

jdk1.7 jdk HashMap jdk concurrentHashMap

jdk1.8 jdk HashMap jdk concurrentHashMap

>1.我们都应该知道 jdk的hashmap是数组 + 链表实现的， hashmap底层会吧 key 和value封装成一个entry对象，然后进而吧这个entry对象放到数组中；

呢么问题来了 吧这个entry对象放到数组中的有一个下标 比方说 arr[1] = "xx", arr[2] = "xxx" , 1和2代表下标位，关于数组的特性就是往数组中存东西的话的确定一个下标

>我们知道ArrayList是一个基于数组实现的容器

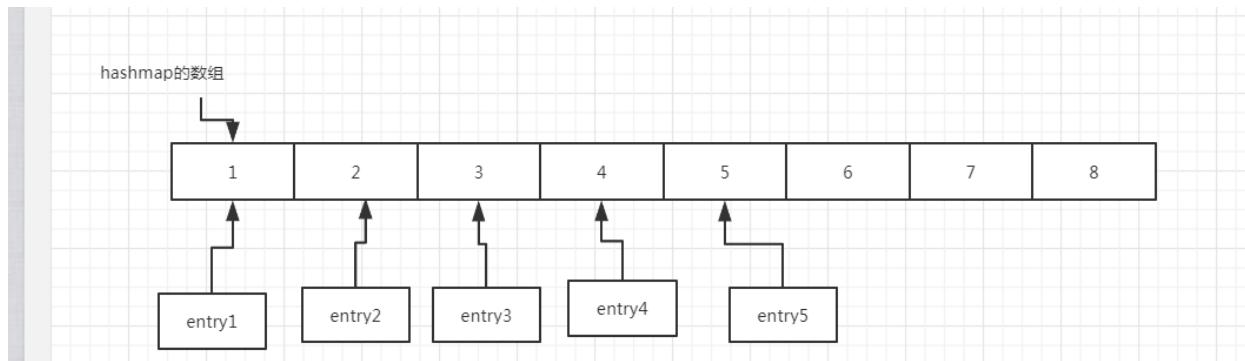
```
1 public boolean add(E e) {  
2     ensureCapacityInternal(size + 1);  
3     elementData[size++] = e;  
4     return true;  
5 }  
6 当我们调用arraylist的add方法时,实际上会调用它elementData[size++] = e;  
7 size默认为0  
8 也就是说 类似于elementData[0] = e;elementData[1] = e1;  
9 arraylist 通过这种自增的方式确定下来的下标看明白了吧，并且从0开始自增
```

呢么 我们的put方法是不是也可以这么实现 当第一个元素放到第一个，第二个entry对象来了放第二个，第三个元素来了放第三个；

这样的话 添加数据的效率比较高，因为他仅仅吧这个元素 放入到自增的下标的位置 并不需要一些其他的操作，答案肯定比不可以

因为hashmap中取数据是get(key)而不像arraylist一样arraylist.get(i),这样的话当我们get (key) 的话的从头开始遍历 找到符合我的key为止 类似于这样的 查询效率低下

图1



也就是对于这个hashmap 得要保证插入数据和查询数据都要高才行，所以说arraylist插入元素的思路在 hashmap 中是不太合适的，那么 put 方法怎么实现的呢

> 1.1

put 肯定要把 key 和 value 存到这个数组中，确定一个下标，那么这个下标怎么确定呢，根据 key 的 hashCode 值，我们可以这么想

1. key ----> 2. hashCode ---> 3. 数字 ----> 4. 计算 index

大致是这么个思路 根据 key 计算成一个 hashCode ，但是 hashCode 太大不适合做下标，所以此时可以用 (key.hashCode) % table.length 取余作为 index

但是这个 index 是有 2 个条件的 第一 index 范围是 {0-7} // 看图 1 ，第二个就是说这个 index 的取值是随机的 不能说某个数字比方说 2 永远不会出现

如果 2 个 key 的 hashCode 一样的话 就可以插入到链表中，现在这个链表是从头插入还是继续跟在尾巴呢，大致写了下加入到尾中

```
1
2 public class Node {
3
4     public Object data;
5     public Node next;
6
7     public Node(Object data, Node next) {
8         this.data = data;
9         this.next = next;
10    }
11
12    public static void main(String[] args) {
13        Node header = new Node(new Object(), null); // 头结点
14        header.next = new Node(new Object(), null); // 尾部插入
15
16    }
17 }
18 插入到连表的尾部
```

这样就是我每次的遍历整个链表找到这个链表的尾结点，从header头结点遍历，看哪个结点的尾结点为空，再加入新的结点

插入慢 我们应该插入到头部 最后将数组的下标位置的引用从旧的结点改为新的结点，这样是比较快的，大致是这个步骤的

大致模拟一下Put的过程

```
put(key,value){  
    int hashCode= key.hashCode  
    int index=hashCode%table.length  
    //----->3个参数 key value 下一个结点的属性  
    Entry entry=new Entry(key,value,null)  
    table[index]=entry  
    ----->  
  
    //第二次插入形成的node结点，吧新节点的引用地址 重新赋值  
    给table[index]//重新赋值操作  
    table[index]=new Entry(key,value,table[index] );  
}
```

好 翻看jdk1.7的hashmap源码

1.构造

```
1 public HashMap(int initialCapacity, float loadFactor) {  
2     if (initialCapacity < 0)  
3         throw new IllegalArgumentException("Illegal initial capacity: " +  
4             initialCapacity);  
5     if (initialCapacity > MAXIMUM_CAPACITY)  
6         initialCapacity = MAXIMUM_CAPACITY;  
7     if (loadFactor <= 0 || Float.isNaN(loadFactor))  
8         throw new IllegalArgumentException("Illegal load factor: " +  
9             loadFactor);  
10  
11     this.loadFactor = loadFactor;
```

```
12     threshold = initialCapacity;
13     init();
14
15 }
16 init(); 在 hashmap 里面什么都不做 但是在 linkedhashmap 中
```

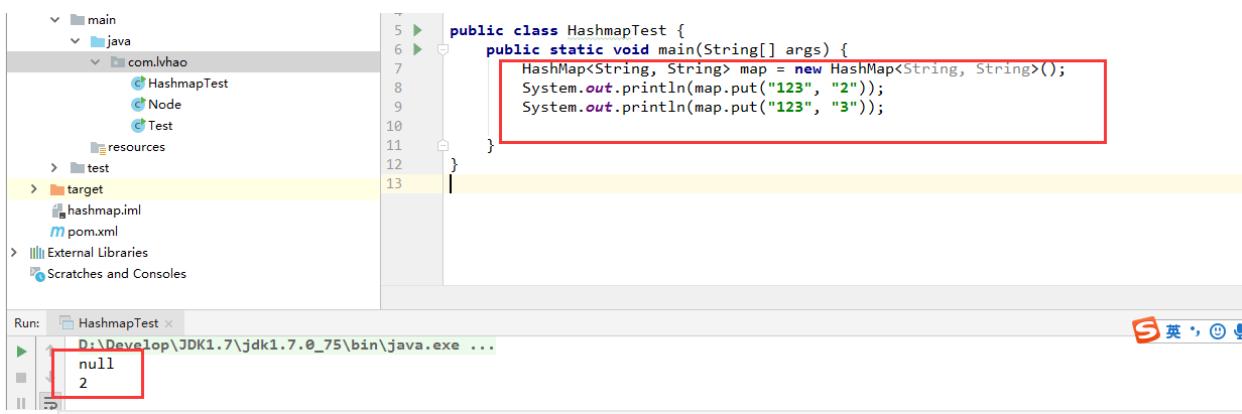
**构造方法就做了赋值操作 { this.loadFactor = loadFactor;
threshold = initialCapacity;}**

一个16,一个0.75 16就是说这个数组的初始值是16 , 0.75与扩容有关 以后再说

```
1 public V put(K key, V value) {
2     //1. table是一个空的
3     if (table == EMPTY_TABLE) {
4         //---->进行初始化数组
5         inflateTable(threshold);
6     }
7     //-----2. 此时代表key 可以为null
8     if (key == null)
9         return putForNullKey(value);
10    //---->3. 根据key计算hash值
11    int hash = hash(key);
12    //---->4.根据hash值以及数组的长度计算 该元素应该在数组中的那个位置
13    int i = indexFor(hash, table.length);
14    //---->5.返回值
15    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
16        Object k;
17        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
18            V oldValue = e.value;
19            e.value = value;
20            e.recordAccess(this);
21            return oldValue;
22        }
23    }
24
25    modCount++;
26    //---->6.增加 将该元素加入到数组结点中
27    addEntry(hash, key, value, i);
28    return null;
```

```
29 }
```

说明一下 hashMap的put是有返回值的



```
public class HashmapTest {
    public static void main(String[] args) {
        HashMap<String, String> map = new HashMap<String, String>();
        System.out.println(map.put("123", "2"));
        System.out.println(map.put("123", "3"));
    }
}
```

```
1
2 整个hashmap的put源码我该从那个地方打开局面呢
3 ----->先从这个循环
4 循环遍历列表
5 for (Entry<K,V> e = table[i]; e != null; e = e.next) {
6     Object k;
7     //->如果此时传进来key的hash 和链表中某结点hash一样 的话 在看key是否相同
8     if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
9         //----->将oldVale 返回
10        V oldValue = e.value;
11        //赋值替换 为新的value
12        e.value = value;
13        e.recordAccess(this);
14        return oldValue;
15    }
16 }
17 {其实put第一次也会遍历 只不过 返回为空}
18 也就是说 当put第二个元素的时候他会遍历这个链表的 如果发现key相同的话
19 就会吧第一次存入的key 返回回去 并且新的value 会把旧的value给覆盖
```

```
1
2
3 inflateTable(16)// 看这个初始化的方法
4 private void inflateTable(int toSize) {
5 //查找大于toSize的最小2的幂数，例如传入的toSize=9，那么capacity为16
6     int capacity = roundUpToPowerOf2(toSize);
7 }
```

```
8 threshold = (int) Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
9 table = new Entry[capacity];
10 initHashSeedAsNeeded(capacity);
11 }
12 }
```

1 //----->看啊，我们这个初始化数组的时候传入的是16 但是真正的数组个数并不是我们传入的 而是capacity

2 我们点进去看这个查找大于toSize的最小2的幂数

```
3 roundUpToPowerOf2 (number)
4 private static int roundUpToPowerOf2(int number) {
5 // assert number >= 0 : "number must be non-negative";
6 return number >= MAXIMUM_CAPACITY
7 ? MAXIMUM_CAPACITY
8 : (number > 1) ? Integer.highestOneBit((number - 1) << 1) : 1;
9 }
```

1 //----->我们先看下他

```
2 Integer.highestOneBit(int number) //我们可以写个方法测下
3 Integer.highestOneBit(8)--->8
4 Integer.highestOneBit(10)--->8
5 Integer.highestOneBit(17)--->16
6 找到小于等于2的幂次方数
7
```

```
1 public static int highestOneBit(int i) {
2 // HD, Figure 3-1
3 i |= (i >> 1);
4 i |= (i >> 2);
5 i |= (i >> 4);
6 i |= (i >> 8);
7 i |= (i >> 16);
8 return i - (i >>> 1);
9 }
```

10 打个比方吧 highestOneBit(17)

```
11 i
12 17 0001 0001 17
13 >>1 0000 1000
14 | 0001 1001 |运算 当两边操作数的位有一边为1时，结果为1，否则为0。如
1100|1010=1110
```

```

15 >>2 0000 0110
16 | 0001 1111
17 >>4 0000 0001
18 | 0001 1111
19 >>8 0000 0000
20 | 0001 1111
21 ----->我们的数字小基本上就是8个一轮了
22
23 i 0001 1111
24 i >>> 1 0000 1111
25 0001 0000
26 ----->这样就能返回16了
27
28 -----
----->----->
29 呢么 我们究竟为什么这么做呢，假设我们一个数字是 001* *****
30 001* ****->这是我之前的数字
31 >>1 0001 ****
32 | 0011 **** -----> 我右移一个位的本质是将我这个数字的低位置变为1
33 >>2 0000 11**
34 | 0011 11** -----> 我右移2个位的本质是将我2个低位置变为1
35 >>4 0000 1111
36 | 0011 1111
37 ----->到最后就会把我 001* ****的所有 * 都变为1 ，
38 呢么他变为1之后呢 他继续右移动
39 0011 1111
40 0001 0000
41 0010 0000 // 此时就能返回想要的结果
42 因为我们 001* ****->我要达到一个结果就是 0010 0000
43 呢么为什么要移动这么多位啊 因为 int 类型 是4个字节 32个bit位
44

```

```

1 highestOneBit()//----->这个是找到小于等于2的幂次方数
2 我们现在找到大于等于2的幂次方数字 他是对这个数字进行处理 然后再去调用这个方法
3
4 17 ----- highestOneBit()---->16
5 左移翻倍 17*2 34----->32
6 但是我们现在想要的是 17----->32
7 此时为啥要-1 呢 当我们想一个数字 16
8 16----- Integer.highestOneBit----->32

```

```
9 但是我们想要的是 16啊  
10 15-----> highestOneBit() 16
```

呢么初始化一个数组为什么一定是2的 幂次方数字

```
1 //----->我们跳出来 看这个  
2 根据hash以及对应数组的length找到对应的数组下标的方法  
3 int i = indexFor(hash, table.length);;  
4  
5 return h & (length-1); //----->他返回这个  
6 这个数组下标 有2个特点 第一个就是说平均分配 或者说是随机分配在数组的下标上  
7 第二个就是说 我们此时这个index 的值是[0,15)  
8 这个h是根据K生成的hash值 我们姑且认为他是  
9 16 0001 0000  
10 15 0000 1111  
11 h: 0101 0001 ----->随便写的啊  
12 & 0000 0101 //&---->都为1 才为1  
13 保证了 最大就是15 {0000,1111]， 高位都是0 只有低4位  
14
```

int i = indexFor(hash, table.length);

根据hash值算在数组中的位置 每一个位置都可能为1或者0

真正参加计算的是15

因为你只有数组容量是2的幂次方数才可以达到 你的indexFor中一个根据hash获取的数组的index

保证了2点 第一点就是 平均 第二点就是说 下标不越界， 此时不用取余是因为位运算方比取余块

```
1 int hash = hash(key);
```

//----->我们看下这个方法 他这个方法没有直接用hashcode 其实 他这个方法是将 hashcode变得更加散列

```
1 16 0001 0000  
2 15 0000 1111  
3 h: 0101 0001 ----->随便写的啊
```

```
4 & 0000 0101 //&-->都为1 才为1
5
6 我们一个hashcode 是32 位的 你这个h 比方说前28bit不管都怎么改变的话 我后四位
只要保证一样的话
7 比方说 h: 0101 0001 和 h: 1111 0001 这2个都代表同一个index 这样算下来 冲突比
较大
8 所以说 如果说高位的变化也可以影响这个Index 的变化 也就是说 我高位28位能够参与
你算数组下标的过程中
9 {现在的问题就是你高28位没有参加到这个计算数组的运算中来}
10 怎么做呢 呢就是右移以及亦或运算
11 这个hash函数，他这个方法是将 hashcode变得更加散列
```

1 当 k=null时候

```
1 putForNullKey(value); 最后说这个方法
2
3 private V putForNullKey(V value) {
4 // 遍历数组的0号位置
5 for (Entry<K,V> e = table[0]; e != null; e = e.next) {
6 // 是不是已经有key=null了 和刚刚一样如果有的话 我替换值
7 if (e.key == null) {
8 V oldValue = e.value;
9 e.value = value;
10 e.recordAccess(this);
11 return oldValue;
12 }
13 }
14 modCount++;
15 addEntry(0, null, value, 0); // 存在 hashmap 的第0个位置
16 return null;
17 }
```

```
1 addEntry(hash, key, value, i); //我们看下这个方法 这个方法就是 key value i
{数组的下标} 放入到数组中
```

```
1 void addEntry(int hash, K key, V value, int bucketIndex) {
2 if ((size >= threshold) && (null != table[bucketIndex])) {
3 resize(2 * table.length);
4 hash = (null != key) ? hash(key) : 0;
5 bucketIndex = indexFor(hash, table.length);
```

```
6 }
7
8     createEntry(hash, key, value, bucketIndex); //----->先看这个 先不用
9     看扩容
9 }
```

```
1 void createEntry(int hash, K key, V value, int bucketIndex) {
2     Entry<K,V> e = table[bucketIndex];
3     table[bucketIndex] = new Entry<>(hash, key, value, e);
4     size++;
5 }
6 // 这不就是 刚开始的想的么
7 插入形成的node结点，吧新节点的引用地址 重新赋值给table[index]//重新赋值操作 此时使用的是头插法
```

```
1 二进制总结
2 | 2边有1个为1 的就是1 可以一直右移动 然后 再 | 让低阶为0
3 & 2边都为1 才能是1 可以判断 2的倍数
4 因为2的幂次方只有1个是1
5 (retries & 1) == 0 是偶数 如果是1的话基数
```