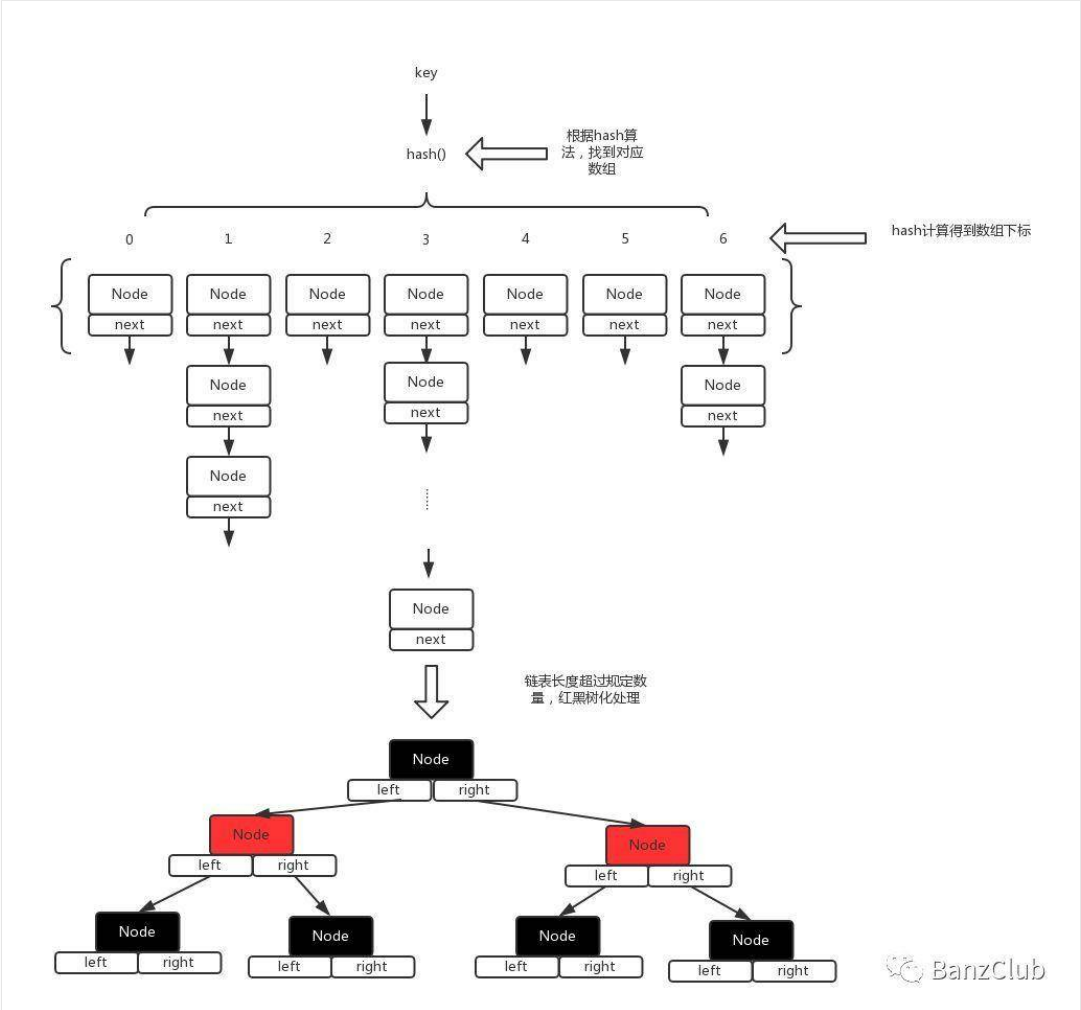
HashMap分析

一、 底层数据结构优化

HashMap在Java1.7里使用的是**数组+链表**的数据结构，在Java1.8里使用的是**数组+链表+红黑树**

HashMap处理“碰撞”增加了**红黑树**这种数据结构，当碰撞结点较少时，**采用链表存储，当较大时（>8个），采用红黑树（特点是查询时间是O（logn））存储（有一个阀值控制，大于阀值(8个)，将链表存储转换成红黑树存储。**



为什么要将链表进行树化操作呢？可以看看1.7版本之前的HashMap实现，hash碰撞之后，将无限增加链表的长度，大家都知道链表的添加、查找、删除时间复杂度是O(n)，这使得HashMap在发生hash碰撞之后，效率变成了链表，而完全用散列实现，在元素比较多的时候，意味着资源的浪费，所以带有自平衡属性红黑树的引入，使得HashMap整体结构和效率更加平衡。二叉树的添加、查找、删除操作的时间复杂度是O(logn)。

二 、HashMap源码解析get/put

jdk1.8 put源码

```
        final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                       boolean evict) {
         Node<K,V>[] tab; Node<K,V> p; int n, i;
        //1
         if ((tab = table) == null || (n = tab.length) == 0)
             n = (tab = resize()).length;
        //2
         if ((p = tab[i = (n - 1) & hash]) == null)
             tab[i] = newNode(hash, key, value, null);
         else {
             Node<K,V> e; K k;
             //3
             if (p.hash == hash &&
                 ((k = p.key) == key || (key != null && key.equals(k))))
                 e = p;
             //4
             else if (p instanceof TreeNode)
                 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
             else {
             //5
                 for (int binCount = 0; ; ++binCount) {
                   //6
                    if ((e = p.next) == null) {
                         p.next = newNode(hash, key, value, null);
                         if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for
1st
                             treeifyBin(tab, hash);
                         break;
                     }
                   //7
                     if (e.hash == hash &&
                         ((k = e.key) == key || (key != null &&
key.equals(k))))
                         break;
                     p = e;
                 }
             }
             //8
             if (e != null) { // existing mapping for key
                 V oldValue = e.value;
                 if (!onlyIfAbsent || oldValue == null)
                     e.value = value;
                 afterNodeAccess(e);
                 return oldValue;
             }
         }
         ++modCount;
          //9
```

```
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

主逻辑：

1. 判断当前数组是否为空，如果为空进行初始化
2. 找到key对应的数组的值，判断是否为null
3. 如果找的的node不为空，判断hash值以及key是否相等，如果相等就返回
4. 如果3不符合条件，判断是否是红黑树，如果是按照树的方式赋值
5. 如果不是红黑树，循环遍历链表
6. 找到下一个node，判断是否为null.如果为null,新建一个ndoe,再判断是超过阀值，如果超过就转变成红黑树
7. 如果下一个node不为空，判断hash值以及key是否相等，如果相等返回，
8. 如果e!=null，说明找到了hash值相等的值，覆盖旧的值
9. 最后判断你是否扩容

jdk1.8 get源码

```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    //1
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        //2
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
       //3
        if ((e = first.next) != null) {
            //4
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            //5
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null &&
key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }//6
    return null;
}
```

主逻辑：

1. 判断node数组是否为空
2. 获取到节点，判断是否hash值和key是否相等.如果相等就返回
3. 寻找下一个节点
4. 判断节点是否是树节点，如果是，根据红黑树，返回节点值
5. 判断节点是否符合条件相等，如果相等返回值
6. 否则返回null

三、Hash算法和容量

HashMap容器的创建，采用了延迟初始化，创建容器时，只是指定了负载系数（loadFactor）和扩展阈值（threshold），真正创建容器，是在put第一个元素的时候；而HashMap的容量被指定为2的整数平方倍，下面来看一下怎么保证容量始终为2的整数平方倍：

- 不指定容量的话，初始为16；
- 如果指定的话，会使用tableSizeFor（）方法转化为一个2的整数平方倍。赋给实例变量threshold，到创建容器时候又将threshold赋给了新容量。

再附官网说明

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

参考资料：

Java的Hashmap https://cloud.tencent.com/developer/article/1467547
Java集合丨重识HashMap https://cloud.tencent.com/developer/article/1446678
HashMap源码 https://cloud.tencent.com/developer/article/1644914