**HashMap**

haspMap在1.8之前是数组+链表，在1.8之后是数组+链表+红黑树，另外1.8在hash算法，还有扩容resize()是也做出了较大的改变，提升了效率。另外1.8由1.7的头插法改为尾插法，解决了链表循环的问题。

**数据结构：哈希表（key:value）：数组+链表(node)+红黑树(TreeNode) HashMap.Node<K,V>[] table**

1．为什么使用31作为hashcode的乘数

1）31是质数，hash碰撞的概率很小，33，39，41也行，不过31就足够了

2）乘积运算可以使用位移提升运算效率：31\*i=(i<<5) – i

2．扰动函数

HashMap为了减少哈希碰撞，对key进行hash值计算后并不是直接使用，而是加入了一个扰动函数，增加了随机性

return key == null ? 0 : (h = key.hashCode()) ^ h >>> 16;

3．初始化容量

hashMap的容量为2的倍数，如果传入初始化容量为17，则会自动变为32。默认容量为0,(在第一次放值的时候为16)

4．负载因子

当容量超过多少时进行扩容。默认为0.75。当负载因子高时，减少空间，但增加碰撞几率，时间变长，负载因子低时，减少碰撞，搜索时间短，但浪费空间。

5. 阈值

当容量超过阈值（总容量\*负载因子）的时候进行扩容。默认为0（实际上为12，第一次添加的时候才会初始化为12）；

5. 插入

1）计算key的hash值，进行hash扰动，获取当前结点的hash值

2）判断当前结点是否为空，或者长度是否为0，是进行扩容操作。

if ((tab = this.table) == null || (n = tab.length) == 0) {  
 n = (tab = this.resize()).length;  
}

3）判断当前下表是否有存放数据，如果有，直接插入，否则进行覆盖或新增

没有存放数据，直接插入

if ((p = tab[i = n - 1 & hash]) == null) {  
 tab[i] = this.newNode(hash, key, value, (HashMap.Node)null);  
}

数据重复，进行覆盖（什么也没做，直接返回）

if (((HashMap.Node)p).hash == hash && ((k = ((HashMap.Node)p).key) == key || key != null && key.equals(k))) {  
 e = p;  
}

4)判断是否为树结点，不是就向链表中插入，是的话就向树插入

插入树结点

else if (p instanceof HashMap.TreeNode) {  
 e = ((HashMap.TreeNode)p).putTreeVal(this, tab, hash, key, value);  
}

//插入链表结点，链表结点数量大于等于8时，转化为树,但是不是所有大于等于8时都转树，只有当map的大小大于等于64的时候才转树，否则直接进行扩容。

while(true) {  
 if ((e = ((HashMap.Node)p).next) == null) {  
 ((HashMap.Node)p).next = this.newNode(hash, key, value, (HashMap.Node)null);

if (binCount >= 7) {  
 this.treeifyBin(tab, hash);  
 }  
 break;  
 }  
  
 if (((HashMap.Node)e).hash == hash && ((k = ((HashMap.Node)e).key) == key || key != null && key.equals(k))) {  
 break;  
 }  
  
 p = e;  
 ++binCount;  
}

5)扩容机制

首先保存旧的数组并new一个新的数组，大小为原来的两倍。然后将旧的数组的内容迁移到新的数组上去。这里resize1.8做出了比较大的改变。在1.7中，旧数组所有的数据都需要根据rehash值重新异或数组长度，然后通过头插法放入新数组。在1.8中，扩容后的位置为原位置或者原位置+旧容量。那么它如何决定是原位置还是旧位置呢，就是将hash值与容量进行&运算，如果等于0，就是原位置，否则是原位置+旧容量。

6．删除

链表进行遍历删除，树涉及树的修复，以及删除后当长度小于6时，进行链表话，不是7而是6是为了防止反复横跳，一下树一下链表

7．查询

通过Hash值得到数组下标，如果没有碰撞直接获取，碰撞了从链表中（遍历）或树中获取

8. 头插法和尾插法

头插法在多线程的情况下，进行扩容的时候，可能会造成环兴链表，而尾插法不会，但是仍然有线程安全的问题。

TreeMap

数据结构：红黑树，每一个结点存放着，key,val,parent

特点：红黑树是有序的，所以TreeMap也是有序的，根据键升序排序，也可以指定排序的比较器。

由于存在排序，所以TreeMap效率不如HashMap（键值无序），Map需要有序的场合才使用TreeMap。

HashMap键值取出时候的顺序是根据数组的顺序取出的，而数组中插入时根据hash值插入的，所以时无序的。

this.next = t[this.index++]

LinkedHashMap

数据结构：双向循环链表+HashMap

false, 所有的Entry按照插入的顺序排列

true, 所有的Entry按照访问的顺序排列 实现LRU缓存

LinkedHashMap 是**HashMap的一个子类**，保存了**记录的插入顺序**，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的.也可以在构造时用带参数，按照应用次数排序。**在遍历的时候会比HashMap慢，**不过有种情况例外，当HashMap容量很大，实际数据较少时，遍历起来可能会比 LinkedHashMap慢，因为LinkedHashMap的遍历速度只和实际数据有关，和容量无关，而HashMap的遍历速度和他的容量有关（遍历数组）。

最大的不同在于重写了父类的Entry,多了before和after结点，维护了插入元素的有序性。插入，删除都是调用父类的方法，不过重写了父类AfterInsertion,进行了结点的维护。

static class Entry<K, V> extends Node<K, V> {  
 LinkedHashMap.Entry<K, V> before;  
 LinkedHashMap.Entry<K, V> after;  
  
 Entry(int hash, K key, V value, Node<K, V> next) {  
 super(hash, key, value, next);  
 }  
}

Hashtable

和HashMap最主要的区别在于Hashtable是线程安全。使用了synchronized保证线程安全，不过锁粒度太大，效率太低，不推荐使用，现使用concurrentHashMap替代。

concurrentHashMap

在JDK1.7版本中，ConcurrentHashMap的数据结构是由一个Segment数组和多个HashEntry组成, Segment数组的意义就是将一个大的table分割成多个小的table来进行加锁，也就是上面的提到的锁分离技术，而每一个Segment元素存储的是HashEntry数组+链表，这个和HashMap的数据存储结构一样。并发度默认为16,

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**，整个看起来就像是优化过且线程安全的HashMap，虽然在JDK1.8中还能看到Segment的数据结构，但是已经简化了属性，只是为了兼容旧版本