Java集合学习手册（1）：Java HashMap

一、概述

从本文你可以学习到：

1. 什么时候会使用HashMap？他有什么特点？
2. 你知道HashMap的工作原理吗？
3. 你知道get和put的原理吗？equals()和hashCode()的都有什么作用？
4. 你知道hash的实现吗？为什么要这样实现？
5. 如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？

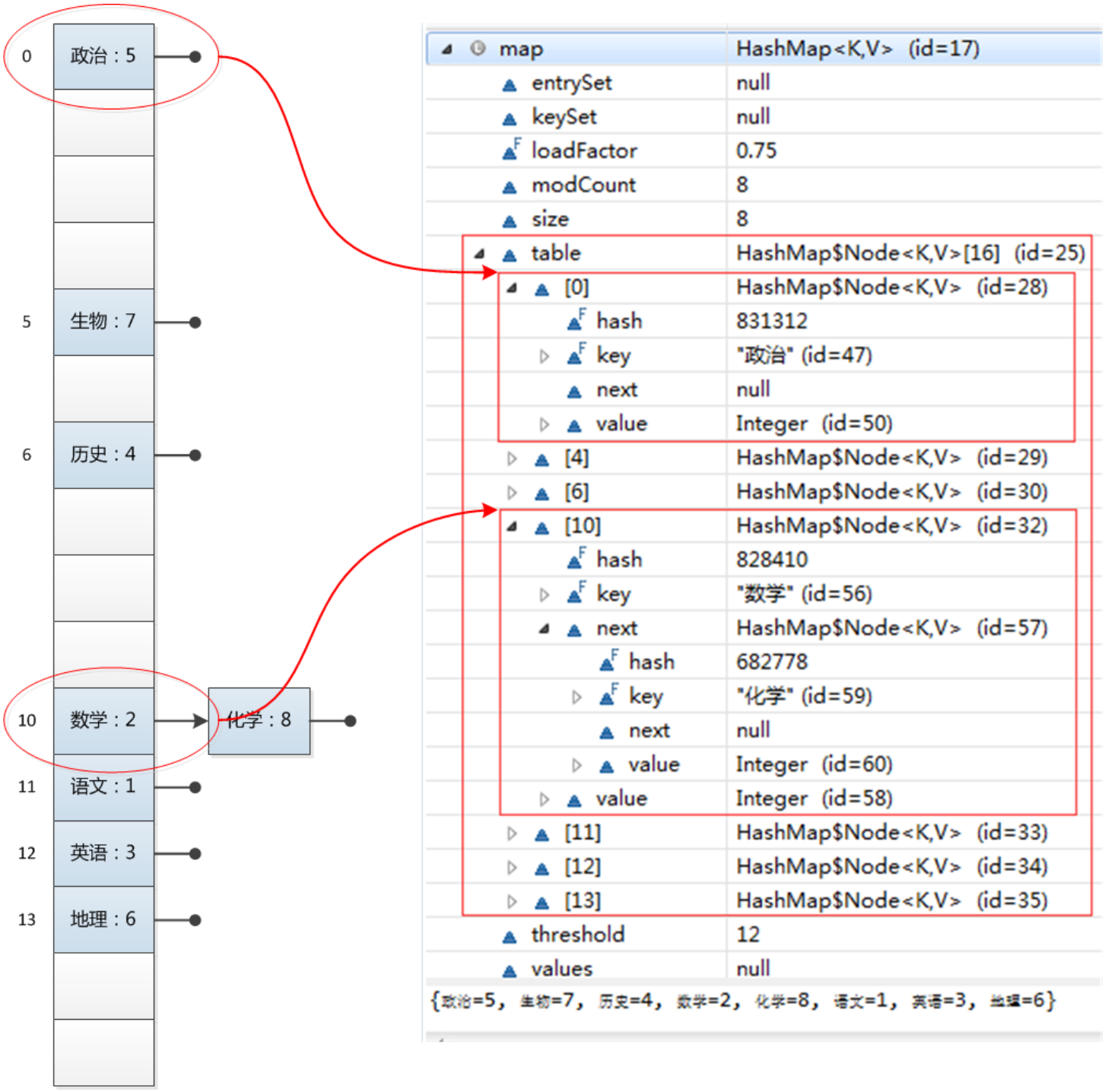
当我们执行下面的操作时：

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("语文", 1);
map.put("数学", 2);
map.put("英语", 3);
map.put("历史", 4);
map.put("政治", 5);
map.put("地理", 6);
map.put("生物", 7);
map.put("化学", 8);
for(Entry<String, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

运行结果是：

```
政治: 5
生物: 7
历史: 4
数学: 2
化学: 8
语文: 1
英语: 3
地理: 6
```

发生了什么呢？下面是一个大致的结构，希望我们对HashMap的结构有一个感性的认识：



在官方文档中是这样描述HashMap的：

> Hash table based **implementation of the Map interface**. This implementation provides all of the optional map operations, and **permits null values and the null key**. (**The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.**) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

几个关键的信息：基于Map接口实现、允许null键/值、非同步、不保证有序(比如插入的顺序)、也不保证序不随时间变化。

在HashMap中有两个很重要的参数，容量(Capacity)和负载因子(Load factor)

- Initial capacity： The capacity is the number of buckets in the hash table, The initial capacity is simply the capacity at the time the hash table is created.
- Load factor： The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

简单的说，Capacity就是bucket的大小，Load factor就是bucket填满程度的最大比例。如果对迭代性能要求很高的话不要把capacity设置过大，也不要把load factor设置过小。当bucket中的entries的数目大于capacity*load factor时就需要调整bucket的大小为当前的2倍。

二、构造函数

HashMap底层维护一个数组，当新建一个HashMap的时候，就会初始化一个数组。我们看一下JDK源码中的HashMap构造函数：

```
public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                               loadFactor);

        // Find a power of 2 >= initialCapacity
        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <<= 1;

        this.loadFactor = loadFactor;
        threshold = (int)Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
        table = new Entry[capacity];
        useAltHashing = sun.misc.VM.isBooted() &&
                (capacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);
        init();
}
```

可以看到其中一行为`table = new Entry[capacity];`。在构造函数中，其创建了一个Entry的数组，其大小为capacity，那么Entry又是什么结构呢？看一下源码：

```
transient Entry<K,V>[] table;
```

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
    final int hash;
    ……
}
```

HashMap中的是通过transient Entry[] table来存储数据，该变量是通过transient进行修饰的。

Entry是一个static class，其中包含了key和value，也就是键值对，另外还包含了一个next的Entry指针。我们可以总结出：Entry就是数组中的元素，每个Entry其实就是一个key-value对，它持有一个指向下一个元素的引用，这就构成了链表。

三、put()方法和get()方法

3.1 put()方法

put函数大致的思路为：

1. 对key的hashCode()做hash，然后再计算index;
2. 如果没碰撞直接放到bucket里；
3. 如果碰撞了，以链表的形式存在buckets后；
4. 如果碰撞导致链表过长(大于等于TREEIFY_THRESHOLD)，就把链表转换成红黑树；
5. 如果节点已经存在就替换old value(保证key的唯一性)
6. 如果bucket满了(超过load factor*current capacity)，就要resize。

具体代码的实现如下：

```
public V put(K key, V value) {
    // 对key的hashCode()做hash
    return putVal(hash(key), key, value, false, true);
}
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    //如果当前map中无数据，执行resize方法。并且返回n
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    ///如果要插入的键值对要存放的这个位置刚好没有元素，那么把他封装成Node对象，放在这个位置上就完事了
    if ((p = tab[i = (n - 1) & hash]) == null)
```

```
            tab[i] = newNode(hash, key, value, null);
        //否则的话，说明这上面有元素
        else {
            Node<K,V> e; K k;
            //如果这个元素的key与要插入的一样，那么就替换一下，也完事。
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            //1.如果当前节点是TreeNode类型的数据，执行putTreeVal方法
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            //还是遍历这条链子上的数据，跟jdk7没什么区别
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        //2.完成了操作后多做了一件事情，判断，并且可能执行treeifyBin方法，tr
eeifyBin()就是将链表转换成红黑树。
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
            // 写入
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        //判断阈值，决定是否扩容
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

一直到JDK7为止，HashMap的结构基于一个数组以及多个链表的实现，hash值冲突的时候，就将对应节点以链表的形式存储。

这样子的HashMap性能上就抱有一定疑问，如果说成百上千个节点在hash时发生碰撞，存储一个链表中，那么如果要查找其中一个节点，那就不可避免的花费O(N)的查找时间，这将是多么大的性能损失。这个问题终于在JDK8中得到了解决。再最坏的情况下，链表查找的时间复杂度为O(n),而红黑树一直是O(logn),这样会提高HashMap的效率。JDK7中HashMap采用的是位桶+链表的方式，即我们常说的散列链表的方式，而JDK8中采用的是位桶+链表/红黑树（有关红黑树请查看红黑树）的方式，也是非线程安全的。当某个位桶的链表的长度达到某个阀值的时候，这个链表就将转换成红黑树。

JDK8中，当同一个hash值的节点数不小于8时，将不再以单链表的形式存储了，会被调整成一颗红黑树。这就是JDK7与JDK8中HashMap实现的最大区别。JDK中Entry的名字变成了Node，原因是和红黑树的实现TreeNode相关联。

```
transient Node<K,V>[] table;
```

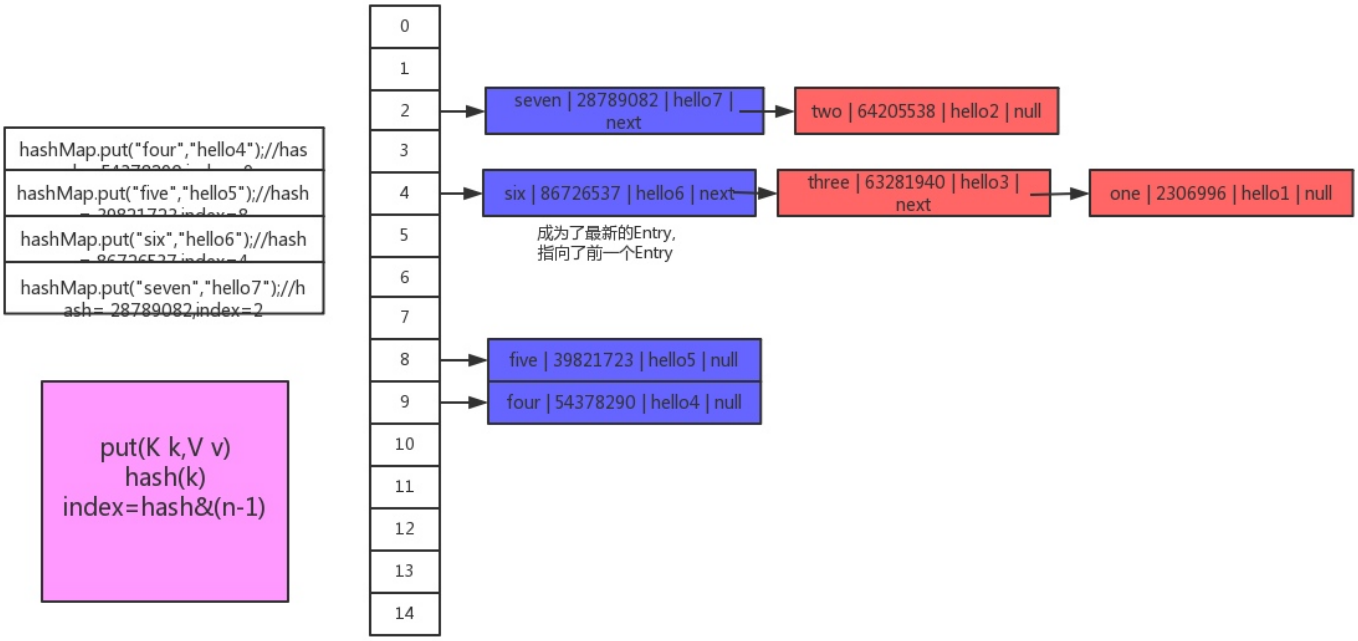
当冲突节点数不小于8-1时，转换成红黑树。

```
static final int TREEIFY_THRESHOLD = 8;
```

总结下put的过程：

当程序试图将一个key-value对放入HashMap中时，程序首先根据该 key 的 hashCode() 返回值决定该 Entry 的存储位置， 该位置就是此对象准备往数组中存放的位置。

如果该位置没有对象存在，就将此对象直接放进数组当中；如果该位置已经有对象存在了，则顺着此存在的对象的链开始寻找(为了判断是否是否值相同，map不允许键值对重复)， 使用 equals 方法进行比较，如果对此链上的 key 通过 equals 比较有一个返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value，但key不会覆盖；如果对此链上的每个对象的 equals 方法比较都为 false，则将该对象放到数组当中，然后将数组中该位置以前存在的那个对象链接到此对象的后面，即新值存放在数组中，旧值在新值的链表上。



3.2 get()方法

在理解了put之后，get就很简单了。大致思路如下：

1. bucket里的第一个节点，直接命中；
2. 如果有冲突，则通过key.equals(k)去查找对应的entry
    - 若为树，则在树中通过key.equals(k)查找，O(logn)；
    - 若为链表，则在链表中通过key.equals(k)查找，O(n)。

具体代码的实现如下：

```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        // 直接命中
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        // 未命中
        if ((e = first.next) != null) {
            // 在树中get
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
```

```
            // 在链表中get
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

3.3 null key存取

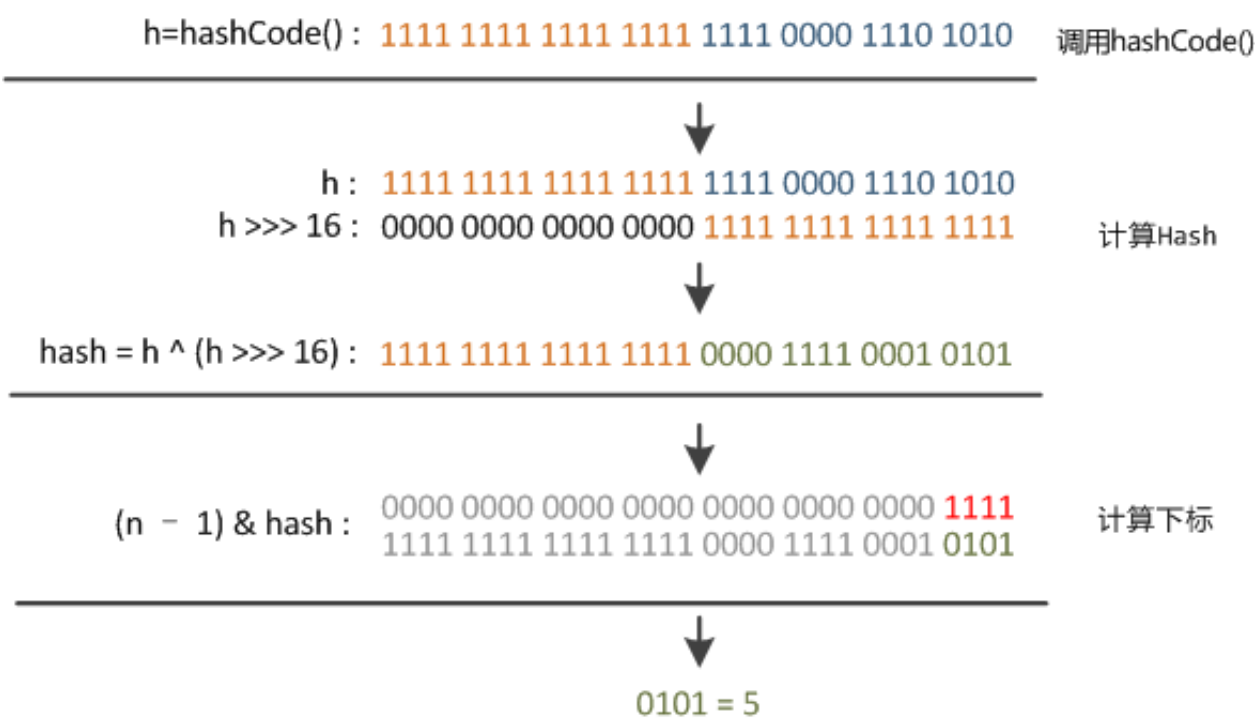
对于put方法来说，HashMap会对null值key进行特殊处理，总是放到table[0]位置
对于get方法来说，同样当key为null时会进行特殊处理，在table[0]的链表上查找key为null的元素

```
private V putForNullKey(V value) {
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
            if (e.key == null) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }
        modCount++;
        addEntry(0, null, value, 0);
        return null;
    }
```

四、hash()与indexFor()

4.1 hash()方法

在get和put的过程中，计算下标时，先对hashCode进行hash操作，然后再通过hash值进一步计算下标，如下图所示：



在对hashCode()计算hash时具体实现是这样的：

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

按位取并，作用上相当于取模mod或者取余%。这意味着数组下标相同，并不表示hashCode相同。

可以看到这个函数大概的作用就是：高16bit不变，低16bit和高16bit做了一个异或。其中代码注释是这样写的：

> Computes key.hashCode() and spreads (XORs) higher bits of hash to lower. Because the table uses power-of-two masking, sets of hashes that vary only in bits above the current mask will always collide. (Among known examples are sets of Float keys holding consecutive whole numbers in small tables.) So we apply a transform that spreads the impact of higher bits downward. There is a tradeoff between speed, utility, and quality of bit-spreading. Because many common sets of hashes are already reasonably distributed (so don’t benefit from spreading), and because we use trees to handle large sets of collisions in bins, we just XOR some shifted bits in the cheapest possible way to reduce systematic lossage, as well as to incorporate impact of the highest bits that would otherwise never be used in index calculations because of table bounds.

4.2 indexFor()方法

在设计hash函数时，因为目前的table长度length n为2的幂，而计算下标的时候，是这样实现的(使用&位操作，而非%求余)：

```
static int indexFor(int h, int n) {
        return h & (n-1);
    }
```

设计者认为这方法很容易发生碰撞。为什么这么说呢？不妨思考一下，在n - 1为15(0x1111)时，其实散列真正生效的只是低4bit的有效位，当然容易碰撞了。

因此，设计者想了一个顾全大局的方法(综合考虑了速度、作用、质量)，就是把高16bit和低16bit异或了一下。设计者还解释到因为现在大多数的hashCode的分布已经很不错了，就算是发生了碰撞也用O(logn)的tree去做了。仅仅异或一下，既减少了系统的开销，也不会造成的因为高位没有参与下标的计算(table长度比较小时)，从而引起的碰撞。

如果还是产生了频繁的碰撞，会发生什么问题呢？作者注释说，他们使用树来处理频繁的碰撞(we use trees to handle large sets of collisions in bins)，在JEP-180中，描述了这个问题：

> Improve the performance of java.util.HashMap under high hash-collision conditions by using balanced trees rather than linked lists to store map entries. Implement the same improvement in the LinkedHashMap class.

之前已经提过，在获取HashMap的元素时，基本分两步：

1. 首先根据hashCode()做hash，然后确定bucket的index；
2. 如果bucket的节点的key不是我们需要的，则通过keys.equals()在链中找。

在Java 8之前的实现中是用链表解决冲突的，在产生碰撞的情况下，进行get时，两步的时间复杂度是O(1)+O(n)。因此，当碰撞很厉害的时候n很大，O(n)的速度显然是影响速度的。

因此在Java 8中，利用红黑树替换链表，这样复杂度就变成了O(1)+O(logn)了，这样在n很大的时候，能够比较理想的解决这个问题，在Java 8：HashMap的性能提升一文中有性能测试的结果。

五、resize()方法

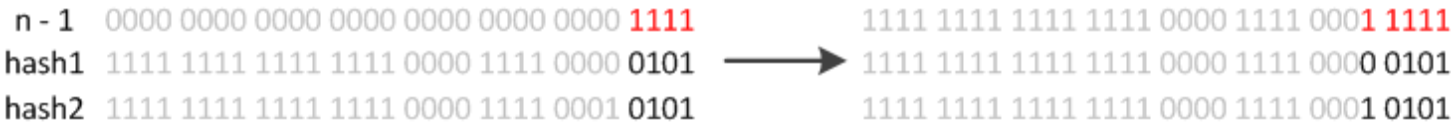
当put时，如果发现目前的bucket占用程度已经超过了Load Factor所希望的比例，那么就会发生resize。在resize的过程，简单的说就是把bucket扩充为2倍，之后重新计算index，把节点再放到新的bucket中。resize的注释是这样描述的：

> Initializes or doubles table size. If null, allocates in accord with initial capacity target held in field threshold. Otherwise, because we are using power-of-two expansion, the

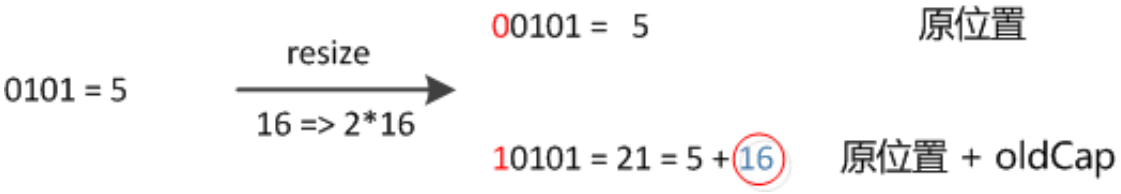
> elements from each bin must either stay at same index, or move with a power of two offset in the new table.

大致意思就是说，当超过限制的时候会resize，然而又因为我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。

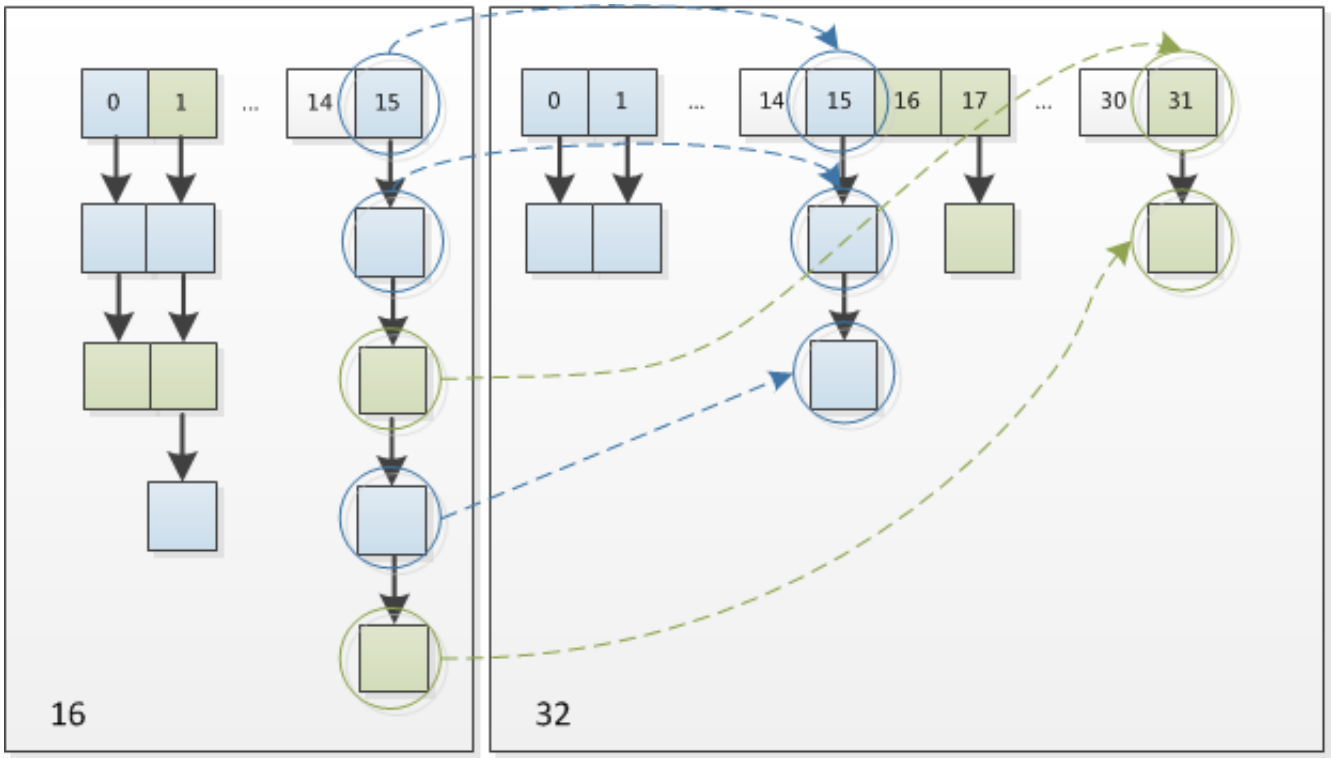
怎么理解呢？例如我们从16扩展为32时，具体的变化如下所示：



因此元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”。可以看看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。

下面是代码的具体实现：

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        // 超过最大值就不再扩充了，就只好随你碰撞去吧
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        // 没超过最大值，就扩充为原来的2倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    // 计算新的resize上限
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        // 把每个bucket都移动到新的buckets中
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
```

```
                        // 原索引
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        // 原索引+oldCap
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    // 原索引放到bucket里
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
                    // 原索引+oldCap放到bucket里
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
    }
    return newTab;
}
```

六、remove()、clear()、containsKey()和containsValue()

6.1 remove()方法

remove方法和put get类似，计算hash，计算index，然后遍历查找，将找到的元素从table[index]链表移除

```
public V remove(Object key) {
```

Entry e = removeEntryForKey(key);
return (e == null ? null : e.value);
}
final Entry removeEntryForKey(Object key) {
int hash = (key == null) ? 0 : hash(key.hashCode());
int i = indexFor(hash, table.length);
Entry prev = table[i];
Entry e = prev;

```
    while (e != null) {
        Entry<K,V> next = e.next;
        Object k;
        if (e.hash == hash &&
            ((k = e.key) == key || (key != null && key.equals(k)))) {
            modCount++;
            size--;
            if (prev == e)
                table[i] = next;
            else
                prev.next = next;
            e.recordRemoval(this);
            return e;
        }
        prev = e;
        e = next;
    }

    return e;
}
```

6.2 clear()方法
clear方法非常简单，就是遍历table然后把每个位置置为**null**，同时修改元素个数为0
需要注意的是clear方法只会清楚里面的元素，并不会重置capactiy

```java
 public void clear() {
        modCount++;
        Entry[] tab = table;
        for (int i = 0; i < tab.length; i++)
            tab[i] = null;
        size = 0;
    }
```

6.3 containsKey()方法

containsKey方法是先计算hash然后使用hash和table.length取摸得到index值，遍历table[index]元素查找是否包含key相同的值

```
public boolean containsKey(Object key) {
        return getEntry(key) != null;
    }
final Entry<K,V> getEntry(Object key) {
        int hash = (key == null) ? 0 : hash(key.hashCode());
        for (Entry<K,V> e = table[indexFor(hash, table.length)];
             e != null;
             e = e.next) {
            Object k;
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                return e;
        }
        return null;
    }
```

6.4 containsValue()方法

containsValue方法就比较粗暴了，就是直接遍历所有元素直到找到value，由此可见HashMap的containsValue方法本质上和普通数组和list的contains方法没什么区别，你别指望它会像containsKey那么高效

```
public boolean containsValue(Object value) {
    if (value == null)
            return containsNullValue();

    Entry[] tab = table;
        for (int i = 0; i < tab.length ; i++)
            for (Entry e = tab[i] ; e != null ; e = e.next)
                if (value.equals(e.value))
                    return true;
    return false;
    }
```

6.5 Fail-Fast机制

我们知道java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了

map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

fail-fast 机制是java集合(Collection)中的一种错误机制。 当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。

例如：当某一个线程A通过 iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出 ConcurrentModificationException异常，产生 fail-fast 事件。

这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容（当然不仅仅是HashMap才会有，其他例如ArrayList也会）的修改都将增加这个值（大家可以再回头看一下其源码，在很多操作中都有modCount++这句），那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。

```
HashIterator() {
    expectedModCount = modCount;
    if (size > 0) { // advance to first entry
    Entry[] t = table;
    while (index < t.length && (next = t[index++]) == null)
        ;
    }
}
```

在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map：

注意到modCount声明为volatile，保证线程之间修改的可见性。

```
final Entry<K,V> nextEntry() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
```

在HashMap的API中指出：

由所有HashMap类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒在将来不确定的时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程

序错误。

在上文中也提到，fail-fast机制，是一种错误检测机制。它只能被用来检测错误，因为JDK并不保证fail-fast机制一定会发生。若在多线程环境下使用 fail-fast机制的集合，建议使用“java.util.concurrent包下的类”去取代“java.util包下的类”。

6.6 两种遍历方式

第一种效率高,以后一定要使用此种方式!

```
java
Map map = new HashMap();
Iterator iter = map.entrySet().iterator();
while (iter.hasNext()) {
Map.Entry entry = (Map.Entry) iter.next();
Object key = entry.getKey();
Object val = entry.getValue();
}
```

第二种效率低,以后尽量少使用!

```
java
  Map map = new HashMap();
Iterator iter = map.keySet().iterator();
while (iter.hasNext()) {
Object key = iter.next();
Object val = map.get(key);
}
```

七、一些问题

我们现在可以回答开始的几个问题，加深对HashMap的理解：

1. 什么时候会使用HashMap？他有什么特点?
   是基于Map接口的实现，存储键值对时，它可以接收null的键值，是非同步的，HashMap存储着Entry(hash, key, value, next)对象。

2. 你知道HashMap的工作原理吗?
   通过hash的方法，通过put和get存储和获取对象。存储对象时，我们将K/V传给put方法时，

它调用hashCode计算hash从而得到bucket位置，进一步存储，HashMap会根据当前bucket的占用情况自动调整容量(超过Load Facotr则resize为原来的2倍)。获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。如果发生碰撞的时候，Hashmap通过链表将产生碰撞冲突的元素组织起来，在Java 8中，如果一个bucket中碰撞冲突的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。

3. 你知道get和put的原理吗？equals()和hashCode()的都有什么作用？
   通过对key的hashCode()进行hashing，并计算下标(n-1 & hash)，从而获得buckets的位置。如果产生碰撞，则利用key.equals()方法去链表或树中去查找对应的节点。

4. 你知道hash的实现吗？为什么要这样实现？
   在Java 1.8的实现中，是通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在bucket的n比较小的时候，也能保证考虑到高低bit都参与到hash的计算中，同时不会有太大的开销。

5. 如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？
   如果超过了负载因子(默认0.75)，则会重新resize一个原来长度两倍的HashMap，并且重新调用hash方法。

6. Java集合小抄
   以Entry[]数组实现的哈希桶数组，用Key的哈希值取模桶数组的大小可得到数组下标。
   插入元素时，如果两条Key落在同一个桶(比如哈希值1和17取模16后都属于第一个哈希桶)，Entry用一个next属性实现多个Entry以单向链表存放，后入桶的Entry将next指向桶当前的Entry。
   查找哈希值为17的key时，先定位到第一个哈希桶，然后以链表遍历桶里所有元素，逐个比较其key值。
   当Entry数量达到桶数量的75%时(很多文章说使用的桶数量达到了75%，但看代码不是)，会成倍扩容桶数组，并重新分配所有原来的Entry，所以这里也最好有个预估值。
   取模用位运算(hash & (arrayLength-1))会比较快，所以数组的大小永远是2的N次方， 你随便给一个初始值比如17会转为32。默认第一次放入元素时的初始值是16。
   iterator()时顺着哈希桶数组来遍历，看起来是个乱序。
   在JDK8里，新增默认为8的阈值，当一个桶里的Entry超过阈值，就不以单向链表而以红黑树来存放以加快Key的查找速度。

7. 一个很棒的面试题

HashMap的工作原理是近年来常见的Java面试题。几乎每个Java程序员都知道HashMap，都知道哪里要用HashMap，知道Hashtable和HashMap之间的区别，那么为何这道面试题如此特殊呢？是因为这道题考察的深度很深。这题经常出现在高级或中高级面试中。投资银行更喜欢问这个问题，甚至会要求你实现HashMap来考察你的编程能力。ConcurrentHashMap和其它同步集

合的引入让这道题变得更加复杂。让我们开始探索的旅程吧！

先来些简单的问题

“你用过HashMap吗？”“什么是HashMap？你为什么用到它？”

几乎每个人都会回答“是的”，然后回答HashMap的一些特性，譬如HashMap可以接受null键值和值，而Hashtable则不能；HashMap是非synchronized;HashMap很快；以及HashMap储存的是键值对等等。这显示出你已经用过HashMap，而且对它相当的熟悉。但是面试官来个急转直下，从此刻开始问出一些刁钻的问题，关于HashMap的更多基础的细节。面试官可能会问出下面的问题：

“你知道HashMap的工作原理吗？”“你知道HashMap的get()方法的工作原理吗？”

你也许会回答“我没有详查标准的Java API，你可以看看Java源代码或者Open JDK。”“我可以用Google找到答案。”

但一些面试者可能可以给出答案，“HashMap是基于hashing的原理，我们使用put(key, value)存储对象到HashMap中，使用get(key)从HashMap中获取对象。当我们给put()方法传递键和值时，我们先对键调用hashCode()方法，返回的hashCode用于找到bucket位置来储存Entry对象。”这里关键点在于指出，HashMap是在bucket中储存键对象和值对象，作为Map.Entry。这一点有助于理解获取对象的逻辑。如果你没有意识到这一点，或者错误的认为仅仅只在bucket中存储值的话，你将不会回答如何从HashMap中获取对象的逻辑。这个答案相当的正确，也显示出面试者确实知道hashing以及HashMap的工作原理。但是这仅仅是故事的开始，当面试官加入一些Java程序员每天要碰到的实际场景的时候，错误的答案频现。下个问题可能是关于HashMap中的碰撞探测(collision detection)以及碰撞的解决方法：

“当两个对象的hashcode相同会发生什么？”从这里开始，真正的困惑开始了，一些面试者会回答因为hashcode相同，所以两个对象是相等的，HashMap将会抛出异常，或者不会存储它们。然后面试官可能会提醒他们有equals()和hashCode()两个方法，并告诉他们两个对象就算hashcode相同，但是它们可能并不相等。一些面试者可能就此放弃，而另外一些还能继续挺进，他们回答“因为hashcode相同，所以它们的bucket位置相同，‘碰撞’会发生。因为HashMap使用链表存储对象，这个Entry(包含有键值对的Map.Entry对象)会存储在链表中。”这个答案非常的合理，虽然有很多种处理碰撞的方法，这种方法是最简单的，也正是HashMap的处理方法。但故事还没有完结，面试官会继续问：

“如果两个键的hashcode相同，你如何获取值对象？”面试者会回答：当我们调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，然后获取值对象。面试官提醒他如果有两个值对象储存在同一个bucket，他给出答案:将会遍历链表直到找到值对象。面试官会问因为你并没有值对象去比较，你是如何确定确定找到值对象的？除非面试者直到HashMap在链表中存储的是键值对，否则他们不可能回答出这一题。

其中一些记得这个重要知识点的面试者会说，找到bucket位置之后，会调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。完美的答案！

许多情况下，面试者会在这个环节中出错，因为他们混淆了hashCode()和equals()方法。因为在此之前hashCode()屡屡出现，而equals()方法仅仅在获取值对象的时候才出现。一些优秀的开发者会指出使用不可变的、声明作final的对象，并且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。

如果你认为到这里已经完结了，那么听到下面这个问题的时候，你会大吃一惊。“如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？”除非你真正知道HashMap的工作原理，否则你将回答不出这道题。默认的负载因子大小为0.75，也就是说，当一个map填满了75%的bucket时候，和其它集合类(如ArrayList等)一样，将会创建原来HashMap大小的两倍的bucket数组，来重新调整map的大小，并将原来的对象放入新的bucket数组中。这个过程叫作rehashing，因为它调用hash方法找到新的bucket位置。

如果你能够回答这道问题，下面的问题来了：“你了解重新调整HashMap大小存在什么问题吗？”你可能回答不上来，这时面试官会提醒你当多线程的情况下，可能产生条件竞争(race condition)。

当重新调整HashMap大小的时候，确实存在条件竞争，因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。这个时候，你可以质问面试官，为什么这么奇怪，要在多线程的环境下使用HashMap呢？：）

热心的读者贡献了更多的关于HashMap的问题：

为什么String, Interger这样的wrapper类适合作为键？ String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个field声明成final就能保证hashCode是不变的，那么请这么做吧。因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。

我们可以使用自定义的对象作为键吗？ 这是前一个问题的延伸。当然你可能使用任何对象作为键，只要它遵守了equals()和hashCode()方法的定义规则，并且当对象插入到Map中之后将不会再改变了。如果这个自定义对象时不可变的，那么它已经满足了作为键的条件，因为当它创建之

后就已经不能改变了。

我们可以使用CocurrentHashMap来代替Hashtable吗？这是另外一个很热门的面试题，因为ConcurrentHashMap越来越多人用了。我们知道Hashtable是synchronized的，但是ConcurrentHashMap同步性能更好，因为它仅仅根据同步级别对map的一部分进行上锁。ConcurrentHashMap当然可以代替HashTable，但是HashTable提供更强的线程安全性。看看这篇博客查看Hashtable和ConcurrentHashMap的区别。

我个人很喜欢这个问题，因为这个问题的深度和广度，也不直接的涉及到不同的概念。让我们再来看看这些问题设计哪些知识点：

- hashing的概念
- HashMap中解决碰撞的方法
- equals()和hashCode()的应用，以及它们在HashMap中的重要性
- 不可变对象的好处
- HashMap多线程的条件竞争
- 重新调整HashMap的大小

总结

HashMap的工作原理
HashMap基于hashing原理，我们通过put()和get()方法储存和获取对象。当我们将键值对传递给put()方法时，它调用键对象的hashCode()方法来计算hashcode，让后找到bucket位置来储存值对象。当获取对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。HashMap使用链表来解决碰撞问题，当发生碰撞了，对象将会储存在链表的下一个节点中。HashMap在每个链表节点中储存键值对对象。

当两个不同的键对象的hashcode相同时会发生什么？ 它们会储存在同一个bucket位置的链表中。键对象的equals()方法用来找到键值对。