**1.Hashtable与HashMap的区别**

|  |
| --- |
| \* Hashtable存储的内容是键值对(key-value)映射，其底层实现是一个Entry数组+链表；  \* Hashtable和HashMap一样也是散列表，存储元素也是键值对；  \* HashMap允许key和value都为null，而Hashtable都不能为null，Hashtable中的映射不是有序的；  \* Hashtable和HashMap扩容的方法不一样，Hashtable中数组默认大小11，扩容方式是 old\*2+1。  \* HashMap中数组的默认大小是16，而且一定是2的指数，增加为原来的2倍。  \* Hashtable继承于Dictionary类（Dictionary类声明了操作键值对的接口方法），实现Map接口（定义键值对接口）；  \* Hashtable大部分类用synchronized修饰，证明Hashtable是线程安全的。 |

**2.属性**

|  |
| --- |
| **每个entry都是单向链表的表头**  **private** **transient** Entry<?,?>[] table;  **当前表中entry数量，如果超过了阈值，就会扩容，调用rehash方法**  **private** **transient** **int** count;  **rehash阈值**  **private** **int** threshold;  **负载因子**  **private** **float** loadFactor;  用来实现"fail-fast"机制的（也就是快速失败）。所谓快速失败就是在并发集合中，其进行  \* 迭代操作时，若有其他线程对其进行结构性的修改，这时迭代器会立马感知到，并且立即抛出  \* ConcurrentModificationException异常  **private** **transient** **int** modCount = 0; |

**3.构造器**

|  |
| --- |
| **private static final int MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8;**  **如果初始化容量小于0，负载因子小于=0或者非小数，则抛异常，如果初始化容量为0，设置为1，给负载因子赋值，以初始化容量初始化table数组，阈值取（初始容量\*负载因子，最大数组长度+1）的最小值**  **public** Hashtable(**int** initialCapacity, **float** loadFactor) {  **if** (initialCapacity < 0)  **throw** **new** IllegalArgumentException("Illegal Capacity: "+  initialCapacity);  **if** (loadFactor <= 0 || Float.*isNaN*(loadFactor))  **throw** **new** IllegalArgumentException("Illegal Load: "+loadFactor);  **if** (initialCapacity==0)  initialCapacity = 1;  **this**.loadFactor = loadFactor;  table = **new** Entry<?,?>[initialCapacity];  threshold = (**int**)Math.*min*(initialCapacity \* loadFactor, ***MAX\_ARRAY\_SIZE*** + 1);  }  设定初始容量，负载因子默认0.75  **public** Hashtable(**int** initialCapacity) {  **this**(initialCapacity, 0.75f);  }  默认容量11，负载因子为0.75  **public** Hashtable() {  **this**(11, 0.75f);  }  通过比较集合的长度\*2与11的大小，取最大值作为容量，负载因子默认0.75f，然后调用putall进行添加，后面介绍  **public** Hashtable(Map<? **extends** K, ? **extends** V> t) {  **this**(Math.*max*(2\*t.size(), 11), 0.75f);  putAll(t);  } |
|  |

**4.查找方法**

|  |
| --- |
| **双重循环，从后开始遍历，遍历table头结点，然后从每个头结点开始查找value相等的元素返回布尔值**  **public** **synchronized** **boolean** contains(Object value) {  **if** (value == **null**) {  **throw** **new** NullPointerException();  }  Entry<?,?> tab[] = table;  **for** (**int** i = tab.length ; i-- > 0 ;) {  **for** (Entry<?,?> e = tab[i] ; e != **null** ; e = e.next) {  **if** (e.value.equals(value)) {  **return** **true**;  }  }  }  **return** **false**;  }  --------------------------------------------------------------  关键在于一个对象的 HashCode可以为负数，这样操作后可以保证它为一个正整数0x7FFFFFFF is 0111 1111 1111 1111 1111 1111 1111 1111 : all 1 except the sign bit.(hash & 0x7FFFFFFF) 将会得到一个正整数，因为你的hash是要作为数组的index的，这样可以避免出现下标为负数而出现异常  **public** **synchronized** **boolean** containsKey(Object key) {  Entry<?,?> tab[] = table;  **int** hash = key.hashCode();  根据key计算hash值，然后与0x7FFFFFFF保证下标正数，此处与hashmap不同，hashmap的求下标，hash&length – 1；作用一致，因为hashmap的长度就是2^n次方  **int** index = (hash & 0x7FFFFFFF) % tab.length;  **for** (Entry<?,?> e = tab[index] ; e != **null** ; e = e.next) {  **if** ((e.hash == hash) && e.key.equals(key)) {  **return** **true**;  }  }  **return** **false**;  }  内部调用contains（value）  **public** **boolean** containsValue(Object value) {  **return** contains(value);  }  这里的判断条件都是hash值相等，key值也相等  **public** **synchronized** V get(Object key) {  Entry<?,?> tab[] = table;  **int** hash = key.hashCode();  **int** index = (hash & 0x7FFFFFFF) % tab.length;  **for** (Entry<?,?> e = tab[index] ; e != **null** ; e = e.next) {  **if** ((e.hash == hash) && e.key.equals(key)) {  **return** (V)e.value;  }  }  **return** **null**;  } |

**5.扩容**

|  |
| --- |
| **private** **static** **final** **int** ***MAX\_ARRAY\_SIZE*** = Integer.***MAX\_VALUE*** - 8;  **当hashtable 中的count超过阈值（容量\*装载因子），会调用rehash增加容量，重新计算每个键值对的hashcode，newCapacity=2\*old + 1；，根据新容量\*装载因子更新阈值**  **protected** **void** rehash() {  **int** oldCapacity = table.length;  Entry<?,?>[] oldMap = table;  **int** newCapacity = (oldCapacity << 1) + 1;  最大只能到***MAX\_ARRAY\_SIZE，不要超过***  **if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0) {  **if** (oldCapacity == ***MAX\_ARRAY\_SIZE***)  // Keep running with MAX\_ARRAY\_SIZE buckets  **return**;  newCapacity = ***MAX\_ARRAY\_SIZE***;  }  Entry<?,?>[] newMap = **new** Entry<?,?>[newCapacity];  modCount++;  threshold = (**int**)Math.*min*(newCapacity \* loadFactor, ***MAX\_ARRAY\_SIZE*** + 1);  table = newMap;  重点是复制过程：外层循环从数组尾部开始，内存循环就是遍历，每个头结点相连的链表  从头结点开始，将原结点复制给e，然后old继续指向下一个，然后根据e的hash计算新的下标，将e的next指向新数组头结点，然后，将e作为新的数组头结点，这样一个一个往头结点插入。  **for** (**int** i = oldCapacity ; i-- > 0 ;) {  **for** (Entry<K,V> old = (Entry<K,V>)oldMap[i] ; old != **null** ; ) {  Entry<K,V> e = old;  old = old.next;  **int** index = (e.hash & 0x7FFFFFFF) % newCapacity;  e.next = (Entry<K,V>)newMap[index];  newMap[index] = e;  }  }  } |

**6.增加元素**

|  |
| --- |
| **Count超过阈值时，进行扩容，扩容后，根据key值对应的hashcode计算hash值，确定下标，得到该下标的头结点。直接通过entry构建一个指向旧头结点的entry对象，然后将其作为新的下标，即都是往头结点上插入数据**  **private** **void** addEntry(**int** hash, K key, V value, **int** index) {  modCount++;  Entry<?,?> tab[] = table;  **if** (count >= threshold) {  rehash();  tab = table;  hash = key.hashCode();  index = (hash & 0x7FFFFFFF) % tab.length;  }  // Creates the new entry.  @SuppressWarnings("unchecked")  Entry<K,V> e = (Entry<K,V>) tab[index];  tab[index] = **new** Entry<>(hash, key, value, e);  count++;  }  Key和value都不能为null，下面手动处理value为null的情况，而key如果为null那么调用key.hashcode()方法就会抛出空指针异常；  根据key求出hash值，再求出下标，在当前下标的链表下找key和hash值都想等的元素，如果找到了替换新值，返回旧value，否则就调用addEntry  **public** **synchronized** V put(K key, V value) {  // Make sure the value is not null  **if** (value == **null**) {  **throw** **new** NullPointerException();  }  // Makes sure the key is not already in the hashtable.  Entry<?,?> tab[] = table;  **int** hash = key.hashCode();  **int** index = (hash & 0x7FFFFFFF) % tab.length;  @SuppressWarnings("unchecked")  Entry<K,V> entry = (Entry<K,V>)tab[index];  **for**(; entry != **null** ; entry = entry.next) {  **if** ((entry.hash == hash) && entry.key.equals(key)) {  V old = entry.value;  entry.value = value;  **return** old;  }  }  addEntry(hash, key, value, index);  **return** **null**;  }  底层就是调用put方法  **public** **synchronized** **void** putAll(Map<? **extends** K, ? **extends** V> t) {  **for** (Map.Entry<? **extends** K, ? **extends** V> e : t.entrySet())  put(e.getKey(), e.getValue());  }  如果key值存在且oldvalue不为null，返回oldvalue，，如果key存在oldvalue为null，设置新值，如果都不存在就调用add  **public** **synchronized** V putIfAbsent(K key, V value) {  Objects.*requireNonNull*(value);  // Makes sure the key is not already in the hashtable.  Entry<?,?> tab[] = table;  **int** hash = key.hashCode();  **int** index = (hash & 0x7FFFFFFF) % tab.length;  @SuppressWarnings("unchecked")  Entry<K,V> entry = (Entry<K,V>)tab[index];  **for** (; entry != **null**; entry = entry.next) {  **if** ((entry.hash == hash) && entry.key.equals(key)) {  V old = entry.value;  **if** (old == **null**) {  entry.value = value;  }  **return** old;  }  }  addEntry(hash, key, value, index);  **return** **null**;  } |

**7.删除元素**

|  |
| --- |
| 找到头节点，记录上一个访问的节点prev和当前节点e，如果e的hash值和key都相等，判断prev是否为null，如果是，说明e是头结点，直接将头结点指向e.next;如果不是，就让prev.next指向e.next;  **public** **synchronized** V remove(Object key) {  Entry<?,?> tab[] = table;  **int** hash = key.hashCode();  **int** index = (hash & 0x7FFFFFFF) % tab.length;  @SuppressWarnings("unchecked")  Entry<K,V> e = (Entry<K,V>)tab[index];  **for**(Entry<K,V> prev = **null** ; e != **null** ; prev = e, e = e.next) {  **if** ((e.hash == hash) && e.key.equals(key)) {  modCount++;  **if** (prev != **null**) {  prev.next = e.next;  } **else** {  tab[index] = e.next;  }  count--;  V oldValue = e.value;  e.value = **null**;  **return** oldValue;  }  }  **return** **null**;  }  与上面不同的是，这个要key，hash，value都相等才能删除  **public** **synchronized** **boolean** remove(Object key, Object value) {  Objects.*requireNonNull*(value);  Entry<?,?> tab[] = table;  **int** hash = key.hashCode();  **int** index = (hash & 0x7FFFFFFF) % tab.length;  @SuppressWarnings("unchecked")  Entry<K,V> e = (Entry<K,V>)tab[index];  **for** (Entry<K,V> prev = **null**; e != **null**; prev = e, e = e.next) {  **if** ((e.hash == hash) && e.key.equals(key) && e.value.equals(value)) {  modCount++;  **if** (prev != **null**) {  prev.next = e.next;  } **else** {  tab[index] = e.next;  }  count--;  e.value = **null**;  **return** **true**;  }  }  **return** **false**;  } |