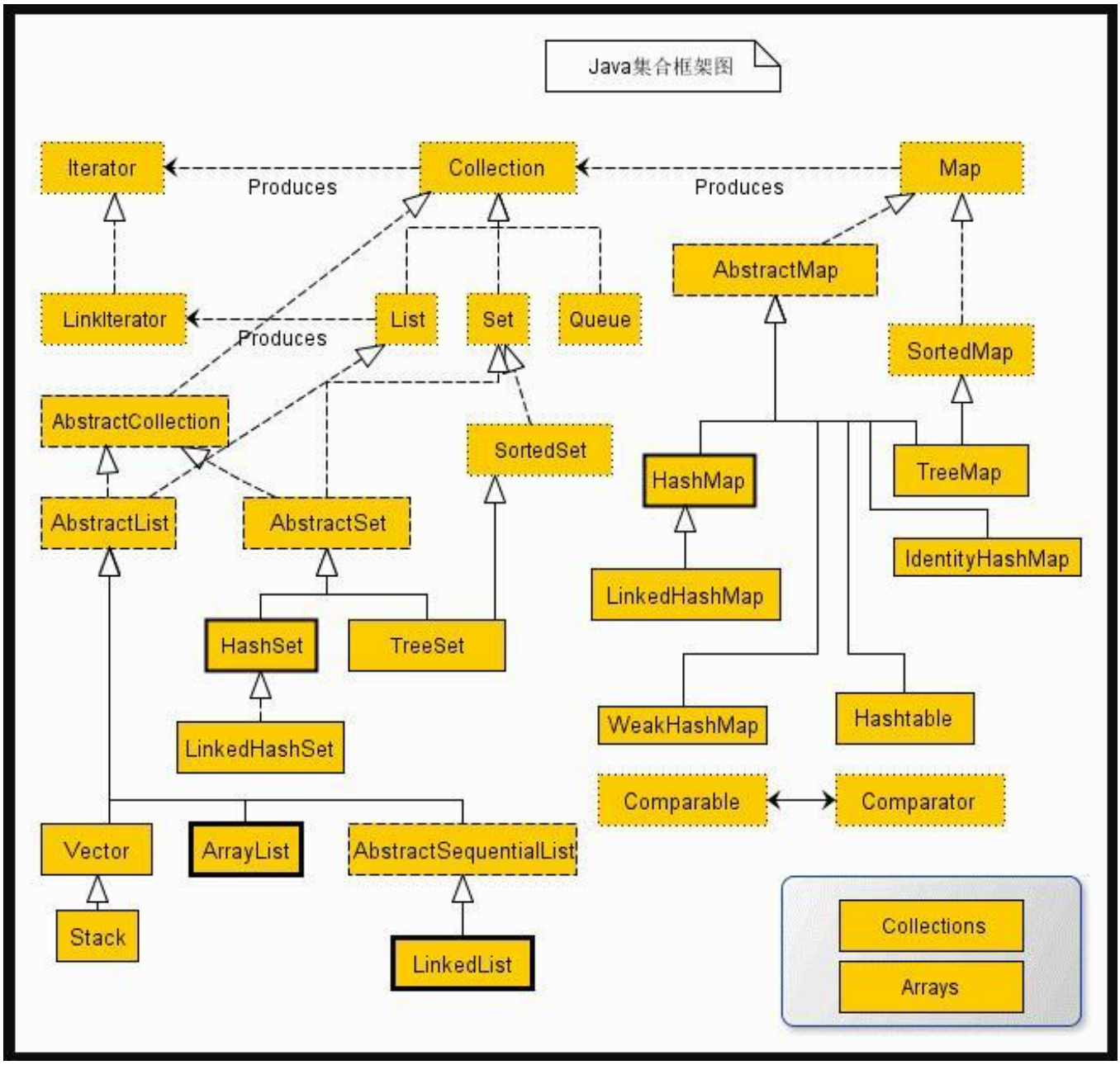
JDK8

集合源码学习

方小白

fangjiaxiaobai@163.com



jdk 源码学习一 java.util.ArrayList (since 1.2)签名

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

可序列化，可以支持快速的随机访问，可以被克隆。

一、成员变量

```
private static final int DEFAULT_CAPACITY = 10; // 默认大小为 10
```

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

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

```
transient Object[] elementData;  // arrayList 中的数据
```

```
private int size;  // 当前状态下的 arraylist 中的数据量。
```

```
Protect transient int modCount = 0; // 记录被修改的次数。
```

二、构造方法

```
public ArrayList(int initialCapacity) {  // 可以指定容量的大小。
        if (initialCapacity > 0) {  //如果指定了容量大小，那么就会创建指定容量的 list
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }
```

```
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA; //使用默认的数据大小
}//但是，DEFAULTCAPACITY_EMPTY_ELEMENTDATA 的大小也是 0，为什么呢？是在第一次调用 add 的时候将其初始化的。见 add 方法。
```

```
public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        if ((size = elementData.length) != 0) {
            // c.toArray might (incorrectly) not return Object[] (see 6260652)
            if (elementData.getClass() != Object[].class)
                elementData = Arrays.copyOf(elementData, size, Object[].class);
        } else {
            // replace with empty array.
            this.elementData = EMPTY_ELEMENTDATA;
        }
    }
```

三、成员方法

1.1 add

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

```
private void ensureCapacityInternal(int minCapacity) { //minCapacity 当前的容量
      if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {//此时即第一次 add
      minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);//设置容量为 10
    }
    ensureExplicitCapacity(minCapacity); // 需要的最小容量。
}
```

```
   private void ensureExplicitCapacity(int minCapacity) {
       modCount++;
       // overflow-conscious code
       if (minCapacity - elementData.length > 0) //需要扩容了。
          grow(minCapacity); //扩容的大小为目前的 size+1
    }
```

```
   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);
    }
```

```
private static int hugeCapacity(int minCapacity) {
       if (minCapacity < 0) // overflow
          throw new OutOfMemoryError();
       return (minCapacity > MAX_ARRAY_SIZE) ?
          Integer.MAX_VALUE :
          MAX_ARRAY_SIZE;// Integer.MAX_VALUE-8
    }
```

扩容的算法是：

1. 指定下次预期扩容的为当前容量的二分之三倍，称之为新容量(newCapacity)，如果需要的最小容量(minCapacity)大于新容量(newCapacity)，那么就以需要的最小容量(minCapacity)扩容。如果新容量(newCapacity)大于了规定的最大数组大小(Integr.MAX_VALUE-8)，那么就将需要的最小容量(minCapacity)和最大数组大小(Integr.MAX_VALUE-8)比较，取大。如果最小容量(minCapacity)大于规定的最大数组大小(Integr.MAX_VALUE-8)，扩容后的数组大小为Integr.MAX_VALUE

1.2 public void add(int index,E element)

```
    public void add(int index, E element) {
       rangeCheckForAdd(index);
       ensureCapacityInternal(size + 1);  // Increments modCount!!
       System.arraycopy(elementData, index, elementData, index + 1,
                        size - index);
       elementData[index] = element;
       size++;
    }
```

```
public static native void arraycopy(Object src,  int  srcPos,
             Object dest, int destPos, int length);
```

添加一个集合中的所有元素的时候，调用的是 System.arraycopy 方法。注意，调用的这个方法是一个本地方法。

1.3 public boolean addAll(Collection<? extends E> c)

```
public boolean addAll(Collection<? extends E> c) {
        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacityInternal(size + numNew);  // Increments modCount
        System.arraycopy(a, 0, elementData, size, numNew);
        size += numNew;
        return numNew != 0;
    }
```

* System.arraycopy

像其他的和添加相关的方法都是差不多了。不一一列举。

2.1 public E remove(int index)

```
    public E remove(int index) {
        rangeCheck(index);
        modCount++;
        E oldValue = elementData(index);
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // clear to let GC do its work
        return oldValue;
    }
```

很简单，不多说。

2.2 public boolean remove(Object o)

```
    public boolean remove(Object o) {
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
                    fastRemove(index);
                    return true;
                }
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }
```

```
private void fastRemove(int index) {
        modCount++;
        int numMoved = size - index - 1;
        if (numMoved > 0)
```

```
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // clear to let GC do its work
    }
```

同样的 内涵之处还是在 System.arraycopy

2.3 public boolean removeAll(Collection<?> c)

```
public boolean removeAll(Collection<?> c) {
        Objects.requireNonNull(c);
        return batchRemove(c, false);
    }
```

```
private boolean batchRemove(Collection<?> c, boolean complement) {
        final Object[] elementData = this.elementData;
        int r = 0, w = 0;
        boolean modified = false;
        try {
            for (; r < size; r++)
                if (c.contains(elementData[r]) == complement)
                    elementData[w++] = elementData[r];
        } finally {
            // Preserve behavioral compatibility with AbstractCollection,
            // even if c.contains() throws.
            if (r != size) {
                System.arraycopy(elementData, r,
                                 elementData, w,
                                 size - r);
                w += size - r;
            }
            if (w != size) {
                // clear to let GC do its work
                for (int i = w; i < size; i++)
                    elementData[i] = null;
                modCount += size - w;
                size = w;
                modified = true;
            }
        }
        return modified;
    }
```

3.1 public E get(int index)

```
public E get(int index) {
    rangeCheck(index);
    return elementData(index);
}
```

```
E elementData(int index) {
        return (E) elementData[index];   // 毕竟是数组。
    }
```

4. public E set(int index, E element)

```
public E set(int index, E element) {
        rangeCheck(index);
        E oldValue = elementData(index);
        elementData[index] = element;   //数组的性质、
        return oldValue;
    }
```

5. public boolean contains(Object o)

```
public boolean contains(Object o) {
        return indexOf(o) >= 0;
    }
```

```
public int indexOf(Object o) {
        if (o == null) {
            for (int i = 0; i < size; i++)
                if (elementData[i]==null)
                    return i;
        } else {
            for (int i = 0; i < size; i++)
                if (o.equals(elementData[i]))
                    return i;
        }
        return -1;
    }
```

四、遍历方式

1. for 循环。
2. fori
3. Iterator、ArrayList.iterator();

```
public Iterator<E> iterator() { return new Itr(); }
```

```
private class Itr implements Iterator<E> {
        int cursor;      // index of next element to return
        int lastRet = -1; // index of last element returned; -1 if no such
        int expectedModCount = modCount;
        public boolean hasNext() {
            return cursor != size;
        }
        @SuppressWarnings("unchecked")
        public E next() {
            checkForComodification();
            int i = cursor;   // cursor 当前元素的指针。
            if (i >= size)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;
            return (E) elementData[lastRet = i];
```

```
        }

        public void remove() {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();
            try {
                ArrayList.this.remove(lastRet);
                cursor = lastRet;
                lastRet = -1;
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }

        @Override
        @SuppressWarnings("unchecked")
        public void forEachRemaining(Consumer<? super E> consumer) {
            Objects.requireNonNull(consumer);
            final int size = ArrayList.this.size;
            int i = cursor;
            if (i >= size) {
                return;
            }
            final Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length) {
                throw new ConcurrentModificationException();
            }
            while (i != size && modCount == expectedModCount) {
                consumer.accept((E) elementData[i++]);
            }
            // update once at end of iteration to reduce heap write traffic
            cursor = i;
            lastRet = i - 1;
            checkForComodification();
        }

        final void checkForComodification() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
    }
```

定义了一个内部类，实现了 Iterator 接口。实现其方法。
用一个变量来记录当前访问的元素的地址，这个变量必须是成员变量。
不定义内部类，直接用 ArrayList 实现 Iterator 接口，也是可以的。

总结：

先说一些老生常谈的事情吧。

1.ArrayList 的实现原理是数组。

2.容量不固定，最大值是 Integer.MAX

3.元素允许为 null。

4.有序(重申:放入和取出是有序的)

5.非线程安全。

并发环境下，要么加锁，要么在初始化时使用 Collection.synchronizeList(new ArrayList());

6.遍历时的效率问题：

for 循环要比迭代器快。原因是 ArrayList 继承了 RandomAccess，支持快速的随机访问，而迭代器都是基于 ArrayList 方法和数组直接操作的。

7.add，remove 值类型的数据时可能会涉及拆装箱操作。

补充一下：

Fail-fast 机制，也叫作快速失败机制，是 java 集合中的一种错误检测机制。

ArrayList 中,有个 modCount 的变量,每次进行 add,set,remove 等操作,都会执行 modCount++.在获取 ArrayList 迭代器时，会将 ArrayList 中的 modCount 保存在迭代中，每次执行 add，set，remove 等操作，都会执行一次检查，都会调用 checkForComodification 方法，对 modCount 进行比较，如果迭代器中的 modCount 和 list 中的 modCount 不同，就会抛出 ConcurrentModificationException。

jdk 源码学习二　java.util.LinkedList (since 1.2)

一、签名

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

1．继承了 AbstractSequentialList<>

2．实现了 Deque：出现在 1.6，继承了 Queue。双端队列容器，不仅可以在尾部插入，删除元素，还可以在头部插入和删除元素。

3．Clone：可以克隆

4．Serializable：可被序列化

二、成员变量

```
trainsient int size = 0;
```

```
Transient Node<E> first;  // 记录第一个。
```

```
Transient Node<E> last;  // 只记录当前的节点，也是最后一个
```

```
protected transient int modCount = 0; //记录当前对 LinkedList 修改的次数
```

```
private static class Node<E> {
       E item;
       Node<E> next;
       Node<E> prev;

       Node(Node<E> prev, E element, Node<E> next) {
           this.item = element;
           this.next = next;
           this.prev = prev;
       }
   }
```

三、构造方法

```
public LinkedList() {
   }
```

```
public LinkedList(Collection<? extends E> c) {
       this();
       addAll(c);
   }
```

四、成员方法

1.1 .add(E e)

```
public boolean add(E e) {
       linkLast(e);
       return true;
   }
```

```
void linkLast(E e) {
       final Node<E> l = last;
       final Node<E> newNode = new Node<>(l, e, null);
       last = newNode;
       if (l == null)
```

```
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
```

1.2 add(int index,E element)

```
public void add(int index, E element) {
        checkPositionIndex(index);

        if (index == size)  // 如果插入的位置是最后一个，
            linkLast(element);
        else
            linkBefore(element, node(index));
    }
```

```
void linkLast(E e) 见上
```

```
void linkBefore(E e, Node<E> succ) {
        // assert succ != null;
        final Node<E> pred = succ.prev;
        final Node<E> newNode = new Node<>(pred, e, succ);
        succ.prev = newNode;
        if (pred == null)
            first = newNode;
        else
            pred.next = newNode;
        size++;
        modCount++;
    }
```

1.3 push();

```
public void push(E e) {
        addFirst(e); // 不知道为什么 不直接使用 linkFirst(e)
    }
```

```
public void addFirst(E e) {
        linkFirst(e);
    }
```

```
private void linkFirst(E e) {
        final Node<E> f = first;
        final Node<E> newNode = new Node<>(null, e, f);//注意 linkLast 的不同之处。
        first = newNode;
        if (f == null)
            last = newNode;
        else
            f.prev = newNode;
        size++;
        modCount++;
    }
```

2.1.remove()

```
public E remove() {
        return removeFirst();
    }
```

```
public E removeFirst() {
        final Node<E> f = first;
        if (f == null)
            throw new NoSuchElementException();
        return unlinkFirst(f);
    }
```

```
private E unlinkFirst(Node<E> f) {
        // assert f == first && f != null;
        final E element = f.item;
        final Node<E> next = f.next;
        f.item = null;
        f.next = null; // help GC
        first = next;
        if (next == null)
            last = null;
        else
            next.prev = null;
        size--;
        modCount++;
        return element;
    }
```

2.2 removeFirst()

2.3 removeLast()

2.4 remove(int index)

```
public E remove(int index) {
        checkElementIndex(index);
        return unlink(node(index));
    }
```

```
Node<E> node(int index) {
        // assert isElementIndex(index);

        if (index < (size >> 1)) {  // 二分查找。
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;
        } else {
            Node<E> x = last;
            for (int i = size - 1; i > index; i--)
                x = x.prev;
            return x;
        }
    }
```

```
E unlink(Node<E> x) {
        // assert x != null;
```

```
        final E element = x.item;
        final Node<E> next = x.next;
        final Node<E> prev = x.prev;

        if (prev == null) {
            first = next;
        } else {
            prev.next = next;
            x.prev = null;
        }

        if (next == null) {
            last = prev;
        } else {
            next.prev = prev;
            x.next = null;
        }

        x.item = null;
        size--;
        modCount++;
        return element;
    }
```

2.5 public E pop()

```
    public E pop() {
            return removeFirst();
    }
```

3.1 public E set(int index,E element)

```
public E set(int index, E element) {
        checkElementIndex(index);
        Node<E> x = node(index);
        E oldVal = x.item;
        x.item = element;
        return oldVal;
    }
```

4.1 public E get(int index)

```
public E get(int index) {
        checkElementIndex(index);
        return node(index).item;
    }
```

4.2 public E peekFirst()

```
public E peekFirst() {
        final Node<E> f = first;
        return (f == null) ? null : f.item;
     }
```

4.3 public E peekLast()

```
public E peekLast() {
        final Node<E> l = last;
```

```
        return (l == null) ? null : l.item;
    }
```

5. public Boolean contains(Object o)

```
public boolean contains(Object o) {
        return indexOf(o) != -1;
}
```

```
public int indexOf(Object o) { // 这种方式就是挨个遍历。
        int index = 0;
        if (o == null) {
            for (Node<E> x = first; x != null; x = x.next) {
                if (x.item == null)
                    return index;
                index++;
            }
        } else {
            for (Node<E> x = first; x != null; x = x.next) {
                if (o.equals(x.item))
                    return index;
                index++;
            }
        }
        return -1;
    }
```

6. public void clear()

```
public void clear() { // 将所有都置为null，help GC
        for (Node<E> x = first; x != null; ) {
            Node<E> next = x.next;
            x.item = null;
            x.next = null;
            x.prev = null;
            x = next;
        }
        first = last = null;
        size = 0;
        modCount++;
}
```

五、遍历方式

1. for
2. fori
3. iterator()

总结：

1. 原理是链表。
2. 有序
3. 非线程安全
4. 元素允许为null
5. 遍历时候for,都会调用node(index)方法。

6．效率问题：

很多文章都再说，arrayList 查找快，增删慢，LinkedList 增删快，查找慢。

这种说法不准确：

（1）LinkedList 做插入、删除的时候，慢在寻址，快在只需要改变前后 Entry 的引用地址

（2）ArrayList 做插入、删除的时候，慢在数组元素的批量 copy，快在寻址

所以，如果待插入、删除的元素是在数据结构的前半段尤其是非常靠前的位置的时候，LinkedList 的效率将大大快过 ArrayList，因为 ArrayList 将批量 copy 大量的元素；越往后，对于 LinkedList 来说，因为它是双向链表，所以在第 2 个元素后面插入一个数据和在倒数第 2 个元素后面插入一个元素在效率上基本没有差别，但是 ArrayList 由于要批量 copy 的元素越来越少，操作速度必然追上乃至超过 LinkedList。

从这个分析看出，如果你十分确定你插入、删除的元素是在前半段，那么就使用 LinkedList；如果你十分确定你删除、删除的元素在比较靠后的位置，那么可以考虑使用 ArrayList。如果你不能确定你要做的插入、删除是在哪儿呢？那还是建议你使用 LinkedList 吧，因为一来 LinkedList 整体插入、删除的执行效率比较稳定，没有 ArrayList 这种越往后越快的情况；二来插入元素的时候，弄得不好 ArrayList 就要进行一次扩容，记住，ArrayList 底层数组扩容是一个既消耗时间又消耗空间的操作，

jdk 源码分析三　java.util. Vector　（since 1.0）

一、签名

```
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

二、成员变量

```
Protect Object[] elementData;
```

```
Protect int elementCount;
```

```
Protect int capacitIncrement;
```

```

```

三、构造方法

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

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

```
public Vector() {
        this(10);  //vector 的默认初始化大小为 10
    }
```

四、成员方法

Vector 和 ArrayList 差不多。不同的是 Vector 的方法上加上了 synchronized 关键字。这表明 Vector 的线程同步的，线程安全的。

1. add()

```
public synchronized void addElement(E obj) {
        modCount++;
        ensureCapacityHelper(elementCount + 1);
        elementData[elementCount++] = obj;
    }
```

```
private void ensureCapacityHelper(int minCapacity) {
        // overflow-conscious code
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }
```

```
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);
    }
```

```
private static int hugeCapacity(int minCapacity) {
        if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
        return (minCapacity > MAX_ARRAY_SIZE) ?
            Integer.MAX_VALUE :
            MAX_ARRAY_SIZE;
    }
```

扩容的算法：如果没有指定扩容大小，那么就默认扩容增量为当前的容量+1，则扩容后的大小为两倍的当前容量+1，称之为新容量。如果新容量小于需要的容量大小，那将这个需要的容量赋值给新容量，如果新容量大于最大的数组大小，那么扩容到最大，即 MAX_ARRAY_SIZE。

```
public synchronized void insertElementAt(E obj, int index) {
        modCount++;
        if (index > elementCount) {
            throw new ArrayIndexOutOfBoundsException(index
                                                     + " > " + elementCount);
        }
        ensureCapacityHelper(elementCount + 1);
        System.arraycopy(elementData,  index,  elementData,  index  +  1,
elementCount - index);
        elementData[index] = obj;
        elementCount++;
    }
```

关键还是在判断容量上。

2. remove()

```
public synchronized boolean removeElement(Object obj) {
        modCount++;
        int i = indexOf(obj);
        if (i >= 0) {
            removeElementAt(i);
            return true;
        }
        return false;
    }
```

```
public int indexOf(Object o) {
        return indexOf(o, 0);
    }
```

```
public synchronized int indexOf(Object o, int index) {
        if (o == null) {
            for (int i = index ; i < elementCount ; i++)
```

```
            if (elementData[i]==null)
                return i;
        } else {
            for (int i = index ; i < elementCount ; i++)
                if (o.equals(elementData[i]))
                return i;
        }
        return -1;
    }
```

```
public synchronized void removeElementAt(int index) {
        modCount++;
        if (index >= elementCount) {
            throw new ArrayIndexOutOfBoundsException(index + " >= " +
                                                     elementCount);
        }
        else if (index < 0) {
            throw new ArrayIndexOutOfBoundsException(index);
        }
        int j = elementCount - index - 1;
        if (j > 0) {
            System.arraycopy(elementData, index + 1, elementData, index, j);
        }
        elementCount--;
        elementData[elementCount] = null; /* to let gc do its work */
    }
```

3. public boolean contains(Object o)

```
public boolean contains(Object o) {
        return indexOf(o, 0) >= 0;
    }
```

```
indexOf(),见上。
```

4.

```
public synchronized E firstElement() {
        if (elementCount == 0) {
            throw new NoSuchElementException();
        }
        return elementData(0);
    }
```

```
public synchronized E lastElement() {
        if (elementCount == 0) {
            throw new NoSuchElementException();
        }
        return elementData(elementCount - 1);
    }
```

五、遍历方式

```
public Enumeration<E> elements() {
```

```
        return new Enumeration<E>() {
            int count = 0;
            public boolean hasMoreElements() {
                return count < elementCount;
            }
            public E nextElement() {
                synchronized (Vector.this) {
                    if (count < elementCount) {
                        return elementData(count++);
                    }
                }
                throw new NoSuchElementException("Vector Enumeration");
            }
        };
    }
```

for

fori

Iterator

总结

允许元素为空。

有序

线程安全

jdk 源码分析四　java.util.HashMap （since 1.2）

本来是想写 HashSet,但是看了 HashSet 的成员变量，你就懂了、、、、

一、签名

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
```

HashMap 通常作为桶式哈希表，当桶变得很大的时候就转化为树节点。一般达到过量数据的时机比较少。所以在桶式哈希表中会尽量推迟树形节点的检测。

树形哈希（所有节点都是树节点），以哈希值排序，但如果都是同类型并且该类型实现了比较器就以比较器的结果为准。TreeNode 是一般节点的两倍。只有当哈希表节点数达到一定数量才使用。

通常第一个节点作为树的根节点，当根节点移除时才更换。

不论哈希列表还是树形哈希，分割还是非树形，都保证相对的访问遍历顺序。

二、成员变量

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;  // 初始大小 16
```

```
static final int MAXIMUM_CAPACITY = 1 << 30; // 最大容量
```

```
static final float DEFAULT_LOAD_FACTOR = 0.75f; //负载系数(装载因子)
```

主要控制空间利用率和冲突。装载因子越大空间利用率更高，冲突可能也会变大，反之则相反。

```
static final int TREEIFY_THRESHOLD = 8; //由链表转换成树的阈值、
```

```
static final int UNTREEIFY_THRESHOLD = 6; // 由树转换成链表的阈值
```

```
static final int MIN_TREEIFY_CAPACITY = 64; //转换树形后表格最小容量，至少是
```

treeify_threshold 的四倍。

```
static class Node<K,V> implements Map.Entry<K,V> { //基本的哈希容器节点。
        final int hash;    // 不可变的 hash 值，由关键字 key 得来。
        final K key;  // 关键字不可变
        V value;
        Node<K,V> next;

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }

        public final K getKey()        { return key; }
        public final V getValue()      { return value; }
        public final String toString() { return key + "=" + value; }

        public final int hashCode() {  // 异或运算。
            return Objects.hashCode(key) ^ Objects.hashCode(value);
        }

        public final V setValue(V newValue) {
            V oldValue = value;
            value = newValue;
            return oldValue;
        }
```

```
        public final boolean equals(Object o) {
            if (o == this)
                return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?,?> e = (Map.Entry<?,?>)o;
                if (Objects.equals(key, e.getKey()) &&
                    Objects.equals(value, e.getValue()))
                    return true;
            }
            return false;
        }
    }
```

```
transient Node<K,V>[] table;  // 不被序列化的节点。Node 类型数组，第一次使用的时候初始化，必要时重新分配空间。长度总是 2 的次幂。
```

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

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }

        public final K getKey()        { return key; }
        public final V getValue()      { return value; }
        public final String toString() { return key + "=" + value; }

        public final int hashCode() {
            return Objects.hashCode(key) ^ Objects.hashCode(value);
        }

        public final V setValue(V newValue) {
            V oldValue = value;
            value = newValue;
            return oldValue;
        }

        public final boolean equals(Object o) {
            if (o == this)
                return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?,?> e = (Map.Entry<?,?>)o;
                if (Objects.equals(key, e.getKey()) &&
```

```
                Objects.equals(value, e.getValue()))
                return true;
        }
        return false;
    }
}
```

```
transient Set<Map.Entry<K,V>> entrySet; // 缓存所有的 EntrySet()
```

```
transient int size; // 当前 map 中的数据量
```

```
transient int modCount; // map 结构的修改次数。实现了 fast-fial 策略。
```

```
int threshold; // 下次重新分配空间 resize()时，table 数组的大小。
```

```
Final float loadFactor;  // hash 表的负载因子。
```

三、静态工具函数。

```
static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }  // 学习一下编码风格(标红处)。
```

计算 key 的 hash 值，并且将高位的 hash 值移到低位。因为使用的掩码是 2 的 n 次幂，高于掩码的位组成的哈希集合总是冲突，所以要把高位移到低位。

```
    static Class<?> comparableClassFor(Object x) {
        if (x instanceof Comparable) {
            Class<?> c; Type[] ts, as; Type t; ParameterizedType p;
            if ((c = x.getClass()) == String.class) // bypass checks
                return c;
            if ((ts = c.getGenericInterfaces()) != null) {
                for (int i = 0; i < ts.length; ++i) {
                    if (((t = ts[i]) instanceof ParameterizedType) &&
                        ((p = (ParameterizedType)t).getRawType() ==
                         Comparable.class) &&
                        (as = p.getActualTypeArguments()) != null &&
                        as.length == 1 && as[0] == c) // type arg is c
                        return c;
                }
            }
        }
        return null;
    }
```

与 x 进行比较，如果 x 是可比较类型,返回 x 的类型，否则返回 null。

```
    static int compareComparables(Class<?> kc, Object k, Object x) {
        return (x == null || x.getClass() != kc ? 0 :
                ((Comparable)k).compareTo(x));
    }
```

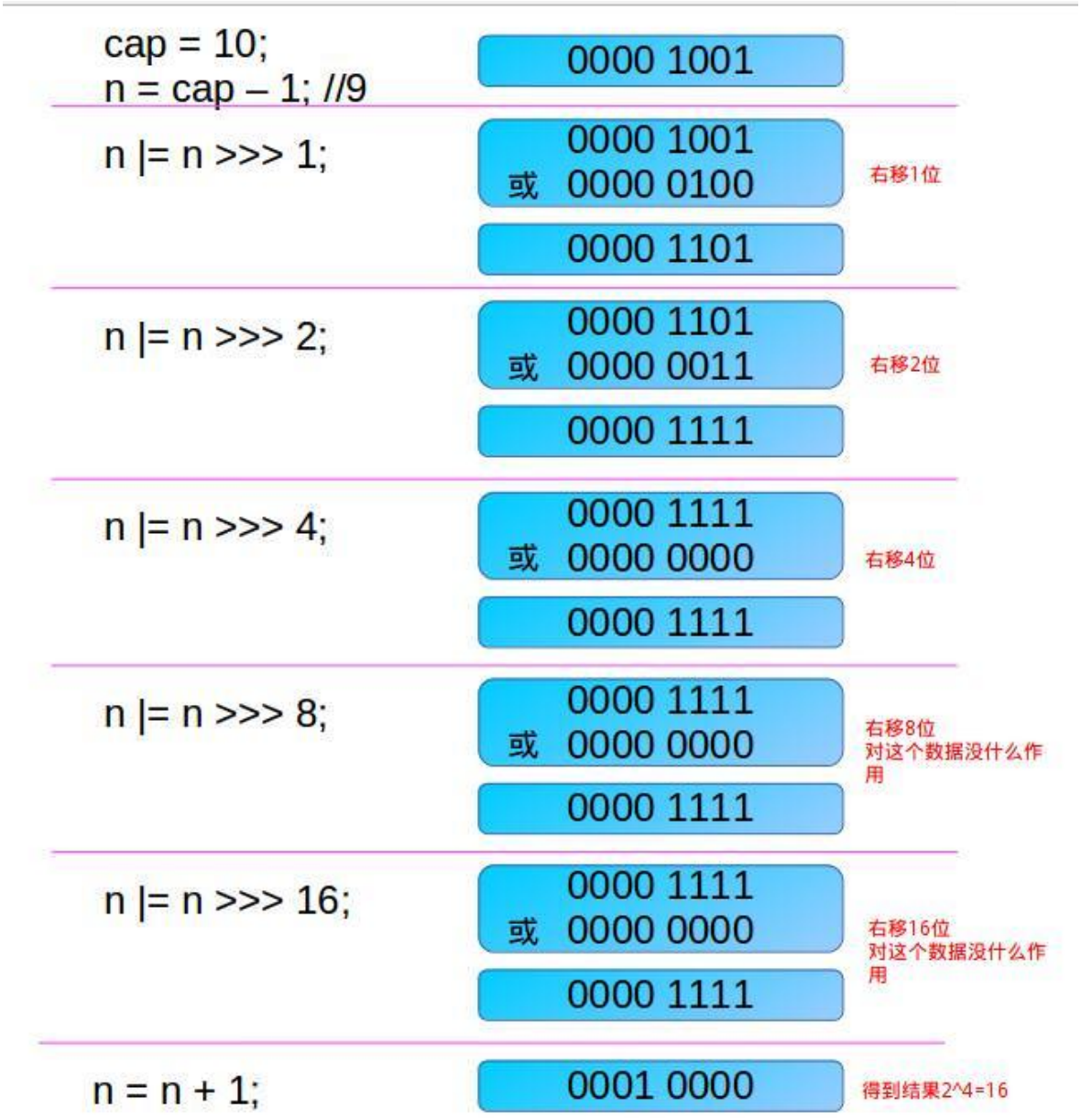
如果 x 和 k 可以比较，返回 k 和 x 的比较结果，否则返回 0.

```
    static final int tableSizeFor(int cap) {
        int n = cap - 1;
        n |= n >>> 1;
```

```
            n |= n >>> 2;
            n |= n >>> 4;
            n |= n >>> 8;
            n |= n >>> 16;
            return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ?
MAXIMUM_CAPACITY : n + 1;
    }
```

对于给定的目标容器返回一个 2 的次幂容量(返回大于 cap 的最小的 2 次幂)*感觉这个地方很深奥。*

百度一张图片吧：



四、构造方法

```
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);
        this.loadFactor = loadFactor;
        this.threshold = tableSizeFor(initialCapacity);
    }
```

根据特定的初始化容量和负载因子的构造函数。

```
public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }
```

根据指定初始化容量和默认装载因子.75 的构造函数

```
public HashMap() {
        this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
    }
```

构造一个默认大小和默认装载因子的 HashMap。

```
public HashMap(Map<? extends K, ? extends V> m) {
        this.loadFactor = DEFAULT_LOAD_FACTOR;
        putMapEntries(m, false);
    }
```

五、成员方法

```
public V put(K key, V value) {
        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 table 为空的话，就先初始化，扩容。
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        if ((p = tab[i = (n - 1) & hash]) == null) // 如果 tab[i]为空，直接放入。
            tab[i] = newNode(hash, key, value, null);
        else { // 如果 hash 后的位置上的值不为空。后接链表。
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k)))) // key
已经存在。
                e = p;
            else if (p instanceof TreeNode) // 如果该节点属于红黑树。(链表的长度>8)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {// tab[i]后还是链表。
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) { //遍历到链表的最后一个元素。
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
```

```
                        treeifyBin(tab, hash); //将链表转换成红黑树。
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null &&
key.equals(k))))//如果 key 已经存在，就覆盖 value。
                    break;
                p = e;
            }
        }

//如果之前判断到 key 已经存在，就进行覆盖 value
        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;
}
```

扩容机制：

```
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) { // 如果原来 table 中容量已经是最大
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
// 如果旧容量*2 小于最大容量阈值，并且旧容量大于默认初始化容量。
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // 新的容量阈值扩大两倍。
    }
    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) { // 把每个table[i]都移动到新的
newtable[i]中。
                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;
                            }
                            else {  // 原索引+oldCap
                                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;
```

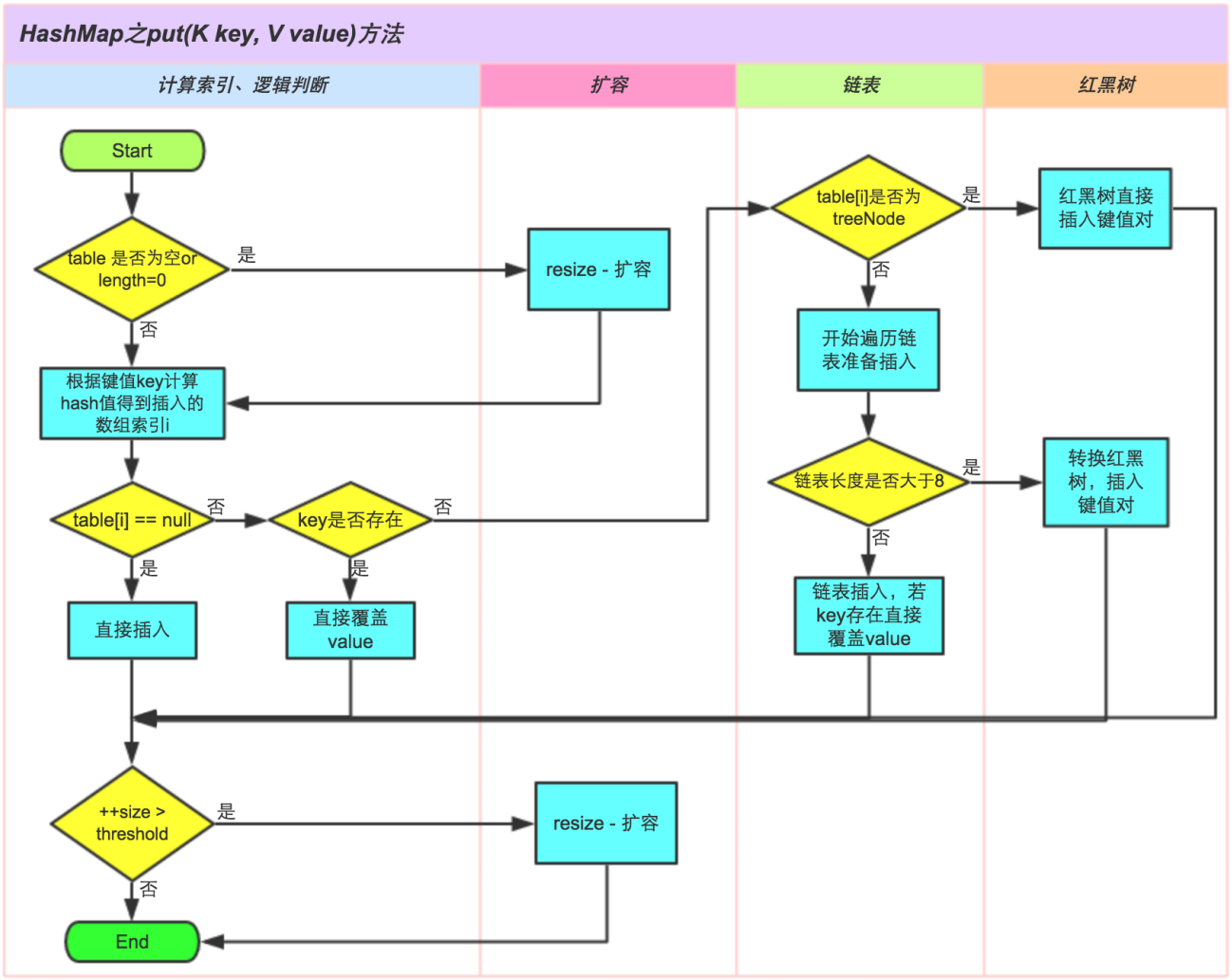
```
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
        return newTab;
    }
```

```
final TreeNode<K,V> putTreeVal(HashMap<K,V> map, Node<K,V>[] tab,
                               int h, K k, V v) {
        Class<?> kc = null;
        boolean searched = false;
        TreeNode<K,V> root = (parent != null) ? root() : this;
        for (TreeNode<K,V> p = root;;) {
            int dir, ph; K pk;
            if ((ph = p.hash) > h)
                dir = -1;
            else if (ph < h)
                dir = 1;
            else if ((pk = p.key) == k || (k != null && k.equals(pk)))
                return p;
            else if ((kc == null &&
                      (kc = comparableClassFor(k)) == null) ||
                     (dir = compareComparables(kc, k, pk)) == 0) {
                if (!searched) {
                    TreeNode<K,V> q, ch;
                    searched = true;
                    if (((ch = p.left) != null &&
                         (q = ch.find(h, k, kc)) != null) ||
                        ((ch = p.right) != null &&
                         (q = ch.find(h, k, kc)) != null))
                        return q;
                }
                dir = tieBreakOrder(k, pk);
            }

            TreeNode<K,V> xp = p;
            if ((p = (dir <= 0) ? p.left : p.right) == null) {
                Node<K,V> xpn = xp.next;
                TreeNode<K,V> x = map.newTreeNode(h, k, v, xpn);
                if (dir <= 0)
                    xp.left = x;
                else
                    xp.right = x;
                xp.next = x;
                x.parent = x.prev = xp;
                if (xpn != null)
                    ((TreeNode<K,V>)xpn).prev = x;
```

```
                    moveRootToFront(tab, balanceInsertion(root, x));
                    return null;
                }
            }
        }
```

分析：



图片来源于网络，见注释。

```
public V remove(Object key) {
        Node<K,V> e;
        return (e = removeNode(hash(key), key, null, false, true)) == null ?
            null : e.value;
    }
```

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

```
final Node<K,V> removeNode(int hash, Object key, Object value,
                           boolean matchValue, boolean movable) {
        Node<K,V>[] tab; Node<K,V> p; int n, index;
```

```
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (p = tab[index = (n - 1) & hash]) != null) {
            Node<K,V> node = null, e; K k; V v;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                node = p;
            else if ((e = p.next) != null) {
                if (p instanceof TreeNode)
                    node = ((TreeNode<K,V>)p).getTreeNode(hash, key);
                else {
                    do {
                        if (e.hash == hash &&
                            ((k = e.key) == key ||
                             (key != null && key.equals(k)))) {
                            node = e;
                            break;
                        }
                        p = e;
                    } while ((e = e.next) != null);
                }
            }
            if (node != null && (!matchValue || (v = node.value) == value ||
                                 (value != null && value.equals(v)))) {
                if (node instanceof TreeNode)
                    ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);
                else if (node == p)
                    tab[index] = node.next;
                else
                    p.next = node.next;
                ++modCount;
                --size;
                afterNodeRemoval(node);
                return node;
            }
        }
        return null;
    }
```

六、遍历方法

这里总结一下 Map 的遍历方法吧，for，fori，iterator 什么的统统不算哈。

就只有两种。一种是 entry，一种是 getKey。

举个栗子：一对夫妻，你想找个那个妻子。。。。。。一种找到他老公，通过他老公去找，第二种是直接拿到他俩的结婚证找到妻子。

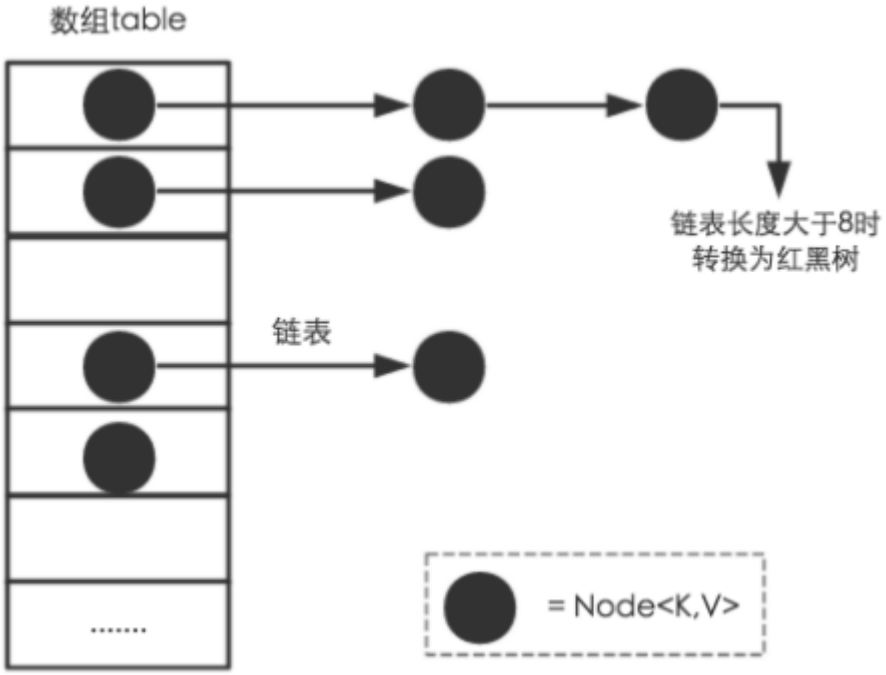
1. hashMap.keySet(); for set……就是 for 循环的方式了。
2. hashMap.entrySet();

总结

hashMap 根据键的 hashCode 值存储数据，大多数情况下可以直接定位它的值。大多数情况下可以直接定位它的值，因而具有很快的访问速度。但是遍历顺序却是不确定的。hashMap 最多只允许一条记录的键为 null，允许多条记录的值为 null。hashMap 非线程安全，即任一时刻可以有多个线程同时写 HashMap，可能会导致数据的不一致，但是可以用 Collections.synchronizedMap()使 HashMap 线程安全。或者使用 ConcurrentHashMap。

要搞清楚 HashMap，首先要清楚 HashMap 是什么，即它的存储结构-字段，其次弄明白它能干什么，即它的功能实现-方法。

Jdk8 中，HashMap 是数组+链表+红黑树实现的，如下图。



那么问题来了，数据底层存储的是什么呢？这样的存储方式有什么好处呢？

1.hashMap 有一个很重要的字段，就是 Node<K,V>.这个 Node 的实现，见上面的字段介绍。途中每个黑点就是一个 Node。

2.大家都知道 HashMap 的底层数据结构使哈希表，哈希表为解决冲突，可以采用开发地址法和链地址法等来解决问题，java 中 HashMap 采用了链地址法。当数据被 hash 后，得到数组下标，把数据放在对应下标元素的链表上。

3.根据源码，threshold 就是在此 loadFactor 和 length 对应下允许的最大元素数目，超过这个数目就要重新扩容，扩容后的 HashMap 容量就是之前容量的两倍。默认的 loadFactor 是 0.75，是对空间和时间的平衡选择，。如果内存空间很多而对事件效率要求很高，可以降低 loadFactor 的值。想反，对内存空间比较紧张而对时间效率要求不高的时候，就可以增加 loadFactor 的值，这个值可以大于 1.

4.为什么 hashMap 的重新扩容或者初始化大小是 2 的 n 次方呢？

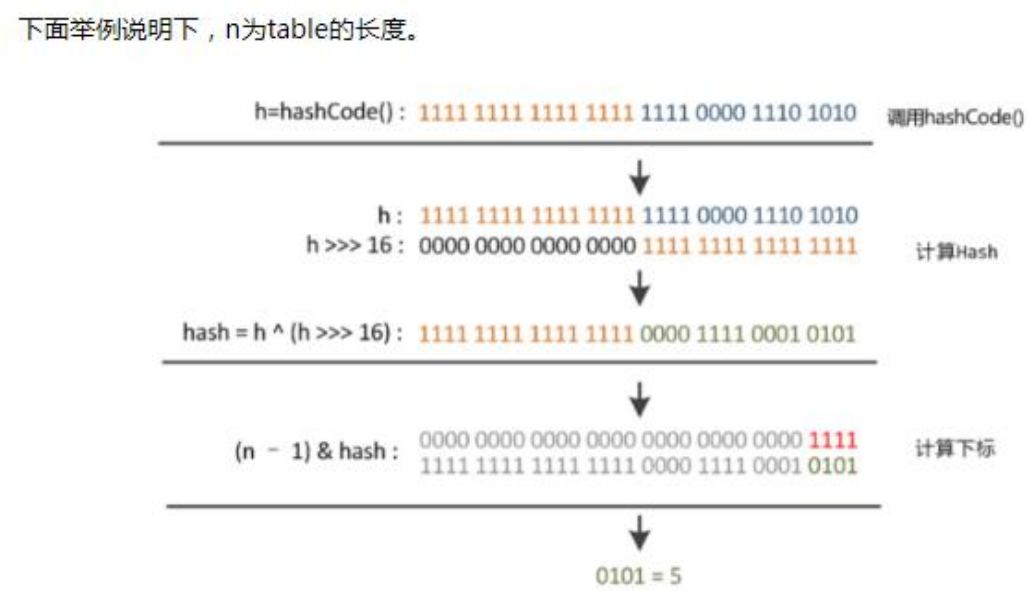
在 HashMap 中，哈希桶数组 table 的长度 length 大小必须为 2 的 n 次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。相对来说素数导致冲突的概率要小于合数，具体证明可以参考 http://blog.csdn.net/liuqiyao_01/article/details/14475159，Hashtable 初始化桶大小为 11，就是桶大小设计为素数的应用（Hashtable 扩容后不能保证还是素数）。HashMap 采用这种非常规设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap 定位哈希桶索引位置时，也加入了高位参与运算的过程。

5.哈希算法：

这里的 Hash 算法本质上就是三步，取 key 的 hashCode 值,，高位运算，取模运算。

对于任意的对象，只要 hashCode()的值相同，那么计算所得的 hash 码也是相同的，我们把 hash 值对数组的长度取模，但是取模的消耗很大，hashMap 采用 h&(length-1)的方式。

这个方法非常巧妙，它通过 h & (table.length -1)来得到该对象的保存位，而 HashMap 底层数组的长度总是 2 的 n 次方，这是 HashMap 在速度上的优化。当 length 总是 2 的 n 次方时，h& (length-1)运算等价于对 length 取模，也就是 h%length，但是&比%具有更高的效率。



6.扩容机制：

我们使用的是 2 次幂的扩展(指长度扩为原来的 2 倍)所以，元素的位置要么是在原位置，要么是在原位置在移动 2 次幂的位置。

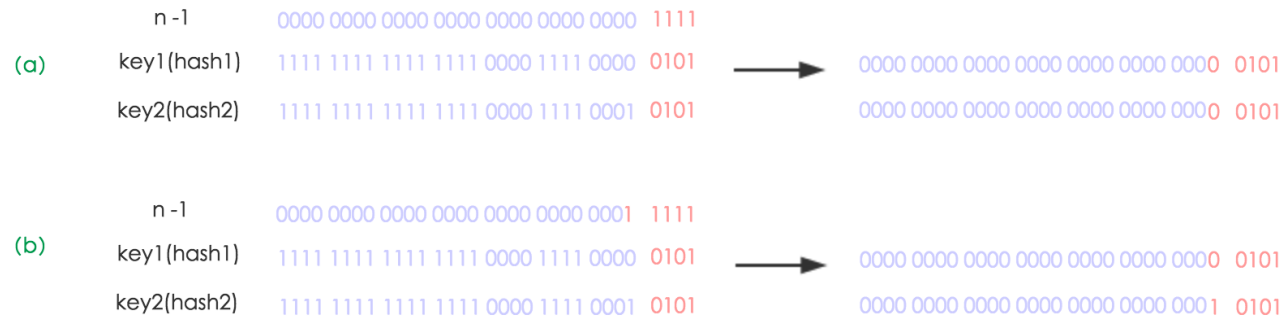
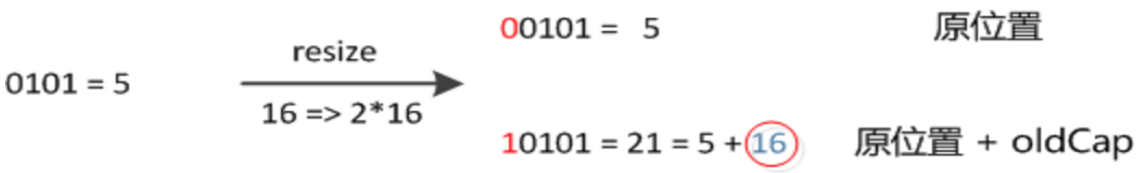
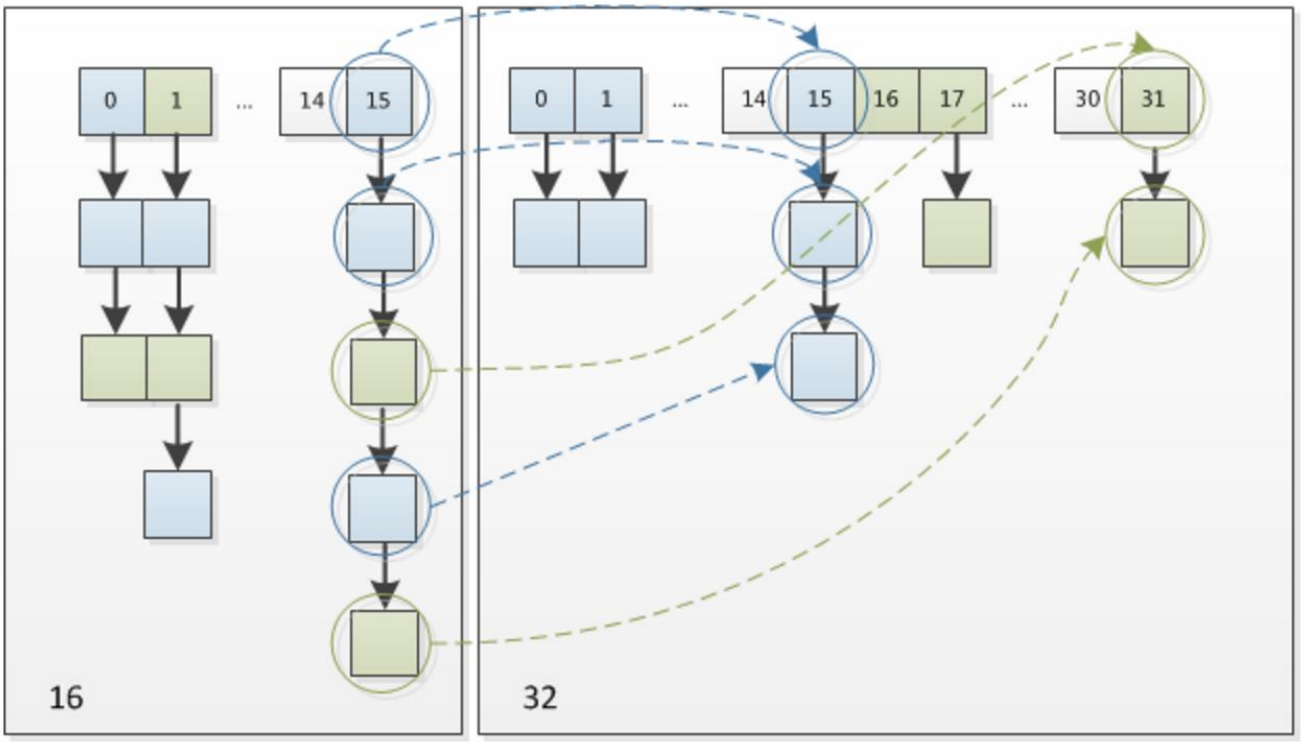


图 a，表示扩容前的 key1 和 key2 两种 key 确定索引的位置，
图 b，表示扩容后的 key1 和 key2 两种 key 确定索引位置的示例。
元素在重新计算 hash 之后，因为 n 变为 2 倍，那么 n-1 的 mask 范围在高位多 1bit(红色)，因此新的 index 就会发生这样的变化：



因此，我们在扩充 HashMap 的时候，不需要像 JDK1.7 的实现那样重新计算 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。这一块就是 JDK1.8 新增的优化点。

7．线程安全机制：

线程不安全，多线程环境中应尽量使用 ConcurrentHashMap。

(1) 扩容是一个特别耗性能的操作，所以当程序员在使用 HashMap 的时候，估算 map 的大小，初始化的时候给一个大致的数值，避免 map 进行频繁的扩容。

(2) 负载因子是可以修改的，也可以大于 1，但是建议不要轻易修改，除非情况非常特殊。

(3) HashMap 是线程不安全的，不要在并发的环境中同时操作 HashMap，建议使用 ConcurrentHashMap。

(4) JDK1.8 引入红黑树大程度优化了 HashMap 的性能。

(5) 还没升级 JDK1.8 的，现在开始升级吧。HashMap 的性能提升仅仅是 JDK1.8 的冰山一角。

借鉴文章：http://www.importnew.com/20386.html （通过这篇文章，知道了什么才是在学习）

jdk 源码分析五 java.util.hashSet (since 1.2)

零、搞完 HashMap 之后，hashSet 就很简单了。这里顺便就把 HashSet 带过了。

一、签名

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
```

二、成员变量

```
private transient HashMap<E,Object> map; //hashSet 底层用 HashMap 实现。
```

```
private static final Object PRESENT = new Object();
```

三、构造方法

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

```
public HashSet(Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
    }
```

```
public HashSet(int initialCapacity, float loadFactor) {
        map = new HashMap<>(initialCapacity, loadFactor);
    }
```

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

```
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
```

四、成员方法：

```
public boolean isEmpty() {
        return map.isEmpty();
    }
```

```
public boolean contains(Object o) {
        return map.containsKey(o);
    }
```

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

```
public boolean remove(Object o) {
        return map.remove(o)==PRESENT;
    }
```

五、遍历方式：

六、总结：

看了 HashMap 之后，在看这个真的一点要敲黑板的地方都没有！~

jdk 源码分析六　java.util.HashTable<K,V>　since 1.0

一、签名

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

Dictionary 接口：所有包含 key 和 value 的抽象父类。

二、成员变量

```
private transient Entry<?,?>[] table; //
```

```
private transient int count; // table 的大小
```

```
private int threshold; // 阈值
```

```
private float loadFactor;  // 装载因子
```

```
private transient int modCount = 0;  //用于 fast-fial 策略，记录结构修改次数。
```

```
private static class Entry<K,V> implements Map.Entry<K,V> { //单向链表。
        final int hash;
        final K key;
        V value;
        Entry<K,V> next;

        protected Entry(int hash, K key, V value, Entry<K,V> next) {
            this.hash = hash;
            this.key =  key;
            this.value = value;
            this.next = next;
        }

        @SuppressWarnings("unchecked")
        protected Object clone() {
            return new Entry<>(hash, key, value,
                                (next==null ? null : (Entry<K,V>) next.clone()));
        }

        // Map.Entry Ops

        public K getKey() {
            return key;
        }

        public V getValue() {
            return value;
        }

        public V setValue(V value) {
            if (value == null)
                throw new NullPointerException();
```

```
            V oldValue = this.value;
            this.value = value;
            return oldValue;
        }

        public boolean equals(Object o) {
            if (!(o instanceof Map.Entry))
                return false;
            Map.Entry<?,?> e = (Map.Entry<?,?>)o;

            return (key==null ? e.getKey()==null : key.equals(e.getKey())) &&
               (value==null ? e.getValue()==null : value.equals(e.getValue()));
        }

        public int hashCode() {
            return hash ^ Objects.hashCode(value);
        }

        public String toString() {
            return key.toString()+"="+value.toString();
        }
    }
```

三、构造方法

```
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);
    }
```

```
public Hashtable(int initialCapacity) {
        this(initialCapacity, 0.75f);
    }
```

```
public Hashtable() {
        this(11, 0.75f);   //这里直接写死了，默认大小就是11.  装载
    }
```

```
public Hashtable(Map<? extends K, ? extends V> t) {
        this(Math.max(2*t.size(), 11), 0.75f);
        putAll(t);
    }
```

四、成员方法

```
public synchronized int size() {
       return count;
   }
```

```
public synchronized boolean isEmpty() {
       return count == 0;
   }
```

```
public synchronized Enumeration<K> keys() {
       return this.<K>getEnumeration(KEYS);
   }
```

```
public synchronized Enumeration<V> elements() {
       return this.<V>getEnumeration(VALUES);
   }
```

```
public synchronized boolean contains(Object value) {
       if (value == null) {
           throw new NullPointerException();
       }

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

```
public synchronized boolean containsKey(Object key) {
       Entry<?,?> tab[] = table;
       int hash = key.hashCode();
       int index = (hash & 0x7FFFFFFF) % tab.length;
       for (Entry<?,?> e = tab[index] ; e != null ; e = e.next) {
           if ((e.hash == hash) && e.key.equals(key)) {
               return true;
           }
       }
       return false;
   }
```

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

```
            }
        }
        return null;
    }
```

```
public synchronized V put(K key, V value) {
        // Make sure the value is not null
        if (value == null) { // 值也不允许为空!
            throw new NullPointerException();
        }

        // Makes sure the key is not already in the hashtable.
        Entry<?,?> tab[] = table;
        int hash = key.hashCode();
        int index = (hash & 0x7FFFFFFF) % tab.length;
        @SuppressWarnings("unchecked")
        Entry<K,V> entry = (Entry<K,V>)tab[index];
        for(; entry != null ; entry = entry.next) {
            if ((entry.hash == hash) && entry.key.equals(key)) {
                V old = entry.value;
                entry.value = value;
                return old;
            }
        }

        addEntry(hash, key, value, index);
        return null;
    }
```

```
private void addEntry(int hash, K key, V value, int index) {
        modCount++;

        Entry<?,?> tab[] = table;
        if (count >= threshold) {
            // Rehash the table if the threshold is exceeded
            rehash();  // 扩容处理。

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

        // Creates the new entry.
        @SuppressWarnings("unchecked")
        Entry<K,V> e = (Entry<K,V>) tab[index];
        tab[index] = new Entry<>(hash, key, value, e);
        count++;
    }
```

五、总结

1．HashTable 是基于 Dictionary 类的、
2．Hashtable 中的方法是同步的，保证了 hashTable 中对象的线程安全。
3．内部实现是数组加链表。默认大小是 11，增加的方式是 old*2+1；
4．Hashtable 中，key 和 value 都不能为空。

Jdk 源码分析七　java.util.LinkedHashMap　since1.4

一、签名

```
public class LinkedHashMap<K,V>
    extends HashMap<K,V>
    implements Map<K,V>
```

实现了 Map 接口，继承了 HashMap。

二、成员变量

```
transient LinkedHashMap.Entry<K,V> head; //// 双向链表的头
```

```
transient LinkedHashMap.Entry<K,V> tail;  // 双向链表的尾
```

```
final boolean accessOrder; //控制读取的顺序，true 表示访问的顺序，false 表示插入的顺序
默认为 false。
```

```

```

三、构造方法

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

```
transient LinkedHashMap.Entry<K,V> head;
```

```
transient LinkedHashMap.Entry<K,V> tail;
```

```
final boolean accessOrder;
```

```
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    } // 注意了　这里是 LinkedHashMap。
```

四、成员方法

```
public V get(Object key) {
        Node<K,V> e;
        if ((e = getNode(hash(key), key)) == null)
            return null;
        if (accessOrder)
            afterNodeAccess(e);
        return 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) {
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
            if ((e = first.next) != null) {
                if (first instanceof TreeNode)
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do {
                    if (e.hash == hash &&
```

```
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

```
void afterNodeAccess(Node<K,V> e) { // move node to last
    LinkedHashMap.Entry<K,V> last;
    if (accessOrder && (last = tail) != e) {
        LinkedHashMap.Entry<K,V> p =
            (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
        p.after = null;
        if (b == null)
            head = a;
        else
            b.after = a;
        if (a != null)
            a.before = b;
        else
            last = b;
        if (last == null)
            head = p;
        else {
            p.before = last;
            last.after = p;
        }
        tail = p;
        ++modCount;
    }
}
```

put 方法和 HashMap 的方法一致。

```
public boolean containsValue(Object value) {
    for (LinkedHashMap.Entry<K,V> e = head; e != null; e = e.after) {
        V v = e.value;
        if (v == value || (value != null && value.equals(v)))
            return true;
    }
    return false;
}
```

contiainsValue 方法，就是遍历整个 linkedHashMap,将每个 Entry 的 value 都比较一下。

五、遍历方式 略

六、总结

1.accessOrder 的作用。

False 的情况下(默认)，按照插入时候的顺序来访问每个元素。

True 的情况，每次调用 get(K k)的时候，都会对 HashTable 发生改变。它会按照访问顺序来改变 HashTable 结构。

Jdk 源码分析八　java.util.TreeMap　since 1.2

一、签名

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
```

NavigableMap：可导航的 Map。 Since 1.6. 它实现继承了 SortedMap,成为了一个具有搜索匹配算法的 Map。和 TreeSet 类似。

二、成员变量

```
private final Comparator<? super K> comparator; // 比较器。
```

```
private transient Entry<K,V> root; // 树的根节点
```

```
private transient int size = 0;  // 树的 entity 数量
```

```
private transient int modCount = 0;
```

```
static final class Entry<K,V> implements Map.Entry<K,V> {
        K key;
        V value;
        Entry<K,V> left;
        Entry<K,V> right;
        Entry<K,V> parent;
        boolean color = BLACK;

        /**
         * Make a new cell with given key, value, and parent, and with
         * {@code null} child links, and BLACK color.
         */
        Entry(K key, V value, Entry<K,V> parent) {
            this.key = key;
            this.value = value;
            this.parent = parent;
        }

        /**
         * Returns the key.
         *
         * @return the key
         */
        public K getKey() {
            return key;
        }

        /**
         * Returns the value associated with the key.
         *
         * @return the value associated with the key
         */
        public V getValue() {
```

```
            return value;
        }

        /**
         * Replaces the value currently associated with the key with the given
         * value.
         *
         * @return the value associated with the key before this method was
         *         called
         */
        public V setValue(V value) {
            V oldValue = this.value;
            this.value = value;
            return oldValue;
        }

        public boolean equals(Object o) {
            if (!(o instanceof Map.Entry))
                return false;
            Map.Entry<?,?> e = (Map.Entry<?,?>)o;

            return valEquals(key,e.getKey()) && valEquals(value,e.getValue());
        }

        public int hashCode() {
            int keyHash = (key==null ? 0 : key.hashCode());
            int valueHash = (value==null ? 0 : value.hashCode());
            return keyHash ^ valueHash;
        }

        public String toString() {
            return key + "=" + value;
        }
    }
```

三、构造方法

```
public TreeMap() {   //构造一个新的，空的 tree map，使用 key 的自然顺序(没有指定比较器)。
        comparator = null;
    }
```

```
public TreeMap(Comparator<? super K> comparator) { // 根据指定的比较器构造一个
        this.comparator = comparator; //新 TreeMap
    }
```

```
public TreeMap(Map<? extends K, ? extends V> m) {  //将给定的 map 构造一个新 treeMap
        comparator = null;  // 但是按照 key 的自然顺序排序。
        putAll(m);
    }
```

```
public TreeMap(SortedMap<K, ? extends V> m) {
        comparator = m.comparator();
```

```
        try {
            buildFromSorted(m.size(), m.entrySet().iterator(), null, null);
        } catch (java.io.IOException cannotHappen) {
        } catch (ClassNotFoundException cannotHappen) {
        }
    }//将给定 SortMap 中的数据根据 SortMap 中的比较器构造一个新的 TreeMap。
```

四、成员方法

```
public V put(K key, V value) {
        Entry<K,V> t = root;
        if (t == null) {
            compare(key, key); // type (and possibly null) check

            root = new Entry<>(key, value, null);
            size = 1;
            modCount++;
            return null;
        }
        int cmp;
        Entry<K,V> parent;
        // split comparator and comparable paths
        Comparator<? super K> cpr = comparator;
        if (cpr != null) {
            do {
                parent = t;
                cmp = cpr.compare(key, t.key);
                if (cmp < 0)
                    t = t.left;
                else if (cmp > 0)
                    t = t.right;
                else
                    return t.setValue(value);
            } while (t != null);
        }
        else {
            if (key == null)
                throw new NullPointerException();
      @SuppressWarnings("unchecked")  // 如果没有指定比较器。就将 key 强转成比较器。
                Comparable<? super K> k = (Comparable<? super K>) key;
            do {
                parent = t;
                cmp = k.compareTo(t.key);
                if (cmp < 0)
                    t = t.left;
                else if (cmp > 0)
                    t = t.right;
                else
                    return t.setValue(value);
```

```
            } while (t != null);
        }
        Entry<K,V> e = new Entry<>(key, value, parent);
        if (cmp < 0)
            parent.left = e;
        else
            parent.right = e;
        fixAfterInsertion(e);
        size++;
        modCount++;
        return null;
    }
```

```
public void putAll(Map<? extends K, ? extends V> map) {
        int mapSize = map.size();
        if (size==0 && mapSize!=0 && map instanceof SortedMap) {
            Comparator<?> c = ((SortedMap<?,?>)map).comparator();
            if (c == comparator || (c != null && c.equals(comparator))) {
                ++modCount;
                try {
                    buildFromSorted(mapSize, map.entrySet().iterator(),
                                    null, null);
                } catch (java.io.IOException cannotHappen) {
                } catch (ClassNotFoundException cannotHappen) {
                }
                return;
            }
        }
        super.putAll(map);
    }
```

```
private void buildFromSorted(int size, Iterator<?> it,
                             java.io.ObjectInputStream str,
                             V defaultVal)
        throws  java.io.IOException, ClassNotFoundException {
        this.size = size;
        root = buildFromSorted(0, 0, size-1, computeRedLevel(size),
                               it, str, defaultVal);
    }
```

```
// 从排序序列中构造 TreeMap 函数。
private final Entry<K,V> buildFromSorted(int level, int lo, int hi,
                                         int redLevel,
                                         Iterator<?> it,
                                         java.io.ObjectInputStream str,
                                         V defaultVal)
        throws  java.io.IOException, ClassNotFoundException { //

// 树的根节点 肯定是排序序列的中间树。
// 递归处理根节点的左树，右树。
```

```
        if (hi < lo) return null;

        int mid = (lo + hi) >>> 1;

        Entry<K,V> left  = null;
        if (lo < mid)
            left = buildFromSorted(level+1, lo, mid - 1, redLevel,
                                   it, str, defaultVal);

        // extract key and/or value from iterator or stream
        K key;
        V value;
        if (it != null) {
            if (defaultVal==null) {
                Map.Entry<?,?> entry = (Map.Entry<?,?>)it.next();
                key = (K)entry.getKey();
                value = (V)entry.getValue();
            } else {
                key = (K)it.next();
                value = defaultVal;
            }
        } else { // use stream
            key = (K) str.readObject();
            value = (defaultVal != null ? defaultVal : (V) str.readObject());
        }

        Entry<K,V> middle =  new Entry<>(key, value, null);

        // color nodes in non-full bottommost level red
        if (level == redLevel)
            middle.color = RED;

        if (left != null) {
            middle.left = left;
            left.parent = middle;
        }

        if (mid < hi) {
            Entry<K,V> right = buildFromSorted(level+1, mid+1, hi, redLevel,
                                               it, str, defaultVal);
            middle.right = right;
            right.parent = middle;
        }

        return middle;
    }
```

查找：

```
final Entry<K,V> getFirstEntry() {
        Entry<K,V> p = root;
        if (p != null)
            while (p.left != null)
                p = p.left;
        return p;
    }  //中序遍历 获取第一个 Entity
```

```
final Entry<K,V> getLastEntry() {
        Entry<K,V> p = root;
        if (p != null)
            while (p.right != null)
                p = p.right;
        return p;
    } // 中序遍历  获取最后一个 entity。
```

删除

```
private void deleteEntry(Entry<K,V> p) {
        modCount++;
        size--;

        // If strictly internal, copy successor's element to p and then make p
        // point to successor.
        if (p.left != null && p.right != null) {
            Entry<K,V> s = successor(p);
            p.key = s.key;
            p.value = s.value;
            p = s;
        } // p has 2 children

        // Start fixup at replacement node, if it exists.
        Entry<K,V> replacement = (p.left != null ? p.left : p.right);

        if (replacement != null) {
            // Link replacement to parent
            replacement.parent = p.parent;
            if (p.parent == null)
                root = replacement;
            else if (p == p.parent.left)
                p.parent.left  = replacement;
            else
                p.parent.right = replacement;

            // Null out links so they are OK to use by fixAfterDeletion.
            p.left = p.right = p.parent = null;

            // Fix replacement
            if (p.color == BLACK)
                fixAfterDeletion(replacement);
```

```
        } else if (p.parent == null) { // return if we are the only node.
            root = null;
        } else { //  No children. Use self as phantom replacement and unlink.
            if (p.color == BLACK)
                fixAfterDeletion(p);

            if (p.parent != null) {
                if (p == p.parent.left)
                    p.parent.left = null;
                else if (p == p.parent.right)
                    p.parent.right = null;
                p.parent = null;
            }
        }
    }
```

```
static <K,V> TreeMap.Entry<K,V> successor(Entry<K,V> t) {
        if (t == null)
            return null;
        else if (t.right != null) {
            Entry<K,V> p = t.right;
            while (p.left != null)
                p = p.left;
            return p;
        } else {
            Entry<K,V> p = t.parent;
            Entry<K,V> ch = t;
            while (p != null && ch == p.right) {
                ch = p;
                p = p.parent;
            }
            return p;
        }
    }
```

五、遍历方式

六、总结

1．红黑树。

从 Entiry 这个内部类可以看出，TreeMap 是使用红黑树这种数据结构来实现的。(By the way, hashMap 中也用到了红黑树。)

大体上说一下红黑树的性质吧，这个我会在以后学习一下的.....

每个节点或是 red，或是 Black。

根节点是 Black 的。

每个叶子节点 NIL 是 black 的。

如果一个节点是 Red，那么它的两个子节点都是 Black。

对于每个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的 Black 节点。

2．TreeMap 在涉及到树形结构变化的时候，所实现的代码都是根据算法导论中的伪代码来的。

如果想研究一下红黑树的变化原理，请参考这篇文章：http://www.importnew.com/20413.html

至此，基础的集合源码 就看到这里了。

当然，我只是大体上看了一下。日后还会仔细研读一下。顺便纠正一下我写的这些小小的总结。

接下来就行，一个比较厉害的类源码了。

ConcurrentHashMap.　这个作为一个后续的任务吧。因为　目前我对并发的理解还不到位，可以说是还没有入门。等我看完了 juc(java.util.concurrent)包中的源码，再来总结一下，ConcurrentHashMap。这个部分的源码，我也得好好的吸收一下。

方小白 2017 年 11 月 18 日。