参考：

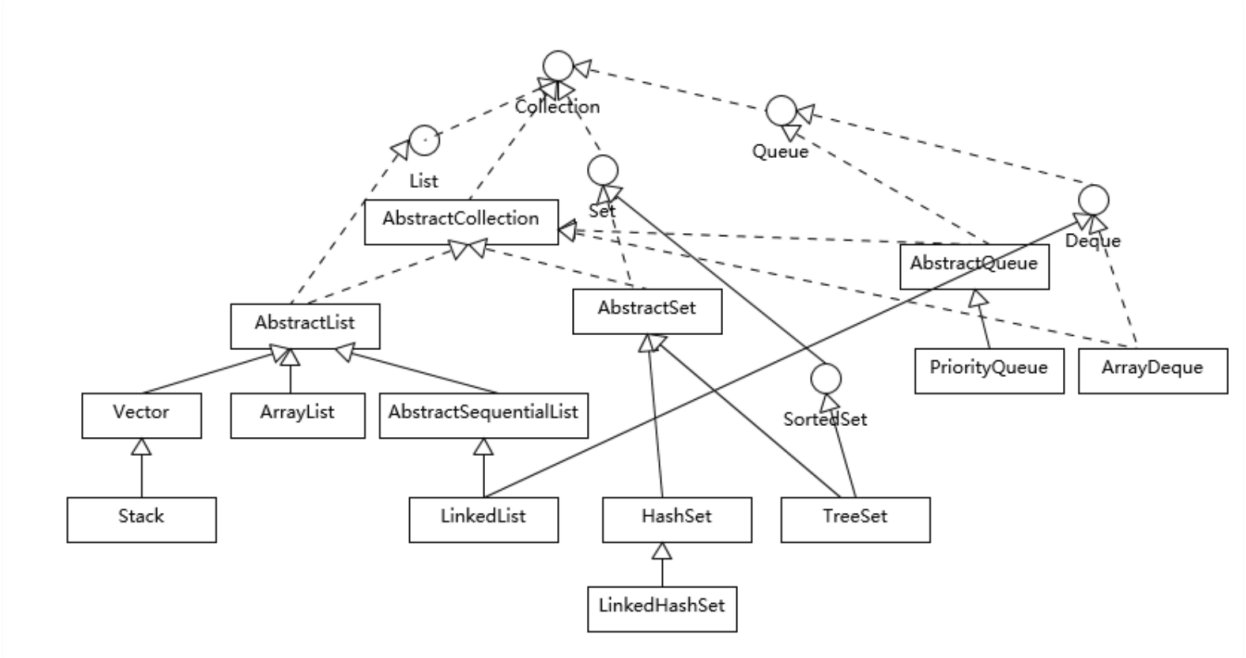
JDK1.8 HashMap源码分析

前言

最近因看了Java核心36讲整整有三讲是用来讲集合相关的知识点的，然后就想着这边统一慢慢整理一下，因为集合的概念太广泛，要整理的比较详细自然篇幅也会非常的长，这里反正就先按部就班的一点点整理。

Collection家族

先上一下这个家族的类图。



总体可以看出Collection又划分出三个子家族：List，Set和Queue。

List家族

这个家族就主要讨论一下Vector，ArrayList和LinkedList。

Vector平常可能用的都不太多，它通过数组实现，扩容默认是按一倍不断扩充，默认创建的话是一个容量为10的数组，它是线程安全的，不过通过synchronized实现线程安全（效率自己体会咯）。

ArrayList则是大家经常使用的，它通样通过数组实现，扩容按50%不断扩充，默认创建的话是一个容量为0的数组，不过第一次进行add时，则扩容为10，它不是线程安全的。

LinkedList通过双向链表实现，既然是链表，就不谈扩不扩的问题了，它也不是线程安全的。

总体来说，Vector和ArrayList是由数组实现，自然在随机访问上的效率大大高于基于链表实现的LinkedList，同时在插入和删除上比不过LinkedList（我认为不涉及数组扩容时的尾插和任何场景的尾删效率两者应该不会相差过大）。

可以看到三者其实都继承了抽象类AbstractList，这里面实现了各个List操作的通用部分。

那么开始解析源码。

Vector默认创建容量10的数组？

```
public Vector() {
    this(10);
}
```

```
public Vector(int initialCapacity) {
    this(initialCapacity, 0);
}
```

```
public Vector(int initialCapacity, int capacityIncrement) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    this.elementData = new Object[initialCapacity];
    this.capacityIncrement = capacityIncrement;
}
```

一波三折，可以看到最后就是指定了初始容量为10。

Vector默认按1倍不断扩容？为了精简篇幅，我们截选关键处的代码。

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                     capacityIncrement : oldCapacity);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

该方法就是Vector的扩容方法，可以看到这里新的容量其实就是2*oldCapacity（默认创建时capacityIncrement是0），所以默认情况就是按1倍扩容。

ArrayList默认创建先是一个容量0的数组，再是个容量10的数组？

```
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

```
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
```

可以看到，我们创建时就是一个0容量的数组，至于为啥初始是10呢，看add操作，这边就一步看。

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

进入ensureCapacityInternal方法。

```
private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}
```

不着急，我们先看参数的处理。

```
private static int calculateCapacity(Object[] elementData, int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity;
    }
```

这一步我们可以看出，若数组是DEFAULTCAPACITY_EMPTY_ELEMENTDATA我们就将DEFAULT_CAPACITY的值给minCapacity。

```
private static final int DEFAULT_CAPACITY = 10;
```

默认的就是10，这时候我们再进ensureExplicitCapacity看。

```
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

这个grow方法自然和Vector的一样是扩容方法。

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

看这一步就清楚了，minCapacity刚刚传的是10，原本是0，所以把10赋给newCapacity，相当于ArrayList创建后不加数据是个0容量的，第一次加之后就扩容为10。

ArrayList按50%扩容？依旧看grow方法。

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

右移一位相当于啥？除以2呗，是不是新的容量就是1.5oldCapacity，所以是按50%扩容的，而且可以看到ArrayList不像Vector可以自己设置扩容量，这边必须按50%扩容。

思考一个问题：List家族中又有线程安全与不安全的区别，又有查询快速和插入、删除快速的区别，这里有个场景，如何去处理超高并发的无锁缓存？

Set家族

这个家族是无序的集合（不要误解TresSet的自然排序，那是把插入的值自动排序，所以相对你插值的顺序也是无序的），是不允许有重复元素的 。

他主要有仨成员，HashSet，LinkedHashSet，TreeSet。

HashSet利用了哈希算法，理想情况下，哈希散列正常，添加、删除等等操作都是常数时间的，它是完全无序的。

LinkedHashSet，与HashSet不同的是它内部构建了一个记录插入顺序的双向链表，所以额外提供了按照插入顺序遍历的能力，除了要额外维护这张链表，其余与HashSet无多少区别。

TreeSet实现了自然排序，插入的结果会按顺序排序，所以添加、删除等操作效率较低（O(log(n)）。

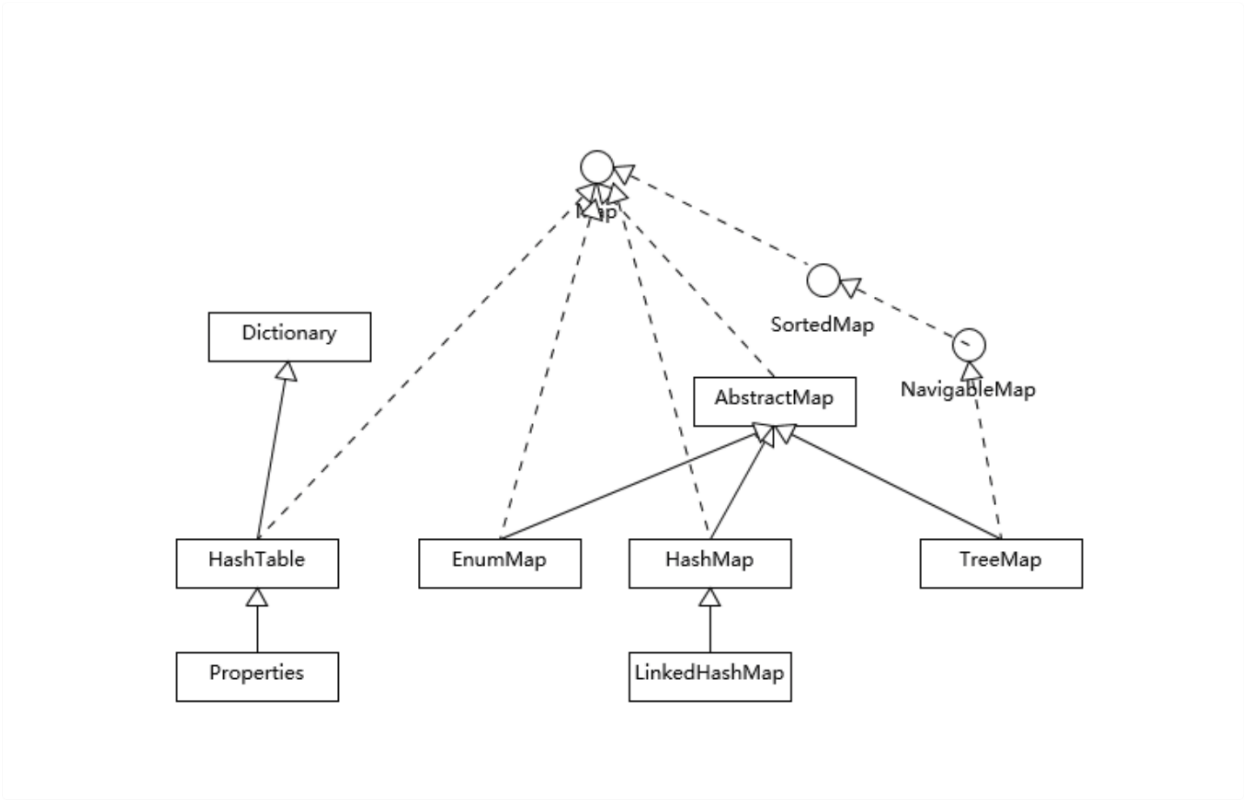
前两者通过HashMap存值，后者通过TreeMap存值（所以存值的讲解等后面的Map整理）。

Queue家族

它是队列结构的实现，支持FIFO或者LIFO，区别于前两者放在util包中，Queue通常出现在并发场合，所以被放置在并发包中。

Map家族

先上这个家族的类图。



这里面主要要了解的就是HashTable、HashMap、TreeMap，后两者我们在Set中也有提及，往下篇幅就开始深入到里面去了解吧。

HashTable是早期Java提供的一个哈希表（散列表，先整理，看有没有需求要对哈希表整理一番）的实现，它是同步的（当然是通过synchronized的方式实现的，性能自己体会），它不支持null键和值。

HashMap是现在最为广泛使用的哈希表的实现，可以看作是HashTable的加强，取消了同步，并且键和值都支持null，通常情况put和get能达到常数时间的性能。

既然HashMap是通常情况下最好的选择，我们来详细研究一下。

我们先看看默认的构造函数。

```
/**
 * Constructs an empty <tt>HashMap</tt> with the default initial capacity
 * (16) and the default load factor (0.75).
 */
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
```

其实是一脸懵逼的，首先它说是初始容量为16，还有个啥负载系数0.75。然后这个构造函数只对负载系数进行了初始化（关键负载系数是干啥的啊）。

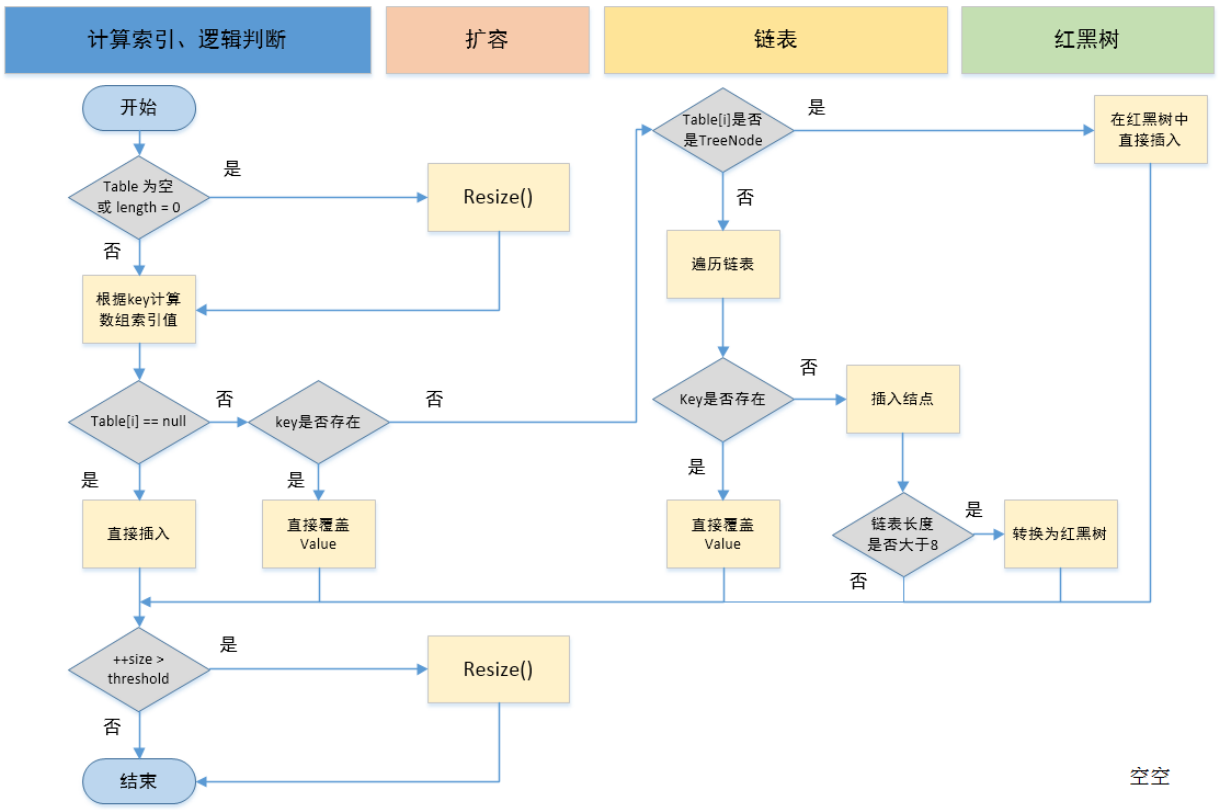
既然构造函数看不懂啥，我们去看看HashMap的put方法看看它是怎么加值的。

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```

```
     * @param hash hash for key
     * @param key the key
     * @param value the value to put
     * @param onlyIfAbsent if true, don't change existing value
     * @param evict if false, the table is in creation mode.
     * @return previous value, or null if none
     */
    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        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;
                }
            }
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

第一个if分支就是解读初始化容量的操作，关键处就在于resize方法。这个方法是HashMap重要的扩容机制的实现，它实现了啥呢，如果表格是0创建初始存储表格，容量不满足需求时jji进行扩容（resize扩容机制看看后面是不是补充下）。

继续解读putVal方法，说实话，这一长串代码看的也头大，我在网上找了张流程图。

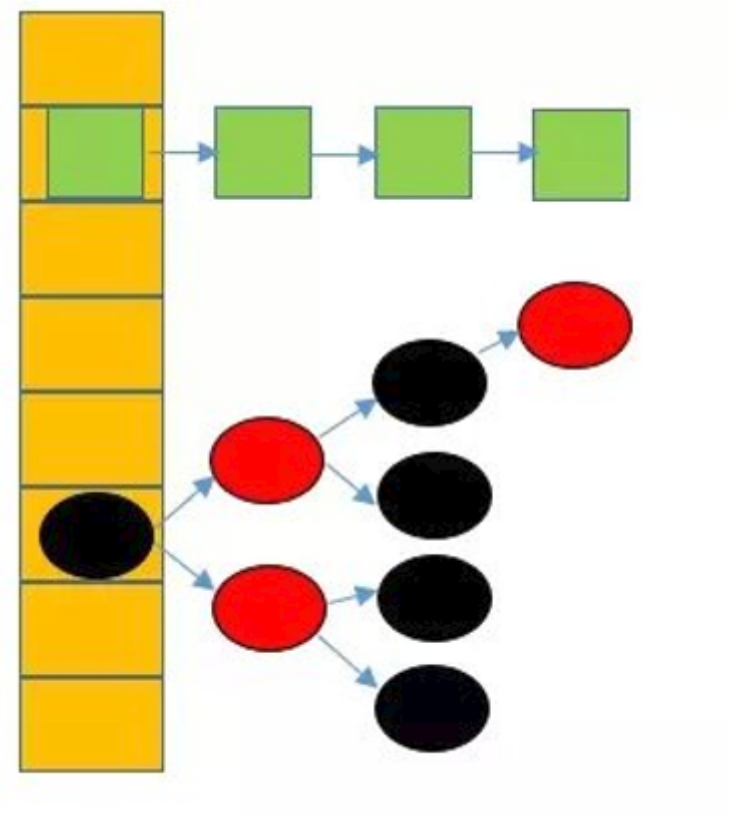


对比这张图结合代码花点时间应该都能看懂七七八八了，针对putTreeVal以及treeifByBin的方法深入还未深入。途中红黑树的我不太懂（头疼），由于key存在而覆盖value并不走途中的分支，而是这个if直接返回旧值。

```
}
if (e != null) { // existing mapping for key
    V oldValue = e.value;
    if (!onlyIfAbsent || oldValue == null)
        e.value = value;
    afterNodeAccess(e);
    return oldValue;
}
```

putVal的最后几句其实挺有意思的，它的扩容不是说现在没空间了，现在才扩容，threshold的值是容量*负载系数（前文提过的没搞懂的初始化参数），负载系数默认是0.75，所以在一定容量的时候就开始扩容，具体默认设置的这个0.75有啥含义等有时间我再去查查。

头已经有点大了，我想会接下来怎么开始。先盗一张别人的图形象的描述下HashMap在jdk1.8的提现吧。



很简单，链表长度小于8时则以链表形式存在，否则就转为红黑树，主要是考虑到链表长度过长时，查询时间会大大降低。

resize方法决定就在这边开始研究下吧，扩容机制也是机器重要的。

```
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) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        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);
        }
        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) {
            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;
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        if (loTail != null) {
                            loTail.next = null;
                            newTab[j] = loHead;
                        }
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
```

```
    return newTab;
}
```

太长不想看系列，难受啊。总体来说分上下两部分，上部分是未初始化初始化容量，下部分是超过阈值（threshold）开始扩容，按2倍扩容。