try {
                    Date date = sdf.get().parse(this.date);
                    System.out.println(date);
                } catch (ParseException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```