HashMap是一个利用哈希表来存储元素的集合,遇到冲突,HashMap采用链地址法来解决,在Java7中,HashMap是由数组+链表构成的,在Java8中HashMap是由数组+链表+红黑树构成,新增了红黑树作为底层数据结构,结构更加复杂,但是效率也更高.

一.HashMap的整体概述与结构

HashMap实现了Map接口,Map接口中定义了一组键值对映射通用的操作,存储一组成对的KEY-VALUE对象,提供KEY到VALUE的映射,Map中的KEY不要求有序,不允许重复,VALUE同样不要求有序,但是可以重复.

Map接口中的方法如下:

该接口中的方法有很多,其中还有很多Java8中新增的与函数式接口相关的方法.

如果之后我们需要自定义一个Map结构,就需要实现Map接口,但是如果在使用的过程中不需要对Map中所有接口方法都进行重写,那么可以选择继承AbstractMap,该抽象类继承Map接口,可以选择自己想要的方法进行重写(装饰设计模式)

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
```

但是实际上HashMap既实现了AbstractMap也实现了Map接口,原因是集合框架的创始人Josh Bloch写错了😂 但是这个错误也就被保留了下来.

二.HashMap中的常用字段

```
//序列化版本号
private static final long serialVersionUID = 362498820763181265L;

//默认HashMap集合初始容量为16 (1<<4)是数字1左移4位,本质上就是1 * 2的四次幂也就是16
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;

//集合的最大容量,如果通过有参构造指定容量的时候超过此数值,默认使用此数值 1073741824
static final int MAXIMUM_CAPACITY = 1 << 30;

//默认的负载因子
static final float DEFAULT_LOAD_FACTOR = 0.75f;

//当桶(BUCKET)上的节点大于这个值的时候会转换为红黑树 [JDK 1.8新增]
static final int TREEIFY_THRESHOLD = 8;

//当桶(BUCKET)上的节点小于这个值的时候会转换为链表 [JDK 1.8新增]
static final int UNTREEIFY_THRESHOLD = 6;

//当集合中的容量大于这个值,表中的桶才能进行树形化,否则桶内元素太多时会进行扩容 [JDK
static final int MIN_TREEIFY_CAPACITY = 64;

//初始化使用的数组,长度总是2的幂次方.
transient Node<K,V>[] table;

//保存缓存的entrySet.
transient Set<Map.Entry<K,V>> entrySet;

//集合中键值对的数量
transient int size;

//记录修改被修改的次数(主要用于迭代循环的时候进行判断)
transient int modCount;

//调整下一个值的大小 (容量 * 加载因子)
int threshold;

//散列表的加载因子
```

```
38 final float loadFactor;
```

字段说明：

```
1 /*
2 初始化使用的数组,长度总是2的幂次方.
3 HashMap是由数字+链表+红黑树组成的,数组指的就是table字段,后面对数组进行初始化的长度
4 */
5 transient Node<K,V>[] table;
```

```
1 /*
2 散列表的加载因子
3 这个数值只要用于衡量HashMap满的程度,计算HashMap的实时装载因子的方法是size/capacity
4 ▲注意:这里不是使用占用桶的数量去除以capacity,capacity是桶的数量,也就是table的长度
5
6 默认的负载因子是0.75,这是对于空间和时间相对平衡的选择,建议不要修改,如果空间大效率要
7 */
8 final float loadFactor;
```

```
1 /*
2 调整下一个值的大小 (容量 * 加载因子)
3
4 当(容量 * 加载因子)大于这个值,则触发resize扩容操作,调整后的HashMap容量是之前的两倍
5 */
6 int threshold;
```

三.HashMap中的构造函数

```
1 /*
2 默认无参构造函数,初始化加载因子为0.75F,这个时候并没有进行数组的初始化.
3 */
4 public HashMap() {
5     this.loadFactor = DEFAULT_LOAD_FACTOR;
6 }
```

```java
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR); //调用本类另外一个构造方法
}

/**
 * @param  initialCapacity 指定初始化容量
 * @param  loadFactor      指定加载因子(默认为0.75)
 */
public HashMap(int initialCapacity, float loadFactor) {
    //初始化容量不可以小于0,否则抛出异常.
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);

    //初始化容量大于2的30次方或者是一个非数值,则抛出异常.
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);
    //将传入的负载因子赋值给成员变量
    this.loadFactor = loadFactor;

    //通过tableSizeFor进行计算,返回一个大于等于initialCapacity的最小二次幂数值.
    this.threshold = tableSizeFor(initialCapacity);
}

//https://www.cnblogs.com/xiyixiaodao/p/14483876.html 参考此篇文章
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

四.HashMap中的确定哈希桶索引的方式

哈希表使用散列函数来确定索引的位置,散列函数设计越好,元素分部越均匀.

HashMap是数组+链表+红黑树的组合,是希望在有限个数组位置时,尽量使每个位置的元素只有一个,当用散列函数获取到索引位置的时候,马上就能知道对应位置的元素是不是想要的.

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

补:i = (table.length - 1) & hash; //补充的这一步是在添加元素的时候计算数组中索引
```

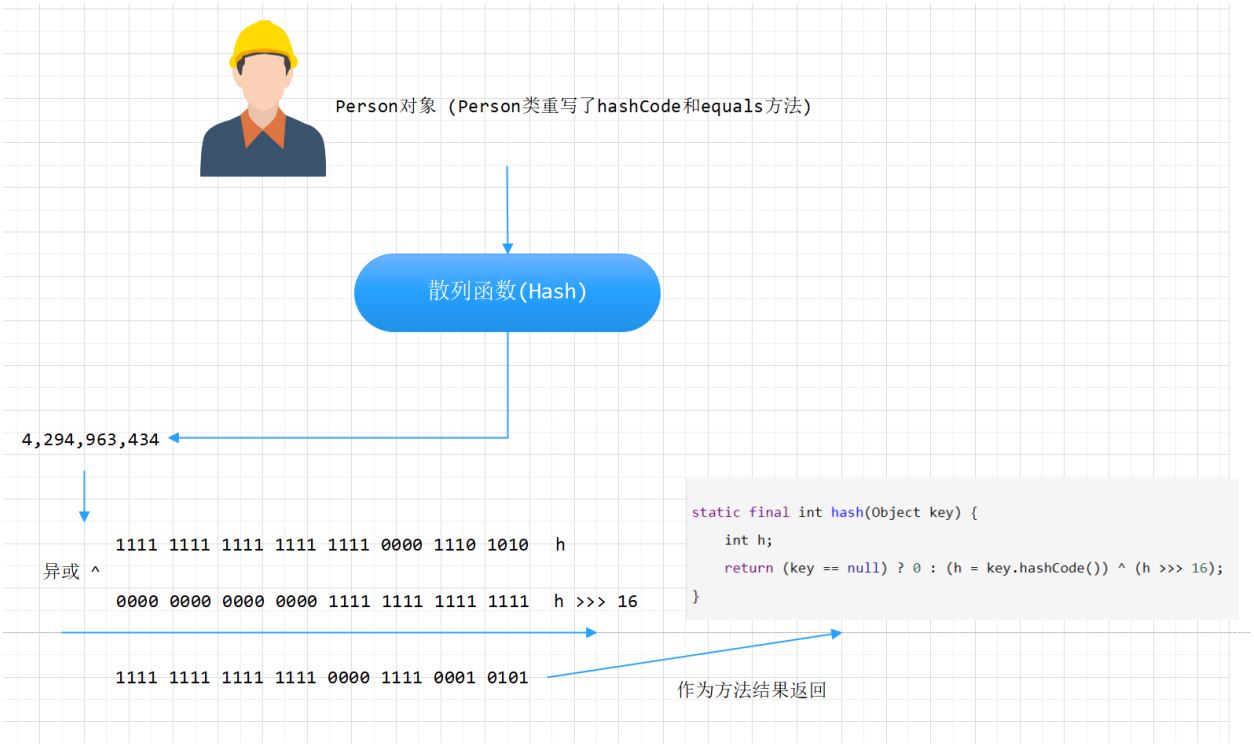
以上分为了如下几步:

1.判断本次元素是不是null值,如果是0,直接返回0作为结果.

2.获取到本次元素的哈希值并且赋值给h.

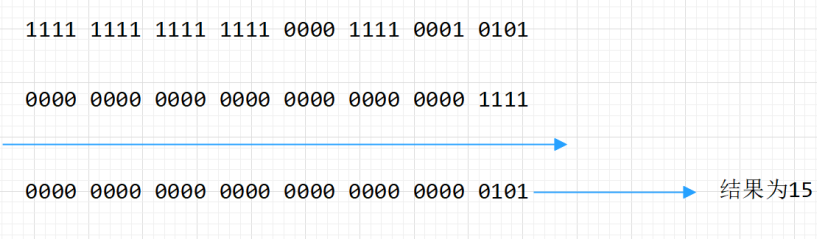
3.获取到元素哈希值中的前16位(高位)　>>>是无符号右移,相当于把前16位移动到了后16位的位置.

4.使用高位与低位进行异或运算获取到最终哈希值结果.



```
补:i = (table.length - 1) & hash; //补充的这一步是在添加元素的时候计算数组中索引的.
```

例:Map中数组长度为16



问题:为什么HashMap的长度一定是2的幂次方?

答:首先为了数组元素分部均匀,可以将hash值对数组的长度取余获取结果,但是计算机都是以二进制进行操作,取模运算相对而言开销比较大.

HashMap巧妙的使用hash & (table.length - 1)的方式来获取索引,n % 32的结果和n & (32 - 1)的结果是一样的,&比%拥有更高的效率.

length为偶数时,length-1为奇数,奇数的二进制最后一位是1,这样便保证了hash & (length-1)的最后一位可能为0,也可能为1(这取决于h的值,即&运算后的结果可能为偶数,也可能为奇数,这样便可以保证散列的均匀性。

而如果length为奇数的话,很明显length-1为偶数,它的最后一位是0,这样hash & (length-1)的最后一位肯定为0,即只能为偶数,这样任何hash值都只会被散列到数组的偶数下标位置上,这便浪费了近一半的空间.

&运算:1为true,0为false,遇0则0.

四.HashMap中的添加的源码解析

```
1  public V put(K key, V value) {
2      return putVal(hash(key), key, value, false, true); //调用了本类的另外一个pu
3  }
4  /*
5   * @param hash Key的哈希值
6   * @param key  Key的值
7   * @param value Value的值
8   * @param onlyIfAbsent 如果是true,表示不要更改现有值
9   * @param evict 如果是false表示处于创建模式
10  */
11 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
12                  boolean evict) {
13     //1.初始化哈希表底层数组tab,节点对象p,int类型的数据n和i.
14     Node<K,V>[] tab; Node<K,V> p; int n, i;
15     //2.判断当前的table是否为null或者长度为0.
16     if ((tab = table) == null || (n = tab.length) == 0)
```

```
        //3.通过resize方法进行扩容,返回的n是扩容后的数组长度.
        n = (tab = resize()).length;
    //3.这里使用到了前面说的获取Key的哈希值的最后一步与当前扩容后的数组长度进行&运算
    if ((p = tab[i = (n - 1) & hash]) == null)
        //4.如果是NULL,则直接将KEY-VALUE通过newNode方法封装为Node对象存储到对应i
        tab[i] = newNode(hash, key, value, null);
    else {
        //5.如果执行到这里说明tab[i]不为NULL.
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p; //6.如果已经有值了,而且是重复值,则直接使用新值覆盖.
        else if (p instanceof TreeNode)   //7.如果当前数组中的索引保存是红黑树,
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            //8.如果不是红黑树,则是链表.
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    //9.如果链表长度大于8,则转换为红黑树.
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                //如果KEY已经存在则直接覆盖老的VALUE值.
                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;
        }
    }
    ++modCount;
    if (++size > threshold) //如果超过最大容量,则进行扩容.
        resize();
    afterNodeInsertion(evict);
```

```
60      return null;
61  }
```

1. 判断键值对数组的table是否为null或者为空,否则执行resize()进行扩容.
2. 根据KEY的值计算哈希值得到插入的数组索引i,如果table[i]是NULL值,则直接新建节点添加(NEXT:6),如果不是NULL,则执行下一步.
3. 判断table[i]的首个元素是否和key一样,如果相同(哈希值和equals方法比较结果)直接覆盖VALUE,否则执行下一步.
4. 判断table[i]是不是一个树节点(TreeNode),如果是树节点说明当前是红黑树,如果是红黑树,则直接在书中插入键值对,否则下一步.
5. 遍历table[i],判断链表的长度是否大于8,大于8,则把链表转换为红黑树,在红黑树中执行插入操作,否则进行链表的插入操作,遍历过程中发现KEY已经存在则直接覆盖VALUE.
6. 插入成功后,判断实际存在的键值对数量是否超过最大容量threshold,如果超过则进行扩容.
7. 如果新插入的KEY不存在,则返回NULL,如果新插入的KEY存在,则返回原VALUE值.