深入 java8 的集合 5：Hashtable 的实现原理

一、概述

上一篇介绍了 Java8 的 HashMap，接下来准备介绍一下 Hashtable。

Hashtable 可以说已经具有一定的历史了，现在也很少使用到 Hashtable 了，更多的是使用 HashMap 或 ConcurrentHashMap。HashTable 是一个线程安全的哈希表，它通过使用 synchronized 关键字来对方法进行加锁，从而保证了线程安全。但这也导致了在单线程环境中效率低下等问题。Hashtable 与 HashMap 不同，它不允许插入 null 值和 null 键。

二、属性

Hashtable 并没有像 HashMap 那样定义了很多的常量，而是直接写死在了方法里（看下去就知道了），所以它的属性相比 HashMap 来说，可以获取的信息还是比较少的。

```
//哈希表

private transient Entry<?,?>[] table;



//记录哈希表中键值对的个数

private transient int count;



//扩容的阈值

private int threshold;



//负载因子

private float loadFactor;
```

三、方法

1、构造方法

```
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: "+l
oadFactor);



    if (initialCapacity==0)

        initialCapacity = 1;

    this.loadFactor = loadFactor;

    table = new Entry<?,?>[initialCapacity];

    threshold = (int)Math.min(initialCapacity * loadFactor, MAX_
ARRAY_SIZE + 1);

}



public Hashtable(int initialCapacity) {

    this(initialCapacity, 0.75f);

}



public Hashtable() {

    this(11, 0.75f);

}
```

二话不说，上来先丢了三个构造函数。从构造函数中，我们可以获取到这些信息：Hashtable 默认的初始化容量为 11（与 HashMap 不同），负载因子默认为 0.75（与 HashMap 相同）。而正因为默认初始化容量的不同，同时也没有对容量做调整的策略，所以可以先推断出，Hashtable 使用的哈希函数跟 HashMap 是不一样的（事实也确实如此）。

2、get 方法

```
public synchronized V get(Object key) {

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

    int hash = key.hashCode();

    //通过哈希函数，计算出 key 对应的桶的位置

    int index = (hash & 0x7FFFFFFF) % tab.length;

    //遍历该桶的所有元素，寻找该 key

    for (Entry<?,?> e = tab[index] ; e != null ; e = e.next) {

        if ((e.hash == hash) && e.key.equals(key)) {

            return (V)e.value;

        }

    }

    return null;

}
```

跟 HashMap 相比，Hashtable 的 get 方法非常简单。我们首先可以看见 get 方法使用了 synchronized 来修饰，所以它能保证线程安全。并且它是通过链表的方式来处理冲突的。另外，我们还可以看见 HashTable 并没有像 HashMap 那样封装一个哈希函数，而是直接把哈希函数写在了方法中。而哈希函数也是比较简单的，它仅对哈希表的长度进行了取模。

3、put 方法

```
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();
    //计算桶的位置
    int index = (hash & 0x7FFFFFFF) % tab.length;
    @SuppressWarnings("unchecked")
    Entry<K,V> entry = (Entry<K,V>)tab[index];
    //遍历桶中的元素，判断是否存在相同的 key
    for(; entry != null ; entry = entry.next) {
        if ((entry.hash == hash) && entry.key.equals(key)) {
            V old = entry.value;
            entry.value = value;
            return old;
        }
    }

    //不存在相同的 key，则把该 key 插入到桶中
```

```
        addEntry(hash, key, value, index);
        return null;
    }


    private void addEntry(int hash, K key, V value, int index) {
        modCount++;

        Entry<?,?> tab[] = table;
        //哈希表的键值对个数达到了阈值，则进行扩容
        if (count >= threshold) {
            // Rehash the table if the threshold is exceeded
            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];
        //把新节点插入桶中（头插法）
        tab[index] = new Entry<>(hash, key, value, e);
```

```
        count++;

    }
```

put 方法一开始就表明了不能有 null 值，否则就会向你抛出一个空指针异常。Hashtable 的 put 方法也是使用 synchronized 来修饰。你可以发现，在 Hashtable 中，几乎所有的方法都使用了 synchronized 来保证线程安全。

4、remove 方法

```
    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.nex
    t) {

            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;
    }
```

remove 方法我已经不想加注释了，跟 get 和 put 的原理差不多。如果看过上一篇的 HashMap 的话，或者理解了上面的 put 方法的话，我相信 remove 方法看一眼就能懂了。

5、rehash 方法

```
protected void rehash() {
    int oldCapacity = table.length;
    Entry<?,?>[] oldMap = table;

    //扩容扩为原来的两倍+1
    int newCapacity = (oldCapacity << 1) + 1;
    //判断是否超过最大容量
    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++;
    //计算下一次 rehash 的阈值
    threshold = (int)Math.min(newCapacity * loadFactor, MAX_ARRA
Y_SIZE + 1);
    table = newMap;


    //把旧哈希表的键值对重新哈希到新哈希表中去
    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;
        }
    }
}
```

Hashtable 的 rehash 方法相当于 HashMap 的 resize 方法。跟 HashMap 那种巧妙的 rehash 方式相比，Hashtable 的 rehash 过程需要对每个键值对都重新计算哈希值，而比起异或和与操作，取模是一个非常耗时的操作，所以这也是导致效率较低的原因之一。