Java集合学习手册（3）：Java HashTable

一、概述

和HashMap一样，Hashtable也是一个散列表，它存储的内容是键值对。

Hashtable在Java中的定义为：

```
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable{}
```

从源码中，我们可以看出，Hashtable继承于Dictionary类，实现了Map, Cloneable, java.io.Serializable接口。其中Dictionary类是任何可将键映射到相应值的类（如 Hashtable）的抽象父类，每个键和值都是对象（源码注释为：The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values. Every key and every value is an object.）。但其Dictionary源码注释是这样的：NOTE: This class is obsolete. New implementations should implement the Map interface, rather than extending this class. 该话指出Dictionary这个类过时了，新的实现类应该实现Map接口。

二、成员变量

Hashtable是通过”拉链法”实现的哈希表。它包括几个重要的成员变量：table, count, threshold, loadFactor, modCount。

- table是一个Entry[]数组类型，而Entry（在HashMap中有讲解过）实际上就是一个单向链表。哈希表的”key-value键值对”都是存储在Entry数组中的。
- count是Hashtable的大小，它是Hashtable保存的键值对的数量。
- threshold是Hashtable的阈值，用于判断是否需要调整Hashtable的容量。threshold的值=”容量*加载因子”。
- loadFactor就是加载因子。
- modCount是用来实现fail-fast机制的。

变量的解释在源码注释中如下：

```
/**
     * The hash table data.
     */
    private transient Entry<K,V>[] table;

    /**
     * The total number of entries in the hash table.
     */
    private transient int count;

    /**
     * The table is rehashed when its size exceeds this threshold.  (The
     * value of this field is (int)(capacity * loadFactor).)
     *
     * @serial
     */
    private int threshold;

    /**
     * The load factor for the hashtable.
     *
     * @serial
     */
    private float loadFactor;

    /**
     * The number of times this Hashtable has been structurally modified
     * Structural modifications are those that change the number of entries in
     * the Hashtable or otherwise modify its internal structure (e.g.,
     * rehash).  This field is used to make iterators on Collection-views of
     * the Hashtable fail-fast.  (See ConcurrentModificationException).
     */
    private transient int modCount = 0;
```

三、构造方法

Hashtable一共提供了4个构造方法：

- public Hashtable(int initialCapacity, float loadFactor)： 用指定初始容量和指定加载因子构造一个新的空哈希表。useAltHashing为boolean，其如果为真，则执行另一散列的字符串键，以减少由于弱哈希计算导致的哈希冲突的发生。
- public Hashtable(int initialCapacity)：用指定初始容量和默认的加载因子 (0.75) 构造一个新的空哈希表。
- public Hashtable()：默认构造函数，容量为11，加载因子为0.75。

- public Hashtable(Map<? extends K, ? extends V> t)：构造一个与给定的 Map 具有相同映射关系的新哈希表。

```
/**
     * Constructs a new, empty hashtable with the specified initial
     * capacity and the specified load factor.
     *
     * @param      initialCapacity   the initial capacity of the hashtable.
     * @param      loadFactor        the load factor of the hashtable.
     * @exception  IllegalArgumentException  if the initial capacity is less
     *             than zero, or if the load factor is nonpositive.
     */
    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)
;
        useAltHashing = sun.misc.VM.isBooted() &&
                (initialCapacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);
    }

    /**
     * Constructs a new, empty hashtable with the specified initial capacity
     * and default load factor (0.75).
     *
     * @param     initialCapacity   the initial capacity of the hashtable.
     * @exception IllegalArgumentException if the initial capacity is less
     *              than zero.
     */
    public Hashtable(int initialCapacity) {
        this(initialCapacity, 0.75f);
    }

    /**
     * Constructs a new, empty hashtable with a default initial capacity (11)
     * and load factor (0.75).
     */
    public Hashtable() {
        this(11, 0.75f);
```

```
    }

    /**
     * Constructs a new hashtable with the same mappings as the given
     * Map.  The hashtable is created with an initial capacity sufficient to
     * hold the mappings in the given Map and a default load factor (0.75).
     *
     * @param t the map whose mappings are to be placed in this map.
     * @throws NullPointerException if the specified map is null.
     * @since   1.2
     */
    public Hashtable(Map<? extends K, ? extends V> t) {
        this(Math.max(2*t.size(), 11), 0.75f);
        putAll(t);
    }
```

四、put方法

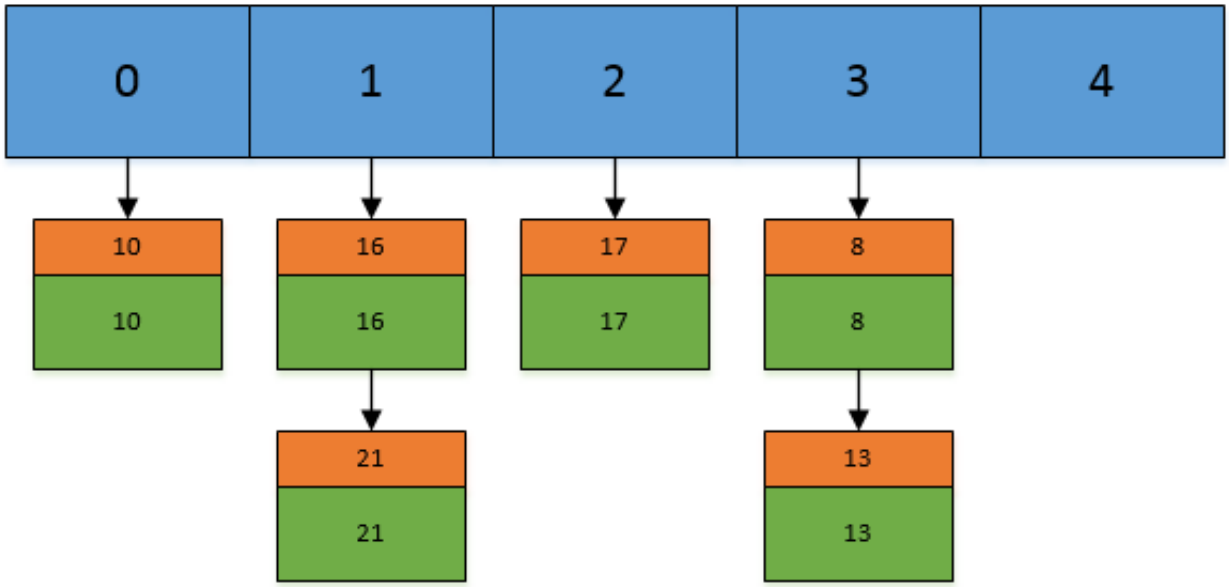
put方法的整个流程为：

- 判断value是否为空，为空则抛出异常；
- 计算key的hash值，并根据hash值获得key在table数组中的位置index，如果table[index]元素不为空，则进行迭代，如果遇到相同的key，则直接替换，并返回旧value；
- 否则，我们可以将其插入到table[index]位置。

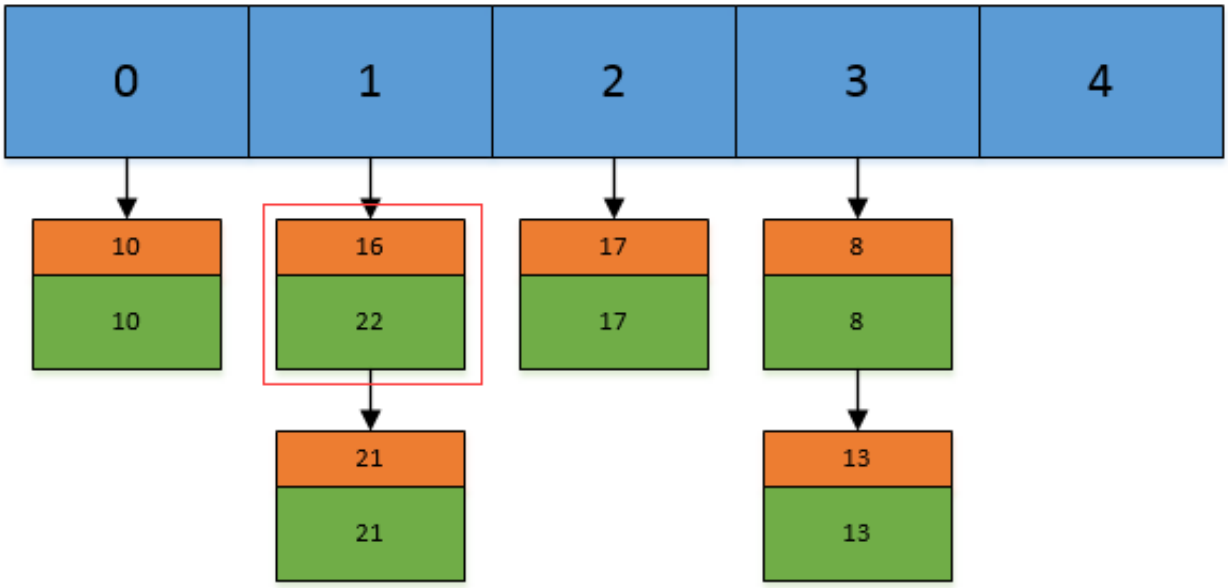
```
public synchronized V put(K key, V value) {
        // Make sure the value is not null确保value不为null
        if (value == null) {
            throw new NullPointerException();
        }
        // Makes sure the key is not already in the hashtable.
        //确保key不在hashtable中
        //首先，通过hash方法计算key的哈希值，并计算得出index值，确定其在table[]中的位置
        //其次，迭代index索引位置的链表，如果该位置处的链表存在相同的key，则替换value，返回旧的value
        Entry tab[] = table;
        int hash = hash(key);
        int index = (hash & 0x7FFFFFFF) % tab.length;
        for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
            if ((e.hash == hash) && e.key.equals(key)) {
                V old = e.value;
                e.value = value;
                return old;
            }
```

```
        }

        modCount++;
        if (count >= threshold) {
            // Rehash the table if the threshold is exceeded
            //如果超过阀值，就进行rehash操作
            rehash();

            tab = table;
            hash = hash(key);
            index = (hash & 0x7FFFFFFF) % tab.length;
        }

        // Creates the new entry.
        //将值插入，返回的为null
        Entry<K,V> e = tab[index];
        // 创建新的Entry节点，并将新的Entry插入Hashtable的index位置，并设置e为新的Entry
的下一个元素
        tab[index] = new Entry<>(hash, key, value, e);
        count++;
        return null;
    }
```

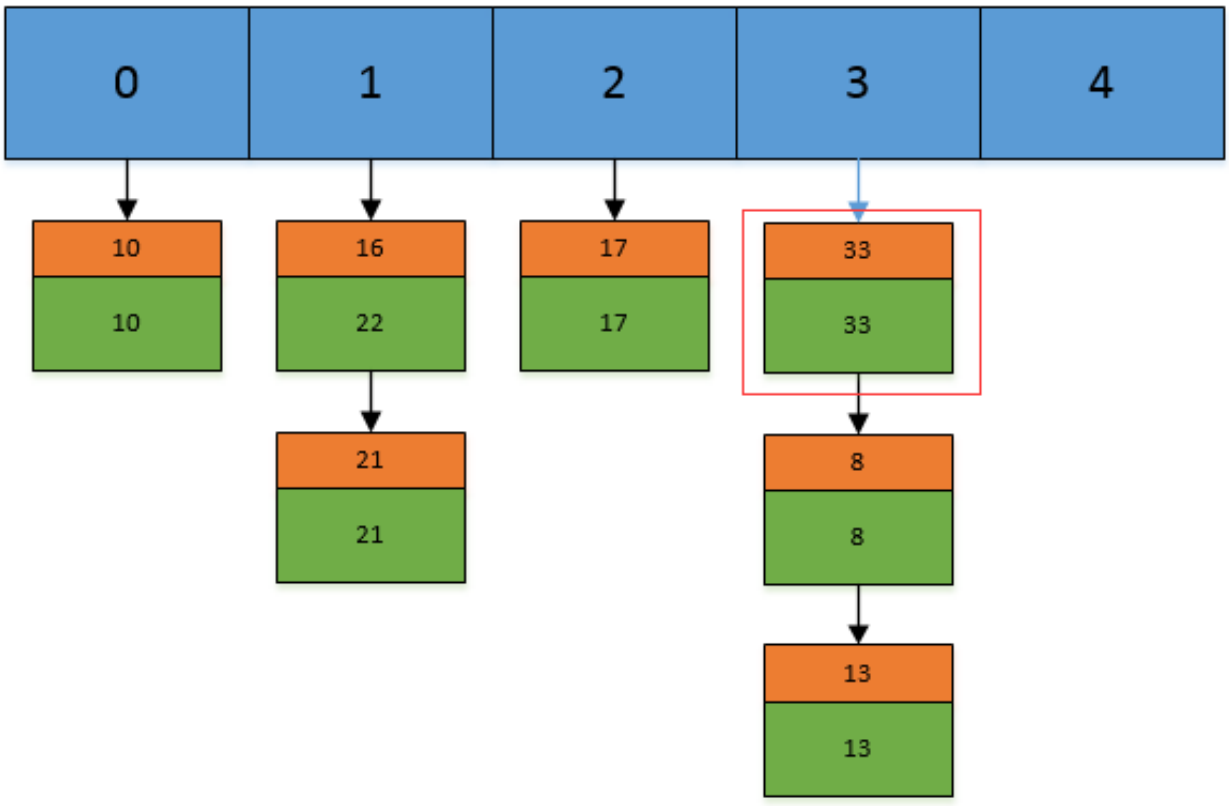
通过一个实际的例子来演示一下这个过程：
假设我们现在Hashtable的容量为5，已经存在了(5,5)，(13,13)，(16,16)，(17,17)，(21,21)这5个键值对，目前他们在Hashtable中的位置如下：



现在，我们插入一个新的键值对，put(16,22)，假设key=16的索引为1.但现在索引1的位置有两个Entry了，所以程序会对链表进行迭代。迭代的过程中，发现其中有一个Entry的key和我们要插入的键值对的key相同，所以现在会做的工作就是将newValue=22替换oldValue=16，然后返回oldValue=16.



然后我们现在再插入一个，put(33,33)，key=33的索引为3，并且在链表中也不存在key=33的Entry，所以将该节点插入链表的第一个位置。



五、get方法

相比较于put方法，get方法则简单很多。其过程就是首先通过hash()方法求得key的哈希值，然后根据hash值得到index索引（上述两步所用的算法与put方法都相同）。然后迭代链表，返回匹配的key的对应的value；找不到则返回null。

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

```
        Entry tab[] = table;
        int hash = hash(key);
        int index = (hash & 0x7FFFFFFF) % tab.length;
        for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
            if ((e.hash == hash) && e.key.equals(key)) {
                return e.value;
            }
        }
        return null;
    }
```

六、遍历方式

Hashtable有多种遍历方式：

```
//1、使用keys()
Enumeration<String> en1 = table.keys();
    while(en1.hasMoreElements()) {
    en1.nextElement();
}

//2、使用elements()
Enumeration<String> en2 = table.elements();
    while(en2.hasMoreElements()) {
    en2.nextElement();
}

//3、使用keySet()
Iterator<String> it1 = table.keySet().iterator();
    while(it1.hasNext()) {
    it1.next();
}

//4、使用entrySet()
Iterator<Entry<String, String>> it2 = table.entrySet().iterator();
    while(it2.hasNext()) {
    it2.next();
}
```