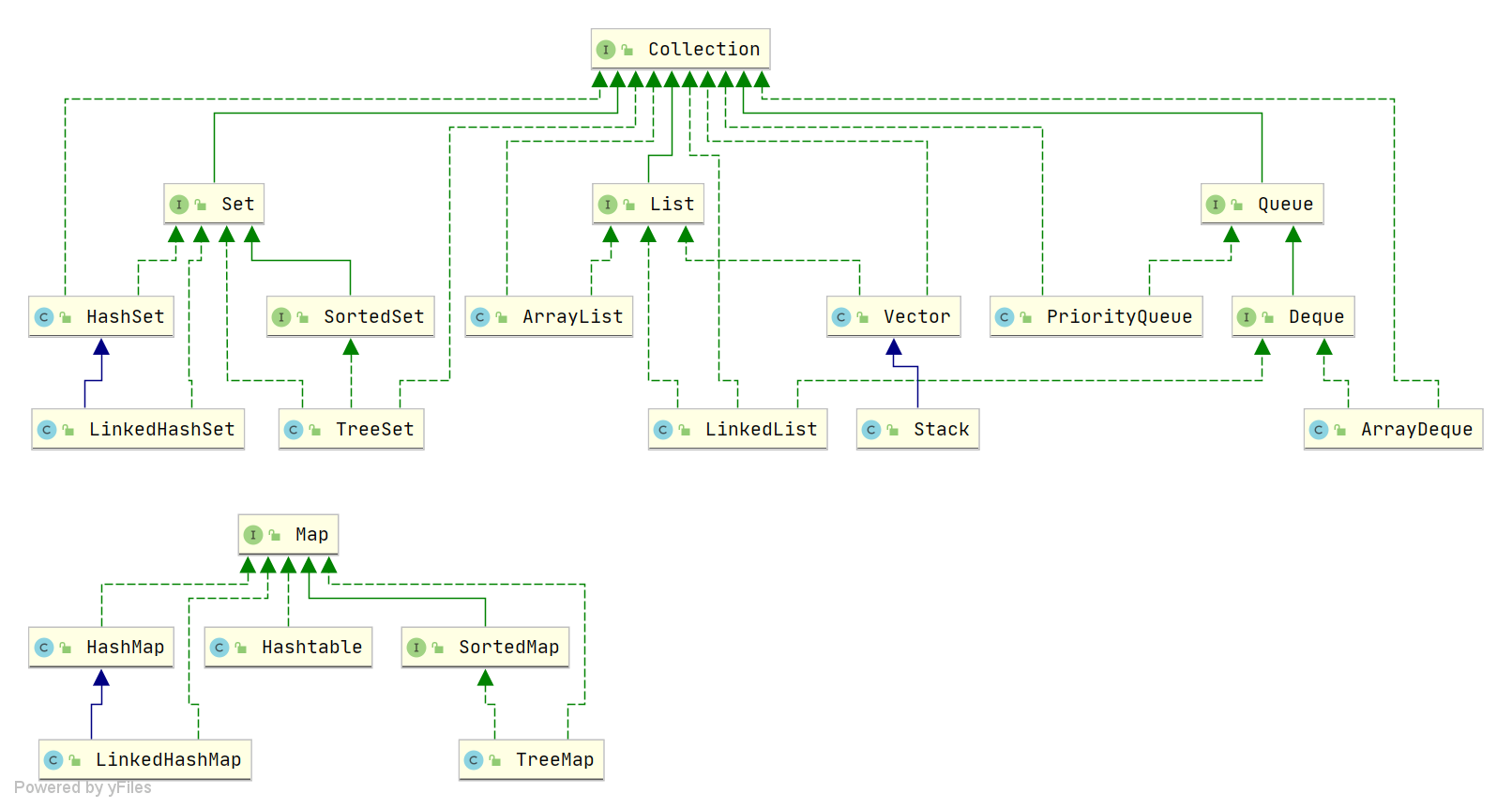
ArrayList:默认10 扩容1.5

Vector：10 2 可以指定扩容量

HashMap：16 2

Hashtable：11 2倍+1

StringBuilder：16 如果参数是String，则str.len+16 扩容2倍+2



ArrayList：

底层实现是数组Object[] elementData，初始容量10，如果指定了capacity(有参构造)，则会创建指定大小的Object数组。属性中有两个已经初始化的Object数组，

DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA和EMPTY\_ELEMENTDATA，如果采用无参构造，则懒惰初始化，elementData指向前者，如果elementData已经初始化但是容量是0(比如有参构造指定容量为0或者用参数为集合的构造方法但集合中元素个数为0的构造方法初始化)，则elementData指向后者。只有无参构造才是懒惰初始化，指定了容量大小的构造不是懒惰初始化，会直接创建数组。

属性：

1. int DEFAULT\_CAPACITY = 10 //默认容量
2. Object[] EMPTY\_ELEMENTDATA = {} //数组已经指定了大小，但元素个数为0时的指向
3. Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {}//数组默认容量初始化时的指向
4. Object[] elementData //数组的引用
5. int size //数组中元素的个数 不是数组的大小

构造方法：

1. 无参构造，使用默认容量，但调用无参构造不会立即创建一个大小为10的数组，而是懒惰初始化，指向类中一个已经初始化的空数组DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA，等第一次添加元素时才会创建数组。
2. 带容量的构造，这种情况就不是默认的了（不管容量参数是否为0），因为你已经指定了容量，如果容量为0，数组为EMPTY\_ELEMENTDATA(这才是真正意义上的空数组)，如果容量大于0，则就创建该容量的数组。
3. 参数为集合的构造，集合元素个数为0，elementData指向空数组，个数大于0，则将集合元素拷贝成数组。

扩容：

扩容为原来的1.5倍(oldCapacity + (oldCapacity >> 1))，如果大小还是不够则直接指定为需要的大小。如果扩容后的容量超过上限(int的最大值-8)，如果需要的容量大于上限，则将容量设置为int的最大值，否则设置成上限。

成员方法：

1. clear(), 将elementData中元素全部设置为null，但是数组的容量不变。
2. add(int index, E element), 先确定是否需要扩容，然后使用arraycopy将index后面的元素全部后移一位，然后将指定元素插入。
3. add(E e)，先确定是否需要扩容，然后将元素插入数组尾部。
4. addAll(Collection<? extends E> c)，将集合c先转为数组，确定是否扩容，然后将集合通过arraycopy拷贝到数组尾部。
5. 添加元素的时候需要判断是否需要扩容，首先要得到数组需要的大小minCapacity(当前元素个数+添加的元素个数)，如果elementData指向DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA，则minCapacity=max(10, minCapacity), 如果当前数组大小length < minCapacity则需要扩容。
6. grow(int minCapacity)，将数组大小扩容为原来的1.5倍，如果还是小于minCapacity, 则数组大小直接指定为minCapacity。如果扩容后的大小newCapacity超过上限，则判断minCapacity和上限的大小，如果超过上限，则newCapacity指定为int的最大值，如果没超过上限，则newCapacity指定为上限。
7. remove(int index)，将index后元素通过arraycopy往前移动一位。最后一个元素设置为null

LinkedList：双向链表(JDK1.6 之前为循环链表，JDK1.7 取消了循环)

属性：

1. private static class Node<E> //有前驱和后继
2. Node<E> first
3. Node<E> last
4. int size = 0

构造方法：

1. 无参构造：空构造

参数为集合的构造：先调空构造，再将集合元素添加到链表尾部。  
成员方法：  
无

Vector：线程安全 加了锁  
属性：  
1. protected Object[] elementData

1. protected int elementCount //size
2. protected int capacityIncrement //每次扩容时增加的量

构造方法：  
1. public Vector() 采用默认容量10的初始化，不是懒惰初始化，直接创建大小为10的数组。

1. public Vector(int initialCapacity) 指定容量，直接创建数组。
2. public Vector(int initialCapacity, int capacityIncrement) 指定容量并指定每次扩容的量

成员方法：

1. 扩容：如果capacityIncrement 为0则扩容为原来的2倍，否则每次增加capacityIncrement 如果大小还是不够则直接指定为需要的大小。如果扩容后的容量超过上限(int的最大值-8)，如果需要的容量大于上限，则将容量设置为int的最大值，否则设置成上限。

HashSet：

属性：

1. private transient HashMap<E,Object> map
2. private static final Object PRESENT = new Object(); //map中的value都指向这个哑元对象。

构造方法：

1. public HashSet() 初始化内部的HashMap。
2. public HashSet(int initialCapacity)
3. public HashSet(int initialCapacity, float loadFactor)
4. HashSet(int initialCapacity, float loadFactor, boolean dummy)

成员方法：  
基本都是直接调用map 的方法。

HashMap：

可以存储null key和value，但NULL key只能有一个。不是线程安全的集合。

属性：

1. Node<K,V>[] table; //Node数组引用 数组大小总是2的幂次倍
2. Set<Map.Entry<k,v>> entrySet; //Node集合引用 将所有的Node结点放到这个集合中，其中Map.Enrty<k,v>是Map接口中的一个接口，Node就是实现了这个接口。
3. int size; //键值对的个数
4. static class Node<K,V> //Node内部类，其中存放了key value next hash（创建结点时通过key计算出来的）
5. static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; //数组默认容量16
6. static final int MAXIMUM\_CAPACITY = 1 << 30; //数组最大容量
7. static final int TREEIFY\_THRESHOLD = 8; //当链表元素个数大于8时转为红黑树
8. static final int UNTREEIFY\_THRESHOLD = 6; //当红黑树元素个数小于6时转为链表
9. static final int MIN\_TREEIFY\_CAPACITY = 64; //当数组容量小于64时，链表不会转为红黑树
10. static final float DEFAULT\_LOAD\_FACTOR = 0.75f; //默认负载因子0.75 当大小为100的数组存放第76个元素时就需要resize。
11. final float loadFactor; //当前负载因子
12. int threshold; //临界值(容量\*填充因子) 当实际大小超过临界值时，会进行扩容。开始初始化map时会懒惰初始化，将你指定的容量通过tableSizeFor得到最小2的幂次方之后，不会立即创建数组，而是将他赋给threshold，所以刚开始的threshold存储的是数组还未初始化时的容量。到添加元素扩容的时候threshold才会变成阈值。

构造方法：

1. HashMap()不会设置threshold的值，这个值默认为0，懒惰初始化第一次初始化时会根据threshold来判断，如果threshold=0，说明你构造时没有指定容量，他就会用默认容量
2. HashMap(int initialCapacity, float loadFactor) 和ArrayList不同，他不会用你指定的容量，而是调用tableSizeFor函数将你指定的容量变为大于他并且最接近的2的幂次方，然后将这个n值赋给threshold，懒惰初始化第一次初始化时，判断threshold这个值，如果大于0，说明你指定容量了，他就不会用默认容量，而是用你这个threshold

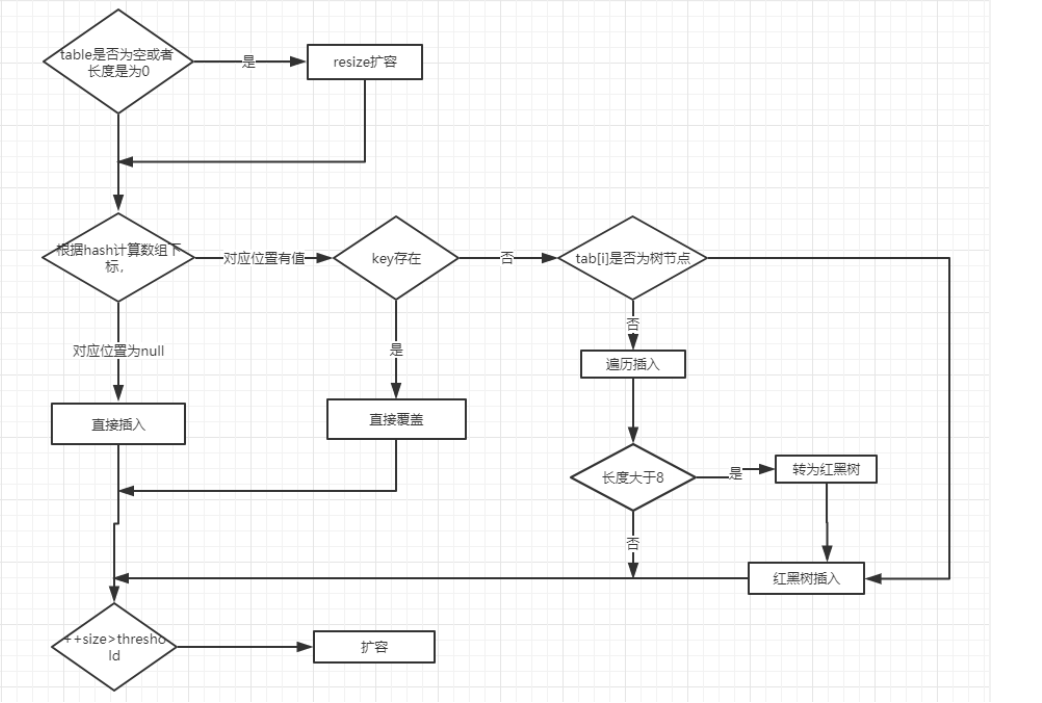
综上，构造方法中的threshold只是代替保存第一次初始化时的容量，方便懒惰初始化

成员方法：

1. V get(Object key) //内部调用了getNode方法
2. Node<K,V> getNode(int hash, Object key) //通过hash值找到位置，然后再找到key。HashCode相同的对象不一定equals，equals的对象hashcode一定相同，在getNode时，先通过hashCode找到存放的桶的位置，然后依次比较hashCode，如果hashCode都不同，则一定不equals，直接跳到下一个元素。如果hashCode相同，则再比较内存地址是否相同(==)，如果相同则直接返回，如果不同则用equals方法比较。使用hashCode比较是为了快速排除元素，使用(==)比较是因为==效率比equals高，内存地址相同了就可以直接返回了。最后前面两个比较都的不到结果，才会不得已使用equals来比较。一切都是为了效率。
3. int hash(Object key) //获得key对应的hash值，是通过key的hashCode()再通过扰动得到的，(h = key.hashCode()) ^ (h >>> 16)，通过异或运算扰动了一次。得到的hash值存放到Node节点中存储的hash属性。为了保证相同的key存放在同一位置，需要重写key的hashCode()方法，使得相同的key对应的hashCode()相同，经过扰动后得到的hash也随之相同，在数组中存放的位置也随之相同。

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

1. static final int tableSizeFor(int cap) //得到大于等于cap的最小2的幂次方。
2. V put(K key, V value) //直接调用了putVal()方法



1. V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict)
2. Node<K,V>[] resize() //扩容方法 方法中的四个变量oldCap，newCap，oldThr，newThr，旧/新容量，旧/新阈值。如果oldCap为0(懒惰初始化)，说明这是初始化之后第一次resize，

如果oldThr不为0说明初始化时指定了容量，因为初始化指定的容量会存放在threshold中。如果oldThr为0，说明是无参构造，则newCap会用默认容量(16)。如果oldCap不为0，则map中已经有元素且超过阈值需要扩容，如果oldCap没有超过上限，则newCap和newThr都扩容为原来的2倍，如果超过上限，则直接返回(任由数组冲突，阈值设为int最大值，不管了)。如果oldCap不为null，则还需要将原来的数据转移到新数组(遍历数组)。

要点：

1. Node在数组中位置：(n - 1) & hash(key) //n是数组的长度
2. Node中存放的hash=hash(key.hashCode)
3. loadFactor 加载因子：loadFactor 加载因子是控制数组存放数据的疏密程度，loadFactor 越趋近于 1，存放元素时会造成更多冲突，导致链表长度增加，查找效率低，数组利用率高。loadFactor 越小，也就是趋近于 0，数组的利用率低，查找效率高。
4. 一个类中的hashCode方法针对的是这个类所创建的对象，比如HashMap中的hashCode方法针对的是你创建的map对象。而HashMap中的hash方法存在的意义：将key的hashCode扰动一下，得到key的hash值，这个hash值会存放在Node类中的hash属性上，但是Node类中也有一个hashCode()方法，这个方法是针对Node对象的，而hash属性是针对key的，这个hash值和n-1做与运算得到Node在数组中的位置。

1.7和1.8的区别：

1. JDK1.8 之前 HashMap 由 数组+链表 组成的， JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。当红黑树结点个数少于6时又会转为链表。
2. 在Java7叫Entry在Java8中叫Node。
3. 1.7采用头插法，1.8采用尾插法。
4. 1.8的hash方法比1.7更简单但是原理不变。1.8的性能更好。