HashMap

- 底层存储结构为hash数组+链表
  - 链表为了解决hash冲突
  - 当底层链表长度>8并且hash数组长度>64就会转化为红黑树
    - 数组长度<64?
      - 是：扩容，并且重写计算hash值
      - 否，将这个数组桶里面的链表转换为红黑树
- 特点
  - 无序存储
  - 键值都可以null
  - 键唯一
  - 底层数据结构通过键值控制
  - jdk1.8之前是链表+数组　1.8之后是链表+数组+红黑树
  - 阈值：链表长度>8 && 数组长度>64 转化为红黑树，否者扩容　效率问题
- 数据结构
  - 存储结构
    - 1.8之前
      - 构造一个长度为16的Entry[] table 用来存储键值对
      - 使用头插法
    - 1.8以后
      - 在第一次put数据的时候创建一个Nde[] table 数组存储键值对
      - 使用尾插法
  - 存储步骤　put一对数值　计算key的hashCode()　算法映射到底层数组长度内　如果数组有原素判断hash值是否相等，如果hash碰撞了，那么就使用它的equals()方法比较是否一样
    - 全部都不一样那么使用头插法
    - 遇见一样的，直接覆盖vlaue
  - 成员变量
    - 序列化版本号　private static final long serialVersionUID = 362498820763181265L
    - 集合初始化容量　static final int DEFAULT_INITIAL_CAPACITY = 1<<4;
      - 必须是二的n次幂
        - 因为当hashMap添加一个数据的时候需要更具key的hash值区判断在数组哪一个下标下面，而这个算法就是取模，但是取模的效率不如位运算，所以做了优化，使用了hash & (length - 1)，这个实际上就是去摸，但是前提是length是2的n次幂
        - 如果不是：那么会找比它大的最近的2的n次幂作为初始容量
          - 通过一个位移或运算，将最高位的后边每一位都变成1
          - 底层还会判断是否大于2的30次方，如果是的话就直接给它2的30次方，如果给的是0那么会返回1
      - 默认容量是16
    - 重点
      - 存放原素个数　transient int size;　用于存放原素个数，这个个数不等于数组长度
      - 当链表的值小于6则会从红黑树转回链表　static final int UNTREEIFY_THRESHOLD = 6;　这种情况只会在rehash之后发生
      - 集合最大容量　static final int MAXIMUM_CAPACITY = 1<<30;　集合最大容量为2的30次方
      - 默认负载因子　static final float DEFAULT_LOAD_FACTOR = 0.75f;
        - 用来决定扩容的阈值
        - 当原素总数大于数组长度 * 负载因子就会扩容
      - 1.8新增 当链表的值超过8则会转为红黑树　static final int TREEIFY_THRESHOLD = 8;
        - 当桶(bucket)上的结点数大于这个值的时候会转成红黑树
        - 因为根据泊松分布，在有限个数内，八个以上存在同一个链表上是非常小的几率的
      - 存储原素的数组　transient Node<K,V>[] table;
        - 1.8以前为Entry[]
        - 1.8以后为Node[]
        - 实现了Map.Entry<K,V>接口
        - table用来初始化，必须是二的n次幂
      - hash表的加载因子也称负载因子　final float loadFactor;
        - 默认为阈值为0.75
          - 太大：导致原素查询效率低下
          - 太小：导致数组利用率很低，存放的数据会很分散
          - 经过大量测试，0.75最合适，不建议修改
        - 由于这个负载因子会印象扩容，并且扩容的话会使用到一系列rehash以及复制数据等操作，所以应该减少扩容的次数，应该在创建对象的时候指定初始容量来减少扩容
    - 了解
      - 用来记录HashMap的修改次数　transient int modCount;
      - 用来调整大小下一个容量的值计算方式为(容量 * 负载因子)　int threshold;　衡量数组是否扩容的一个标准
  - 构造方法
    - 无参构造　将默认加载因子0.75赋值给loadFactor
    - 有参构造：(int initialCapacity)　指定容量大小的构造函数，并且传入默认加载因子
    - 有参构造：(int initialCapacity, float loadFactor)　指定容量大小，指定加载因子　不推荐指定加载因子
    - 有参构造：(Map<? extends K, ? extends V> m)
      - 传入一个map构造一个hashmap
      - 默认负载因子0.75
      - 这个构造发放里面会调用一个putMapEntries方法
        - 这个方法判断hashMap里面的数组是否为空，然后将原素添加进数组
        - 这里有个点：当table == null的时候，会判断是否需要扩容
          - float ft = ((float) s / loadFactor) + 1.0F;
          - 这里为什么要加1?
            - 因为后面需要判断是否扩容，这里保证了在后面传入第一个参数的时候不必要扩容，直接在创建的时候就创建两倍大就行了
            - 因为后面会根据这个值设置数组长度，比如这里如果不加1等于8那么后面数组第一次添加原素就会扩容，如果这里加了1，那么后面调用tableISizeFor方法的时候就会将长度设置为16
- 面试题
  - 1. hash表底层采用何种算法计算hash值？还有那些算法可以计算hash值?
    - 使用key的hashCode()方法获得hash值结合数组长度，通过后面的方法获得数组下标 hash & (length - 1)
      - 使用无符号右移(>>>)
      - 按位异或(^)
      - 按位与(&)
    - 一切其他的计算方法
      - 取余数
      - 伪随机数法
      - 平方取中法
    - 因为效率问题所以底层选择了位运算的方式计算数组下标映射
  - 当两个对象hashCode相等会怎样?　调用key的equals()判断值是否相等
    - 相等就更改value
    - 不相等就插入这个原素在链表　链表长度>8 && 数组长度>64转换为红黑树
  - 何时发生hash碰撞，什么是hash碰撞，如何解决hash碰撞?　当key计算的hash值相等就会发生hash碰撞
    - jdk 1.8之前采用链表解决hash碰撞
    - jdk 1.8之后采用链表 + 红黑树解决hash碰撞
  - 当两个键的hash值相同，如何存储键值对?　当hash值相同，通过equals判断key内容是否相同
    - 相同：覆盖value
    - 不相同：将键值对添加到hash表中
  - 在不断添加原素的过程中会发生扩容问题?　当put原素数超过负载因子（默认为0.75），就会触发扩容，容量为原来的2倍，然后将数据复制过来
  - 传统hashMap的缺点，1.8为什么引入红黑树?　因为性能问题，链表的查询速度为O(n),红黑树为O(logn)，在不频繁旋转红黑树的情况下，红黑树速度是比链表快的