[**Java 集合框架 HashSet 和 HashMap 源码剖析**](http://www.codeceo.com/article/java-hashset-hashmap.html)

2016-05-27    分类：[JAVA开发](http://www.codeceo.com/article/category/develop/java)、[编程开发](http://www.codeceo.com/article/category/develop)、[首页精华](http://www.codeceo.com/article/category/pick)[2人评论](http://www.codeceo.com/article/java-hashset-hashmap.html#comments)     来源：[CarpenterLee](http://www.blogjava.net/CarpenterLee/archive/2016/04/27/430268.html)

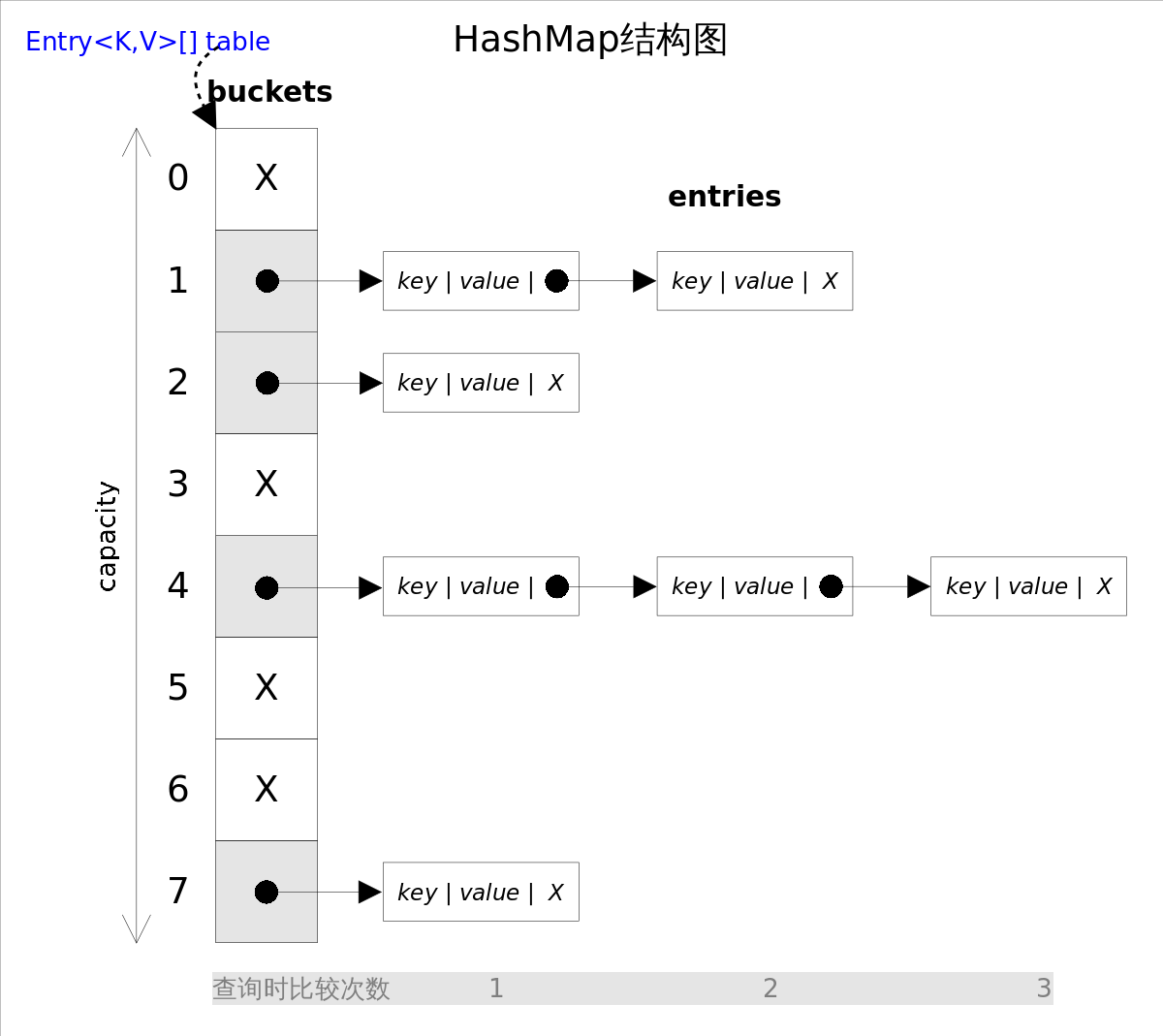
分享到：更多15

**总体介绍**

之所以把HashSet和HashMap放在一起讲解，是因为二者在Java里有着相同的实现，前者仅仅是对后者做了一层包装，也就是说**HashSet里面有一个HashMap（适配器模式）**。因此本文将重点分析HashMap。

HashMap实现了Map接口，允许放入null元素，除该类未实现同步外，其余跟Hashtable大致相同，跟TreeMap不同，该容器不保证元素顺序，根据需要该容器可能会对元素重新哈希，元素的顺序也会被重新打散，因此不同时间迭代同一个HashMap的顺序可能会不同。

根据对冲突的处理方式不同，哈希表有两种实现方式，一种开放地址方式（Open addressing），另一种是冲突链表方式（Separate chaining with linked lists）。**Java HashMap采用的是冲突链表方式**。



从上图容易看出，如果选择合适的哈希函数，put()和get()方法可以在常数时间内完成。但在对HashMap进行迭代时，需要遍历整个table以及后面跟的冲突链表。因此对于迭代比较频繁的场景，不宜将HashMap的初始大小设的过大。

有两个参数可以影响HashMap的性能：初始容量（inital capacity）和负载系数（load factor）。初始容量指定了初始table的大小，负载系数用来指定自动扩容的临界值。当entry的数量超过capacity\*load\_factor时，容器将自动扩容并重新哈希。对于插入元素较多的场景，将初始容量设大可以减少重新哈希的次数。

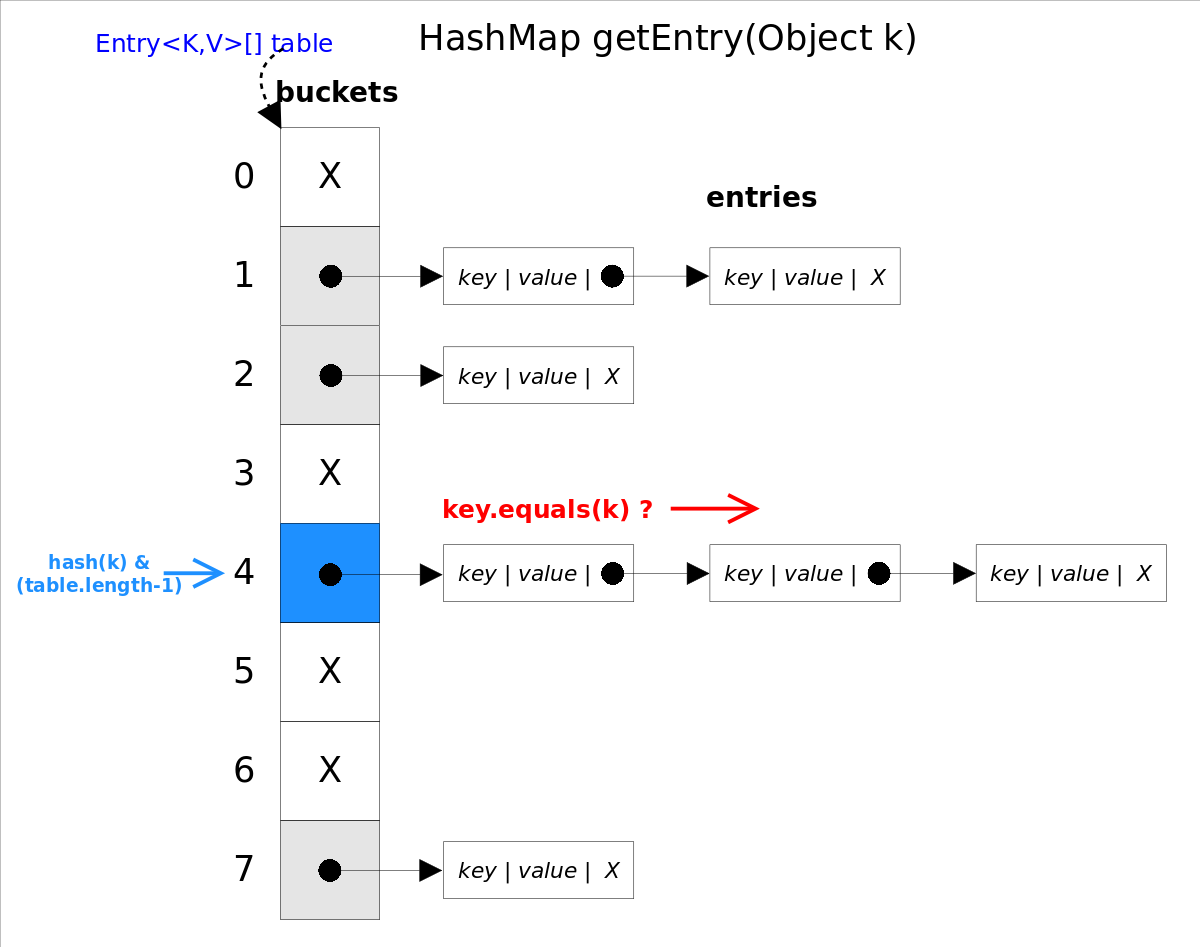
将对向放入到HashMap或HashSet中时，有两个方法需要特别关心：hashCode()和equals()。**hashCode()方法决定了对象会被放到哪个bucket里，当多个对象的哈希值冲突时，equals()方法决定了这些对象是否是“同一个对象”**。所以，如果要将自定义的对象放入到HashMap或HashSet中，需要@Override hashCode()和equals()方法。

**方法剖析**

**get()**

get(Object key)方法根据指定的key值返回对应的value，该方法调用了getEntry(Object key)得到相应的entry，然后返回entry.getValue()。因此getEntry()是算法的核心。

算法思想是首先通过hash()函数得到对应bucket的下标，然后依次遍历冲突链表，通过key.equals(k)方法来判断是否是要找的那个entry。



上图中hash(k)&(table.length-1)等价于hash(k)%table.length，原因是HashMap要求table.length必须是2的指数，因此table.length-1就是二进制低位全是1，跟hash(k)相与会将哈希值的高位全抹掉，剩下的就是余数了。

//getEntry()方法

final Entry<K,V> getEntry(Object key) {

......

int hash = (key == null) ? 0 : hash(key);

for (Entry<K,V> e = table[hash&(table.length-1)];//得到冲突链表

e != null; e = e.next) {//依次遍历冲突链表中的每个entry

Object k;

//依据equals()方法判断是否相等

if (e.hash == hash &&

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

return e;

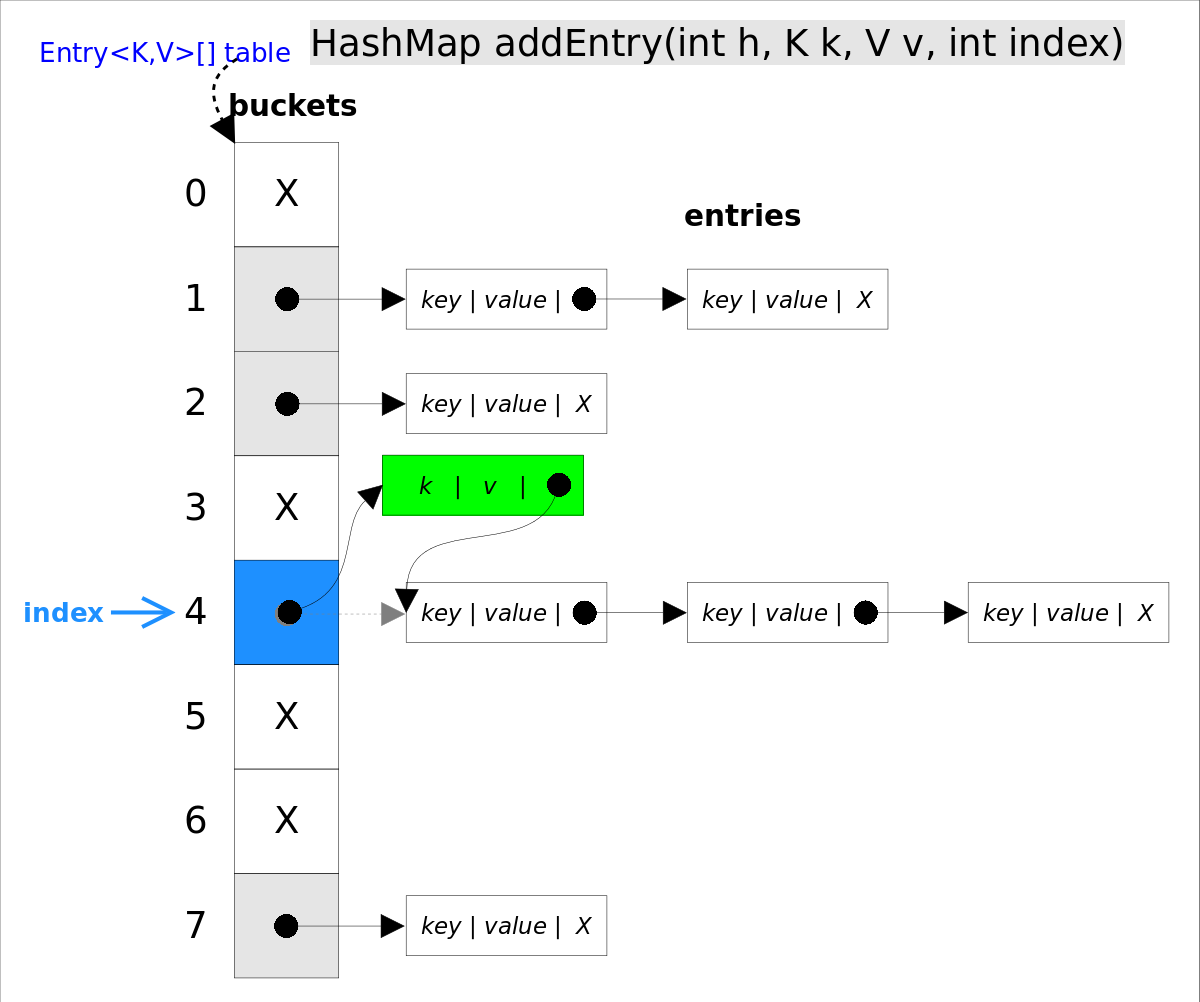
}

return null;

}

**put()**

put(K key, V value)方法是将指定的key, value对添加到map里。该方法首先会对map做一次查找，看是否包含该元组，如果已经包含则直接返回，查找过程类似于getEntry()方法；如果没有找到，则会通过addEntry(int hash, K key, V value, int bucketIndex)方法插入新的entry，插入方式为**头插法**。



//addEntry()

void addEntry(int hash, K key, V value, int bucketIndex) {

if ((size >= threshold) && (null != table[bucketIndex])) {

resize(2 \* table.length);//自动扩容，并重新哈希

hash = (null != key) ? hash(key) : 0;

bucketIndex = hash & (table.length-1);//hash%table.length

}

//在冲突链表头部插入新的entry

Entry<K,V> e = table[bucketIndex];

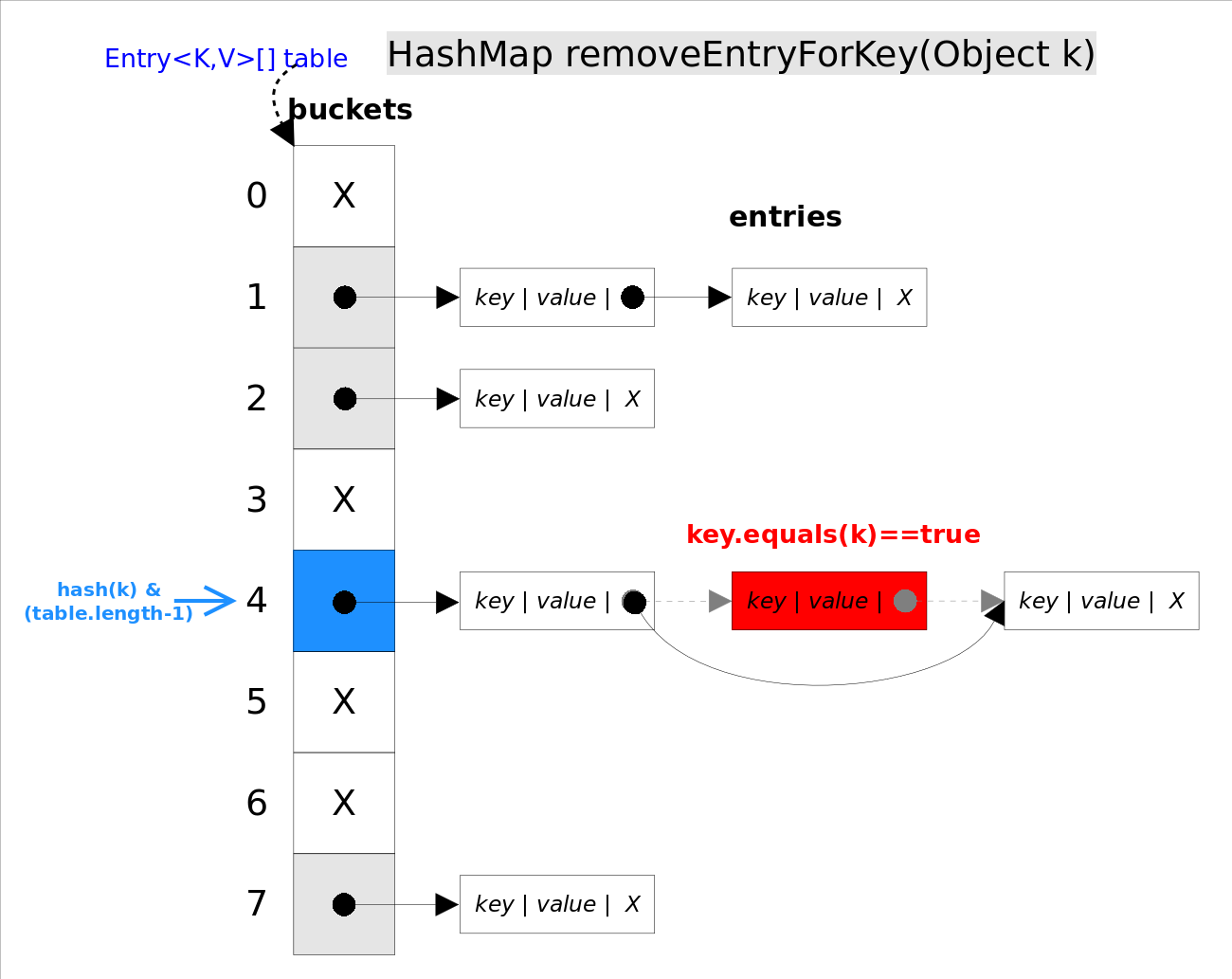
table[bucketIndex] = new Entry<>(hash, key, value, e);

size++;

}

**remove()**

remove(Object key)的作用是删除key值对应的entry，该方法的具体逻辑是在removeEntryForKey(Object key)里实现的。removeEntryForKey()方法会首先找到key值对应的entry，然后删除该entry（修改链表的相应指针）。查找过程跟getEntry()过程类似。



//removeEntryForKey()

final Entry<K,V> removeEntryForKey(Object key) {

......

int hash = (key == null) ? 0 : hash(key);

int i = indexFor(hash, table.length);//hash&(table.length-1)

Entry<K,V> prev = table[i];//得到冲突链表

Entry<K,V> e = prev;

while (e != null) {//遍历冲突链表

Entry<K,V> next = e.next;

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k)))) {//找到要删除的entry

modCount++; size--;

if (prev == e) table[i] = next;//删除的是冲突链表的第一个entry

else prev.next = next;

return e;

}

prev = e; e = next;

}

return e;

}

**HashSet**

前面已经说过HashSet是对HashMap的简单包装，对HashSet的函数调用都会转换成合适的HashMap方法，因此HashSet的实现非常简单，只有不到300行代码。这里不再赘述。

//HashSet是对HashMap的简单包装

public class HashSet<E>

{

......

private transient HashMap<E,Object> map;//HashSet里面有一个HashMap

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();

public HashSet() {

map = new HashMap<>();

}

......

public boolean add(E e) {//简单的方法转换

return map.put(e, PRESENT)==null;

}

......

}