HashMap底层的实现（hashmap和hashset的关系）

----------哈希表底层使用的数组和普通数组的差别----------

①：虽然哈希表底层实现使用数组（之所以不用链表的原因是考虑到内存的集中利用，猜的），但是这个数组使用下标遍历没有意义，而且数组中元素分布不均匀，不能使用数组的遍历方法，哈希表访问数据的方式也是下标，但是它的下标是不连续的，即下标之间没有串联起来，互不通讯，不能使用下标递增的方式访问，只能通过散列函数算得hash值（过大的整数），然后再根据数组长度结合hash值求得下标，通过这个下标才得以访问里面的数据；需要注意的是，在底层判断Key是否重复的时候考虑到了解决散列冲突的链地址法，过程是先通过key值的hash值结合数组长度算出下标值，然后到数组的该下标下找是否已经存在key值，这里使用的遍历其实就是遍历链地址法的链表，会比较key值的hashcode值和values值，如果相等认为是相同对象，则把key值所对应的老的value值替换并返回；

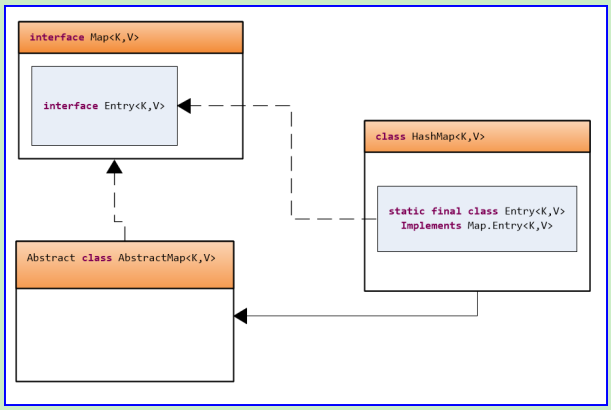
1、底层的实现

HashMap底层使用数组实现的，创建了一个Entry<K,V>类型的数组Entry<K,V>[ ]，即table数组；

```
static final Entry<?,?>[] EMPTY_TABLE = {};

/**
 * The table, resized as necessary. Length MUST Always be a power of two.
 */
transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY_TABLE;
```

而Entry<K,V>是HashMap的内部类（**和c++的链表的实现非常相似**），**hash**属于它的属性，next类似于指针，是为了给链地址法解决散列冲突准备的（详情见下面），但是使用数组实现的，（不使用链表是为了考虑内存的利用，集中管理内存）



哈希表的存储过程：

1）哈希表底层是一个数组，我们必须知道集合中的一个对象元素到底要存储到哈希表中的数组的哪个位置，也就是需要一个下标。

（由于对象调用hashCode（）函数生成的一个过大的整数，但是怎么可能作为数组的下标呢？

在底层会根据生成的过大的哈希值（其实就是一个大的int类型的数据）结合数组长度计算出对应的空间下标；）

2）哈希表会根据集合中的每个对象元素的内容计算得出一个整数值。由于集合中的对象元素类型是任意的，而现在这里使用的算法必须是任意类型的元素都可以使用的算法。能够让任意类型的对象元素都可以使用的算法肯定在任意类型的对象所属类的父类中，即上帝类Object中，这个算法就是Object类中的hashCode()函数。

结论：要给HashSet集合中保存对象，需要调用对象的hashCode函数（注意下面的源代码使用的短路与）。

【内部类结构】

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next; //保存下一个Entry节点的地址，供迭代器和查找使用
    int hash;  //保存此节点的哈希值,判断两个节点是否相同，先判断hash值是否
               相同，若相同，再判断equals值是否相等

    /**
     * Creates new entry.
     */
    Entry(int h, K k, V v, Entry<K,V> n) {
        value = v;
        next = n;
        key = k;
        hash = h;
    }
```

hash值（过大的整数）就是通过hashcode( )计算的。下面将演示如何将一个元素插入到哈希表中:

```
public V put(K key, V value) {
    if (table == EMPTY_TABLE) {
        inflateTable(threshold);
    }
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key);
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;        //比较两个元素用到了hashcode和equals
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    modCount++;
    addEntry(hash, key, value, i);  遍历上面的哈希表，如果没有重复的key，那就添加一个
    return null;
}
```

先判断哈希表是否为空表，为空做处理；

接着判断插入的key值是否为null，做特殊处理；

通过内部的函数hash函数得到hash值，此函数里面调用了hashcode（）函数，源代码如下：

```
final int hash(Object k) {
    int h = hashSeed;
    if (0 != h && k instanceof String) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode();

    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

得到hash值（过大的整数），**然后在底层会根据生成的过大的哈希值（其实就是一个大的int类型的数据）结合数组长度计算出对应的空间下标**；这是通过indexFor（）函数实现的：

```
static int indexFor(int h, int length) {
    // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";
    return h & (length-1);
}
```

然后得到下标值 i 值，接着在哈希表中找是否已经有与要插入的Key值相等的键值，遍历一遍，（注意，这里的遍历不是遍历整个哈希表，而是遍历为了解决冲突的链地址法的链表，所以才使用链表的遍历方法，）比较key的hash值和equals关系（短路与），

如果key值通过上述方法比较相同的话，用key值对应的value值覆盖原来的老值，并返回老值；

如果没有，就添加一个新的Entry<K,V>节点；addEntry()代码如下：

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize(2 * table.length);
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = indexFor(hash, table.length);
    }

    createEntry(hash, key, value, bucketIndex);
}
```

先对要插入的数组的容量进行判断，如果长度不够，扩容，然后重新根据数组的新长度和hash值计算下标（此时已经判断过key值的重复性了，所以这里的key值是不重复的，也就是新的Key值），

然后调用createEntry（）函数，源代码如下：

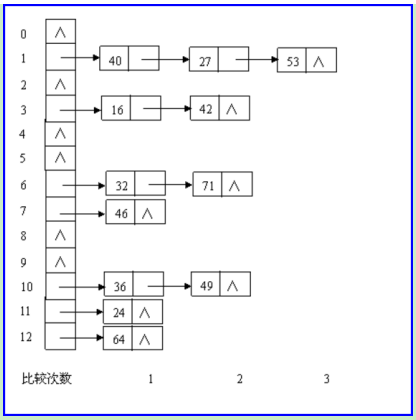
```
void createEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<>(hash, key, value, e);
    size++;
}
```

注意这里的赋值语句：Entry<K,V> e = table[bucketIndex]，是把table[bucketIndex]变量里储存的地址值（以前存在的节点，有可能是一个Entry链表的头地址或者只有一个节点的链表头地址，或者为null，散列冲突）赋值给e了，而不是这个盒子的地址（table[bucketIndex]地址），所以不会关联下面的修改；

这里的bucketIndex就是要插入的key值通过计算得到的i下表值，然后调用Entry()函数创建新节点，调用上面的内部类的构造函数，完成插入：

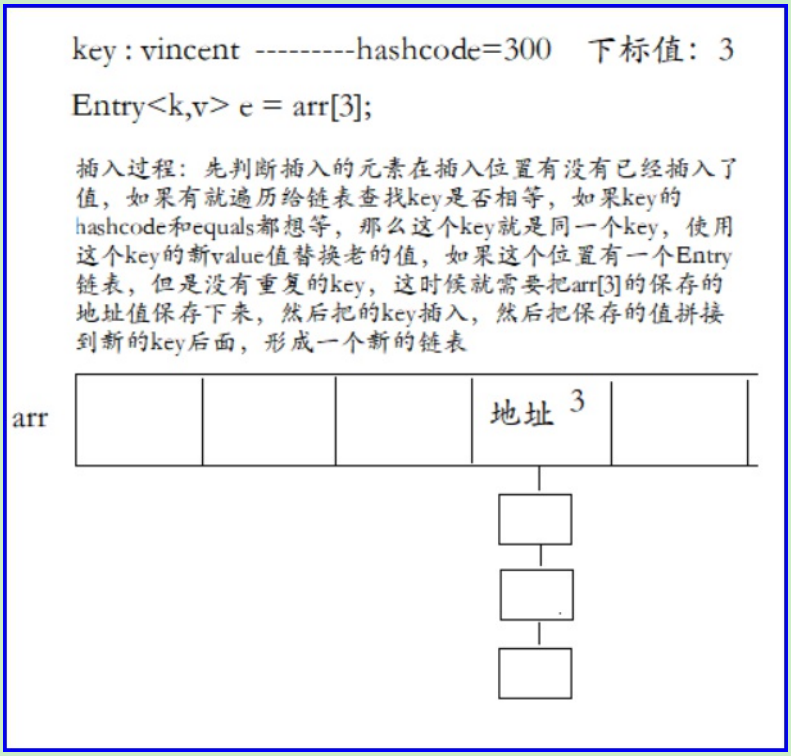
```
Entry(int h, K k, V v, Entry<K,V> n) {
    value = v;
    next = n;
    key = k;
    hash = h;
}
```

链地址法处理散列冲突：




```
void createEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<>(hash, key, value, e);
    size++;
}
```

再找个算法中包含了链地址法处理过程，如果要插入的**Entry**节点的通过**hashcode**计算得到的下标值在**table**数组的某个位置，此位置上保存的是插入的**Entry**的地址，如果在插入时，此前已经判断过了重复**key**的问题，此位置已经有了**Entry**节点或者**Entry**链表节点的链首地址，那么先把这个地址保存到上述的**e**中，然后插入新节点，再插入新节点的时候把保存下来的**Entry**链（有可能为空）接上去，



2、hashmap和hashset的关系

hashset的底层是通过hashmap实现的：源码如下：

```
public HashSet() {
    map = new HashMap<>();
}
```

它只保留了key值，用一个虚拟的值代替value值，例如它的添加函数源码如下：

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

如果此前哈希表中不存在key值，看上面的put函数，返回null，null == null，结果为true;

否则为false(存在就会覆盖原来的value，返回老的value值，老的value == null成立与否。一般不成立);

set集合可以存储null但只能有一个（key值）；

PRESENT值是一个虚拟的值

```
// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();
```

3、自定义类型的插入：

在hash系的集合中，底层实现原理都是使用hashmap实现的，在HashMap进行插入的时候都要进行Key的比较，比较规则如下：

```
for (Entry<K,V> e = table[i]; e != null; e = e.next) {
    Object k;          //比较两个元素用到了hashcode和equals
    if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
```

先比较hash值，hash函数内部调用了hashcode函数，如果相同，则比较equals方法，注意这里使用了短路与，就是说equals方法不一定调用，所以自定义类型的插入一定要重写的两个方法是：

hashCode( )

equals( )方法；