HashMap(JDK1.8)

HashMap源码分析

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
static final int hash(Object key) {
    int h;
    //判断我们传入的键是否为空
    //我们需要思考为什么要无符号右移动16位
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    //考虑(n - 1) & hash为什么不会数组越界
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
```

```
47      ++modCount;
48      if (++size > threshold)
49          resize();
50      afterNodeInsertion(evict);
51      return null;
52  }
```

为什么默认长度是16，以及为什么每次扩容是2的幂次方
(1)好处1:获取元素存储位置的索引是通过(n - 1) & hash来获取的
注意最后return的是h&(n-1)。如果n不为2的幂，比如15。那么length-1的2进制就会变成1110。在h为随机数的情况下，和1110做&操作。尾数永远为0。那么0001、1001、1101等尾数为1的位置就永远不可能被entry占用。这样会造成浪费，不随机等问题。 n-1 二进制中为1的位数越多，那么分布就平均
(2)好处2:扩容之后不需要重新通过hash计算元素的位置,举个例子来说明以下

原来的hash值是5，看以下计算过程

```
1     0000 0000 0000 0000 0000 0000 0000 1111 //(n-1)=15二进制表示
2  &  0000 0000 0000 0000 0000 0000 0000 0101 //hash值5二进制表示
3  ___________________________________________________________________
4     0000 0000 0000 0000 0000 0000 0000 0101
5
6   //看扩容之后的计算(扩容到32)
7     0000 0000 0000 0000 0000 0000 0001 1111 //(32-1)=31二进制表示
8  &  0000 0000 0000 0000 0000 0000 0000 0101 //hash值5二进制表示
9  ___________________________________________________________________
10    0000 0000 0000 0000 0000 0000 0001 0101
```

扩容之后的位置=原位置(5) + oldCap(原集合的容量) = 21

原来的hash值是14

```
1     0000 0000 0000 0000 0000 0000 0000 1111 //(n-1)=15二进制表示
2  &  0000 0000 0000 0000 0000 0000 0000 1110 //hash值14二进制表示
3  ___________________________________________________________________
4     0000 0000 0000 0000 0000 0000 0000 1110
5
6   //看扩容之后的计算(扩容到32)
7     0000 0000 0000 0000 0000 0000 0001 1111 //(32-1)=31二进制表示
8  &  0000 0000 0000 0000 0000 0000 0000 1110 //hash值5二进制表示
9  ___________________________________________________________________
10    0000 0000 0000 0000 0000 0000 0000 1110
```

扩容之后的位置=原位置(14)
总结规律:
resize过程中不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引=原索引+oldCap

(3)好处3:位运算比较快

HashMap不扩容行不行？有什么问题吗?

面试术语
在JDK1.8中HashMap的实现是通过(数组+链表+二叉树)来实现的
首先存入数据调用put方法的时候会通过hash方法对键做一次hash操作，最后把原键的hash值和原键hash值无符号右移16位的值按位异或的结果，作为最后返回的hash值，这么做的原因是为了取得的hash值更加的均匀减少hash冲突的次数，在调用putVal存储我们的值，判断Node节点的数组是否创建，如果没有创建，

我们也没有设置HashMap的initialCapacity初始化容量和负载因子loadFactor，则创建默认的容量为16，负载因子为0.75的数组，接下来分为2中情况
1.通过返回的hash值按位异或我们创建的数组的长度减1的做过作为元素在数组中的坐标，如果该位置没有存放元素，我们直接创建Node节点存储数据
2.如果数组索引的位置存在元素又分为三中情况
(1)键的hash值和我们要存入的数据的键的hash值相同，并且值也相同，说明我们存入的数据重复，这种情况下，会把新的值覆盖原来的值
(2)判断获取到的元素是否为TreeNode结构，如果是挂载到TreeNode中
(3)遍历table数组，找出尾节点，把我们的数据挂载到为节点中
到此存入数据的常规流程完毕
可以接着分析如果链表中的数据过多，自动转换为二叉树的过程在treeifyBin中我们先会判断链表的长度默认如果超过了8个(TREEIFY_THRESHOLD)会自动转换为二叉树
构建的过程比较简单，遍历我们的数组重新整理为二叉树的数据结构，这么设计的原因是为了查询快
接着分析数据的扩容过程

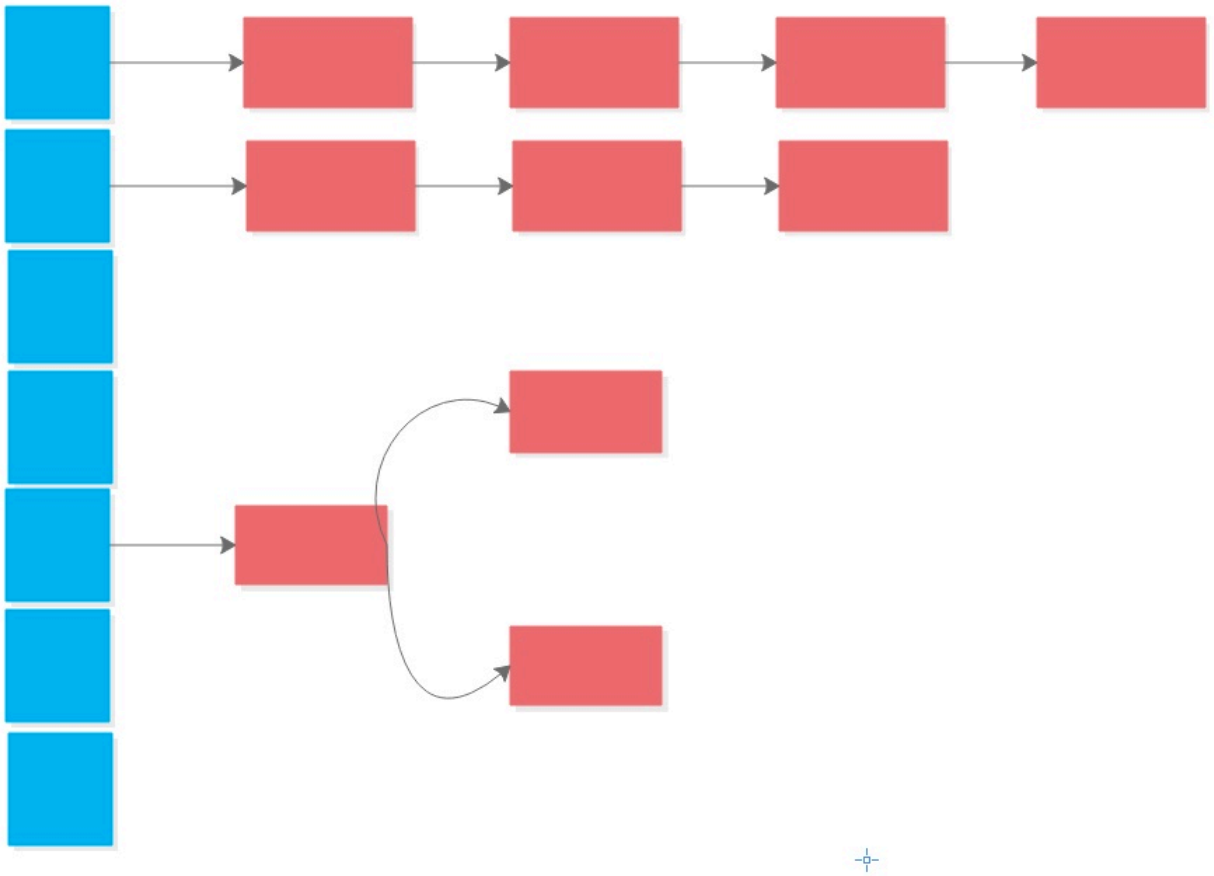
```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order
```

```
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
    }
    return newTab;
}
```

HashMap优化

1.防止过多的扩容，如果我们知道集合的容量，可以在初始化的时候设置容量

HashMap存储结构图



HashMap存在的一些问题

1.HashMap在做put的时候引起的数据丢失问题