**List，Set,Map三者的区别及总结**

* **List：对付顺序的好帮手**

**存储一组不唯一，有序的对象**

* **Set: 注重独一无二的性质**

**存储的对象不能重复**

* **Map: 用Key来搜索的专家**

**使用键值对存储，通过key直接定位到value**

**两个Key可以引用相同的对象，但Key不能重复**

**ArrayList,Vector, LinkedList的存储性能和特性？**

* ArrayList和Vector底层都是

1. 使用数组方式存储数据，因为数组内存地址连续，可以通过计算得到元素内存地址，所以查找快。
2. 插入元素会导致数组元素移动内存操作，所以存读快而插入删除慢；

//内存地址连续，前面动了后面都得动

* Vector的方法添加了同步修饰符，所以Vector是线程安全的，但性能较差
* 当需要增长时,Vector 默认增长为原来一培，ArrayList增长原来的一半

LinkedList底层

1. 使用双向链表实现存储查询需要进行向前或向后遍历，但插入数据只需要记录本项的前后项，所以插入快但查询慢
2. 它还提供了专门用于操作表头和表尾元素的方法，可以当作堆、栈、队列和双向队列使用

（将内存中零散的内存单元通过附加的引用关联起来，形成一个可按序号索引的线性结构，内存利用率高）

堆其实就是一棵完全二叉树（若设二叉树的深度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边） //满二叉树

**HashMap 和 Hashtable 的区别**

* HashMap底层结构是数组+链表/红黑树：先通过链表解决地址冲突问题O(n)，链表长度为8和散列表容量大于64会变成红黑树O(logN)

计算index方法：index = hash & (tab.length – 1)

HashTable：

计算index方法index = (hash & 0x7FFFFFFF) % tab.length //int类型最大值

* HashTable线程安全，但在修改数据时锁住整个HashTable，性能低

HashMap ~

* Hashmap可以有一个null键，可以有一个或多个键所对应的值为null

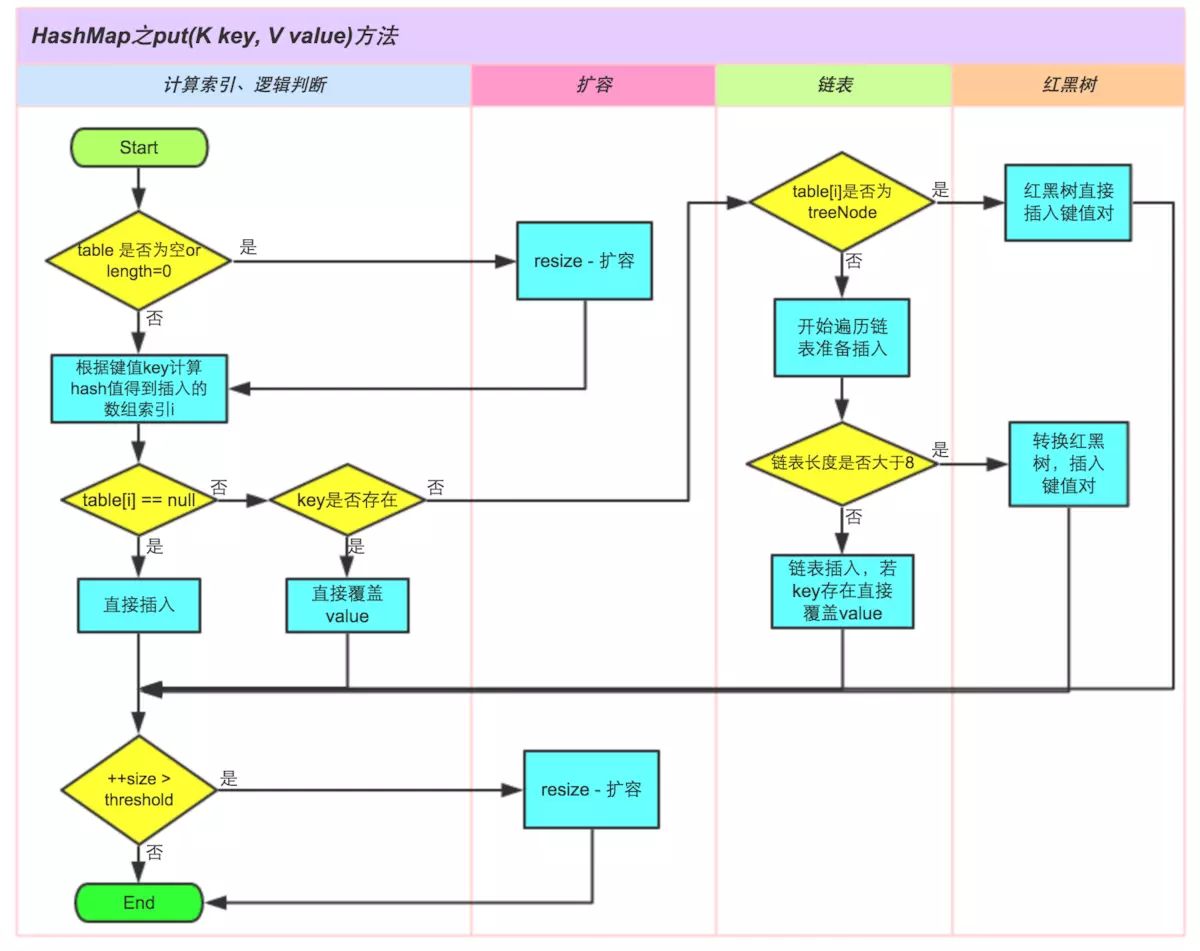
HashTable对象的key、value均不可为null

红黑树存储有序的数据

最长路径不超过最短路径的两倍

通过旋转和节点变色实现自平衡

**HashMap的put方法的具体流程？**



初始化散列表

通过key计算hash，在与数组长度-1做异或运算，得到在Entry数组位置

②开始遍历链表，如果key已存在则退出遍历，用新值代替旧值，否则插入~

**Hashmap的容量为什么是2的幂次**

1. 为了存取效率达到最高，就要要求hashMap每次保存的时候尽量平均分散到数组的各个位置，可以通过hashMap对数组长度取余运算

而如果容量为2的N次方的话，那么hash&(length-1) == hash%length

如果容量不为2的N次方，有些位置是存不到数据的

最后一位是0和任何的hash相与，最后的一位都得不到1 9/3位置得不到

1. 10进制计算，速度就比不上二进制运算

**HashMap是使用了哪些方法来有效解决哈希冲突的**

1. 使用链地址法来链接拥有相同hash值的数据；

2. 使用2次扰动函数来降低哈希冲突的概率，使得数据分布更平均；

（hash函数异或运算/求地址与运算）

3. 引入红黑树进一步降低遍历的时间复杂度，使得遍历更快



如果使用hashCode与长度取余，那么相当于参与运算的只有hashCode的低位，高位是没有起到任何作用的，所以最好让hashCode取值出的高位也参与运算，进一步降低hash碰撞的概率，使得数据分布更平均，我们把这样的操作称为扰动 不同1同为0

**为什么是两次扰动呢？**

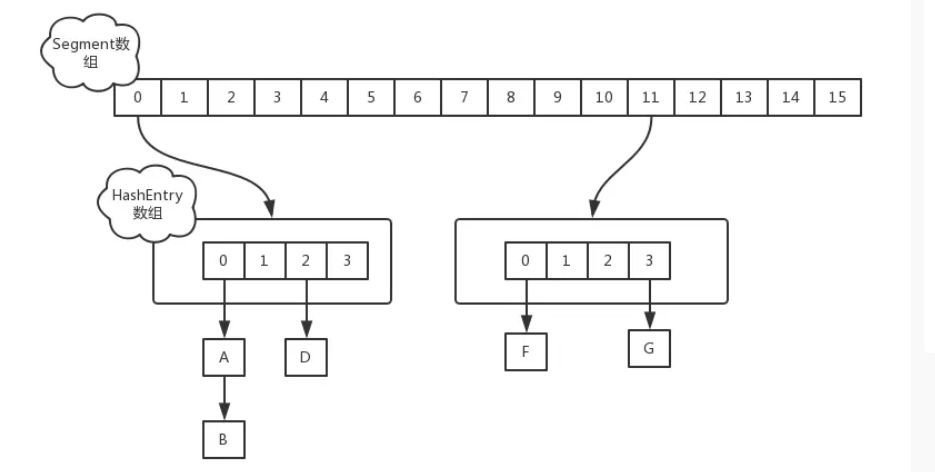
这样就是加大哈希值低位的随机性，使得分布更均匀，从而提高对应数组存储下标位置的随机性和均匀性(与)，最终减少Hash冲突，两次就够了，已经达到了高位低位同时参与运算的目的；

**HashMap/Table 和 ConcurrentHashMap的区别**

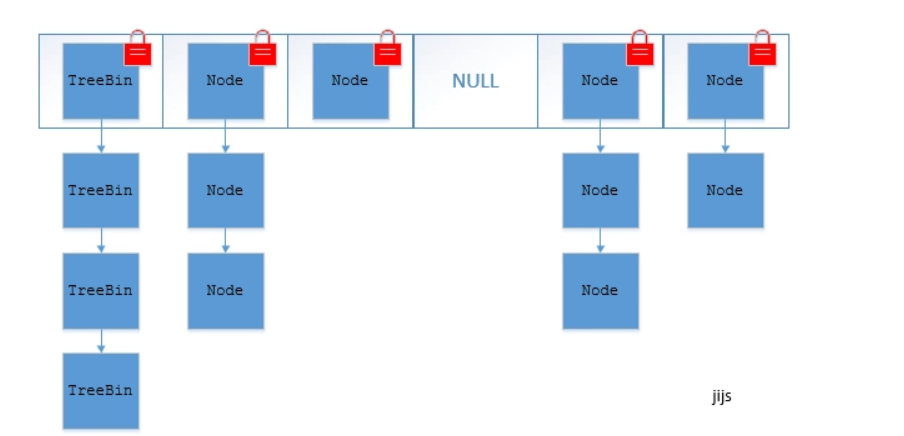
1. 在JDK1.7中，ConcurrentHashMap采用HashEntry[ ]+ Segment的方式进行实现

前者用来存放键值对，后者用来当锁

每个 Segment 守护一个HashEntry 数组里得元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 锁



1. 在JDK1.8中，取消了Segment分段锁，采用Node【】+ CAS + Syn来保证并发安全



锁粒度变小：原来是对需要进行数据操作的Segment加锁，现对每个数组元素加锁

长度超过8：链表转化为红黑树:查询时间复杂度从原来的遍历链表O(n)，变成遍历红黑树O(logN)

但HashTable的同步锁syn是锁住整个表，所以它更加精细，并发性能更好，而HashMap没有锁机制，不是线程安全的

CAS算法：比较与交换，能够保证一次读写操作是原子的, 当旧值和内存值相同时，将内存值修改为新值

1. HashMap的键值对可以null，但是ConCurrentHashMap都不允许

**Put方法**

1、判断Node[]数组是否初始化，没有则进行初始化操作

2、通过hash定位数组的索引坐标，是否有Node节点，如果没有则使用CAS进行添加（链表的头节点），添加失败则进入下次循环。

3、检查到内部正在扩容，就帮助它一块扩容。

4、如果当前头节点!=null，则使用synchronized锁住此元素（链表/红黑树的头元素）。如果是Node（链表结构）则执行链表的添加操作；如果是TreeNode（树型结构）则执行树添加操作。

最后

5、判断链表长度已经达到临界值8，超过这个值就需要把链表转换为树结构

6、检查是否需要扩容

**HashSet 和 HashMap 区别**

//因为要比较

**HashSet如何检查重复**

HashSet的底层是HashMap，添加时把数据作为K值，而V值一直使用同一个对象保存





由于HashMap的K值不允许重复，且在HashMap中如果K/V相同时，会用新的V覆盖掉旧的V，然后返回旧的V，那么在HashSet中执行这一句话始终会返回一个false，导致插入失败，这样就保证了数据的不可重复性；

为什么不直接使用equals()进行比较呢？如果只使用equals()，意味着需要迭代整个集合进行比较操作，如果集合中有1万个元素，就需要进行1万次比较

通过对象的物理地址转换成一个整数，然后该整数通过hash函数的算法就得到了**hashcode**。所以，hashcode就是在hash表中对应的位置

hash表中的8个位置，就好比8个桶，每个桶里能装很多的对象

两个对象的HashCode相同，不代表两个对象就相同，只能说明这两个对象在散列存储结构中，存放于同一个位置

**比较器**

**自然排序(内部比较器)**

**要实现Comparable接口**

TreeSet dset = new TreeSet();

dset.add(new Dog(2));

dset.add(new Dog(1));

Dog类要实现java.lang.Comparable的compareTo(T o1)方法

**定制排序(外部比较器**)

**要实现Comparator接口, 实现Compare(T o1, T o2)方法;** com per ri ter

**Arrays.sort(stu数组, stuComparator规则类)**

**区别：**

**创建类时没考虑到类的对象要比较没实现Comparable接口** comper ble

**或者**

**对类已经实现的排序规则不满意，则另建一个实现Comparator接口的类传递规则进去**

**自然：需要类自己实现comparable接口和compareTo(Object obj)方法**

**定制：需要实现comparator接口和compare(Object obj1, Object obj2)方法**

**对当前类已有的排序规则不满意可以实现Comparator接口，把排序规则传递进去**

TreeMap存键值对的键ky实现Comparable接口根据键对元素进行排序

也可以自定义排序规则：要实现Comparator接口

**两个对象值(x.equals(y) == true)，但却可有不同的hash code，这句话对不对？**

不对，equals方法比较两个对象相等时，它们hashcode相同，反之不然

因为equals可能被重写，所以equals和hashcode()方法应当一起被重写