# HashTable

Hashtable实现了Map接口，用来映射key-value；key和value都不可以为null；为了从散列表成功存储和检索对象，用作键的对象必须实现hashCode方法和equals方法；Hashtable有2个影响性能的因数：初始容量和加载因子；通常默认的加载因子是0.75，这个值是在时间和空间上的一个折中，加载因子更高会节省空间，但是会增加访问和操作的时间成本；

初始容量控是使用空间和扩容操作之间的折中，扩容是耗时的；如果容量大于MAX\_ARRAY\_SIZE（int MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8）定义的最大数组长度后，就不进行扩容了，设置初始容量太高可能浪费空间

Hashtable在修改结构的方法上都使用synchronzed,用来保证操作的线程安全性

Hashtable内部结构就是数组+链表，（ hashmap：数组+链表+红黑树）

## Hashtable变量

**private** **transient** Entry<?,?>[] table;存储key-value节点的数组

**private** **transient** **int** count;存放节点的个数

**private** **int** threshold;扩容阈值

**private** **float** loadFactor;加载因子

**private** **transient** **int** modCount = 0;修改次数

## 节点对象

存放key,value,key的hash值，指向下一个节点的引用

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

**final** **int** hash;

**final** K key;

V value;

Entry<K,V> next;

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

**this**.hash = hash;

**this**.key = key;

**this**.value = value;

**this**.next = next;

}

}

## 构造函数

指定初始数组容量和加载因子，如果构造函数中没有指定，那么默认容量就是11，默认加载因子就是0.75

在初始化过程中对数组table就进行初始化

**public** Hashtable() {

**this**(11, 0.75f);

}

**public** Hashtable(**int** initialCapacity) {

**this**(initialCapacity, 0.75f);

}

**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);

}

构造函数参数是Map接口对象，根据传入的对象计算出初始容量，使用默认的加载因子创建Hashtable对象

然后调用putAll方法将参数中的值存入当前的数组中

**public** Hashtable(Map<? **extends** K, ? **extends** V> t) {

**this**(Math.*max*(2\*t.size(), 11), 0.75f);

putAll(t);

}

## Put方法

Put方法，将指定的key和value存放到Map中（如果key存在就返回原value，key不存在返回null）

1. 判断value是否为null，如果为null直接抛出异常，不为null进行之后的操作
2. 调用key的hashCode方法，这里间接判断的key是否为null,如果key为null，那么就抛出空指针异常，结束添加操作；key不为null，返回has值进行之后的操作
3. 通过key的hash值计算出存放在数组的位置（hash值和0x7FFFFFFF（2147483647）进行与运算然后和数组长度进行取模，结果就是位置(hash & 0x7FFFFFFF) % tab.length）
4. 计算出节点位置后，取出对应位置的节点链表对象（Entry<K,V> entry = (Entry<K,V>)tab[index];），

遍历链表的节点，判断key是否已经存在，存在的话就将value替换原value即可，返回原value

Key不存在，那么就在创建一个新的节点，让这个新建的节点的next指向当前节点的链表对象；最后将新建的这个节点赋值给数组当前的位置（tab[index] = **new** Entry<>(hash, key, value, e);

1. 判断是否扩容

当前节点数量不超过扩容阈值，不进行扩容

超过就进行扩容，rehash()方法创建新的数组，将原数组上的内容转移到新的数组上

5.1计算扩容后数组容量

新容量等于原容量扩大2倍在+1，判断新容量是否大于MAX\_ARRAY\_SIZE；大于时在判断原数组容量是否等于MAX\_ARRAY\_SIZE，如果等于，那么就是原数组已经到达了最大数组容量，不进行扩容了，直接添加

5.2要进行扩容，

创建新数组，计算出新的扩容阈值

遍历原数组不为null的节点，用hash值重新计算出在新数组上的下标位置（**int** index = (e.hash & 0x7FFFFFFF) % newCapacity;），然后在新数组上对应位置赋值

**protected** **void** rehash() {

**int** oldCapacity = table.length;

Entry<?,?>[] oldMap = table;

**int** newCapacity = (oldCapacity << 1) + 1;

**if** (newCapacity - *MAX\_ARRAY\_SIZE* > 0) {

**if** (oldCapacity == *MAX\_ARRAY\_SIZE*)

**return**;

newCapacity = *MAX\_ARRAY\_SIZE*;

}

Entry<?,?>[] newMap = **new** Entry<?,?>[newCapacity];

modCount++;

threshold = (**int**)Math.*min*(newCapacity \* loadFactor, *MAX\_ARRAY\_SIZE* + 1);

table = newMap;

**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];这步是什么意思

newMap[index] = e;

}

}

}

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

**if** (value == **null**) {

**throw** **new** NullPointerException();

}

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**;

}

**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;

}

Entry<K,V> e = (Entry<K,V>) tab[index];

tab[index] = **new** Entry<>(hash, key, value, e);

count++;

}

## 其他方法

**Values()方法返回Hashtable中所有value的集合**

通过Collections的synchronizedCollection方法创建线程安全的集合对象ValueCollection

**public** Collection<V> values() {

**if** (values==**null**)

values = Collections.*synchronizedCollection*(**new** ValueCollection(),**this**);

**return** values;

}

ValueCollection类，继承AbstractCollection抽象类，重写了iterator方法返回迭代对象

Iterator方法内部调用getIterator方法来创建迭代对象；当hashtable中还没有节点数量为0时，创建一个空的迭代对象（EmptyIterator），不为0时，创建Enumerator对象；Enumerator实现了Iterator节点，重写hasNext,next方法来遍历所有节点的value；

EmptyIterator类的next方法和hasNext方法

Next方法内部就是调用nextElement方法，遍历数组的每个链表的每个节点判断是否存在next节点根据type类型返回key或者value

**public** T nextElement() {

Entry<?,?> et = entry;

**int** i = index;

Entry<?,?>[] t = table;

/\* Use locals for faster loop iteration \*/

**while** (et == **null** && i > 0) {

et = t[--i];

}

entry = et;

index = i;

**if** (et != **null**) {

Entry<?,?> e = lastReturned = entry;

entry = e.next;

**return** type == *KEYS* ? (T)e.key : (type == *VALUES* ? (T)e.value : (T)e);

}

**throw** **new** NoSuchElementException("Hashtable Enumerator");

}

hasNext方法内部调用hasMoreElements方法

从数组的最高位下标开始，取出不为null的节点作为指针节点，然后依次遍历这个链表知道next为null,然后从下一个下标位置继续判断

**public** **boolean** hasMoreElements() {

Entry<?,?> e = entry;

**int** i = index;

Entry<?,?>[] t = table;

/\* Use locals for faster loop iteration \*/

**while** (e == **null** && i > 0) {

e = t[--i];

}

entry = e;

index = i;

**return** e != **null**;

}

生成迭代对象,key和value都是使用这个方法，type表示返回value或者key

**private** <T> Iterator<T> getIterator(**int** type) {

**if** (count == 0) {

**return** Collections.*emptyIterator*();

} **else** {

**return** **new** Enumerator<>(type, **true**);

}

}

value存放的集合

**private** **class** ValueCollection **extends** AbstractCollection<V> {

**public** Iterator<V> iterator() {

**return** getIterator(*VALUES*);

}

**public** **int** size() {

**return** count;

}

**public** **boolean** contains(Object o) {

**return** containsValue(o);

}

**public** **void** clear() {

Hashtable.**this**.clear();

}

}

keySet（）方法，返回key的set集合（实现方式和value类似，使用的是同一个迭代对象）

定义内部类KeySet在继承体系中实现了Iterable接口，重写了iterator方法，返回迭代对象；迭代对象和value的是一样的，只是在遍历的时候返回的是节点的key(return type == KEYS ? (T)e.key : (type == VALUES ? (T)e.value : (T)e);)

**public** Set<K> keySet() {

**if** (keySet == **null**)

keySet = Collections.*synchronizedSet*(**new** KeySet(), **this**);

**return** keySet;

}