**HashMap学习**

数据结构中有数组和链表来实现对数据的存储。但数组存储空间是连续的， 空间复杂度较大，寻址容易，插入和删除困难。链表正好相反，存储空间分散，插入和删除容易，寻址困哪。HashMap中使用的数据结构正好综合了数组与链表的优点。数组中的每一个元素存储的是一个链表的头结点。如下图：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | | | 96 |  | | |  | | | | 396 | |  |
| 1 |
| 2 |
| 3 |  | 363 | | | |  | | |  | | 19 | |  | | |  | 39 |  |
| 4 |
| 5 |  | | 35 | | | |  | | |

**HashMap中的一些变量：**

//当初始化HashMap时未指定容量，HashMap的默认容量为16

**static** **final** **int** ***DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4;

//HashMap的默认最大容量

**static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 30;

//默认负载因子

**static** **final** **float** ***DEFAULT\_LOAD\_FACTOR*** = 0.75f;

//当一个链表的碰撞次数大于该属性时 将会使用red-black树来存储

**static** **final** **int** ***TREEIFY\_THRESHOLD*** = 8;

//HashMap中用来存储各个链表头结点的数组

**transient** Node<K,V>[] table;

**transient** Set<Map.Entry<K,V>> entrySet;

//HashMap中key-value的对数

**transient** **int** size;

//HashMap修改结构的次数

**transient** **int** modCount;

// 极限值 当size大于极限值时会扩容，极限值由负载因子和初始容量决定

**int** threshold;

HashMap的构造函数：

//指定初始化容量时 极限值为大于初始化容量的2的n次幂

**public** HashMap(**int** initialCapacity, **float** loadFactor) {}

**public** HashMap(**int** initialCapacity) { //会调用第一个 }

**public** HashMap() { //参数为空时 负载因子为0.75 初始容量大小为16

**this**.loadFactor = ***DEFAULT\_LOAD\_FACTOR***; }

**HashMap的数据结构：Node<K,V>**

**HashMap使用Node静态内部类来存储节点**

**static** **class** Node<K,V> **implements** Map.Entry<K,V> {

**final** **int** hash; //相同链表的hash值相同 但key不一定相同

**final** K key; //对应的key

V value; //对应的值

Node<K,V> next; //该节点的下一个节点 如果为null，该节点//为链表的尾节点

Node(**int** hash, K key, V value, Node<K,V> next) {

**this**.hash = hash;

**this**.key = key;

**this**.value = value;

**this**.next = next;

}

**public** **final** K getKey() { **return** key; }

**public** **final** V getValue() { **return** value; }

**public** **final** String toString() { **return** key + "=" + value; }

使用Node<K,V>[] table来存储各个链表或者树的头结点

**HashMap如何是将元素存储的(put方法)以及怎样解决哈希冲突的**：

**public** V put(K key, V value) {

//通过key作为参数得到一个hash码 这个hash码与table的长度决定了该元素存储于哪个位置的链表

**return** putVal(*hash*(key), key, value, **false**, **true**);

}

**static** **final** **int** hash(Object key) {

**int** h;

**return** (key == **null**) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

Putval方法：插入元素并返回key对应的值，修改返回旧值，插入返回null

**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)

//这个if循环只会在第一次调用put方法时才会进入

n = (tab = resize()).length;

**if** ((p = tab[i = (n - 1) & hash]) == **null**)

//通过key计算得打的hash值与table长度得到链表位置

//这种情况下该位置没有链表 此时这个元素就作为头结点存储

tab[i] = newNode(hash, key, value, **null**);

**else** {

//发生哈希冲入时会执行以下代码

Node<K,V> e; K k;

**if** (p.hash == hash &&((k = p.key) == key || (key != **null** && key.equals(k))))

e = p; //此时并未添加元素 而是修改key对应的值

**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**);

//冲突次数大于等于8 不在使用链表存储 而是使用//red-black树来进行存储

**if** (binCount >= ***TREEIFY\_THRESHOLD*** - 1) treeifyBin(tab, hash);

**break**;

}

**if** (e.hash == hash &&((k = e.key) == key || (key != **null** && key.equals(k)))) //这个if是插入元素与链表中的元素一致，此时应该修改key对应的值

**break**;

p = e;

}

}

**if** (e != **null**) { // 修改key对应的值

V oldValue = e.value;

//因为参数onlyIfAbsent为false，会修改值

**if** (!onlyIfAbsent || oldValue == **null**)

e.value = value;

afterNodeAccess(e);

**return** oldValue; //返回旧值

}

}

//代码运行到这儿表示插入数据

++modCount; //HashMap结构修改 变量值自增1

**if** (++size > threshold) //检查个数是否大于极限值

resize(); //扩容

afterNodeInsertion(evict);

**return** **null**; //插入新数据返回null

}

**HashMap的get方法**：

**public** V get(Object key) {

Node<K,V> e;

//通过key计算hash值得到存储该元素的链表

**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;

**if** ((tab = table) != **null** && (n = tab.length) > 0 &&

(first = tab[(n - 1) & hash]) != **null**) {

**if** (first.hash == hash &&((k = first.key) == key || (key != **null** && key.equals(k))))

**return** first;//如果头结点符合 就返回头结点

**if** ((e = first.next) != **null**) {

**if** (first **instanceof** TreeNode)

**return** ((TreeNode<K,V>)first).getTreeNode(hash, key);

**do** {//遍历节点的key与参数key不仅hash码要相同

//还要equals方法为true

**if** (e.hash == hash &&

((k = e.key) == key || (key != **null** && key.equals(k))))

**return** e;//循环到链表的尾部位置

} **while** ((e = e.next) != **null**);

}

}

**return** **null**;//未找到就返回null值

}

Containskey方法 就是调用getNode方法来判断key对应的节点是否为空来判断map中是否有key值

**Remove方法**

**public** V remove(Object key) {

Node<K,V> e;

**return** (e = removeNode(*hash*(key), key, **null**, **false**, **true**)) == **null** ?

**null** : e.value;

}

**final** Node<K,V> removeNode(**int** hash, Object key, Object value,**boolean** matchValue, **boolean** movable) {

Node<K,V>[] tab; Node<K,V> p; **int** n, index;

**if** ((tab = table) != **null** && (n = tab.length) > 0 &&(p = tab[index = (n - 1) & hash]) != **null**) {

Node<K,V> node = **null**, e; K k; V v;

**if** (p.hash == hash &&

((k = p.key) == key || (key != **null** && key.equals(k))))

node = p;

**else** **if** ((e = p.next) != **null**) {

**if** (p **instanceof** TreeNode)

node = ((TreeNode<K,V>)p).getTreeNode(hash, key);

**else** {

**do** {

**if** (e.hash == hash &&

((k = e.key) == key ||

(key != **null** && key.equals(k)))) {

node = e;

**break**;

}

p = e;

} **while** ((e = e.next) != **null**);

}

}

//上面代码是为了找出key对应的Node以及该Node的前一个节点p

**if** (node != **null** && (!matchValue || (v = node.value) == value ||//只要Node不为空就会执行以下代码

(value != **null** && value.equals(v)))) {

**if** (node **instanceof** TreeNode)

((TreeNode<K,V>)node).removeTreeNode(**this**, tab, movable);

**else** **if** (node == p)//该种情况为链表只有一个头结点

tab[index] = node.next;

**else**

//修改p的下一个节点为Node的下一个几点

p.next = node.next; ++modCount;

--size;

afterNodeRemoval(node);

**return** node;

}

}

**return** **null**;

}

HashMap resize方法在put方法插入元素后根据size与threshold比较有可能会调用，这个方法主要用来扩充table数组的长度。

**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; //极限值变为原来的2倍

}

**else** **if** (oldThr > 0)

newCap = oldThr; //新容量为旧极限值

**else** {

//调用无参构造时第一次调用resize会进入这块代码

newCap = ***DEFAULT\_INITIAL\_CAPACITY***;

newThr = (**int**)(***DEFAULT\_LOAD\_FACTOR*** \* ***DEFAULT\_INITIAL\_CAPACITY***);//极限值由容量与负载因子得到

}

**if** (newThr == 0) {

//指定容量时会执行这块代码

**float** ft = (**float**)newCap \* loadFactor;

newThr = (newCap < ***MAXIMUM\_CAPACITY*** && ft < (**float**)***MAXIMUM\_CAPACITY*** ?

(**int**)ft : Integer.***MAX\_VALUE***);

}

//无论是否指定初始容量，最终容量会为2的n次方幂

threshold = newThr;

@SuppressWarnings({"rawtypes","unchecked"})

Node<K,V>[] newTab = (Node<K,V>[])**new** Node[newCap];

table = newTab;//扩容后的数组 此时没有元素

**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**)

//通过hash值与新数组的长度计算得出在新数组的位置

newTab[e.hash & (newCap - 1)] = e;

**else** **if** (e **instanceof** TreeNode)

((TreeNode<K,V>)e).split(**this**, newTab, j, oldCap);

**else** { // 确认链表头结点在新数组中的位置是//否改变

Node<K,V> loHead = **null**, loTail = **null**;

Node<K,V> hiHead = **null**, hiTail = **null**;

Node<K,V> next;

**do** {

next = e.next;

//原索引

**if** ((e.hash & oldCap) == 0) {

**if** (loTail == **null**)

loHead = e;

**else**

loTail.next = e;

loTail = e;

}

//原索引+oldCap

**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 迭代：

迭代HashMap的key最终会用到HashIterator类

HashIterator() {

expectedModCount = modCount;

Node<K,V>[] t = table;

current = next = **null**;

index = 0;

**if** (t != **null** && size >0){

//如果迭代的HashMap中key-value对不为0

//找出数组中存放链表最小的下标，并将链表的头结点赋给next

**do** {} **while** (index < t.length && (next = t[index++]) == **null**);

}

}

**public** **final** **boolean** hasNext() {

**return** next != **null**;

}

**final** Node<K,V> nextNode() {

Node<K,V>[] t;

Node<K,V> e = next;

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

**if** (e == **null**)

**throw** **new** NoSuchElementException();

**if** ((next = (current = e).next) == **null** && (t = table) != **null**) {

//如果遍历完一个链表 寻找数组中下标大于index存放链表最小的下标，

//并将链表的头结点赋给next

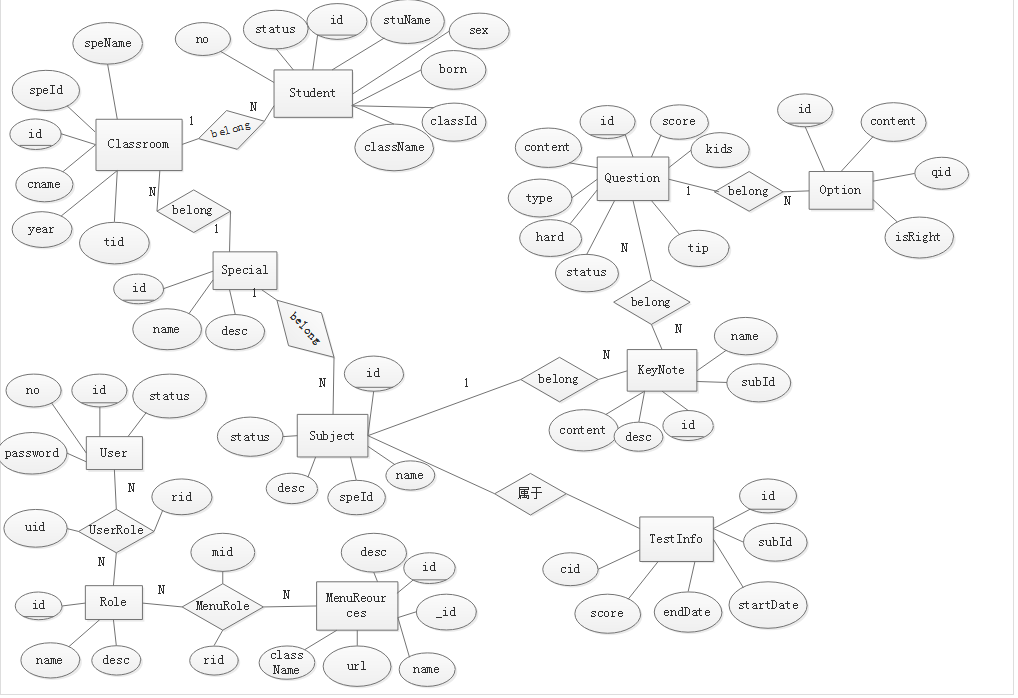
**do** {} **while** (index < t.length && (next = t[index++]) == **null**);

}

**return** e;

}

还有关于TreeNode这个内部类代码现在还没看懂.



ER图没画完 后面画不来了