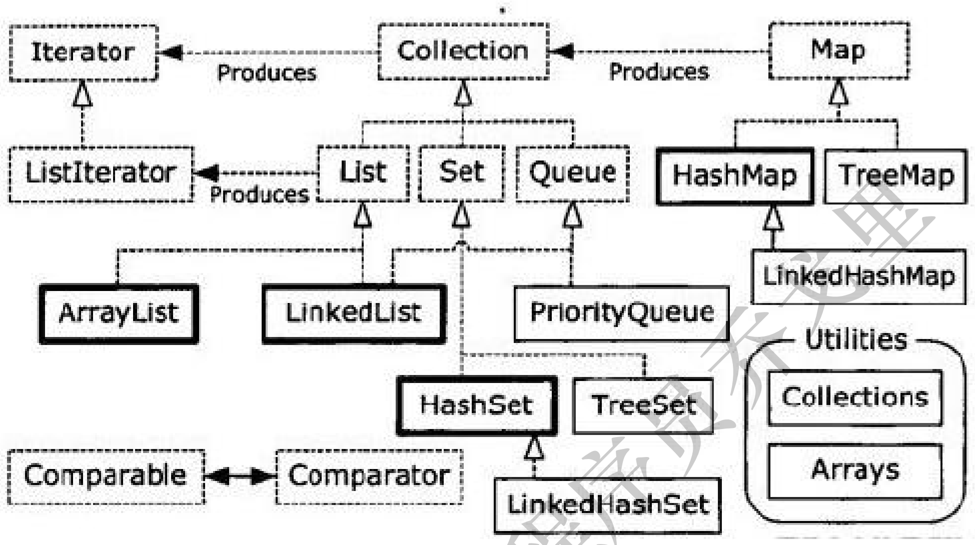
加入**互联网 IT** 求职、技术交流、资料共享 QQ 群 **691206173**，3T 编程资料等你来拿，

群主（群号：**691206173**）：本硕就读于**哈尔滨工业大学**，计算机专业，2019 硕士毕业，已拿**百度** java 后台开发 offer，另外还有去哪儿，华为，茄子快传，vipkid,秒针，创新工厂这些公司的 offer。公众号中的文章有分享群主找工作的经验，java 学习/C++学习的指导路线，以及未来在百度的学习成长之路，满满都是干货，除了干货分享还有 3T 编程资料**(java/C++/算法/php/机器学习/大数据/人工智能/面试等）**等你来拿，另外还有微信交流群以及群主的**个人微信（抽空提供一对一指导意见），当然也希望你能帮忙在朋友圈转发推广一下公众号，如果实在不方便，也不勉强，最近开通了流量主，帮忙点击一下文章末尾的广告，也算是对我的一个小小的鼓励，**欢迎您的关注。

集合框架

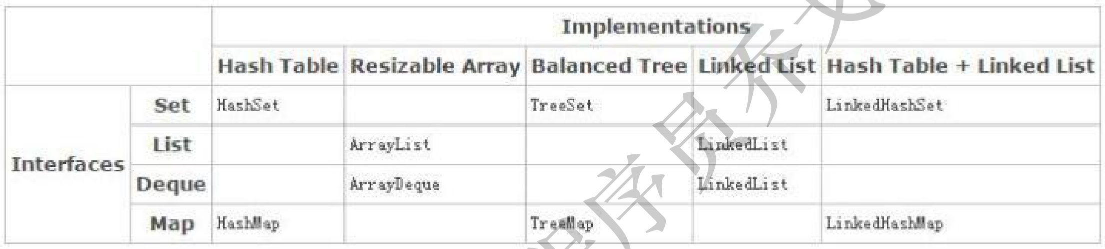


Iterator
Produces
Collection
Produces
Map
ListIterator
Produces
List
Set
Queue
HashMap
TreeMap
LinkedHashMap
ArrayList
LinkedList
PriorityQueue
Utilities
Collections
Arrays
HashSet
TreeSet
Comparable
Comparator
LinkedHashSet

接口

常见接口

Map 接口和 Collection 接口是所有集合框架的父接口；
Collection 接口的子接口包括：Set 接口、List 接口和 Queue 接口；
Map 接口的实现类主要有：HashMap、TreeMap、LinkedHashMap、Hashtable、ConcurrentHashMap 以及 Properties 等；
Set 接口的实现类主要有：HashSet、TreeSet、LinkedHashSet 等；
List 接口的实现类主要有：ArrayList、LinkedList、Stack、Vector 以及 CopyOnWriteArrayList 等；
Queue 接口的主要实现类有：ArrayDeque、ArrayBlockingQueue、LinkedBlockingQueue、PriorityQueue 等；



		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Deque		ArrayDeque		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

List 接口和 Set 接口的区别

List 元素是有序的，可以重复；Set 元素是无序的，不可以重复。

队列、Set、Map 区别

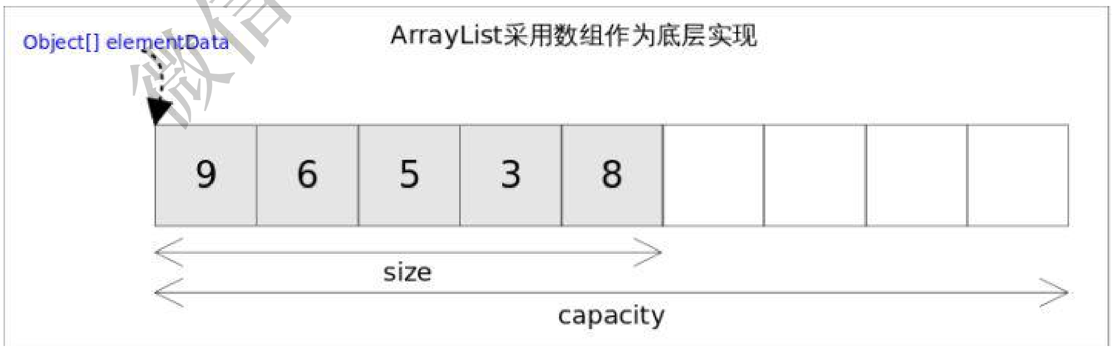
List 有序列表
Set 无序集合
Map 键值对的集合
Queue 队列 FIFO

List

有顺序，可重复

ArrayList

基于数组实现，无容量的限制。
在执行插入元素时可能要扩容，在删除元素时并不会减小数组的容量，在查找元素时要遍历数组，对于非 null 的元素采取 equals 的方式寻找。
是非线程安全的。
注意点：
（1）ArrayList 随机元素时间复杂度 O(1)，插入删除操作需大量移动元素，效率较低
（2）为了节约内存，当新建容器为空时，会共享 Object[] EMPTY_ELEMENTDATA = {}和 Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {}空数组
（3）容器底层采用数组存储，每次扩容为 1.5 倍
（4）ArrayList 的实现中大量地调用了 Arrays.copyof()和 System.arraycopy()方法，其实 Arrays.copyof()内部也是调用 System.arraycopy()。System.arraycopy()为 Native 方法
（5）两个 ToArray 方法
Object[] toArray()方法。该方法有可能会抛出 java.lang.ClassCastException 异常
<T> T[] toArray(T[] a)方法。该方法可以直接将 ArrayList 转换得到的 Array 进行整体向下转型
（6）ArrayList 可以存储 null 值
（7）ArrayList 每次修改（增加、删除）容器时，都是修改自身的 modCount；在生成迭代器时，迭代器会保存该 modCount 值，迭代器每次获取元素时，会比较自身的 modCount 与 ArrayList 的 modCount 是否相等，来判断容器是否已经被修改，如果被修改了则抛出异常（fast-fail 机制）。



成员变量

```
/**
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY = 10;

/**
 * Shared empty array instance used for empty instances.
 */
private static final Object[] EMPTY_ELEMENTDATA = {};

/**
 * Shared empty array instance used for default sized empty instances. We
 * distinguish this from EMPTY_ELEMENTDATA to know how much to inflate when
 * first element is added.
 */
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

/**
 * The array buffer into which the elements of the ArrayList are stored.
 * The capacity of the ArrayList is the length of this array buffer. Any
 * empty ArrayList with elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
 * will be expanded to DEFAULT_CAPACITY when the first element is added.
 */
transient Object[] elementData; // non-private to simplify nested class access

/**
 * The size of the ArrayList (the number of elements it contains).
 *
 * @serial
 */
private int size;
```

构造方法

```
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}
```

添加 add(e)

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

即使初始化时指定大小 小于 10 个，添加元素时会调整大小，保证 capacity 不会少于 10 个。

```
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }

    ensureExplicitCapacity(minCapacity);
}
```

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

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

扩容

```
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);
}
```

Arrays.copyOf 底层是 System.arrayCopy

```
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]>
newType) {
    @SuppressWarnings("unchecked")
    T[] copy = ((Object)newType == (Object)Object[].class)
        ? (T[]) new Object[newLength]
        : (T[]) Array.newInstance(newType.getComponentType(), newLength);
    System.arraycopy(original, 0, copy, 0,
                     Math.min(original.length, newLength));
    return copy;
}
```

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

添加 add(index,e)

```
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++;
}
```

```
private void rangeCheckForAdd(int index) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

删除 remove(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
}
```

删除 remove(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;
}
```

获取

```java
public E get(int index) {
    rangeCheck(index);

    return elementData(index);
}
```

```java
private void rangeCheck(int index) {
    if (index >= size)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

```java
E elementData(int index) {
    return (E) elementData[index];
}
```

更新

```java
public E set(int index, E element) {
    rangeCheck(index);

    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
```

遍历

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

/**
 * An optimized version of AbstractList.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;
```

```java
public boolean hasNext() {
    return cursor != size;
}

@SuppressWarnings("unchecked")
public E next() {
    checkForComodification();
    int i = cursor;
    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();
    }
}
```

包含

```
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;
}
```

LinkedList

基于双向链表机制
在插入元素时，须创建一个新的 Entry 对象，并切换相应元素的前后元素的引用；在查找元素时，须遍历链表；在删除元素时，须遍历链表，找到要删除的元素，然后从链表上将此元素删除即可。
是非线程安全的。

注意：
（1）LinkedList 有两个构造参数，一个为无参构造，只是新建一个空对象，第二个为有参构造，新建一个空对象，然后把所有元素添加进去。
（2）LinkedList 的存储单元为一个名为 Node 的内部类，包含 pre 指针，next 指针，和 item 元素，实现为双向链表
（3）LinkedList 的删除、添加操作时间复杂度为 O(1)，查找时间复杂度为 O(n)，查找函数有一定优化，容器会先判断查找的元素是离头部较近，还是尾部较近，来决定从头部开始遍历还是尾部开始遍历
（4）LinkedList 实现了 Deque 接口，因此也可以作为栈、队列和双端队列来使用。
（5）LinkedList 可以存储 null 值

成员变量

```
transient int size = 0;
transient Node<E> first;
transient Node<E> last;
```

构造方法

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

添加 add(e)

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

把一个元素添加到最后一个位置

```
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++;
}
```

添加 add(index,e)

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

    if (index == size)
        linkLast(element);
    else
        linkBefore(element, node(index));
}
```

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

    if (index < (size >> 1)) {
        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;
    }
}
```

```java
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++;
}
```

删除 remove(o)

```java
public boolean remove(Object o) {
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null) {
                unlink(x);
                return true;
            }
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item)) {
                unlink(x);
                return true;
            }
        }
    }
    return false;
}
```

```java
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;
}
```

删除 remove(index)

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

获取

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

更新

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

遍历

```
public ListIterator<E> listIterator(int index) {
    checkPositionIndex(index);
    return new ListItr(index);
}

private class ListItr implements ListIterator<E> {
    private Node<E> lastReturned;
    private Node<E> next;
    private int nextIndex;
    private int expectedModCount = modCount;

    ListItr(int index) {
        // assert isPositionIndex(index);
        next = (index == size) ? null : node(index);
        nextIndex = index;
    }

    public boolean hasNext() {
        return nextIndex < size;
    }

    public E next() {
        checkForComodification();
        if (!hasNext())
            throw new NoSuchElementException();

        lastReturned = next;
        next = next.next;
        nextIndex++;
        return lastReturned.item;
    }

    public boolean hasPrevious() {
        return nextIndex > 0;
    }

    public E previous() {
        checkForComodification();
        if (!hasPrevious())
            throw new NoSuchElementException();

        lastReturned = next = (next == null) ? last : next.prev;
        nextIndex--;
        return lastReturned.item;
    }

    public int nextIndex() {
        return nextIndex;
    }

    public int previousIndex() {
        return nextIndex - 1;
    }

    public void remove() {
        checkForComodification();
        if (lastReturned == null)
            throw new IllegalStateException();

        Node<E> lastNext = lastReturned.next;
        unlink(lastReturned);
        if (next == lastReturned)
            next = lastNext;
        else
            nextIndex--;
        lastReturned = null;
        expectedModCount++;
    }

    public void set(E e) {
        if (lastReturned == null)
            throw new IllegalStateException();
        checkForComodification();
        lastReturned.item = e;
    }

    public void add(E e) {
        checkForComodification();
        lastReturned = null;
        if (next == null)
            linkLast(e);
        else
            linkBefore(e, next);
        nextIndex++;
        expectedModCount++;
    }

    public void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (modCount == expectedModCount && nextIndex < size) {
            action.accept(next.item);
            lastReturned = next;
            next = next.next;
            nextIndex++;
        }
        checkForComodification();
    }

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

包含

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

```
public int indexOf(Object o) {
    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;
}
```

Vector

基于 synchronized 实现的线程安全的 ArrayList, 但在插入元素时容量扩充的机制和 ArrayList 稍有不同，并可通过传入 capacityIncrement 来控制容量的扩充。

成员变量

```java
protected Object[] elementData;
protected int elementCount;
protected int capacityIncrement;
```

构造方法

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

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

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

添加

```java
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}
```

删除

```java
public boolean remove(Object o) {
    return removeElement(o);
}
```

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

扩容

```java
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);
}
```

获取

```java
public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    return elementData(index);
}
```

更新

```java
public synchronized E set(int index, E element) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
```

包含

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

Stack

基于 Vector 实现，支持 LIFO。

类声明

```
public class Stack<E> extends Vector<E> {}
```

push

```
public E push(E item) {
    addElement(item);
    return item;
}
```

pop

```
public synchronized E pop() {
    E  obj;
    int len = size();
    obj = peek();
    removeElementAt(len - 1);
    return obj;
}
```

peek

```
public synchronized E peek() {
    int len = size();
    if (len == 0)
        throw new EmptyStackException();
    return elementAt(len - 1);
}
```

CopyOnWriteArrayList

是一个线程安全、并且在读操作时无锁的 ArrayList。
很多时候，我们的系统应对的都是读多写少的并发场景。CopyOnWriteArrayList 容器允许并发读，读操作是无锁的，性能较高。至于写操作，比如向容器中添加一个元素，则**首先将当前容器复制一份，然后在新副本上执行写操作，结束之后再将原容器的引用指向新容器。**

优点
1）采用读写分离方式，读的效率非常高
2）CopyOnWriteArrayList 的迭代器是基于创建时的数据快照的，故数组的增删改不会影响到迭代器

缺点
1）内存占用高，每次执行写操作都要将原容器拷贝一份，数据量大时，对内存压力较大，可能会引起频繁 GC
2）只能保证数据的最终一致性，不能保证数据的实时一致性。写和读分别作用在新老不同容器上，在写操作执行过程中，读不会阻塞但读取到的却是老容器的数据。

成员变量

```java
/** The lock protecting all mutators */
final transient ReentrantLock lock = new ReentrantLock();

/** The array, accessed only via getArray/setArray. */
private transient volatile Object[] array;
```

构造方法

```java
public CopyOnWriteArrayList() {
    setArray(new Object[0]);
}
```

```java
final void setArray(Object[] a) {
    array = a;
}
```

添加（有锁，锁内重新创建数组）

```java
final Object[] getArray() {
    return array;
}
```

```java
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        lock.unlock();
    }
}
```

存在则添加（有锁，锁内重新创建数组）

先保存一份数组 snapshot，如果 snapshot 中存在，则直接返回。
如果不存在，那么加锁，获取当前数组 current，比较 snapshot 与 current，遍历它们共同长度内的元素，如果发现 current 中某一个元素等于 e，那么直接返回（当然 current 与 snapshot 相同就不必看了）；
之后再遍历 current 单独的部分，如果发现 current 中某一个元素等于 e，那么直接返回；
此时可以去创建一个长度+1 的新数组，将 e 加入。

```java
public boolean addIfAbsent(E e) {
    Object[] snapshot = getArray();
    return indexOf(e, snapshot, 0, snapshot.length) >= 0 ? false :
        addIfAbsent(e, snapshot);
}
```

```java
private boolean addIfAbsent(E e, Object[] snapshot) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] current = getArray();
        int len = current.length;
        if (snapshot != current) {
            // Optimize for lost race to another addXXX operation
            int common = Math.min(snapshot.length, len);
            for (int i = 0; i < common; i++)
                //如果 snapshot 与 current 元素不同但 current 与 e 相同，那么直接返回
（扫描 0 到 common）
                if (current[i] != snapshot[i] && eq(e, current[i]))
                    return false;
            // 如果 current 中存在 e，那么直接返回（扫描 commen 到 len）
            if (indexOf(e, current, common, len) >= 0)
                    return false;
        }
        Object[] newElements = Arrays.copyOf(current, len + 1);
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        lock.unlock();
    }
}
```

删除（有锁，锁内重新创建数组）

```
public E remove(int index) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        E oldValue = get(elements, index);
        int numMoved = len - index - 1;
        if (numMoved == 0)
            setArray(Arrays.copyOf(elements, len - 1));
        else {
            Object[] newElements = new Object[len - 1];
            System.arraycopy(elements, 0, newElements, 0, index);
            System.arraycopy(elements, index + 1, newElements, index,
                             numMoved);
            setArray(newElements);
        }
        return oldValue;
    } finally {
        lock.unlock();
    }
}
```

获取（无锁）

```java
public E get(int index) {
    return get(getArray(), index);
}
```

```java
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

更新（有锁，锁内重新创建数组）

```java
public E set(int index, E element) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        E oldValue = get(elements, index);

        if (oldValue != element) {
            int len = elements.length;
            Object[] newElements = Arrays.copyOf(elements, len);
            newElements[index] = element;
            setArray(newElements);
        } else {
            // 为了保持“volatile”的语义，任何一个读操作都应该是一个写操作的结果，
            也就是读操作看到的数据一定是某个写操作的结果（尽管写操作没有改变数据本
身）。
            所以这里即使不设置也没有问题，仅仅是为了一个语义上的补充（就如源码中的注
释所言）
            // Not quite a no-op; ensures volatile write semantics
            setArray(elements);
        }
        return oldValue;
    } finally {
        lock.unlock();
    }
}
```

包含（无锁）

```java
public boolean contains(Object o) {
    Object[] elements = getArray();
    return indexOf(o, elements, 0, elements.length) >= 0;
}
```

```java
private static int indexOf(Object o, Object[] elements,
                           int index, int fence) {
    if (o == null) {
        for (int i = index; i < fence; i++)
            if (elements[i] == null)
                return i;
    } else {
        for (int i = index; i < fence; i++)
            if (o.equals(elements[i]))
                return i;
    }
    return -1;
}
```

遍历（遍历的是获取 iterator 时的数组快照）

```java
public Iterator<E> iterator() {
    return new COWIterator<E>(getArray(), 0);
}
```

```java
static final class COWIterator<E> implements ListIterator<E> {
    /** Snapshot of the array */
    private final Object[] snapshot;
    /** Index of element to be returned by subsequent call to next.  */
    private int cursor;

    private COWIterator(Object[] elements, int initialCursor) {
        cursor = initialCursor;
        snapshot = elements;
    }

    public boolean hasNext() {
        return cursor < snapshot.length;
    }
    public boolean hasPrevious() {
        return cursor > 0;
    }

    @SuppressWarnings("unchecked")
    public E next() {
        if (! hasNext())
            throw new NoSuchElementException();
        return (E) snapshot[cursor++];
    }

    @SuppressWarnings("unchecked")
    public E previous() {
        if (! hasPrevious())
            throw new NoSuchElementException();
        return (E) snapshot[--cursor];
    }

    public int nextIndex() {
        return cursor;
    }

    public int previousIndex() {
        return cursor-1;
    }

    /**
     * Not supported. Always throws UnsupportedOperationException.
     * @throws UnsupportedOperationException always; {@code remove}
     *         is not supported by this iterator.
     */
    public void remove() {
        throw new UnsupportedOperationException();
    }

    /**
     * Not supported. Always throws UnsupportedOperationException.
     * @throws UnsupportedOperationException always; {@code set}
     *         is not supported by this iterator.
     */
    public void set(E e) {
        throw new UnsupportedOperationException();
    }
    /**
     * Not supported. Always throws UnsupportedOperationException.
     * @throws UnsupportedOperationException always; {@code add}
     *         is not supported by this iterator.
     */
    public void add(E e) {
        throw new UnsupportedOperationException();
    }

    @Override
    public void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        Object[] elements = snapshot;
        final int size = elements.length;
        for (int i = cursor; i < size; i++) {
            @SuppressWarnings("unchecked") E e = (E) elements[i];
            action.accept(e);
        }
        cursor = size;
    }
}
```

List 实现类之间的区别

(1) 对于需要快速插入，删除元素，应该使用 LinkedList。
(2) 对于需要快速随机访问元素，应该使用 ArrayList。
(3) 对于“单线程环境” 或者 “多线程环境，但 List 仅仅只会被单个线程操作”，此时应该使用非同步的类(如 ArrayList)。
对于“多线程环境，且 List 可能同时被多个线程操作”，此时，应该使用同步的类(如 Vector、CopyOnWriteArrayList)。

Set

没有顺序，不可重复

HashSet（底层是 HashMap）

Set 不允许元素重复。
基于 HashMap 实现，无容量限制。
是非线程安全的。

成员变量

```
private transient HashMap<E,Object> map;

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

构造方法

```
/**
 * Constructs a new, empty set; the backing <tt>HashMap</tt> instance has
 * default initial capacity (16) and load factor (0.75).
 */
public HashSet() {
    map = new HashMap<>();
}
```

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

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

添加

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

删除

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

遍历

```java
public Iterator<E> iterator() {
    return map.keySet().iterator();
}
```

包含

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

TreeSet（底层是 TreeMap）

基于TreeMap实现，支持排序（**自然排序 或者 根据创建TreeSet 时提供的 Comparator 进行排序**）。
是非线程安全的。

成员变量

```java
/**
 * The backing map.
 */
private transient NavigableMap<E,Object> m;

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

构造方法

```java
public TreeSet() {
    this(new TreeMap<E,Object>());
}
```

```java
public TreeSet(Comparator<? super E> comparator) {
    this(new TreeMap<>(comparator));
}
```

添加

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

删除

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

遍历

```java
public Iterator<E> iterator() {
    return m.navigableKeySet().iterator();
}
```

包含

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

获取开头

```java
public E first() {
    return m.firstKey();
}
```

获取结尾

```java
public E last() {
    return m.lastKey();
}
```

子集

```java
public NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                              E toElement,   boolean toInclusive) {
    return new TreeSet<>(m.subMap(fromElement, fromInclusive,
                                   toElement,   toInclusive));
}
```
默认是含头不含尾

```
public SortedSet<E> subSet(E fromElement, E toElement) {
    return subSet(fromElement, true, toElement, false);
}
```

LinkedHashSet（继承自 HashSet，底层是 LinkedHashMap）

LinkedHashSet 继承自 HashSet，源码更少、更简单，唯一的区别是 LinkedHashSet 内部使用的是 LinkHashMap。这样做的意义或者好处就是 LinkedHashSet 中的元素顺序是可以保证的，也就是说**遍历序和插入序是一致的**。

类声明

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

构造方法

```java
public LinkedHashSet(int initialCapacity, float loadFactor) {
    super(initialCapacity, loadFactor, true);
}

/**
 * Constructs a new, empty linked hash set with the specified initial
 * capacity and the default load factor (0.75).
 *
 * @param   initialCapacity   the initial capacity of the LinkedHashSet
 * @throws  IllegalArgumentException if the initial capacity is less
 *              than zero
 */
public LinkedHashSet(int initialCapacity) {
    super(initialCapacity, .75f, true);
}

/**
 * Constructs a new, empty linked hash set with the default initial
 * capacity (16) and load factor (0.75).
 */
public LinkedHashSet() {
    super(16, .75f, true);
}
```

super 指的是 HashSet 的 default 访问级别的构造方法

```java
/**
 * Constructs a new, empty linked hash set.  (This package private
 * constructor is only used by LinkedHashSet.) The backing
 * HashMap instance is a LinkedHashMap with the specified initial
 * capacity and the specified load factor.
 *
 * @param      initialCapacity   the initial capacity of the hash map
 * @param      loadFactor        the load factor of the hash map
 * @param      dummy             ignored (distinguishes this
 *             constructor from other int, float constructor.)
 * @throws     IllegalArgumentException if the initial capacity is less
 *             than zero, or if the load factor is nonpositive
 */
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
```

BitSet（位集，底层是 long 数组，用于替代 List<Boolean>）

BitSet 是位操作的对象，值只有 0 或 1 即 false 和 true，内部维护了一个 long 数组，初始只有一个 long，所以 BitSet 最小的 size 是 64（8 个字节 64 个位，可以存储 64 个数字），当随着存储的元素越来越多，BitSet 内部会动态扩充，最终内部是由 N 个 long 来存储，这些针对操作都是透明的。
默认情况下，BitSet 的所有位都是 false 即 0。
不是线程安全的。
用 1 位来表示一个数据是否出现过，0 为没有出现过，1 表示出现过。使用的时候既可根据某一个是否为 0 表示，此数是否出现过。

一个 1GB 的空间，有 8*1024*1024*1024 = 8.58*10^9bit，也就是 1GB 的空间可以表示 85 亿多个数。
常见的应用是那些需要对海量数据进行一些统计工作的时候，比如日志分析、用户数统计等等，如统计 40 亿个数据中没有出现的数据，将 40 亿个不同数据进行排序，海量数据去重等等。

JDK 选择 **long 数组**作为 BitSet 的内部存储结构是出于性能的考虑，因为 **BitSet 提供 and 和 or 这种操作，需要对两个 BitSet 中的所有 bit 位做 and 或者 or，实现的时候需要遍历所有的数组元素。使用 long 能够使得循环的次数降到最低**，所以 Java 选择使用 long 数组作为 BitSet 的内部存储结构。

`BitSet()` 创建一个新的位 set。
`BitSet(int nbits)` 创建一个位 set，它的初始大小足以显式表示索引范围

在 0 到 nbits-1 的位。	
void	and(BitSet set) 对此目标位 set 和参数位 set 执行逻辑与操作。
void	andNot(BitSet set) 清除此 BitSet 中所有的位，其相应的位在指定的 BitSet 中已设置。
int	cardinality() 返回此 BitSet 中设置为 true 的位数。
void	clear() 将此 BitSet 中的所有位设置为 false。
void	clear(int bitIndex) 将索引指定处的位设置为 false。
void	clear(int fromIndex, int toIndex) 将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的位设置为 false。
Object	clone() 复制此 BitSet，生成一个与之相等的新 BitSet。
boolean	equals(Object obj) 将此对象与指定的对象进行比较。
void	flip(int bitIndex) 将指定索引处的位设置为其当前值的补码。
void	flip(int fromIndex, int toIndex) 将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的每个位设置为其当前值的补码。
boolean	get(int bitIndex) 返回指定索引处的位值。

BitSet	get(int fromIndex, int toIndex) 返回一个新的 BitSet，它由此 BitSet 中从 fromIndex（包括）到 toIndex（不包括）范围内的位组成。
int	hashCode() 返回此位 set 的哈希码值。
boolean	intersects(BitSet set) 如果指定的 BitSet 中有设置为 true 的位，并且在此 BitSet 中也将其设置为 true，则返回 ture。
boolean	isEmpty() 如果此 BitSet 中没有包含任何设置为 true 的位，则返回 ture。
int	length() 返回此 BitSet 的"逻辑大小"：BitSet 中最高设置位的索引加 1。
int	nextClearBit(int fromIndex) 返回第一个设置为 false 的位的索引，这发生在指定的起始索引或之后的索引上。
int	nextSetBit(int fromIndex) 返回第一个设置为 true 的位的索引，这发生在指定的起始索引或之后的索引上。
void	or(BitSet set) 对此位 set 和位 set 参数执行逻辑**或**操作。
void	set(int bitIndex) 将指定索引处的位设置为 true。
void	set(int bitIndex, boolean value) 将指定索引处的位设置为指定的值。

void	set(int fromIndex, int toIndex) 将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的位设置为 true。
void	set(int fromIndex, int toIndex, boolean value) 将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的位设置为指定的值。
int	size() 返回此 BitSet 表示位值时实际使用空间的位数。
String	toString() 返回此位 set 的字符串表示形式。
void	xor(BitSet set) 对此位 set 和位 set 参数执行逻辑**异或**操作。

去重示例

```
public static void containChars(String str) {
    BitSet used = new BitSet();
    for (int i = 0; i < str.length(); i++)
        used.set(str.charAt(i)); // set bit for char
    StringBuilder sb = new StringBuilder();
    sb.append("[");
    int size = used.size();
    for (int i = 0; i < size; i++) {
        if (used.get(i)) {
            sb.append((char) i);
        }
    }
    sb.append("]");
    System.out.println(sb.toString());
}

public static void main(String[] args) {
```

```java
    containChars("abcdfab");
}
```

```
[abcdf]
```

排序示例

```java
public static void sortArray(int[] array) {

    BitSet bitSet = new BitSet(2 << 13);
    // 虽然可以自动扩容，但尽量在构造时指定估算大小,默认为 64
    System.out.println("BitSet size: " + bitSet.size());

    for (int i = 0; i < array.length; i++) {
        bitSet.set(array[i]);
    }
    //剔除重复数字后的元素个数
    int bitLen = bitSet.cardinality();

    //进行排序，即把 bit 为 true 的元素复制到另一个数组
    int[] orderedArray = new int[bitLen];
    int k = 0;
    for (int i = bitSet.nextSetBit(0); i >= 0; i = bitSet.nextSetBit(i + 1))
{
        orderedArray[k++] = i;
    }

    System.out.println("After ordering: ");
    for (int i = 0; i < bitLen; i++) {
        System.out.print(orderedArray[i] + "\t");
    }
}

public static void main(String[] args) {
    int[] array = new int[]{423, 700, 9999, 2323, 356, 6400, 1, 2, 3, 2, 2, 2,
2};
    sortArray(array);
}
```

```
BitSet size: 16384
After ordering:
1	2	3	356	423	700	2323	6400	9999
```

CopyOnWriteArraySet（底层是 CopyOnWriteArrayList）

基于 CopyOnWriteArrayList 实现，其唯一的不同是在 add 时调用的是 CopyOnWriteArrayList 的 addIfAbsent 方法。
在每次 add 的时候都要进行数组的遍历，因此其性能会略低于 CopyOnWriteArrayList。

成员变量

```
private final CopyOnWriteArrayList<E> al;
```

构造方法

```
public CopyOnWriteArraySet() {
    al = new CopyOnWriteArrayList<E>();
}
```

添加

```
public boolean add(E e) {
    return al.addIfAbsent(e);
}
```

删除

```
public boolean remove(Object o) {
    return al.remove(o);
}
```

遍历

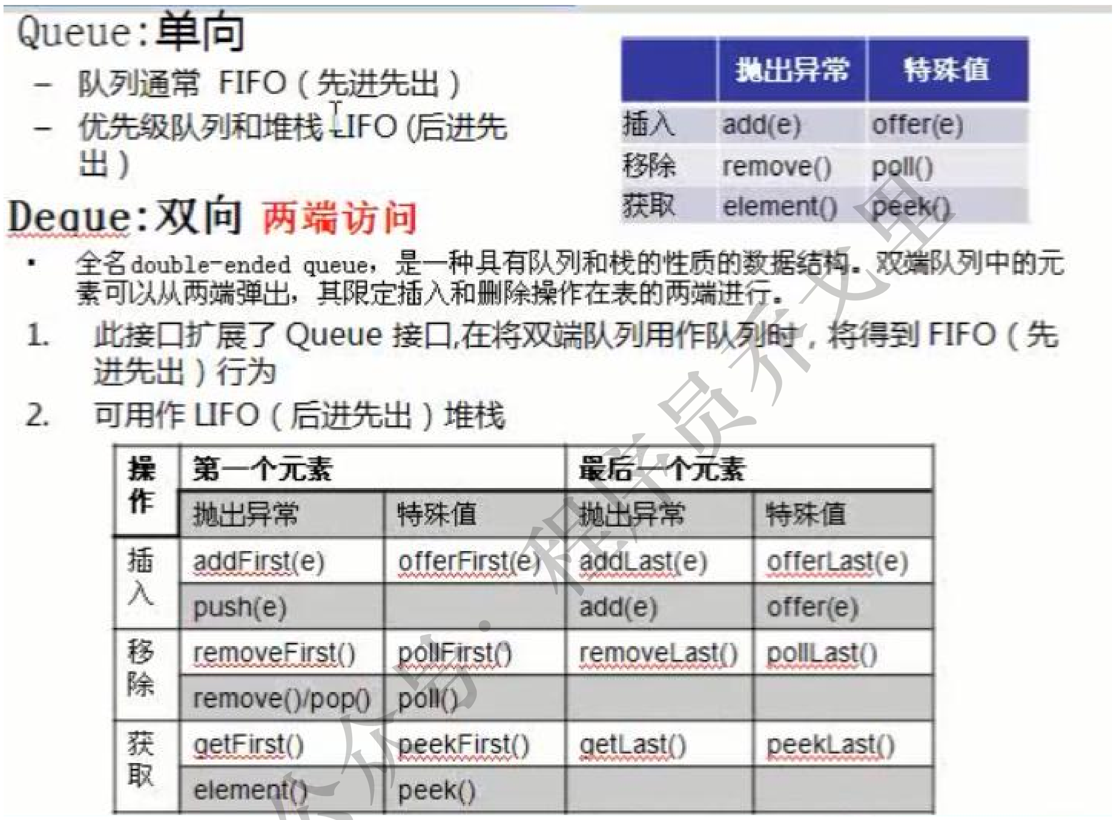
```
public Iterator<E> iterator() {
    return al.iterator();
}
```

包含

```java
public boolean contains(Object o) {
    return al.contains(o);
}
```

Queue

先进先出”（FIFO—first in first out）的线性表
LinkedList 类实现了 Queue 接口，因此我们可以把 LinkedList 当成 Queue 来用。
Java 里有一个叫做 Stack 的类，却没有叫做 Queue 的类（它是个接口名字）。当需要使用栈时，Java 已不推荐使用 Stack，而是**推荐使用更高效的 ArrayDeque**；既然 Queue 只是一个接口，当需要使用队列时也就首选 ArrayDeque 了（次选是 LinkedList）。



Queue:单向

- 队列通常 FIFO（先进先出）
- 优先级队列和堆栈 LIFO (后进先出）

	抛出异常	特殊值
插入	add(e)	offer(e)
移除	remove()	poll()
获取	element()	peek()

Deque:双向 **两端访问**

- 全名double-ended queue，是一种具有队列和栈的性质的数据结构。双端队列中的元素可以从两端弹出，其限定插入和删除操作在表的两端进行。

1. 此接口扩展了 Queue 接口,在将双端队列用作队列时，将得到 FIFO（先进先出）行为
2. 可用作 LIFO（后进先出）堆栈

操作	第一个元素		最后一个元素	
	抛出异常	特殊值	抛出异常	特殊值
插入	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
	push(e)		add(e)	offer(e)
移除	removeFirst()	pollFirst()	removeLast()	pollLast()
	remove()/pop()	poll()		
获取	getFirst()	peekFirst()	getLast()	peekLast()
	element()	peek()		

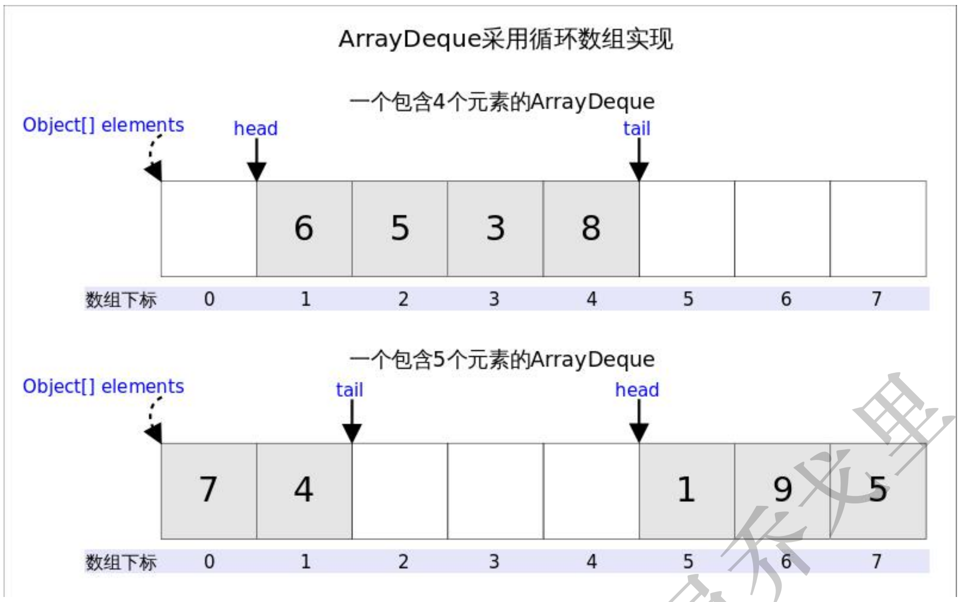
Deque 既可以作为栈使用，也可以作为队列使用。

Queue Method	**Equivalent Deque Method**	**说明**
add(e)	addLast(e)	向队尾插入元素，失败则抛出异常
remove()	removeFirst()	获取并删除队首元素，失败则抛出异常
element()	getFirst()	获取但不删除队首元素，失败则抛出异常
offer(e)	offerLast(e)	向队尾插入元素，失败则返回 false
poll()	pollFirst()	获取并删除队首元素，失败则返回 null
peek()	peekFirst()	获取但不删除队首元素，失败则返回 null

Stack Method	**Equivalent Deque Method**	**说明**
push(e)	addFirst(e)	向栈顶插入元素，失败则抛出异常
无	offerFirst(e)	向栈顶插入元素，失败则返回 false
pop()	removeFirst()	获取并删除栈顶元素，失败则抛出异常
无	pollFirst()	获取并删除栈顶元素，失败则返回 null
peek()	peekFirst()	获取但不删除栈顶元素，失败则抛出异常
无	peekFirst()	获取但不删除栈顶元素，失败则返回 null

ArrayDeque 和 LinkedList 是 Deque 的两个通用实现。

1）ArrayDeque（底层是循环数组，有界队列）



head 指向首端第一个有效元素，tail 指向尾端第一个可以插入元素的空位。因为是循环数组，所以 head 不一定总等于 0，tail 也不一定总是比 head 大。

成员变量

```
transient Object[] elements; // non-private to simplify nested class access
transient int head;
transient int tail;
private static final int MIN_INITIAL_CAPACITY = 8;
```

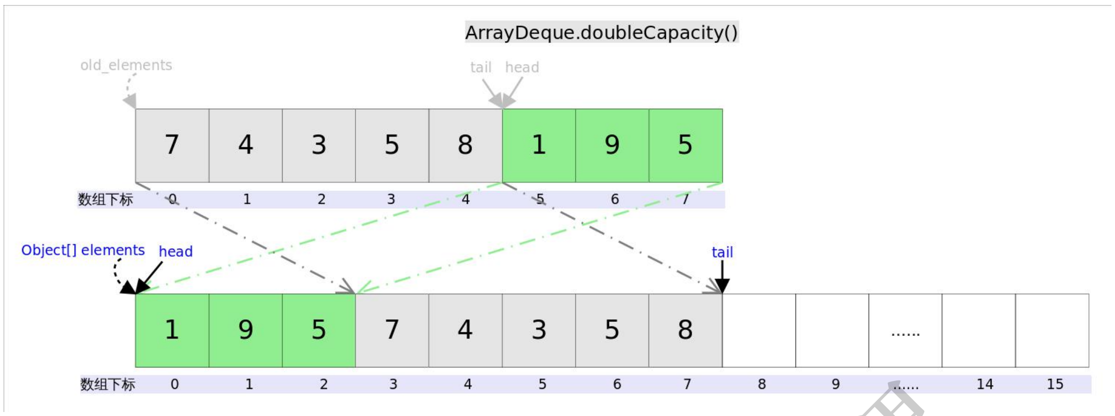
构造方法

```
public ArrayDeque() {
    elements = new Object[16];
}
```

```
public ArrayDeque(int numElements) {
    allocateElements(numElements);
}
```

```
/**
 * Allocates empty array to hold the given number of elements.
 *
 * @param numElements  the number of elements to hold
 */
private void allocateElements(int numElements) {
    int initialCapacity = MIN_INITIAL_CAPACITY;
    // Find the best power of two to hold elements.
    // Tests "<=" because arrays aren't kept full.
    if (numElements >= initialCapacity) {
        initialCapacity = numElements;
        initialCapacity |= (initialCapacity >>>  1);
        initialCapacity |= (initialCapacity >>>  2);
        initialCapacity |= (initialCapacity >>>  4);
        initialCapacity |= (initialCapacity >>>  8);
        initialCapacity |= (initialCapacity >>> 16);
        initialCapacity++;

        if (initialCapacity < 0)   // Too many elements, must back off
            initialCapacity >>>= 1;// Good luck allocating 2 ^ 30 elements
    }
    elements = new Object[initialCapacity];
}
```

扩容



```java
/**
 * Doubles the capacity of this deque.  Call only when full, i.e.,
 * when head and tail have wrapped around to become equal.
 */
private void doubleCapacity() {
    assert head == tail;
    int p = head;
    int n = elements.length;
    int r = n - p; // number of elements to the right of p
    int newCapacity = n << 1;
    if (newCapacity < 0)
        throw new IllegalStateException("Sorry, deque too big");
    Object[] a = new Object[newCapacity];
    System.arraycopy(elements, p, a, 0, r);
    System.arraycopy(elements, 0, a, r, p);
    elements = a;
    head = 0;
    tail = n;
}
```

offer

```java
public boolean offer(E e) {
    return offerLast(e);
}
```

```java
public boolean offerLast(E e) {
    addLast(e);
    return true;
}
```

```java
public void addLast(E e) {
    if (e == null)
        throw new NullPointerException();
    elements[tail] = e;
    if ( (tail = (tail + 1) & (elements.length - 1)) == head)
        doubleCapacity();
}
```

poll

```java
public E poll() {
    return pollFirst();
}
```

```java
public E pollFirst() {
    int h = head;
    @SuppressWarnings("unchecked")
    E result = (E) elements[h];
    // Element is null if deque empty
    if (result == null)
        return null;
    elements[h] = null;     // Must null out slot
    head = (h + 1) & (elements.length - 1);
    return result;
}
```

peek

```java
public E peek() {
    return peekFirst();
}
```

```java
public E peekFirst() {
    // elements[head] is null if deque empty
    return (E) elements[head];
}
```

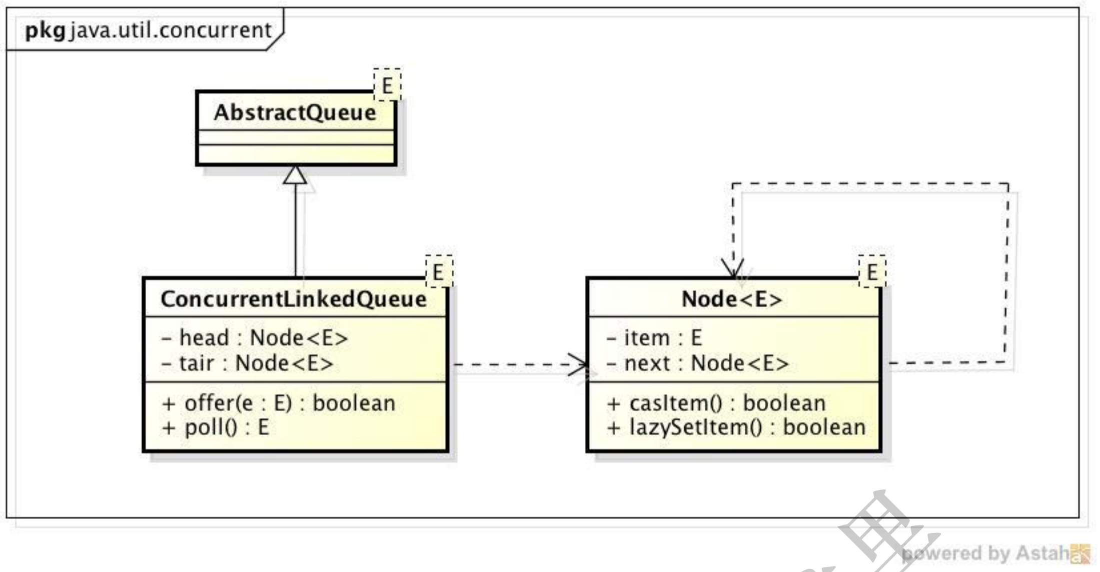
ConcurrentLinkedQueue（底层是链表，基于 CAS 的非阻塞队列，无界队列）

ConcurrentLinkedQueue 是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。它采用了“wait－free”算法（非阻塞）来实现。

1．使用 CAS 原子指令来处理对数据的并发访问，这是非阻塞算法得以实现的基础。
2. head/tail 并非总是指向队列的头 / 尾节点，也就是说允许队列处于不一致状态。 这个特性把入队 / 出队时，原本需要一起原子化执行的两个步骤分离开来，从而缩小了入队 / 出队时需要原子化更新值的范围到唯一变量。这是非阻塞算法得以实现的关键。
3. 以批处理方式来更新 head/tail，从整体上减少入队 / 出队操作的开销。
4. ConcurrentLinkedQueue 的迭代器是弱一致性的，这在并发容器中是比较普遍的现象，主要是指在一个线程在遍历队列结点而另一个线程尝试对某个队列结点进行修改的话不会抛出 ConcurrentModificationException，这也就造成在遍历某个尚未被修改的结点时，在 next 方法返回时可以看到该结点的修改，但在遍历后再对该结点修改时就看不到这种变化。

1. 在入队时最后一个结点中的 next 域为 null
2. 队列中的所有未删除结点的 item 域不能为 null 且从 head 都可以在 O(N)时间内遍历到
3. 对于要删除的结点，不是将其引用直接置为空，而是将其的 item 域先置为 null(迭代器在遍历是会跳过 item 为 null 的结点)
4. 允许 head 和 tail 滞后更新，也就是上文提到的 head/tail 并非总是指向队列的头 / 尾节点（这主要是为了减少 CAS 指令执行的次数，但同时会增加 volatile 读的次数，但是这种消耗较小）。具体而言就是，当在队列中插入一个元素是，会检测 tail 和最后一个结点之间的距离是否在两个结点及以上(内部称之为 hop)；而在出队时，对 head 的检测就是与队列的第一个结点的距离是否达到两个，有则将 head 指向第一个结点并将 head 原来指向的结点的 next 域指向自己，这样就能断开与队列的联系从而帮助 GC
head 节点并不是总指向第一个结点，tail 也并不是总指向最后一个节点。

源码过于复杂，可以先跳过。



成员变量

```java
private transient volatile Node<E> head;
private transient volatile Node<E> tail;
```

构造方法

```java
public ConcurrentLinkedQueue() {
    head = tail = new Node<E>(null);
}
```

Node#CAS 操作

在 obj 的 offset 位置比较 object field 和期望的值，如果相同则更新。这个方法的操作应该是原子的，因此提供了一种不可中断的方式更新 object field。

如果 node 的 next 值为 cmp，则将其更新为 val

```java
boolean casNext(Node<E> cmp, Node<E> val) {
    return UNSAFE.compareAndSwapObject(this, nextOffset, cmp, val);
}
```

```java
boolean casItem(E cmp, E val) {
    return UNSAFE.compareAndSwapObject(this, itemOffset, cmp, val);
}
```

```
private boolean casHead(Node<E> cmp, Node<E> val) {
    return UNSAFE.compareAndSwapObject(this, headOffset, cmp, val);
}
```

```
void lazySetNext(Node<E> val) {
    UNSAFE.putOrderedObject(this, nextOffset, val);
}
```

offer（无锁）

```
/**
 * Inserts the specified element at the tail of this queue.
 * As the queue is unbounded, this method will never return {@code false}.
 *
 * @return {@code true} (as specified by {@link Queue#offer})
 * @throws NullPointerException if the specified element is null
 */
public boolean offer(E e) {
    checkNotNull(e);
    final Node<E> newNode = new Node<E>(e);

    for (Node<E> t = tail, p = t;;) {
        Node<E> q = p.next;
        // q/p.next/tail.next 为 null，则说明 p 是尾节点，则插入
        if (q == null) {
            // CAS 插入 p.next = newNode，多线程环境下只有一个线程可以设置成功
            // 此时 tail.next = newNode
            if (p.casNext(null, newNode)) {
                // CAS 成功说明新节点已经放入链表
                // 如果 p 不为 t，说明当前线程是之前 CAS 失败后又重试 CAS 成功的，tail
= newNode
                if (p != t) // hop two nodes at a time
                    casTail(t, newNode);  // Failure is OK.
                return true;
            }
            // Lost CAS race to another thread; re-read next
        }
        else if (p == q)
            //多线程操作时候，由于 poll 时候会把老的 head 变为自引用，然后 head 的 next
变为新 head，所以这里需要重新找新的 head，因为新的 head 后面的节点才是激活的节点
            // p = head , t = tail
            p = (t != (t = tail)) ? t : head;
        else
            // 对上一次 CAS 失败的线程而言，t.next/p.next/tail.next/q 不是 null 了
            // 副作用是 p = q，p 和 q 都指向了尾节点，进入第三次循环
            p = (p != t && t != (t = tail)) ? t : q;
    }
}
```

poll（无锁）

```
public E poll() {
    restartFromHead:
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {
            // 保存当前节点的值
            E item = p.item;
            // 当前节点有值则 CAS 置为 null, p.item = null
            if (item != null && p.casItem(item, null)) {
                // CAS 成功代表当前节点已经从链表中移除

                if (p != h) // hop two nodes at a time
                    updateHead(h, ((q = p.next) != null) ? q : p);
                return item;
            } // 当前队列为空时则返回 null
            else if ((q = p.next) == null) {
                updateHead(h, p);
                return null;
            } // 自引用了，则重新找新的队列头节点
            else if (p == q)
                continue restartFromHead;
            else
                p = q;
        }
    }
}
```

```
final void updateHead(Node<E> h, Node<E> p) {
    if (h != p && casHead(h, p))
        h.lazySetNext(h);
}
```

peek（无锁）

```
public E peek() {
    restartFromHead:
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {
            E item = p.item;
            if (item != null || (q = p.next) == null) {
                updateHead(h, p);
                return item;
            }
            else if (p == q)
                continue restartFromHead;
            else
                p = q;
        }
    }
}
```

size（遍历计算大小，效率低）

```
public int size() {
    int count = 0;
    for (Node<E> p = first(); p != null; p = succ(p))
        if (p.item != null)
            // Collection.size() spec says to max out
            if (++count == Integer.MAX_VALUE)
                break;
    return count;
}
```

ConcurrentLinkedDeque（底层是双向链表，基于 CAS 的非阻塞队列，无界队列）

2）PriorityQueue（底层是数组，逻辑上是小顶堆，无界队列）

PriorityQueue 底层实现的数据结构是“堆”，堆具有以下两个性质：
任意一个节点的值总是不大于（最大堆）或者不小于（最小堆）其父节点的值；堆是一棵完全二叉树
基于数组实现的二叉堆，对于数组中任意位置的 n 上元素，其左孩子在[2n+1]位置上，右孩子[2(n+1)]位置，它的父亲则在[(n-1)/2]上，而根的位置则是[0]。

1）时间复杂度：remove()方法和 add()方法时间复杂度为 O(logn)，remove(Object obj)和 contains()方法需要 O(n)时间复杂度，取队头则需要 O(1)时间
2）在初始化阶段会执行建堆函数，最终建立的是最小堆，每次出队和入队操作不能保证队列元素的有序性，只能保证队头元素和新插入元素的有序性，如果需要有序输出队列中的元素，则只要调用 Arrays.sort()方法即可
3）可以使用 Iterator 的迭代器方法输出队列中元素
4）PriorityQueue 是非同步的，要实现同步需要调用 java.util.concurrent 包下的 PriorityBlockingQueue 类来实现同步
5）在队列中不允许使用 null 元素
6）PriorityQueue 默认是一个小顶堆，然而可以通过传入自定义的 Comparator 函数来实现大顶堆

替代：用 TreeMap 复杂度太高，有没有更好的方法。hash 方法，但是队列不是定长的，如果改变了大小要 rehash 代价太大，还有什么方法？用堆实现，那每次 get put 复杂度是多少（lgN）

成员变量

```java
transient Object[] queue; // non-private to simplify nested class access

/**
 * The number of elements in the priority queue.
 */
private int size = 0;

/**
 * The comparator, or null if priority queue uses elements'
 * natural ordering.
 */
private final Comparator<? super E> comparator;

/**
 * The number of times this priority queue has been
 * <i>structurally modified</i>.  See AbstractList for gory details.
 */
transient int modCount = 0; // non-private to simplify nested class access
```

构造方法

```
public PriorityQueue() {
    this(DEFAULT_INITIAL_CAPACITY, null);
}

/**
 * Creates a {@code PriorityQueue} with the specified initial
 * capacity that orders its elements according to their
 * {@linkplain Comparable natural ordering}.
 *
 * @param initialCapacity the initial capacity for this priority queue
 * @throws IllegalArgumentException if {@code initialCapacity} is less
 *         than 1
 */
public PriorityQueue(int initialCapacity) {
    this(initialCapacity, null);
}

/**
 * Creates a {@code PriorityQueue} with the default initial capacity and
 * whose elements are ordered according to the specified comparator.
 *
 * @param  comparator the comparator that will be used to order this
 *         priority queue.  If {@code null}, the {@linkplain Comparable
 *         natural ordering} of the elements will be used.
 * @since 1.8
 */
public PriorityQueue(Comparator<? super E> comparator) {
    this(DEFAULT_INITIAL_CAPACITY, comparator);
}

/**
 * Creates a {@code PriorityQueue} with the specified initial capacity
 * that orders its elements according to the specified comparator.
 *
 * @param  initialCapacity the initial capacity for this priority queue
 * @param  comparator the comparator that will be used to order this
 *         priority queue.  If {@code null}, the {@linkplain Comparable
 *         natural ordering} of the elements will be used.
 * @throws IllegalArgumentException if {@code initialCapacity} is
 *         less than 1
 */
public PriorityQueue(int initialCapacity,
                     Comparator<? super E> comparator) {
    // Note: This restriction of at least one is not actually needed,
    // but continues for 1.5 compatibility
    if (initialCapacity < 1)
        throw new IllegalArgumentException();
    this.queue = new Object[initialCapacity];
    this.comparator = comparator;
}
```

扩容

Double size if small; else grow by 50%

```
private void grow(int minCapacity) {
    int oldCapacity = queue.length;
    // Double size if small; else grow by 50%
    int newCapacity = oldCapacity + ((oldCapacity < 64) ?
                                     (oldCapacity + 2) :
                                     (oldCapacity >> 1));
    // overflow-conscious code
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    queue = Arrays.copyOf(queue, 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;
}
```

offer

```java
public boolean offer(E e) {
    if (e == null)
        throw new NullPointerException();
    modCount++;
    int i = size;
    if (i >= queue.length)
        grow(i + 1);
    size = i + 1;
    if (i == 0)
        queue[0] = e;
    else
        siftUp(i, e);
    return true;
}
```

```java
private void siftUp(int k, E x) {
    if (comparator != null)
        siftUpUsingComparator(k, x);
    else
        siftUpComparable(k, x);
}
```

```java
private void siftUpUsingComparator(int k, E x) {
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        Object e = queue[parent];
        if (comparator.compare(x, (E) e) >= 0)
            break;
        queue[k] = e;
        k = parent;
    }
    queue[k] = x;
}
```

```java
private void siftUpComparable(int k, E x) {
    Comparable<? super E> key = (Comparable<? super E>) x;
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        Object e = queue[parent];
        if (key.compareTo((E) e) >= 0)
            break;
        queue[k] = e;
        k = parent;
    }
    queue[k] = key;
}
```

poll

```
public E poll() {
    if (size == 0)
        return null;
    int s = --size;
    modCount++;
    E result = (E) queue[0];
    E x = (E) queue[s];
    queue[s] = null;
    if (s != 0)
        siftDown(0, x);
    return result;
}
```

```
private void siftDown(int k, E x) {
    if (comparator != null)
        siftDownUsingComparator(k, x);
    else
        siftDownComparable(k, x);
}
```

```
private void siftDownUsingComparator(int k, E x) {
    int half = size >>> 1;
    while (k < half) {
        int child = (k << 1) + 1;
        Object c = queue[child];
        int right = child + 1;
        if (right < size &&
            comparator.compare((E) c, (E) queue[right]) > 0)
            c = queue[child = right];
        if (comparator.compare(x, (E) c) <= 0)
            break;
        queue[k] = c;
        k = child;
    }
    queue[k] = x;
}
```

```
private void siftDownComparable(int k, E x) {
    Comparable<? super E> key = (Comparable<? super E>)x;
    int half = size >>> 1;        // loop while a non-leaf
    while (k < half) {
        int child = (k << 1) + 1; // assume left child is least
        Object c = queue[child];
        int right = child + 1;
        if (right < size &&
            ((Comparable<? super E>) c).compareTo((E) queue[right]) > 0)
            c = queue[child = right];
        if (key.compareTo((E) c) <= 0)
            break;
        queue[k] = c;
        k = child;
    }
    queue[k] = key;
}
```

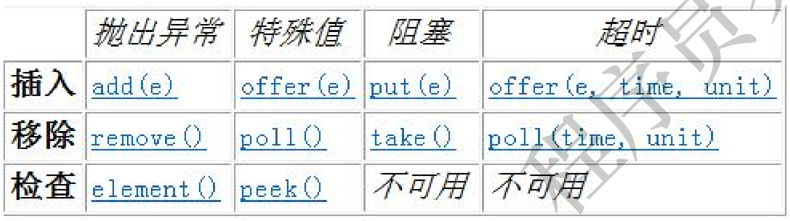
peek

```
public E peek() {
    return (size == 0) ? null : (E) queue[0];
}
```

3）BlockingQueue

对于许多多线程问题，都可以通过使用一个或多个队列以优雅的方式将其形式化
生产者线程向队列插入元素，消费者线程则取出它们。使用队列，可以安全地从一个线程向另一个线程传递数据。
比如转账
一个线程将转账指令放入队列
一个线程从队列中取出指令执行转账，只有这个线程可以访问银行对象的内部。因此不需要同步

当试图向队列中添加元素而队列已满，或是想从队列移出元素而队列为空的时候，阻塞队列导致线程阻塞
在协调多个线程之间的合作时，阻塞队列是很有用的。
工作者线程可以周期性地将中间结果放入阻塞队列，其他工作者线程取出中间结果并进一步修改。队列会自动平衡负载，大概第一个线程集比第二个运行的慢，那么第二个线程集在等待结果时会阻塞，反之亦然



	抛出异常	特殊值	阻塞	超时
插入	add(e)	offer(e)	put(e)	offer(e, time, unit)
移除	remove()	poll()	take()	poll(time, unit)
检查	element()	peek()	不可用	不可用

1）LinkedBlockingQueue 的容量是没有上边界的，是一个双向队列
2）ArrayBlockingQueue 在构造时需要指定容量，并且有一个参数来指定是否需要公平策略
3）PriorityBlockingQueue 是一个带优先级的队列，元素按照它们的优先级顺序被移走。该队列没有容量上限。
4）DelayQueue 包含实现了 **Delayed** 接口的对象
5）TransferQueue 接口允许生产者线程等待，直到消费者准备就绪可以接收一个元素。如果生产者调用 **transfer** 方法，那么这个调用会阻塞，直到插入的元素被消费者取出之后才停止阻塞。
LinkedTransferQueue 类实现了这个接口

ArrayBlockingQueue（底层是数组，阻塞队列，一把锁两个Condition，有界同步队列）

基于数组、先进先出、线程安全的集合类，特点是可实现指定时间的阻塞读写，并且容量是可限制的。

成员变量

```java
/** The queued items */
final Object[] items;

/** items index for next take, poll, peek or remove */
int takeIndex;

/** items index for next put, offer, or add */
int putIndex;

/** Number of elements in the queue */
int count;

/*
 * Concurrency control uses the classic two-condition algorithm
 * found in any textbook.
 */

/** Main lock guarding all access */
final ReentrantLock lock;

/** Condition for waiting takes */
private final Condition notEmpty;

/** Condition for waiting puts */
private final Condition notFull;

/**
 * Shared state for currently active iterators, or null if there
 * are known not to be any.  Allows queue operations to update
 * iterator state.
 */
transient Itrs itrs = null;
```

构造方法

```
public ArrayBlockingQueue(int capacity) {
    this(capacity, false);
}

/**
 * Creates an {@code ArrayBlockingQueue} with the given (fixed)
 * capacity and the specified access policy.
 *
 * @param capacity the capacity of this queue
 * @param fair if {@code true} then queue accesses for threads blocked
 *        on insertion or removal, are processed in FIFO order;
 *        if {@code false} the access order is unspecified.
 * @throws IllegalArgumentException if {@code capacity < 1}
 */
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}
```

put（有锁，队列满则阻塞）

```
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}
```

```
private void enqueue(E x) {
    // assert lock.getHoldCount() == 1;
    // assert items[putIndex] == null;
    final Object[] items = this.items;
    items[putIndex] = x;
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
    notEmpty.signal();
}
```

take（有锁，队列空则阻塞）

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```

```
private E dequeue() {
    // assert lock.getHoldCount() == 1;
    // assert items[takeIndex] != null;
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
    E x = (E) items[takeIndex];
    items[takeIndex] = null;
    if (++takeIndex == items.length)
        takeIndex = 0;
    count--;
    if (itrs != null)
        itrs.elementDequeued();
    notFull.signal();
    return x;
}
```

offer（有锁，最多阻塞一段时间）

```
public boolean offer(E e, long timeout, TimeUnit unit)
    throws InterruptedException {
```

```java
    checkNotNull(e);
    long nanos = unit.toNanos(timeout);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length) {
            if (nanos <= 0)
                return false;
            nanos = notFull.awaitNanos(nanos);
        }
        enqueue(e);
        return true;
    } finally {
        lock.unlock();
    }
}
```

poll（有锁，最多阻塞一段时间）

```java
public E poll(long timeout, TimeUnit unit) throws InterruptedException {
    long nanos = unit.toNanos(timeout);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0) {
            if (nanos <= 0)
                return null;
            nanos = notEmpty.awaitNanos(nanos);
        }
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```

peek（有锁）

```java
public E peek() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return itemAt(takeIndex); // null when queue is empty
    } finally {
        lock.unlock();
    }
```

```
final E itemAt(int i) {
    return (E) items[i];
}
```

遍历（构造迭代器加锁，遍历迭代器也加锁）

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

```
private class Itr implements Iterator<E> {
    /** Index to look for new nextItem; NONE at end */
    private int cursor;

    /** Element to be returned by next call to next(); null if none */
    private E nextItem;

    /** Index of nextItem; NONE if none, REMOVED if removed elsewhere */
    private int nextIndex;

    /** Last element returned; null if none or not detached. */
    private E lastItem;

    /** Index of lastItem, NONE if none, REMOVED if removed elsewhere */
    private int lastRet;

    /** Previous value of takeIndex, or DETACHED when detached */
    private int prevTakeIndex;

    /** Previous value of iters.cycles */
    private int prevCycles;

    /** Special index value indicating "not available" or "undefined" */
    private static final int NONE = -1;

    /**
     * Special index value indicating "removed elsewhere", that is,
     * removed by some operation other than a call to this.remove().
     */
    private static final int REMOVED = -2;

    /** Special value for prevTakeIndex indicating "detached mode" */
    private static final int DETACHED = -3;

    Itr() {
        // assert lock.getHoldCount() == 0;
        lastRet = NONE;
        final ReentrantLock lock = ArrayBlockingQueue.this.lock;
        lock.lock();
        try {
            if (count == 0) {
                // assert itrs == null;
                cursor = NONE;
                nextIndex = NONE;
                prevTakeIndex = DETACHED;
            } else {
                final int takeIndex = ArrayBlockingQueue.this.takeIndex;
                prevTakeIndex = takeIndex;
                nextItem = itemAt(nextIndex = takeIndex);
                cursor = incCursor(takeIndex);
                if (itrs == null) {
                    itrs = new Itrs(this);
                } else {
                    itrs.register(this); // in this order
                    itrs.doSomeSweeping(false);
                }
                prevCycles = itrs.cycles;
                // assert takeIndex >= 0;
                // assert prevTakeIndex == takeIndex;
                // assert nextIndex >= 0;
                // assert nextItem != null;
            }
        } finally {
            lock.unlock();
        }
    }
}
```

LinkedBlockingQueue（底层是链表，阻塞队列，两把锁，各自对应一个 Condition，无界同步队列）

另一种 BlockingQueue 的实现，基于链表，没有容量限制。
由于出队只操作队头，入队只操作队尾，这里巧妙地使用了两把锁，**对于 put 和 offer 入队操作使用一把锁，对于 take 和 poll 出队操作使用一把锁，避免了出队、入队时互相竞争锁的现象**，因此 LinkedBlockingQueue 在高并发读写都多的情况下，性能会较 ArrayBlockingQueue 好很多，在遍历以及删除的情况下则要两把锁都要锁住。
多 CPU 情况下可以在同一时刻既消费又生产。

成员变量

```java
/** The capacity bound, or Integer.MAX_VALUE if none */
private final int capacity;

/** Current number of elements */
private final AtomicInteger count = new AtomicInteger();

/**
 * Head of linked list.
 * Invariant: head.item == null
 */
transient Node<E> head;

/**
 * Tail of linked list.
 * Invariant: last.next == null
 */
private transient Node<E> last;

/** Lock held by take, poll, etc */
private final ReentrantLock takeLock = new ReentrantLock();

/** Wait queue for waiting takes */
private final Condition notEmpty = takeLock.newCondition();

/** Lock held by put, offer, etc */
private final ReentrantLock putLock = new ReentrantLock();

/** Wait queue for waiting puts */
private final Condition notFull = putLock.newCondition();
```

构造方法

```java
public LinkedBlockingQueue() {
    this(Integer.MAX_VALUE);
}

/**
 * Creates a {@code LinkedBlockingQueue} with the given (fixed) capacity.
 *
 * @param capacity the capacity of this queue
 * @throws IllegalArgumentException if {@code capacity} is not greater
 *         than zero
 */
public LinkedBlockingQueue(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
    last = head = new Node<E>(null);
}
```

put（加 putLock 锁，队列满则阻塞）

```java
/**
 * Inserts the specified element at the tail of this queue, waiting if
 * necessary for space to become available.
 *
 * @throws InterruptedException {@inheritDoc}
 * @throws NullPointerException {@inheritDoc}
 */
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    // Note: convention in all put/take/etc is to preset local var
    // holding count negative to indicate failure unless set.
    int c = -1;
    Node<E> node = new Node<E>(e);
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly();
    try {
        /*
         * Note that count is used in wait guard even though it is
         * not protected by lock. This works because count can
         * only decrease at this point (all other puts are shut
         * out by lock), and we (or some other waiting put) are
         * signalled if it ever changes from capacity. Similarly
         * for all other uses of count in other wait guards.
         */
        while (count.get() == capacity) {
            // 阻塞，直至有剩余空间
            notFull.await();
        }
        enqueue(node);
        c = count.getAndIncrement();
        if (c + 1 < capacity)
            // 还有剩余空间时，唤醒其他生产者
            notFull.signal();
    } finally {
        putLock.unlock();
    }
    if (c == 0)
        // c 是放入当前元素之前队列的容量，现在新添加一个元素，那么唤醒消费者进行消
费
        signalNotEmpty();
}
```

```
private void enqueue(Node<E> node) {
    // assert putLock.isHeldByCurrentThread();
    // assert last.next == null;
    last = last.next = node;
}
```

```
private void signalNotEmpty() {
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        // 唤醒消费线程
        notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
}
```

take（加 takeLock 锁，队列空则阻塞）

```java
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        while (count.get() == 0) {
            // 队列空则阻塞
            notEmpty.await();
        }
        x = dequeue();
        c = count.getAndDecrement();
        if (c > 1)
            // 还有元素则唤醒其他消费者
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
    if (c == capacity)
        // c 是消费当前元素之前队列的容量，现在的容量是 c-1，可以继续放入元素，唤醒生产者进行生产
        signalNotFull();
    return x;
}
```

```java
private E dequeue() {
    // assert takeLock.isHeldByCurrentThread();
    // assert head.item == null;
    Node<E> h = head;
    Node<E> first = h.next;
    h.next = h; // help GC
    head = first;
    E x = first.item;
    first.item = null;
    return x;
}
```

```java
private void signalNotFull() {
    final ReentrantLock putLock = this.putLock;
    putLock.lock();
    try {
        // 唤醒生产者
        notFull.signal();
    } finally {
        putLock.unlock();
    }
}
```

peek（加 takeLock 锁）

```
public E peek() {
    if (count.get() == 0)
        return null;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        Node<E> first = head.next;
        if (first == null)
            return null;
        else
            return first.item;
    } finally {
        takeLock.unlock();
    }
}
```

remove（加两把锁）

```
/**
 * Locks to prevent both puts and takes.
 */
void fullyLock() {
    putLock.lock();
    takeLock.lock();
}

/**
 * Unlocks to allow both puts and takes.
 */
void fullyUnlock() {
    takeLock.unlock();
    putLock.unlock();
}
```

```
public boolean remove(Object o) {
    if (o == null) return false;
    fullyLock();
    try {
        for (Node<E> trail = head, p = trail.next;
             p != null;
             trail = p, p = p.next) {
            if (o.equals(p.item)) {
                unlink(p, trail);
                return true;
            }
        }
        return false;
    } finally {
        fullyUnlock();
    }
}
```

遍历（加两把锁）

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

private class Itr implements Iterator<E> {
    /*
     * Basic weakly-consistent iterator.  At all times hold the next
     * item to hand out so that if hasNext() reports true, we will
     * still have it to return even if lost race with a take etc.
     */

    private Node<E> current;
    private Node<E> lastRet;
    private E currentElement;

    Itr() {
        fullyLock();
        try {
            current = head.next;
            if (current != null)
                currentElement = current.item;
        } finally {
            fullyUnlock();
        }
    }

    public boolean hasNext() {
        return current != null;
    }

    /**
     * Returns the next live successor of p, or null if no such.
     *
     * Unlike other traversal methods, iterators need to handle both:
     * - dequeued nodes (p.next == p)
     * - (possibly multiple) interior removed nodes (p.item == null)
     */
    private Node<E> nextNode(Node<E> p) {
        for (;;) {
            Node<E> s = p.next;
            if (s == p)
                return head.next;
            if (s == null || s.item != null)
                return s;
            p = s;
        }
    }

    public E next() {
        fullyLock();
        try {
            if (current == null)
                throw new NoSuchElementException();
            E x = currentElement;
            lastRet = current;
            current = nextNode(current);
            currentElement = (current == null) ? null : current.item;
            return x;
        } finally {
            fullyUnlock();
        }
    }
    public void remove() {
        if (lastRet == null)
            throw new IllegalStateException();
        fullyLock();
        try {
            Node<E> node = lastRet;
            lastRet = null;
            for (Node<E> trail = head, p = trail.next;
                 p != null;
                 trail = p, p = p.next) {
                if (p == node) {
                    unlink(p, trail);
                    break;
                }
            }
        } finally {
            fullyUnlock();
        }
    }
}
```

LinkedBlockingDeque（底层是双向链表，阻塞队列，一把锁两个 Condition，无界同步队列）

LinkedBlockingDeque 是一个基于链表的双端阻塞队列。和 LinkedBlockingQueue 类似，区别在于该类实现了 Deque 接口，而 LinkedBlockingQueue 实现了 Queue 接口。
LinkedBlockingDeque 内部只有一把锁以及该锁上关联的两个条件，所以可以推断同一时刻只有一个线程可以在队头或者队尾执行入队或出队操作（类似于 ArrayBlockingQueue）。可以发现这点和 LinkedBlockingQueue 不同，LinkedBlockingQueue 可以同时有两个线程在两端执行操作。

LinkedBlockingDeque 和 LinkedBlockingQueue 的相同点在于：
1. 基于链表
2. 容量可选，不设置的话，就是 Int 的最大值

和 LinkedBlockingQueue 的不同点在于：
1. 双端链表和单链表
2. 不存在哨兵节点
3. 一把锁+两个条件

LinkedBlockingDeque 和 ArrayBlockingQueue 的相同点在于：使用一把锁+两个条件维持队列的同步。

PriorityBlockingQueue（底层是数组，出队时队空则阻塞；无界队列，不存在队满情况，一把锁一个 Condition）

支持优先级的无界阻塞队列。默认情况下元素采用自然顺序升序排序，当然我们也可以通过构造函数来指定 Comparator 来对元素进行排序。需要注意的是 PriorityBlockingQueue 不能保证同优先级元素的顺序。

成员变量

```
private static final int DEFAULT_INITIAL_CAPACITY = 11;

/**
 * The maximum size of array to allocate.
 * Some VMs reserve some header words in an array.
 * Attempts to allocate larger arrays may result in
 * OutOfMemoryError: Requested array size exceeds VM limit
 */
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

/**
 * Priority queue represented as a balanced binary heap: the two
 * children of queue[n] are queue[2*n+1] and queue[2*(n+1)].  The
 * priority queue is ordered by comparator, or by the elements'
 * natural ordering, if comparator is null: For each node n in the
 * heap and each descendant d of n, n <= d.  The element with the
 * lowest value is in queue[0], assuming the queue is nonempty.
 */
private transient Object[] queue;

/**
 * The number of elements in the priority queue.
 */
private transient int size;

/**
 * The comparator, or null if priority queue uses elements'
 * natural ordering.
 */
private transient Comparator<? super E> comparator;

/**
 * Lock used for all public operations
 */
private final ReentrantLock lock;

/**
 * Condition for blocking when empty
 */
private final Condition notEmpty;

/**
 * Spinlock for allocation, acquired via CAS.
 */
private transient volatile int allocationSpinLock;

/**
 * A plain PriorityQueue used only for serialization,
 * to maintain compatibility with previous versions
 * of this class. Non-null only during serialization/deserialization.
 */
private PriorityQueue<E> q;
```

构造方法

```
public PriorityBlockingQueue() {
    this(DEFAULT_INITIAL_CAPACITY, null);
}

/**
 * Creates a {@code PriorityBlockingQueue} with the specified
 * initial capacity that orders its elements according to their
 * {@linkplain Comparable natural ordering}.
 *
 * @param initialCapacity the initial capacity for this priority queue
 * @throws IllegalArgumentException if {@code initialCapacity} is less
 *         than 1
 */
public PriorityBlockingQueue(int initialCapacity) {
    this(initialCapacity, null);
}

/**
 * Creates a {@code PriorityBlockingQueue} with the specified initial
 * capacity that orders its elements according to the specified
 * comparator.
 *
 * @param initialCapacity the initial capacity for this priority queue
 * @param  comparator the comparator that will be used to order this
 *         priority queue.  If {@code null}, the {@linkplain Comparable
 *         natural ordering} of the elements will be used.
 * @throws IllegalArgumentException if {@code initialCapacity} is less
 *         than 1
 */
public PriorityBlockingQueue(int initialCapacity,
                             Comparator<? super E> comparator) {
    if (initialCapacity < 1)
        throw new IllegalArgumentException();
    this.lock = new ReentrantLock();
    this.notEmpty = lock.newCondition();
    this.comparator = comparator;
    this.queue = new Object[initialCapacity];
}
```

扩容（基于 CAS+Lock，CAS 控制创建新的数组原子执行，Lock 控制数组替换原子执行）

```
private void tryGrow(Object[] array, int oldCap) {
    lock.unlock(); // must release and then re-acquire main lock
    Object[] newArray = null;
    if (allocationSpinLock == 0 &&
        UNSAFE.compareAndSwapInt(this, allocationSpinLockOffset,
                                 0, 1)) {
        try {
            int newCap = oldCap + ((oldCap < 64) ?
                                   (oldCap + 2) : // grow faster if small
                                   (oldCap >> 1));
            if (newCap - MAX_ARRAY_SIZE > 0) {    // possible overflow
                int minCap = oldCap + 1;
                if (minCap < 0 || minCap > MAX_ARRAY_SIZE)
                    throw new OutOfMemoryError();
                newCap = MAX_ARRAY_SIZE;
            }
            if (newCap > oldCap && queue == array)
                newArray = new Object[newCap];
        } finally {
            allocationSpinLock = 0;
        }
    }
```

```java
    if (newArray == null) // back off if another thread is allocating
        Thread.yield();
    lock.lock();
    if (newArray != null && queue == array) {
        queue = newArray;
        System.arraycopy(array, 0, newArray, 0, oldCap);
    }
}
```

put（有锁）

```java
public void put(E e) {
    offer(e); // never need to block
}
```

```java
public boolean offer(E e) {
    if (e == null)
        throw new NullPointerException();
    final ReentrantLock lock = this.lock;
    lock.lock();
    int n, cap;
    Object[] array;
    while ((n = size) >= (cap = (array = queue).length))
        tryGrow(array, cap);
    try {
        Comparator<? super E> cmp = comparator;
        if (cmp == null)
            siftUpComparable(n, e, array);
        else
            siftUpUsingComparator(n, e, array, cmp);
        size = n + 1;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
    return true;
}
```

take（有锁）

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    E result;
    try {
        while ( (result = dequeue()) == null)
            notEmpty.await();
    } finally {
        lock.unlock();
    }
    return result;
}
```

```
private E dequeue() {
    int n = size - 1;
    if (n < 0)
        return null;
    else {
        Object[] array = queue;
        E result = (E) array[0];
        E x = (E) array[n];
        array[n] = null;
        Comparator<? super E> cmp = comparator;
        if (cmp == null)
            siftDownComparable(0, x, array, n);
        else
            siftDownUsingComparator(0, x, array, n, cmp);
        size = n;
        return result;
    }
}
```

peek（有锁）

```
public E peek() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return (size == 0) ? null : (E) queue[0];
    } finally {
        lock.unlock();
    }
}
```

DelayQueue（底层是 PriorityQueue，无界阻塞队列，过期元素方可移除，基于 Lock）

```
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
    implements BlockingQueue<E> {

    private final transient ReentrantLock lock = new ReentrantLock();
    private final PriorityQueue<E> q = new PriorityQueue<E>();
```

DelayQueue 队列中每个元素都有个过期时间，并且队列是个优先级队列，当从队列获取元素时候，只有过期元素才会出队列。
每个元素都必须实现 Delayed 接口

```
public interface Delayed extends Comparable<Delayed> {

    /**
     * Returns the remaining delay associated with this object, in the
     * given time unit.
     *
     * @param unit the time unit
     * @return the remaining delay; zero or negative values indicate
     * that the delay has already elapsed
     */
    long getDelay(TimeUnit unit);
}
```

getDelay 方法返回对象的残留延迟，负值表示延迟结束
元素只有在延迟用完的时候才能从 DelayQueue 移出。还必须实现 Comparable 接口。

一个典型场景是重试机制的实现，比如当调用接口失败后，把当前调用信息放入 delay=10s 的元素，然后把元素放入队列，那么这个队列就是一个重试队列，一个线程通过 take 方法获取需要重试的接口，take 返回则接口进行重试，失败则再次放入队列，同时也可以在元素加上重试次数。

成员变量

```
private final transient ReentrantLock lock = new ReentrantLock();
private final PriorityQueue<E> q = new PriorityQueue<E>();

private Thread leader = null;

private final Condition available = lock.newCondition();
```

构造方法

```java
public DelayQueue() {}
```

put

```java
public void put(E e) {
    offer(e);
}
```

```java
public boolean offer(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        q.offer(e);
        if (q.peek() == e) {
            leader = null;
            // 通知最先等待的线程
            available.signal();
        }
        return true;
    } finally {
        lock.unlock();
    }
}
```

take

获取并移除队列首元素，如果队列没有过期元素则等待。
第一次调用 take 时候由于队列空，所以调用（2）把当前线程放入 available 的条件队列等待，当执行 offer 并且添加的元素就是队首元素时候就会通知最先等待的线程激活，循环重新获取队首元素，这时候 first 假如不空，则调用 getdelay 方法看该元素海剩下多少时间就过期了，如果 delay<=0 则说明已经过期，则直接出队返回。否则看 leader 是否为 null，不为 null 则说明是其他线程也在执行 take 则把该线程放入条件队列，否则是当前线程执行的 take 方法，则调用(5) await 直到剩余过期时间到（这期间该线程会释放锁，所以其他线程可以 offer 添加元素，也可以 take 阻塞自己），剩余过期时间到后，该线程会重新竞争得到锁，重新进入循环。
（6）说明当前 take 返回了元素，如果当前队列还有元素则调用 singal 激活条件队列里面可能有的等待线程。leader 那么为 null，那么是第一次调用 take 获取过期元素的线程，第一次调用的线程调用设置等待时间的 await 方法等待数据过期，后面调用 take 的线程则调用 await

直到 signal。

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            // 1）获取但不移除队首元素
            E first = q.peek();
            if (first == null)
                // 2）无元素，则阻塞
                available.await();
            else {
                long delay = first.getDelay(NANOSECONDS);
                // 3）有元素，且已经过期，则移除
                if (delay <= 0)
                    return q.poll();
                first = null; // don't retain ref while waiting
                // 4）
                if (leader != null)
                    available.await();
                else {
                    Thread thisThread = Thread.currentThread();
                    // 5）
                    leader = thisThread;
                    try {
                        // 继续阻塞延迟的时间
                        available.awaitNanos(delay);
                    } finally {
                        if (leader == thisThread)
                            leader = null;
                    }
                }
            }
        }
    } finally {
        if (leader == null && q.peek() != null)
            available.signal();
        lock.unlock();
    }
}
```

peek

SynchronousQueue(只存储一个元素,阻塞队列,基于 CAS)

实现了 BlockingQueue，是一个阻塞队列。
一个只存储一个元素的的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入一直处于阻塞状态，吞吐量高于 LinkedBlockingQueue。
SynchronousQueue 内部并没有数据缓存空间，你不能调用 peek()方法来看队列中是否有数据元素，因为数据元素只有当你试着取走的时候才可能存在，不取走而只想偷窥一下是不行的，当然遍历这个队列的操作也是不允许的。队列头元素是第一个排队要插入数据的线程，而不是要交换的数据。**数据是在配对的生产者和消费者线程之间直接传递的，并不会将数据缓冲数据到队列中。可以这样来理解：生产者和消费者互相等待对方，握手，然后一起离开。**

// 如果为 true，则等待线程以 FIFO 的顺序竞争访问；否则顺序是未指定的。
// SynchronousQueue<Integer> sc =new SynchronousQueue<>(true);//fair -
SynchronousQueue<Integer> sc = new SynchronousQueue<>(); // 默认不指定的话是 false，不公平的

4）TransferQueue（特殊的 BlockingQueue）

生产者会一直阻塞直到所添加到队列的元素被某一个消费者所消费（不仅仅是添加到队列里就完事）
当我们不想生产者过度生产消息时，TransferQueue 可能非常有用，可避免发生 OutOfMemory 错误。在这样的设计中，消费者的消费能力将决定生产者产生消息的速度。

```java
public interface TransferQueue<E> extends BlockingQueue<E> {
    /**
     * 立即转交一个元素给消费者，如果此时队列没有消费者，那就 false
     */
    boolean tryTransfer(E e);

    /**
     * 转交一个元素给消费者，如果此时队列没有消费者，那就阻塞
     */
    void transfer(E e) throws InterruptedException;

    /**
     * 带超时的 tryTransfer
     */
    boolean tryTransfer(E e, long timeout, TimeUnit unit)
        throws InterruptedException;

    /**
     * 是否有消费者等待接收数据，瞬时状态，不一定准
     */
    boolean hasWaitingConsumer();

    /**
     * 返回还有多少个等待的消费者，跟上面那个一样，都是一种瞬时状态，不一定准
     */
    int getWaitingConsumerCount();
}
```

LinkedTransferQueue（底层是链表，阻塞队列，无界同步队列）

LinkedTransferQueue 实现了 TransferQueue 接口，这个接口继承了 BlockingQueue。之前 BlockingQueue 是队列满时再入队会阻塞，而这个接口实现的功能是队列不满时也可以阻塞，实现一种有阻塞的入队功能。
LinkedTransferQueue 实际上是 ConcurrentLinkedQueue、SynchronousQueue（公平模式）和 LinkedBlockingQueue 的超集。而且 LinkedTransferQueue 更好用，因为它不仅仅综合了这几个类的功能，同时也提供了更高效的实现。

5）Queue 实现类之间的区别

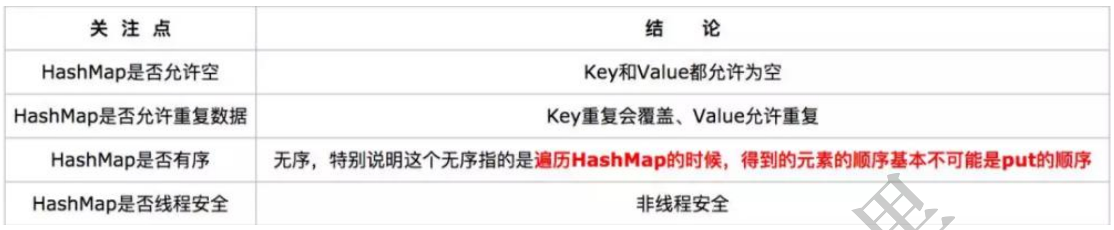
非线程安全的：ArrayDeque、LinkedList、PriorityQueue
线程安全的：ConcurrentLinkedQueue、ConcurrentLinkedDeque、ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue
线程安全的又分为阻塞队列和非阻塞队列，阻塞队列提供了 put、take 等会阻塞当前线程的方法，比如 ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue，也有 offer、poll 等阻塞一段时间候返回的方法；
非阻塞队列是使用 CAS 机制保证 offer、poll 等可以线程安全地入队出队，并且不需要加锁，不会阻塞当前线程，比如 ConcurrentLinkedQueue、ConcurrentLinkedDeque。

ArrayBlockingQueue 和 LinkedBlockingQueue 区别

1. 队列中锁的实现不同
   ArrayBlockingQueue 实现的队列中的锁是没有分离的，即生产和消费用的是同一个锁；
   LinkedBlockingQueue 实现的队列中的锁是分离的，即生产用的是 putLock，消费是 takeLock
2. 底层实现不同
前者基于数组，后者基于链表
3. 队列边界不同
   ArrayBlockingQueue 实现的队列中必须指定队列的大小，是有界队列
   LinkedBlockingQueue 实现的队列中可以不指定队列的大小，但是默认是 Integer.MAX_VALUE，是无界队列

Map

HashMap（底层是数组+链表/红黑树，无序键值对集合，非线程安全）



关 注 点	结 论
HashMap是否允许空	Key和Value都允许为空
HashMap是否允许重复数据	Key重复会覆盖、Value允许重复
HashMap是否有序	无序，特别说明这个无序指的是遍历HashMap的时候，得到的元素的顺序基本不可能是put的顺序
HashMap是否线程安全	非线程安全

基于哈希表实现，链地址法。
loadFactor 默认为 0.75，threshold（阈）为 12，并创建一个大小为 16 的 Entry 数组。
在遍历时是无序的，如需有序，建议使用 TreeMap。
采用数组方式存储 key、value 构成的 Entry 对象，无容量限制。
基于 key hash 寻找 Entry 对象存放在数组中的位置，对于 hash 冲突采用链表/红黑树的方式来解决。
HashMap 在插入元素时可能会扩大数组的容量，在扩大容量时需要重新计算 hash，并复制对象到新的数组中。
是非线程安全的。

// 1. 哈希冲突时采用链表法的类，一个哈希桶多于 8 个元素改为 TreeNode
static class Node<K,V> implements Map.Entry<K,V>
// 2. 哈希冲突时采用红黑树存储的类，一个哈希桶少于 6 个元素改为 Node
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V>

某个桶对应的链表过长的话搜索效率低，改为红黑树效率会提高。

为何按位与而不是取摸 hashmap 的 iterator 读取时是否会读到另一个线程 put 的数据 红黑树；hashmap 报 ConcurrentModificationException 的情况

Hash 冲突中链表结构的数量大于 8 个，则调用树化转为红黑树结构，红黑树查找稍微快些；红黑树结构的数量小于 6 个时，则转为链表结构
如果加载因子越大，对空间的利用更充分，但是查找效率会降低（链表长度会越来越长）；如果加载因子太小，那么表中的数据将过于稀疏（很多空间还没用，就开始扩容了），对空间造成严重浪费。如果我们在构造方法中不指定，则系统默认加载因子为 0.75，这是一个比较理想的值，一般情况下我们是无需修改的。
一般对哈希表的散列很自然地会想到用 hash 值对 length 取模（即除法散列法），Hashtable 中也是这样实现的，这种方法基本能保证元素在哈希表中散列的比较均匀，但取模会用到除法运算，效率很低，HashMap 中则通过 h&(length-1)的方法来代替取模，同样实现了均匀的散列，但效率要高很多，这也是 HashMap 对 Hashtable 的一个改进。
哈希表的容量一定要是 2 的整数次幂。首先，length 为 2 的整数次幂的话，h&(length-1)就

相当于对 length 取模，这样便保证了散列的均匀，同时也提升了效率；其次，**length 为 2 的整数次幂的话，为偶数，这样 length-1 为奇数，奇数的最后一位是 1，这样便保证了 h&(length-1)的最后一位可能为 0，也可能为 1**（这取决于 h 的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，而如果 length 为奇数的话，很明显 length-1 为偶数，它的最后一位是 0，这样 h&(length-1)的最后一位肯定为 0，即只能为偶数，**这样任何 hash 值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间，**因此，length 取 2 的整数次幂，是为了使不同 hash 值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。

Map#Entry（接口）

```
interface Entry<K,V> {
    K getKey();

    V getValue();

    V setValue(V value);

    boolean equals(Object o);
    int hashCode();
    public static <K extends Comparable<? super K>, V>
Comparator<Map.Entry<K,V>> comparingByKey() {
        return (Comparator<Map.Entry<K, V>> & Serializable)
            (c1, c2) -> c1.getKey().compareTo(c2.getKey());
    }

    public static <K, V extends Comparable<? super V>>
Comparator<Map.Entry<K,V>> comparingByValue() {
        return (Comparator<Map.Entry<K, V>> & Serializable)
            (c1, c2) -> c1.getValue().compareTo(c2.getValue());
    }

    public static <K, V> Comparator<Map.Entry<K, V>> comparingByKey(Comparator<?
super K> cmp) {
        Objects.requireNonNull(cmp);
        return (Comparator<Map.Entry<K, V>> & Serializable)
            (c1, c2) -> cmp.compare(c1.getKey(), c2.getKey());
    }

    public static <K, V> Comparator<Map.Entry<K, V>>
comparingByValue(Comparator<? super V> cmp) {
        Objects.requireNonNull(cmp);
        return (Comparator<Map.Entry<K, V>> & Serializable)
            (c1, c2) -> cmp.compare(c1.getValue(), c2.getValue());
    }
}
```

HashMap#Node（Map.Entry 的实现，链表的基本元素）

```
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;
    }
}
```

HashMap#TreeNode（Map.Entry 的实现，红黑树的基本元素）

```java
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent;  // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;    // needed to unlink next upon deletion
    boolean red;
    TreeNode(int hash, K key, V val, Node<K,V> next) {
        super(hash, key, val, next);
    }
    //...
}
```

LinkedHashMap#Entry

```java
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }
}
```

成员变量

```java
/**
 * The default initial capacity - MUST be a power of two.
 */
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

/**
 * The maximum capacity, used if a higher value is implicitly specified
 * by either of the constructors with arguments.
 * MUST be a power of two <= 1<<30.
 */
static final int MAXIMUM_CAPACITY = 1 << 30;

/**
 * The load factor used when none specified in constructor.
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;

/**
 * The bin count threshold for using a tree rather than list for a
 * bin.  Bins are converted to trees when adding an element to a
 * bin with at least this many nodes. The value must be greater
 * than 2 and should be at least 8 to mesh with assumptions in
 * tree removal about conversion back to plain bins upon
 * shrinkage.
 */
static final int TREEIFY_THRESHOLD = 8;

/**
 * The bin count threshold for untreeifying a (split) bin during a
 * resize operation. Should be less than TREEIFY_THRESHOLD, and at
 * most 6 to mesh with shrinkage detection under removal.
 */
static final int UNTREEIFY_THRESHOLD = 6;

/**
 * The smallest table capacity for which bins may be treeified.
 * (Otherwise the table is resized if too many nodes in a bin.)
 * Should be at least 4 * TREEIFY_THRESHOLD to avoid conflicts
 * between resizing and treeification thresholds.
 */
static final int MIN_TREEIFY_CAPACITY = 64;
```

```java
/**
 * The table, initialized on first use, and resized as
 * necessary. When allocated, length is always a power of two.
 * (We also tolerate length zero in some operations to allow
 * bootstrapping mechanics that are currently not needed.)
 */
transient Node<K,V>[] table;

/**
 * Holds cached entrySet(). Note that AbstractMap fields are used
 * for keySet() and values().
 */
transient Set<Map.Entry<K,V>> entrySet;

/**
 * The number of key-value mappings contained in this map.
 */
transient int size;

/**
 * The number of times this HashMap has been structurally modified
 * Structural modifications are those that change the number of mappings in
 * the HashMap or otherwise modify its internal structure (e.g.,
 * rehash).  This field is used to make iterators on Collection-views of
 * the HashMap fail-fast.  (See ConcurrentModificationException).
 */
transient int modCount;

/**
 * The next size value at which to resize (capacity * load factor).
 *
 * @serial
 */
// (The javadoc description is true upon serialization.
// Additionally, if the table array has not been allocated, this
// field holds the initial array capacity, or zero signifying
// DEFAULT_INITIAL_CAPACITY.)
// HashMap 的阈值，用于判断是否需要调整 HashMap 的容量（threshold = 容量*装载因子）
int threshold;

/**
 * The load factor for the hash table.
 *
 * @serial
 */
final float loadFactor;
```

AbstractMap

```
transient Set<K>        keySet;
transient Collection<V> values;
```

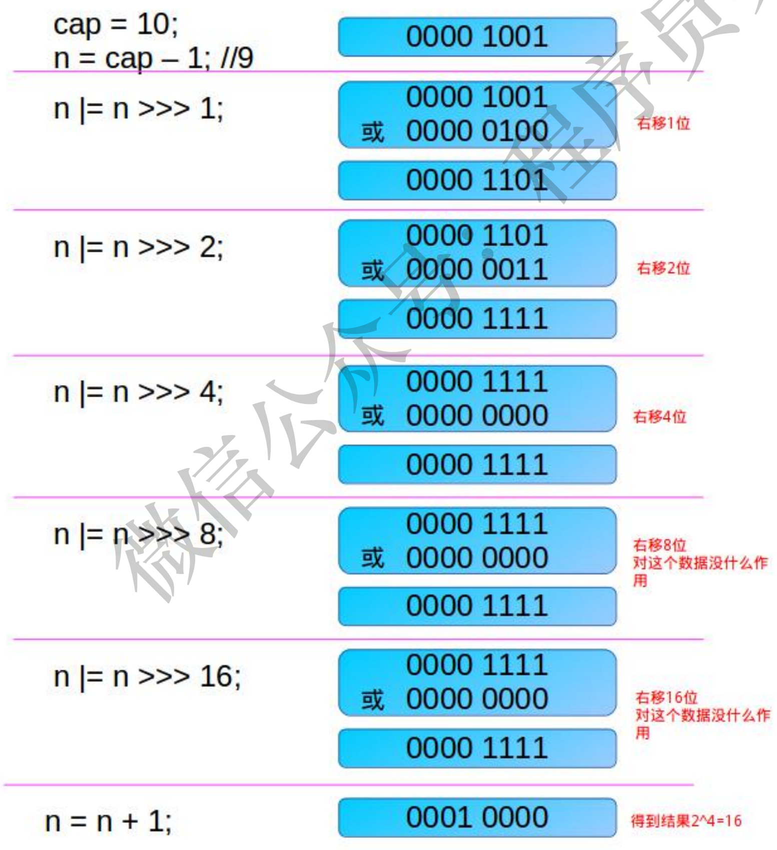
构造方法

注意哪怕是指定了初始容量，也不会直接初始化 table，而是在第一次 put 时调用 resize 来初始化 table，resize 里会将 threshold 视为初始容量。

```
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;
    // 阈值为不小于容量的 2 的幂次
    this.threshold = tableSizeFor(initialCapacity);
}

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

/**
 * 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
}
```

tableSizeFor（找到大于等于 initialCapacity 的最小的 2 的幂次以及原因）

```
/**
 * Returns a power of two size for the given target capacity.
 */
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;
}
```

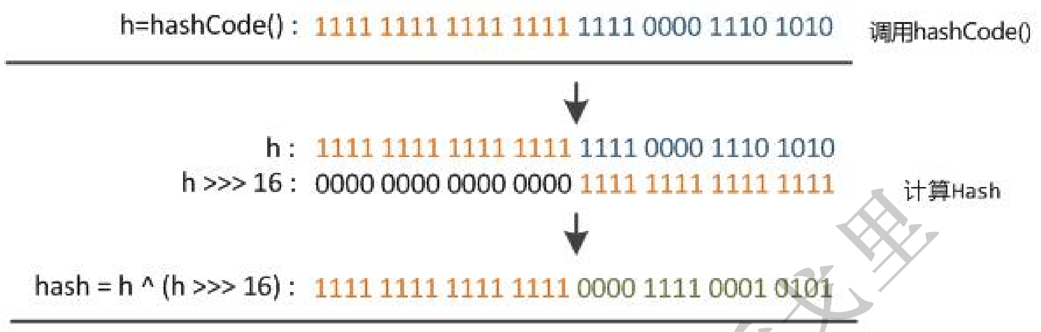


hash（hash 算法，算法比较高效、均匀）

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

key 的 hash 值高 16 位不变，低 16 位与高 16 位异或作为 key 的最终 hash 值。（h >>> 16，表示无符号右移 16 位，高位补 0，任何数跟 0 异或都是其本身，因此 key 的 hash 值高 16 位不变。）

保证了对象的 hashCode 的高 16 位的变化能反应到低 16 位中，


h=hashCode() : 1111 1111 1111 1111 1111 0000 1110 1010
调用hashCode()
h : 1111 1111 1111 1111 1111 0000 1110 1010
h >>> 16 : 0000 0000 0000 0000 1111 1111 1111 1111
计算Hash
hash = h ^ (h >>> 16) : 1111 1111 1111 1111 0000 1111 0001 0101

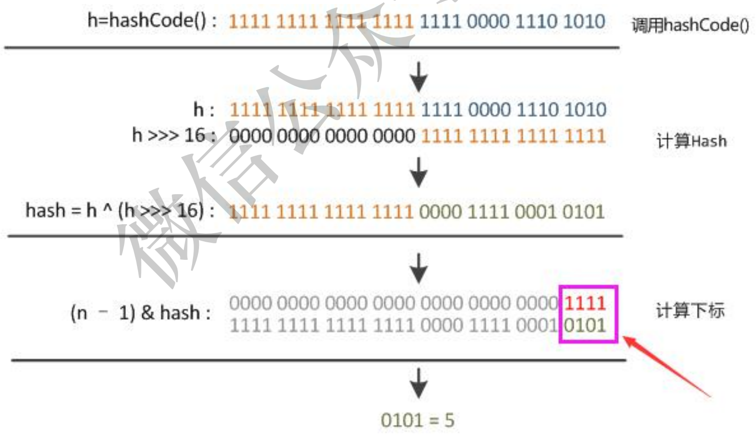
hash to index

如何根据 hash 值计算 index？（put 和 get 中的代码）

n = table.length;

index = (n-1)& hash;

当 n 总是 2 的 n 次方时，hash & (n-1)运算等价于 h%n，但是&比%具有更高的效率。


h=hashCode() : 1111 1111 1111 1111 1111 0000 1110 1010
调用hashCode()
h : 1111 1111 1111 1111 1111 0000 1110 1010
h >>> 16 : 0000 0000 0000 0000 1111 1111 1111 1111
计算Hash
hash = h ^ (h >>> 16) : 1111 1111 1111 1111 0000 1111 0001 0101
(n - 1) & hash : 0000 0000 0000 0000 0000 0000 0000 1111
1111 1111 1111 1111 0000 1111 0001 0101
计算下标
0101 = 5

put

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

// onlyIfAbsent 如果为 true，只有在 hashmap 没有该 key 的时候才添加

// evict 如果为 false， hashmap 为创建模式；只有在使用 Map 集合作为构造器创建 LinkedHashMap 或 HashMap 时才会为 false。
// 这两个参数均为实现 java8 的新接口而设置

```
Node<K,V> newNode(int hash, K key, V value, Node<K,V> next) {
    return new Node<>(hash, key, value, next);
}
```

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; // table
    Node<K,V> p;  // node pointer
    int n, i; // n 为 length, i 为 node index
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // index 处没有元素，则直接放入新节点
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        // index 处有元素
        Node<K,V> e;
        K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            // 假如 key 是相同的，那么替换 value 即可
            e = p;
        else if (p instanceof TreeNode)
            // key 不同，但如果 p 是红黑树根节点，那么将新节点放入红黑树
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            // key 不同，但如果 p 是链表头节点，那么判断链表中是否有该节点，如没有，
则将新节点插入到链表尾部
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    // 插入后如果发现已经链表长度已经适合转为红黑树了，则转换
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                // 链表中某元素 key 和 key 相同，则替换 value 即可
                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;
}
```

扩容 resize

// 扩容函数，如果 hash 桶为空，初始化默认大小，否则双倍扩容
// 注意 !! 因为扩容为 2 的倍数，根据 hash 桶的计算方法，元素哈希值不变
// 所以元素在新的 hash 桶的下标，要不跟旧的 hash 桶下标一致，要不增加 1 倍。
cap : capacity
thr : threshold

```java
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;
    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) {
                // j 位置原本元素存在
                oldTab[j] = null;
                if (e.next == null)
                    // 如果该位置没有形成链表，则再次计算 index，放入新 table
                    // 假设扩容前的 table 大小为 2 的 N 次方，有上述 put 方法解析可知，
元素的 table 索引为其 hash 值的后 N 位确定
                    那么扩容后的 table 大小即为 2 的 N+1 次方，则其中元素的 table 索引
为其 hash 值的后 N+1 位确定，比原来多了一位
                    因此，table 中的元素只有两种情况：
                    元素 hash 值第 N+1 位为 0：不需要进行位置调整
                    元素 hash 值第 N+1 位为 1：调整至原索引的两倍位置
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    // 如果该位置形成了红黑树，则 split
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order
                    // 如果该位置形成了链表，则分成两个链表，分别放在
0~oldCap,oldCap~oldCap*2 位置处
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        // 用于确定元素 hash 值第 N+1 位是否为 0：
                        若为 0，则使用 loHead 与 loTail，将元素移至新 table 的原索引
处
                        若不为 0，则使用 hiHead 与 hiHead，将元素移至新 table 的两倍
索引处
                        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;
}
```

get（O(logn)）

```
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) {
        // table 不为空，且 hash 对应 index 元素不为空
        // 如果 index 位置就是我们要找的 key，则直接返回
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        // 如果不是，则从链表或红黑树的角度继续找
        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;
}
```

remove

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

value=null,matchValue=false,movable=true

```
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;
        // 1) 如果 hash 对应 index 即为我们要找的 key，则找到
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            node = p;
       // 2) 从链表或红黑树的角度继续找
        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);
            }
        }
        // 找到后，根据找到的位置不同 相应地进行删除
        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;
}
```

containsKey

```
public boolean containsKey(Object key) {
    return getNode(hash(key), key) != null;
}
```

containsValue

```
public boolean containsValue(Object value) {
    Node<K,V>[] tab; V v;
    if ((tab = table) != null && size > 0) {
        for (int i = 0; i < tab.length; ++i) {
            for (Node<K,V> e = tab[i]; e != null; e = e.next) {
                if ((v = e.value) == value ||
                    (value != null && value.equals(v)))
                    return true;
            }
        }
    }
    return false;
}
```

a）链表转红黑树 treeifyBin

```
/**
 * Replaces all linked nodes in bin at index for given hash unless
 * table is too small, in which case resizes instead.
 */
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        TreeNode<K,V> hd = null, tl = null;
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (tl == null)
                hd = p;
            else {
                p.prev = tl;
                tl.next = p;
            }
            tl = p;
        } while ((e = e.next) != null);
        if ((tab[index] = hd) != null)
            hd.treeify(tab);
    }
}
```

b）红黑树转链表 TreeNode#untreeify

```
final Node<K,V> untreeify(HashMap<K,V> map) {
    Node<K,V> hd = null, tl = null;
    for (Node<K,V> q = this; q != null; q = q.next) {
        Node<K,V> p = map.replacementNode(q, null);
        if (tl == null)
            hd = p;
        else
            tl.next = p;
        tl = p;
    }
    return hd;
}
```

c）红黑树 查找

```java
final TreeNode<K,V> getTreeNode(int h, Object k) {
    return ((parent != null) ? root() : this).find(h, k, null);
}
```

```java
/**
 * Finds the node starting at root p with the given hash and key.
 * The kc argument caches comparableClassFor(key) upon first use
 * comparing keys.
 */
final TreeNode<K,V> find(int h, Object k, Class<?> kc) {
    TreeNode<K,V> p = this;
    do {
        int ph, dir; K pk;
        TreeNode<K,V> pl = p.left, pr = p.right, q;
        if ((ph = p.hash) > h)
            p = pl;
        else if (ph < h)
            p = pr;
        else if ((pk = p.key) == k || (k != null && k.equals(pk)))
            return p;
        else if (pl == null)
            p = pr;
        else if (pr == null)
            p = pl;
        else if ((kc != null ||
                  (kc = comparableClassFor(k)) != null) &&
                 (dir = compareComparables(kc, k, pk)) != 0)
            p = (dir < 0) ? pl : pr;
        else if ((q = pr.find(h, k, kc)) != null)
            return q;
        else
            p = pl;
    } while (p != null);
    return null;
}
```

d）红黑树 添加

```java
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;
        }
    }
}
```

e）红黑树 删除

```java
/**
 * Removes the given node, that must be present before this call.
 * This is messier than typical red-black deletion code because we
 * cannot swap the contents of an interior node with a leaf
 * successor that is pinned by "next" pointers that are accessible
 * independently during traversal. So instead we swap the tree
 * linkages. If the current tree appears to have too few nodes,
 * the bin is converted back to a plain bin. (The test triggers
 * somewhere between 2 and 6 nodes, depending on tree structure).
 */
final void removeTreeNode(HashMap<K,V> map, Node<K,V>[] tab,
                          boolean movable) {
    int n;
    if (tab == null || (n = tab.length) == 0)
        return;
    int index = (n - 1) & hash;
    TreeNode<K,V> first = (TreeNode<K,V>)tab[index], root = first, rl;
    TreeNode<K,V> succ = (TreeNode<K,V>)next, pred = prev;
    if (pred == null)
        tab[index] = first = succ;
    else
        pred.next = succ;
    if (succ != null)
        succ.prev = pred;
    if (first == null)
        return;
    if (root.parent != null)
        root = root.root();
    if (root == null || root.right == null ||
        (rl = root.left) == null || rl.left == null) {
        tab[index] = first.untreeify(map);  // too small
        return;
    }
    TreeNode<K,V> p = this, pl = left, pr = right, replacement;
    if (pl != null && pr != null) {
        TreeNode<K,V> s = pr, sl;
        while ((sl = s.left) != null) // find successor
            s = sl;
        boolean c = s.red; s.red = p.red; p.red = c; // swap colors
        TreeNode<K,V> sr = s.right;
        TreeNode<K,V> pp = p.parent;
    if (s == pr) { // p was s's direct parent
        p.parent = s;
        s.right = p;
    }
    else {
        TreeNode<K,V> sp = s.parent;
        if ((p.parent = sp) != null) {
            if (s == sp.left)
                sp.left = p;
            else
                sp.right = p;
        }
        if ((s.right = pr) != null)
            pr.parent = s;
    }
    p.left = null;
    if ((p.right = sr) != null)
        sr.parent = p;
    if ((s.left = pl) != null)
        pl.parent = s;
    if ((s.parent = pp) == null)
        root = s;
    else if (p == pp.left)
        pp.left = s;
    else
        pp.right = s;
    if (sr != null)
        replacement = sr;
    else
        replacement = p;
}
else if (pl != null)
    replacement = pl;
else if (pr != null)
    replacement = pr;
else
    replacement = p;
if (replacement != p) {
    TreeNode<K,V> pp = replacement.parent = p.parent;
    if (pp == null)
        root = replacement;
    else if (p == pp.left)
        pp.left = replacement;
    else
```

```
            pp.right = replacement;
        p.left = p.right = p.parent = null;
    }

    TreeNode<K,V> r = p.red ? root : balanceDeletion(root, replacement);

    if (replacement == p) {  // detach
        TreeNode<K,V> pp = p.parent;
        p.parent = null;
        if (pp != null) {
            if (p == pp.left)
                pp.left = null;
            else if (p == pp.right)
                pp.right = null;
        }
    }
    if (movable)
        moveRootToFront(tab, r);
}
```

f）红黑树 遍历

使用 next 指针，类似链表方式，便可遍历红黑树。

遍历（先迭代 table，再迭代 bucket->链表/红黑树）

keySet

keySet().iterator()

```
public Set<K> keySet() {
    Set<K> ks = keySet;
    if (ks == null) {
        ks = new KeySet();
        keySet = ks;
    }
    return ks;
}
```

```
final class KeySet extends AbstractSet<K> {
    public final Iterator<K> iterator()     { return new KeyIterator(); }
}
```

KeyIterator 实现了 Iterator 接口，并继承了 HashIterator。前者仅适用于 KeySet 的迭代，后者适合所有基于 HashMap 的迭代。

HashMap#HashIterator

```
abstract class HashIterator {
    Node<K,V> next;        // next entry to return
    Node<K,V> current;     // current entry
    int expectedModCount;  // for fast-fail
    int index;             // current slot

    HashIterator() {
        expectedModCount = modCount;
        Node<K,V>[] t = table;
        current = next = null;
        index = 0;
        if (t != null && size > 0) { // advance to first entry
            do {} while (index < t.length && (next = t[index++]) == null);
        }
    }

    public final boolean hasNext() {
        return next != null;
    }

    final Node<K,V> nextNode() {
        Node<K,V>[] t;
        Node<K,V> e = next;
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        if (e == null)
            throw new NoSuchElementException();
        // next 的 next 为空的话，则继续遍历 table，否则就返回 next 的 next（链表或红
黑树的下一个节点）
        if ((next = (current = e).next) == null && (t = table) != null) {
            do {} while (index < t.length && (next = t[index++]) == null);
        }
        return e;
    }

    public final void remove() {
        Node<K,V> p = current;
        if (p == null)
            throw new IllegalStateException();
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        current = null;
        K key = p.key;
        removeNode(hash(key), key, null, false, false);
        expectedModCount = modCount;
    }
}
```

HashMap#KeyIterator

```
final class KeyIterator extends HashIterator
    implements Iterator<K> {
    public final K next() { return nextNode().key; }
}
```

entrySet

```
public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> es;
    return (es = entrySet) == null ? (entrySet = new EntrySet()) : es;
}
```

使用的是该迭代器：

```
final class EntryIterator extends HashIterator
    implements Iterator<Map.Entry<K,V>> {
    public final Map.Entry<K,V> next() { return nextNode(); }
}
```

多线程环境下的问题

1.8 中 hashmap 的确不会因为多线程 put 导致死循环（1.7 代码中会这样子），但是依然有其他的弊端，比如数据丢失等等。因此多线程情况下还是建议使用 ConcurrentHashMap。

数据丢失：当多线程 put 的时候，当 index 相同而又同时达到链表的末尾时，另一个线程 put 的数据会把之前线程 put 的数据覆盖掉，就会产生数据丢失。

```
            if ((e = p.next) == null) {
                p.next = newNode(hash, key, value, null);
            }
```

Hashtable

Hashtable 同样是基于哈希表实现的，同样每个元素是一个 key-value 对，其内部也是通过单链表解决冲突问题，容量不足（超过了阈值）时，同样会自动增长。
Hashtable 也是 JDK1.0 引入的类，是线程安全的，能用于多线程环境中。
Hashtable 同样实现了 Serializable 接口，它支持序列化，实现了 Cloneable 接口，能被克隆。

Hashtable#Entry

```
private static class Entry<K,V> implements Map.Entry<K,V> {
    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;
```

```java
        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();
    }
}
```

成员变量

```java
/**
 * The hash table data.
 */
private transient Entry<?,?>[] table;

/**
 * The total number of entries in the hash table.
 */
private transient int count;

/**
 * The table is rehashed when its size exceeds this threshold.  (The
 * value of this field is (int)(capacity * loadFactor).)
 *
 * @serial
 */
private int threshold;

/**
```

```
 * The load factor for the hashtable.
 *
 * @serial
 */
private float loadFactor;

/**
 * The number of times this Hashtable has been structurally modified
 * Structural modifications are those that change the number of entries in
 * the Hashtable or otherwise modify its internal structure (e.g.,
 * rehash).  This field is used to make iterators on Collection-views of
 * the Hashtable fail-fast.  (See ConcurrentModificationException).
 */
private transient int modCount = 0;
```

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);
}

/**
 * Constructs a new, empty hashtable with the specified initial capacity
 * and default load factor (0.75).
 *
 * @param     initialCapacity   the initial capacity of the hashtable.
 * @exception IllegalArgumentException if the initial capacity is less
 *              than zero.
 */
public Hashtable(int initialCapacity) {
    this(initialCapacity, 0.75f);
}
```

```java
/**
 * Constructs a new, empty hashtable with a default initial capacity (11)
 * and load factor (0.75).
 */
public Hashtable() {
    this(11, 0.75f);
}
```

11？
Hashtable 的容量增加逻辑是乘 2+1，保证奇数。
在应用数据分布在等差数据集合(如偶数)上时，如果公差与桶容量有公约数 n，则至少有(n-1)/n 数量的桶是利用不到的。

hash to index

```java
int hash = key.hashCode();
int index = (hash & 0x7FFFFFFF) % tab.length;
```

取与之后一定是一个非负数
0x7FFFFFFF is 0111 1111 1111 1111 1111 1111 1111 1111 : all 1 except the sign bit.
 (hash & 0x7FFFFFFF) will result in a positive integer.
 (hash & 0x7FFFFFFF) % tab.length will be in the range of the tab length.

put（有锁）

```java
public synchronized V put(K key, V value) {
    // Make sure the value is not null
    if (value == null) {
        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;
}
```

```java
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();

        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++;
}
```

扩容 rehash

```java
protected void rehash() {
    int oldCapacity = table.length;
    Entry<?,?>[] oldMap = table;

    // overflow-conscious code
    int newCapacity = (oldCapacity << 1) + 1;
    if (newCapacity - MAX_ARRAY_SIZE > 0) {
        if (oldCapacity == MAX_ARRAY_SIZE)
            // Keep running with MAX_ARRAY_SIZE buckets
            return;
        newCapacity = MAX_ARRAY_SIZE;
    }
    Entry<?,?>[] newMap = new Entry<?,?>[newCapacity];

    modCount++;
    threshold = (int)Math.min(newCapacity * loadFactor, MAX_ARRAY_SIZE + 1);
    table = newMap;

    for (int i = oldCapacity ; i-- > 0 ;) {
        for (Entry<K,V> old = (Entry<K,V>)oldMap[i] ; old != null ; ) {
            Entry<K,V> e = old;
            old = old.next;
            // 所有元素重新散列
            int index = (e.hash & 0x7FFFFFFF) % newCapacity;
            e.next = (Entry<K,V>)newMap[index];
            newMap[index] = e;
        }
    }
}
```

get（有锁）

```
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;
}
```

remove（有锁）

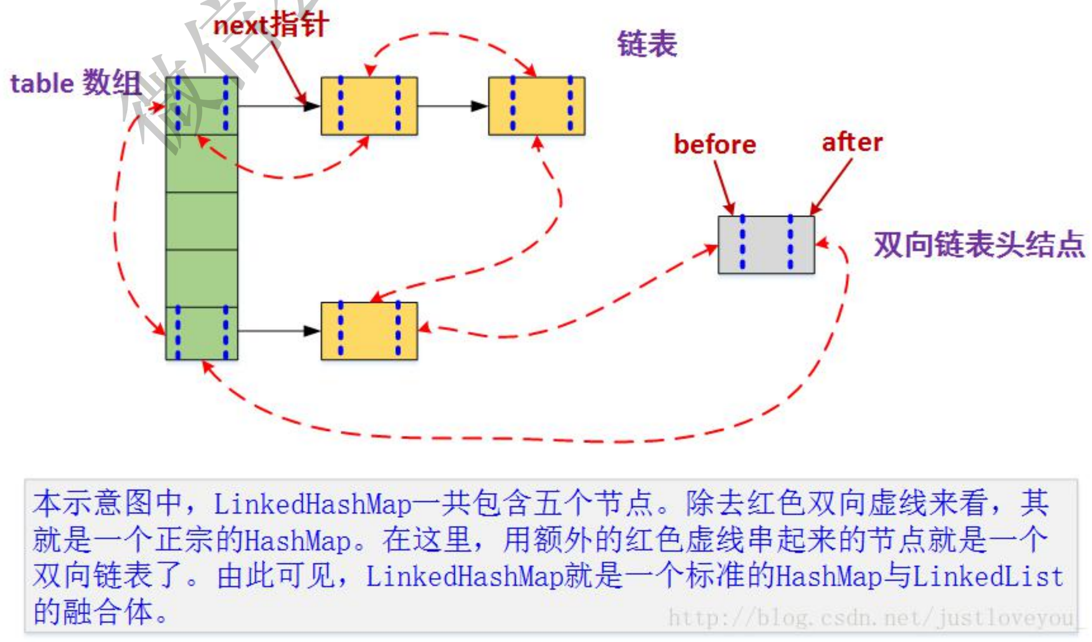
```java
public synchronized V remove(Object key) {
    Entry<?,?> tab[] = table;
    int hash = key.hashCode();
    int index = (hash & 0x7FFFFFFF) % tab.length;
    @SuppressWarnings("unchecked")
    Entry<K,V> e = (Entry<K,V>)tab[index];
    for(Entry<K,V> prev = null ; e != null ; prev = e, e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            modCount++;
            if (prev != null) {
                prev.next = e.next;
            } else {
                tab[index] = e.next;
            }
            count--;
            V oldValue = e.value;
            e.value = null;
            return oldValue;
        }
    }
    return null;
}
```

LinkedHashMap（底层是(数组+链表/红黑树)+环形双向链表，继承自 HashMap）

LinkedHashMap 是 key 键有序的 HashMap 的一种实现。它除了使用哈希表这个数据结构，使用环形双向链表来保证 key 的顺序。
HashMap 是无序的，也就是说，迭代 HashMap 所得到的元素顺序并不是它们最初放置到 HashMap 的顺序。HashMap 的这一缺点往往会造成诸多不便，因为在有些场景中，我们确需要用到一个可以保持插入顺序的 Map。庆幸的是，JDK 为我们解决了这个问题，它为 HashMap 提供了一个子类 —— LinkedHashMap。虽然 LinkedHashMap 增加了时间和空间上的开销，但是它通过维护一个额外的双向链表保证了迭代顺序。特别地，该迭代顺序可以是插入顺序，也可以是访问顺序。因此，根据链表中元素的顺序可以将 LinkedHashMap 分为：**保持插入顺序的 LinkedHashMap 和 保持访问顺序（LRU，get 后调整链表序，最新获取的放在最后）的 LinkedHashMap**，其中 LinkedHashMap 的默认实现是按插入顺序排序的。

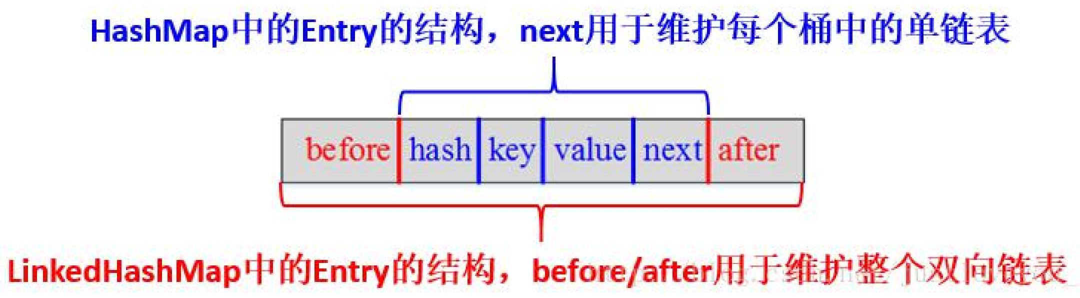
特点：
一般来说，如果需要使用的 Map 中的 key 无序，选择 HashMap；如果要求 key 有序，则选择 TreeMap。
但是选择 TreeMap 就会有性能问题，因为 TreeMap 的 get 操作的时间复杂度是 O(log(n))的，相比于 HashMap 的 O(1)还是差不少的，LinkedHashMap 的出现就是为了平衡这些因素，使得能够以 O(1)时间复杂度增加查找元素，又能够保证 key 的有序性

实现原理：
将所有 Entry 节点链入一个双向链表的 HashMap。在 LinkedHashMap 中，所有 put 进来的 Entry 都保存在哈希表中，但由于它又额外定义了一个以 head 为头结点的双向链表，因此对于每次 put 进来 Entry，除了将其保存到哈希表上外，还会将其插入到双向链表的尾部。



本示意图中，LinkedHashMap一共包含五个节点。除去红色双向虚线来看，其就是一个正宗的HashMap。在这里，用额外的红色虚线串起来的节点就是一个双向链表了。由此可见，LinkedHashMap就是一个标准的HashMap与LinkedList的融合体。

LinkedHashMap#Entry


HashMap中的Entry的结构，next用于维护每个桶中的单链表
before
hash
key
value
next
after
LinkedHashMap中的Entry的结构，before/after用于维护整个双向链表

```
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }
}
```

成员变量

```
/**
 * The head (eldest) of the doubly linked list.
 */
transient LinkedHashMap.Entry<K,V> head;

/**
 * The tail (youngest) of the doubly linked list.
 */
transient LinkedHashMap.Entry<K,V> tail;

/**
 * The iteration ordering method for this linked hash map: <tt>true</tt>
 * for access-order, <tt>false</tt> for insertion-order.
 *
 * @serial
 */
final boolean accessOrder;
```

构造方法

```java
public LinkedHashMap(int initialCapacity, float loadFactor) {
    super(initialCapacity, loadFactor);
    accessOrder = false;
}

/**
 * Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance
 * with the specified initial capacity and a default load factor (0.75).
 *
 * @param  initialCapacity the initial capacity
 * @throws IllegalArgumentException if the initial capacity is negative
 */
public LinkedHashMap(int initialCapacity) {
    super(initialCapacity);
    accessOrder = false;
}

/**
 * Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance
 * with the default initial capacity (16) and load factor (0.75).
 */
public LinkedHashMap() {
    super();
    accessOrder = false;
}

/**
 * Constructs an empty <tt>LinkedHashMap</tt> instance with the
 * specified initial capacity, load factor and ordering mode.
 *
 * @param  initialCapacity the initial capacity
 * @param  loadFactor      the load factor
 * @param  accessOrder     the ordering mode - <tt>true</tt> for
 *         access-order, <tt>false</tt> for insertion-order
 * @throws IllegalArgumentException if the initial capacity is negative
 *         or the load factor is nonpositive
 */
public LinkedHashMap(int initialCapacity,
                     float loadFactor,
                     boolean accessOrder) {
    super(initialCapacity, loadFactor);
    this.accessOrder = accessOrder;
}
```

put

同 HashMap，但重写了 afterNodeInsertion。

```
void afterNodeInsertion(boolean evict) { // possibly remove eldest
    LinkedHashMap.Entry<K,V> first;
    if (evict && (first = head) != null && removeEldestEntry(first)) {
        K key = first.key;
        removeNode(hash(key), key, null, false, true);
    }
}
```

//可以自行重写该方法

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    return false;
}
```

```
public class LRUHashMap<K, V> extends LinkedHashMap<K, V>{


        private final int MAX_CACHE_SIZE;

        public BaseLRUCache(int cacheSize) {
            super(cacheSize, 0.75f, true);
            MAX_CACHE_SIZE = cacheSize;
        }

        @Override
        protected boolean removeEldestEntry(Map.Entry eldest) {
            return size() > MAX_CACHE_SIZE;
        }


}
```

get

```
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;
}
```

```
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;
    }
}
```

remove

同 HashMap，但重写了 afterNodeRemoval。

```
void afterNodeRemoval(Node<K,V> e) { // unlink
    LinkedHashMap.Entry<K,V> p =
        (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
    p.before = p.after = null;
    if (b == null)
        head = a;
    else
        b.after = a;
    if (a == null)
        tail = b;
    else
```

```
        a.before = b;
}
```

遍历（迭代环形双向链表）

entrySet

```
public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> es;
    return (es = entrySet) == null ? (entrySet = new LinkedEntrySet()) : es;
}
```
它使用的是该迭代器：
```
abstract class LinkedHashIterator {
    LinkedHashMap.Entry<K,V> next;
    LinkedHashMap.Entry<K,V> current;
    int expectedModCount;

    LinkedHashIterator() {
        next = head;
        expectedModCount = modCount;
        current = null;
    }

    public final boolean hasNext() {
        return next != null;
    }

    final LinkedHashMap.Entry<K,V> nextNode() {
        LinkedHashMap.Entry<K,V> e = next;
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        if (e == null)
            throw new NoSuchElementException();
        current = e;
        next = e.after;
        return e;
    }

    public final void remove() {
        Node<K,V> p = current;
        if (p == null)
            throw new IllegalStateException();
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        current = null;
        K key = p.key;
        removeNode(hash(key), key, null, false, false);
        expectedModCount = modCount;
    }
}
```

```
final class LinkedEntryIterator extends LinkedHashIterator
    implements Iterator<Map.Entry<K,V>> {
    public final Map.Entry<K,V> next() { return nextNode(); }
}
```

TreeMap（底层是红黑树）

支持排序的 Map 实现。
基于红黑树实现， 无容量限制。
是非线程安全的。

TreeMap 是根据 key 进行排序的, 它的排序和定位需要依赖比较器或覆写 Comparable 接口, 也因此不需要 key 覆写 hashCode 方法和 equals 方法, 就可以排除掉重复的 key, 而 HashMap 的 key 则需要通过覆写 hashCode 方法和 equals 方法来确保没有重复的 key
TreeMap 的查询、插入、删除效率均没有 HashMap 高，一般只有要对 key 排序时才使用 TreeMap。
TreeMap 的 key 不能为 null，而 HashMap 的 key 可以为 null。

TreeMap#Entry

```
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;
    }
}
```

成员变量

```
private final Comparator<? super K> comparator;

private transient Entry<K,V> root;

/**
 * The number of entries in the tree
 */
private transient int size = 0;

/**
 * The number of structural modifications to the tree.
 */
private transient int modCount = 0;
```

构造方法

```
public TreeMap() {
    comparator = null;
}

public TreeMap(Comparator<? super K> comparator) {
    this.comparator = comparator;
}
```

put

```java
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")
            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;
}
```

get

```
public V get(Object key) {
    Entry<K,V> p = getEntry(key);
    return (p==null ? null : p.value);
}
```

```
final Entry<K,V> getEntry(Object key) {
    // Offload comparator-based version for sake of performance
    if (comparator != null)
        return getEntryUsingComparator(key);
    if (key == null)
        throw new NullPointerException();
    @SuppressWarnings("unchecked")
        Comparable<? super K> k = (Comparable<? super K>) key;
    Entry<K,V> p = root;
    while (p != null) {
        int cmp = k.compareTo(p.key);
        if (cmp < 0)
            p = p.left;
        else if (cmp > 0)
            p = p.right;
        else
            return p;
    }
    return null;
}
```

```
final Entry<K,V> getEntryUsingComparator(Object key) {
    @SuppressWarnings("unchecked")
        K k = (K) key;
```

```java
    Comparator<? super K> cpr = comparator;
    if (cpr != null) {
        Entry<K,V> p = root;
        while (p != null) {
            int cmp = cpr.compare(k, p.key);
            if (cmp < 0)
                p = p.left;
            else if (cmp > 0)
                p = p.right;
            else
                return p;
        }
    }
    return null;
}
```

remove

```java
public V remove(Object key) {
    Entry<K,V> p = getEntry(key);
    if (p == null)
        return null;

    V oldValue = p.value;
    deleteEntry(p);
    return oldValue;
}
```

```java
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;
        }
    }
}
```

containsKey

```
public boolean containsKey(Object key) {
    return getEntry(key) != null;
}
```

containsValue

```java
public boolean containsValue(Object value) {
    for (Entry<K,V> e = getFirstEntry(); e != null; e = successor(e))
        if (valEquals(value, e.value))
            return true;
    return false;
}
```

```java
final Entry<K,V> getFirstEntry() {
    Entry<K,V> p = root;
    if (p != null)
        while (p.left != null)
            p = p.left;
    return p;
}
```

```java
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;
    }
}
```

```java
static final boolean valEquals(Object o1, Object o2) {
    return (o1==null ? o2==null : o1.equals(o2));
}
```

遍历

```
public Set<Map.Entry<K,V>> entrySet() {
    EntrySet es = entrySet;
    return (es != null) ? es : (entrySet = new EntrySet());
}
```

```
class EntrySet extends AbstractSet<Map.Entry<K,V>> {
    public Iterator<Map.Entry<K,V>> iterator() {
        return new EntryIterator(getFirstEntry());
    }
}
```

```
abstract class PrivateEntryIterator<T> implements Iterator<T> {
    Entry<K,V> next;
    Entry<K,V> lastReturned;
    int expectedModCount;

    PrivateEntryIterator(Entry<K,V> first) {
        expectedModCount = modCount;
        lastReturned = null;
        next = first;
    }

    public final boolean hasNext() {
        return next != null;
    }

    final Entry<K,V> nextEntry() {
        Entry<K,V> e = next;
        if (e == null)
            throw new NoSuchElementException();
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        next = successor(e);
        lastReturned = e;
        return e;
    }

    final Entry<K,V> prevEntry() {
        Entry<K,V> e = next;
        if (e == null)
            throw new NoSuchElementException();
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        next = predecessor(e);
        lastReturned = e;
        return e;
    }

    public void remove() {
        if (lastReturned == null)
            throw new IllegalStateException();
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        // deleted entries are replaced by their successors
        if (lastReturned.left != null && lastReturned.right != null)
            next = lastReturned;
        deleteEntry(lastReturned);
        expectedModCount = modCount;
        lastReturned = null;
    }
}
```

```
final class EntryIterator extends PrivateEntryIterator<Map.Entry<K,V>> {
    EntryIterator(Entry<K,V> first) {
        super(first);
    }
    public Map.Entry<K,V> next() {
        return nextEntry();
    }
}
```

ConcurrentHashMap（底层是数组+链表/红黑树，基于 CAS+synchronized）

JDK1.7 前：分段锁
基于 currentLevel 划分出了多个 Segment 来对 key-value 进行存储，从而避免每次 put 操作都得锁住整个数组。在默认的情况下，最佳情况下可以允许 16 个线程并发无阻塞地操作集合对象，尽可能地减少并发时的阻塞现象。
put、remove 会加锁。get 和 containsKey 不会加锁。
计算 size：在不加锁的情况下遍历所有的段，读取其 count 以及 modCount，这两个属性都是 volatile 类型的，并进行统计，再遍历一次所有的段，比较 modCount 是否有改变。如有改变，则再尝试两次机上动作。
如执行了三次上述动作，仍然有问题，则遍历所有段，分别进行加锁，然后进行计算，计算完毕后释放所有锁，从而完成计算动作。

JDK1.8 后：CAS+synchronized
bin 是桶 bucket 的意思

ConcurrentHashMap 是延迟初始化的，只有在插入数据时，整个 HashMap 才被初始化为 2 的次方大小个桶(bin)，每个 bin 包含哈希值相同的一系列 Node(一般含有 0 或 1 个 Node)。
每个 bin 的第一个 Node 作为这个 bin 的锁，Hash 值为零或者负的将被忽略；
每个 bin 的第一个 Node 插入用到 CAS 原理，这是在 ConcurrentHashMap 中最常发生的操作，**其余的插入、删除、替换操作对 bin 中的第一个 Node 加锁**，进行操作
ConcurrentHashMap 的 size()函数一般比较少用，同时为了提高增删查改的效率，容器并未在内部保存一个 size 值，而且采用每次调用 size()函数时累加各个 bin 中 Node 的个数计算得到，而且这一过程不加锁，即得到的 size 值不一定是最新的。

ConcurrentHashMap#Node

Node 是最核心的内部类，它包装了 key-value 键值对，所有插入 ConcurrentHashMap 的数据都包装在这里面。它与 HashMap 中的定义很相似，但是但是有一些差别：它对 value 和 next 属性设置了 volatile 属性；它不允许调用 setValue 方法直接改变 Node 的 value 域；它增加了 find 方法辅助 map.get()方法。

```java
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val; // value 和 next 是 volatile 的
    volatile Node<K,V> next;

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

    public final K getKey()       { return key; }
    public final V getValue()     { return val; }
    public final int hashCode()   { return key.hashCode() ^ val.hashCode(); }
    public final String toString(){ return key + "=" + val; }
    public final V setValue(V value) {
        throw new UnsupportedOperationException();
    }

    public final boolean equals(Object o) {
        Object k, v, u; Map.Entry<?,?> e;
        return ((o instanceof Map.Entry) &&
                (k = (e = (Map.Entry<?,?>)o).getKey()) != null &&
                (v = e.getValue()) != null &&
                (k == key || k.equals(key)) &&
                (v == (u = val) || v.equals(u)));
    }

    /**
     * Virtualized support for map.get(); overridden in subclasses.
     */
    Node<K,V> find(int h, Object k) {
        Node<K,V> e = this;
        if (k != null) {
            do {
                K ek;
                if (e.hash == h &&
                    ((ek = e.key) == k || (ek != null && k.equals(ek))))
                    return e;
            } while ((e = e.next) != null);
        }
        return null;
    }
}
```

ConcurrentHashMap#TreeNode

当链表长度过长的时候，会转换为 TreeNode。但是与 HashMap 不相同的是，它并不是直接转换为红黑树，而是**把这些结点包装成 TreeNode 放在 TreeBin 对象中**，由 TreeBin 完成对红黑树的包装。而且 TreeNode 在 ConcurrentHashMap 继承自 Node 类，而并非 HashMap 中的继承自 LinkedHashMap.Entry<K,V>类，也就是说 TreeNode 带有 next 指针，这样做的目的是方便基于 TreeBin 的访问。

```
static final class TreeNode<K,V> extends Node<K,V> {
    TreeNode<K,V> parent;  // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;    // needed to unlink next upon deletion
    boolean red;

    TreeNode(int hash, K key, V val, Node<K,V> next,
             TreeNode<K,V> parent) {
        super(hash, key, val, next);
        this.parent = parent;
    }

    Node<K,V> find(int h, Object k) {
        return findTreeNode(h, k, null);
    }

    /**
     * Returns the TreeNode (or null if not found) for the given key
     * starting at given root.
     */
    final TreeNode<K,V> findTreeNode(int h, Object k, Class<?> kc) {
        if (k != null) {
            TreeNode<K,V> p = this;
            do  {
                int ph, dir; K pk; TreeNode<K,V> q;
                TreeNode<K,V> pl = p.left, pr = p.right;
                if ((ph = p.hash) > h)
                    p = pl;
                else if (ph < h)
                    p = pr;
                else if ((pk = p.key) == k || (pk != null && k.equals(pk)))
                    return p;
                else if (pl == null)
                    p = pr;
                else if (pr == null)
                    p = pl;
                else if ((kc != null ||
                          (kc = comparableClassFor(k)) != null) &&
                         (dir = compareComparables(kc, k, pk)) != 0)
                    p = (dir < 0) ? pl : pr;
                else if ((q = pr.findTreeNode(h, k, kc)) != null)
                    return q;
                else
                    p = pl;
            } while (p != null);
        }
        return null;
    }
}
```

ConcurrentHashMap#TreeBin

这个类并不负责包装用户的 key、value 信息，而是包装的很多 TreeNode 节点。它代替了 TreeNode 的根节点，也就是说在**实际的 ConcurrentHashMap“数组”中，存放的是 TreeBin 对象，而不是 TreeNode 对象，这是与 HashMap 的区别**。另外这个类还带有了读写锁。

可以看到在构造 TreeBin 节点时，仅仅指定了它的 hash 值为 TREEBIN 常量，这也就是个标识位；同时也看到我们熟悉的红黑树构造方法。

```
/**
 * TreeNodes used at the heads of bins. TreeBins do not hold user
 * keys or values, but instead point to list of TreeNodes and
 * their root. They also maintain a parasitic read-write lock
 * forcing writers (who hold bin lock) to wait for readers (who do
 * not) to complete before tree restructuring operations.
 */
static final class TreeBin<K,V> extends Node<K,V> {
    TreeNode<K,V> root;
    volatile TreeNode<K,V> first;
    volatile Thread waiter;
```

```java
volatile int lockState;
// values for lockState
static final int WRITER = 1; // set while holding write lock
static final int WAITER = 2; // set when waiting for write lock
static final int READER = 4; // increment value for setting read lock

/**
 * Tie-breaking utility for ordering insertions when equal
 * hashCodes and non-comparable. We don't require a total
 * order, just a consistent insertion rule to maintain
 * equivalence across rebalancings. Tie-breaking further than
 * necessary simplifies testing a bit.
 */
static int tieBreakOrder(Object a, Object b) {
    int d;
    if (a == null || b == null ||
        (d = a.getClass().getName().
         compareTo(b.getClass().getName())) == 0)
        d = (System.identityHashCode(a) <= System.identityHashCode(b) ?
             -1 : 1);
    return d;
}

/**
 * Creates bin with initial set of nodes headed by b.
 */
TreeBin(TreeNode<K,V> b) {
    super(TREEBIN, null, null, null);
    this.first = b;
    TreeNode<K,V> r = null;
    for (TreeNode<K,V> x = b, next; x != null; x = next) {
        next = (TreeNode<K,V>)x.next;
        x.left = x.right = null;
        if (r == null) {
            x.parent = null;
            x.red = false;
            r = x;
        }
        else {
            K k = x.key;
            int h = x.hash;
            Class<?> kc = null;
            for (TreeNode<K,V> p = r;;) {
                int dir, ph;
```

```
                    K pk = p.key;
                    if ((ph = p.hash) > h)
                        dir = -1;
                    else if (ph < h)
                        dir = 1;
                    else if ((kc == null &&
                              (kc = comparableClassFor(k)) == null) ||
                             (dir = compareComparables(kc, k, pk)) == 0)
                        dir = tieBreakOrder(k, pk);
                        TreeNode<K,V> xp = p;
                    if ((p = (dir <= 0) ? p.left : p.right) == null) {
                        x.parent = xp;
                        if (dir <= 0)
                            xp.left = x;
                        else
                            xp.right = x;
                        r = balanceInsertion(r, x);
                        break;
                    }
                }
            }
        }
        this.root = r;
        assert checkInvariants(root);
    }
}
```

ConcurrentHashMap#ForwardingNode

```java
/**
 * A node inserted at head of bins during transfer operations.
 */
static final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }

    Node<K,V> find(int h, Object k) {
        // loop to avoid arbitrarily deep recursion on forwarding nodes
        outer: for (Node<K,V>[] tab = nextTable;;) {
            Node<K,V> e; int n;
            if (k == null || tab == null || (n = tab.length) == 0 ||
                (e = tabAt(tab, (n - 1) & h)) == null)
                return null;
            for (;;) {
                int eh; K ek;
                if ((eh = e.hash) == h &&
                    ((ek = e.key) == k || (ek != null && k.equals(ek))))
                    return e;
                if (eh < 0) {
                    if (e instanceof ForwardingNode) {
                        tab = ((ForwardingNode<K,V>)e).nextTable;
                        continue outer;
                    }
                    else
                        return e.find(h, k);
                }
                if ((e = e.next) == null)
                    return null;
            }
        }
    }
}
```

ConcurrentHashMap#ReservationNode

```java
/**
 * A place-holder node used in computeIfAbsent and compute
 */
static final class ReservationNode<K,V> extends Node<K,V> {
    ReservationNode() {
        super(RESERVED, null, null, null);
    }

    Node<K,V> find(int h, Object k) {
        return null;
    }
}
```

节点类型

hash 值大于等于 0，则是链表节点，Node
hash 值为-1　　MOVED，则是 forwarding nodes，存储 nextTable 的引用。只有 table 发生扩容的时候，ForwardingNode 才会发挥作用，作为一个占位符放在 table 中表示当前节点为 null 或则已经被移动。
hash 值为-2　　TREEBIN，则是红黑树根，TreeBin 类型
hash 值为-3　　RESERVED，则是 reservation nodes，

```
static final int MOVED     = -1; // hash for forwarding nodes
static final int TREEBIN   = -2; // hash for roots of trees
static final int RESERVED  = -3; // hash for transient reservations
```

成员变量

```java
/**
 * The largest possible table capacity.  This value must be
 * exactly 1<<30 to stay within Java array allocation and indexing
 * bounds for power of two table sizes, and is further required
 * because the top two bits of 32bit hash fields are used for
 * control purposes.
 */
private static final int MAXIMUM_CAPACITY = 1 << 30;

/**
 * The default initial table capacity.  Must be a power of 2
 * (i.e., at least 1) and at most MAXIMUM_CAPACITY.
 */
private static final int DEFAULT_CAPACITY = 16;

/**
 * The largest possible (non-power of two) array size.
 * Needed by toArray and related methods.
 */
static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

/**
 * The default concurrency level for this table. Unused but
 * defined for compatibility with previous versions of this class.
 */
private static final int DEFAULT_CONCURRENCY_LEVEL = 16;

/**
 * The load factor for this table. Overrides of this value in
 * constructors affect only the initial table capacity.  The
 * actual floating point value isn't normally used -- it is
 * simpler to use expressions such as {@code n - (n >>> 2)} for
 * the associated resizing threshold.
 */
private static final float LOAD_FACTOR = 0.75f;

/**
 * The bin count threshold for using a tree rather than list for a
 * bin.  Bins are converted to trees when adding an element to a
 * bin with at least this many nodes. The value must be greater
 * than 2, and should be at least 8 to mesh with assumptions in
 * tree removal about conversion back to plain bins upon
 * shrinkage.
 */
static final int TREEIFY_THRESHOLD = 8;

/**
 * The bin count threshold for untreeifying a (split) bin during a
 * resize operation. Should be less than TREEIFY_THRESHOLD, and at
 * most 6 to mesh with shrinkage detection under removal.
 */
static final int UNTREEIFY_THRESHOLD = 6;

/**
 * The smallest table capacity for which bins may be treeified.
 * (Otherwise the table is resized if too many nodes in a bin.)
 * The value should be at least 4 * TREEIFY_THRESHOLD to avoid
 * conflicts between resizing and treeification thresholds.
 */
static final int MIN_TREEIFY_CAPACITY = 64;

/**
 * Minimum number of rebinnings per transfer step. Ranges are
 * subdivided to allow multiple resizer threads.  This value
 * serves as a lower bound to avoid resizers encountering
 * excessive memory contention.  The value should be at least
 * DEFAULT_CAPACITY.
 */
private static final int MIN_TRANSFER_STRIDE = 16;

/**
 * The number of bits used for generation stamp in sizeCtl.
 * Must be at least 6 for 32bit arrays.
 */
private static int RESIZE_STAMP_BITS = 16;

/**
 * The maximum number of threads that can help resize.
 * Must fit in 32 - RESIZE_STAMP_BITS bits.
 */
private static final int MAX_RESIZERS = (1 << (32 - RESIZE_STAMP_BITS)) - 1;

/**
 * The bit shift for recording size stamp in sizeCtl.
 */
```

```java
private static final int RESIZE_STAMP_SHIFT = 32 - RESIZE_STAMP_BITS;
/*
 * Encodings for Node hash fields. See above for explanation.
 */
static final int MOVED     = -1; // hash for forwarding nodes
static final int TREEBIN   = -2; // hash for roots of trees
static final int RESERVED  = -3; // hash for transient reservations
static final int HASH_BITS = 0x7fffffff; // usable bits of normal node hash

/** Number of CPUS, to place bounds on some sizings */
static final int NCPU = Runtime.getRuntime().availableProcessors();

/**
 * The array of bins. Lazily initialized upon first insertion.
 * Size is always a power of two. Accessed directly by iterators.
 */
transient volatile Node<K,V>[] table;

/**
 * The next table to use; non-null only while resizing.
 */
private transient volatile Node<K,V>[] nextTable;

/**
 * Base counter value, used mainly when there is no contention,
 * but also as a fallback during table initialization
 * races. Updated via CAS.
 */
private transient volatile long baseCount;

/**
 * Table initialization and resizing control.  When negative, the
 * table is being initialized or resized: -1 for initialization,
 * else -(1 + the number of active resizing threads).  Otherwise,
 * when table is null, holds the initial table size to use upon
 * creation, or 0 for default. After initialization, holds the
 * next element count value upon which to resize the table.
   负数代表正在进行初始化或扩容操作
   -1 代表正在初始化
   -N 表示有N-1 个线程正在进行扩容操作
   正数或0 代表hash 表还没有被初始化，这个数值表示初始化或下一次进行扩容的大小，这
一点类似于扩容阈值的概念。还后面可以看到，它的值始终是当前ConcurrentHashMap 容量的
0.75 倍，这与loadfactor 是对应的。
 */
```

```java
private transient volatile int sizeCtl;

/**
 * The next table index (plus one) to split while resizing.
 */
private transient volatile int transferIndex;

/**
 * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
 */
private transient volatile int cellsBusy;

/**
 * Table of counter cells. When non-null, size is a power of 2.
 */
private transient volatile CounterCell[] counterCells;

// views
private transient KeySetView<K,V> keySet;
private transient ValuesView<K,V> values;
private transient EntrySetView<K,V> entrySet;
```

构造方法

```java
public ConcurrentHashMap() {
}

/**
 * Creates a new, empty map with an initial table size
 * accommodating the specified number of elements without the need
 * to dynamically resize.
 *
 * @param initialCapacity The implementation performs internal
 * sizing to accommodate this many elements.
 * @throws IllegalArgumentException if the initial capacity of
 * elements is negative
 */
public ConcurrentHashMap(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap;
}

/**
 * Creates a new map with the same mappings as the given map.
 *
 * @param m the map
 */
public ConcurrentHashMap(Map<? extends K, ? extends V> m) {
    this.sizeCtl = DEFAULT_CAPACITY;
    putAll(m);
}

/**
 * Creates a new, empty map with an initial table size based on
 * the given number of elements ({@code initialCapacity}) and
 * initial table density ({@code loadFactor}).
 *
 * @param initialCapacity the initial capacity. The implementation
 * performs internal sizing to accommodate this many elements,
 * given the specified load factor.
 * @param loadFactor the load factor (table density) for
 * establishing the initial table size
 * @throws IllegalArgumentException if the initial capacity of
 * elements is negative or the load factor is nonpositive
 *
 * @since 1.6
 */
public ConcurrentHashMap(int initialCapacity, float loadFactor) {
    this(initialCapacity, loadFactor, 1);
}

/**
 * Creates a new, empty map with an initial table size based on
 * the given number of elements ({@code initialCapacity}), table
 * density ({@code loadFactor}), and number of concurrently
 * updating threads ({@code concurrencyLevel}).
 *
 * @param initialCapacity the initial capacity. The implementation
 * performs internal sizing to accommodate this many elements,
 * given the specified load factor.
 * @param loadFactor the load factor (table density) for
 * establishing the initial table size
 * @param concurrencyLevel the estimated number of concurrently
 * updating threads. The implementation may use this value as
 * a sizing hint.
 * @throws IllegalArgumentException if the initial capacity is
 * negative or the load factor or concurrencyLevel are
 * nonpositive
 */
public ConcurrentHashMap(int initialCapacity,
                         float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    if (initialCapacity < concurrencyLevel)   // Use at least as many bins
        initialCapacity = concurrencyLevel;   // as estimated threads
    long size = (long)(1.0 + (long)initialCapacity / loadFactor);
    int cap = (size >= (long)MAXIMUM_CAPACITY) ?
        MAXIMUM_CAPACITY : tableSizeFor((int)size);
    this.sizeCtl = cap;
}
```

CAS

```
private static final sun.misc.Unsafe U;
```

Unsafe 类的几个 CAS 方法，可以原子性地修改对象的某个属性值

```
/**
 * Atomically update Java variable to <tt>x</tt> if it is currently
 * holding <tt>expected</tt>.
 * @return <tt>true</tt> if successful
 */
public final native boolean compareAndSwapObject(Object o, long offset,
                                                 Object expected,
                                                 Object x);

/**
 * Atomically update Java variable to <tt>x</tt> if it is currently
 * holding <tt>expected</tt>.
 * @return <tt>true</tt> if successful
 */
public final native boolean compareAndSwapInt(Object o, long offset,
                                              int expected,
                                              int x);

/**
 * Atomically update Java variable to <tt>x</tt> if it is currently
 * holding <tt>expected</tt>.
 * @return <tt>true</tt> if successful
 */
public final native boolean compareAndSwapLong(Object o, long offset,
                                               long expected,
                                               long x);
/**
 * Fetches a reference value from a given Java variable, with volatile
 * load semantics. Otherwise identical to {@link #getObject(Object, long)}
 */
public native Object getObjectVolatile(Object o, long offset);

/**
 * Stores a reference value into a given Java variable, with
 * volatile store semantics. Otherwise identical to {@link #putObject(Object,
long, Object)}
 */
public native void    putObjectVolatile(Object o, long offset, Object x);
```

Unsafe.getObjectVolatile 可以直接获取指定内存的数据，保证了每次拿到数据都是最新的。

三个核心方法

ConcurrentHashMap 定义了三个原子操作，用于对指定位置的节点进行操作。正是这些原子操作保证了 ConcurrentHashMap 的线程安全。

```
static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
    return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
}

static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
                                    Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}

static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {
    U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);
}
```

初始化

对于 ConcurrentHashMap 来说，调用它的构造方法仅仅是设置了一些参数而已。而整个 table 的初始化是在向 ConcurrentHashMap 中插入元素的时候发生的。如调用 **put**、**computeIfAbsent**、**compute**、**merge** 等方法的时候，调用时机是检查 table==null。

初始化方法主要应用了关键属性 sizeCtl 如果这个值<0，表示其他线程正在进行初始化，就放弃这个操作。在这也可以看出 ConcurrentHashMap 的初始化只能由一个线程完成。如果获得了初始化权限，就用 CAS 方法将 sizeCtl 置为-1，防止其他线程进入。初始化数组后，将 sizeCtl 的值改为 0.75*n。

```java
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        // 利用 CAS 方法把 sizectl 的值置为-1 表示本线程正在进行初始化
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    // 相当于 0.75*n 设置一个扩容的阈值
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

spread（hash）

h 是某个对象的 hashCode 返回值

```
static final int spread(int h) {
    return (h ^ (h >>> 16)) & HASH_BITS;
}
```

```
static final int HASH_BITS = 0x7fffffff; // usable bits of normal node hash
```

类似于 Hashtable+HashMap 的 hash 实现，Hashtable 中也是和一个魔法值取与，保证结果一定为正数；HashMap 中也是将 hashCode 与其移动低 n 位的结果再取异或，保证了对象的 hashCode 的高 16 位的变化能反应到低 16 位中，

size 相关

成员变量

```
@sun.misc.Contended static final class CounterCell {
    volatile long value;
    CounterCell(long x) { value = x; }
}
```

```
/**
 * Base counter value, used mainly when there is no contention,
 * but also as a fallback during table initialization
 * races. Updated via CAS.
 */
private transient volatile long baseCount;
/**
 * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
 */
private transient volatile int cellsBusy;

/**
 * Table of counter cells. When non-null, size is a power of 2.
 */
private transient volatile CounterCell[] counterCells;
```

每个 CounterCell 都对应一个 bucket，CounterCell 中的 long 值就是对应 bucket 的 binCount。

计算总大小就是将所有 bucket 的 binCount 求和，而每个 binCount 都存储在 CounterCell#value 中，每当 put 或者 remove 时都会更新节点所在 bucket 对应的 CounterCell#value。

size()

没有直接返回 baseCount 而是统计一次这个值，而这个值其实也是一个大概的数值，因此可能在统计的时候有其他线程正在执行插入或删除操作。

```
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}
```

在 baseCount 基础上再加上所有 counterCell 的值求和。

而在 addCount 时，会先尝试 CAS 更新 baseCount，如果有冲突，则再尝试 CAS 更新随机的一个 counterCell 中的 value，这样求和就是正确的 size 了。

```
final long sumCount() {
    CounterCell[] as = counterCells;
    CounterCell a;
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                // 所有 counter 的值求和
                sum += a.value;
        }
    }
    return sum;
}
```

put（若 bucket 第一个结点插入则使用 CAS，否则加锁）

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

整体流程就是首先定义不允许 key 或 value 为 null 的情况放入 。对于每一个放入的值，首先利用 spread 方法对 key 的 hashcode 进行一次 hash 计算，由此来确定这个值在 table 中的位置。

1）如果这个位置是空的，那么直接放入，而且不需要加锁操作。

2）如果这个位置存在结点，说明发生了 hash 碰撞，首先判断这个节点的类型。

    a）如果是 MOVED 节点，则表示正在扩容，帮助进行扩容

    b）如果是链表节点(hash >=0）,则得到的结点就是 hash 值相同的节点组成的链表的头节点。需要依次向后遍历确定这个新加入的值所在位置。如果遇到 hash 值与 key 值都与新加入节点是一致的情况，则只需要更新 value 值即可。否则依次向后遍历，直到链表尾插入这个结点。 如果加入这个节点以后链表长度大于 8，就把这个链表转换成红黑树。

    c）如果这个节点的类型已经是树节点的话，直接调用树节点的插入方法进行插入新的值。

3）addCount 增加计数值

```
/** Implementation for put and putIfAbsent */
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode());
    int binCount = 0;
    // 死循环，只有插入成功时才会跳出
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            // table 为空则初始化（延迟初始化）
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // hash to index 后正好为空，则 CAS 放入；如果失败那么进入下次循环继续
尝试
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        // 如果 index 处非空，且 hash 为 MOVED（表示该节点是 ForwardingNode），则表
示有其它线程正在扩容，则一起进行扩容操作。
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        // 如果 index 处非空，且为链表节点或树节点
        else {
            V oldVal = null;
            // 对某个 bucket 上执行添加操作仅需要锁住第一个 Node 即可（可以保证不会
多线程同时对某个 bucket 进行写入）
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    // 1) 如果是链表节点，那么插入到链表中
                    if (fh >= 0) {
                        // binCount 是该 bucket 中元素个数
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,
                                                          value, null);
                                break;
                            }
                        }
                    }
                    // 2)如果是红黑树树根，那么插入到红黑树中
                    else if (f instanceof TreeBin) {
                        Node<K,V> p;
                        binCount = 2;
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                       value)) != null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                }
            }
            // 插入节点/释放锁之后，如果大小合适调整为红黑树，那么将链表转为红黑树
            if (binCount != 0) {
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    // 将当前 ConcurrentHashMap 的元素数量+1 ， 如果超过阈值, 那么进行扩容
    addCount(1L, binCount);
    return null;
}
```

treeifyBin（有锁，数组较小则扩容，较大则转为红黑树）

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
        // 如果数组长度 n 小于阈值 MIN_TREEIFY_CAPACITY, 默认是 64, 则会调用
tryPresize 方法把数组长度扩大到原来的两倍, 并触发 transfer 方法, 重新调整节点的位置
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            tryPresize(n << 1);
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```
tryPreSize

```java
/**
 * Tries to presize table to accommodate the given number of elements.
 *
 * @param size number of elements (doesn't need to be perfectly accurate)
 */
private final void tryPresize(int size) {
    int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
        tableSizeFor(size + (size >>> 1) + 1);
    int sc;
    while ((sc = sizeCtl) >= 0) {
        Node<K,V>[] tab = table; int n;
        if (tab == null || (n = tab.length) == 0) {
            n = (sc > c) ? sc : c;
            if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
                try {
                    if (table == tab) {
                        @SuppressWarnings("unchecked")
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        table = nt;
                        sc = n - (n >>> 2);
                    }
                } finally {
                    sizeCtl = sc;
                }
            }
        }
        else if (c <= sc || n >= MAXIMUM_CAPACITY)
            break;
        else if (tab == table) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                Node<K,V>[] nt;
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
        }
    }
}
```

扩容

tryPresize

tryPresize 在 putAll 以及 treeifyBin 中调用

```java
private final void tryPresize(int size) {
    // c 是扩容之后预计表的大小
    int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
        tableSizeFor(size + (size >>> 1) + 1);
    int sc;
    // 没有正在初始化或扩容
    while ((sc = sizeCtl) >= 0) {
        Node<K,V>[] tab = table; int n;
        if (tab == null || (n = tab.length) == 0) {
            n = (sc > c) ? sc : c;
            // 期间没有其他线程对表操作，则 CAS 将 SIZECTL 状态置为-1，表示正在进行
初始化
            if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
                try {
                    if (table == tab) {
                        @SuppressWarnings("unchecked")
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        table = nt;
                        // 即 0.75*n
                        sc = n - (n >>> 2);
                    }
                } finally {
                    sizeCtl = sc;
                }
            }
        }
        // 若欲扩容值不大于原阈值，或现有容量>=最值，则 do nothing
        else if (c <= sc || n >= MAXIMUM_CAPACITY)
            break;
        // table 不为空，且在此期间其他线程未修改 table
        else if (tab == table) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                Node<K,V>[] nt;
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    // 协助扩容
                    transfer(tab, nt);
            }
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                // 发起扩容
                transfer(tab, null);
        }
    }
}
```

addCount

x=1，check=bucketCount

```
private final void addCount(long x, int check) {
    // 计数值加 x
    // 利用 CAS 方法更新 baseCount 的值
    CounterCell[] as; long b, s;
    if ((as = counterCells) != null ||
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        // 如果 CAS 更新 baseCount 失败或者 counterCells 不为空，那么尝试 CAS 更新当前线程的 hashCode 对应的 bucket 的 value
        CounterCell a; long v; int m;
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            !(uncontended =
              U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            // 如果两次 CAS 都失败了，那么调用 fullAddCount 方法
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        s = sumCount();
    }
    // 以上与扩容无关，如果 check 值大于等于 0 则需要检查是否需要进行扩容操作
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            // 如果 sizeCtl 是小于 0 的，说明有其他线程正在执行扩容操作，nextTable
一定不为空
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                // 协助扩容
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            // 当前线程是唯一的或是第一个发起扩容的线程  此时 nextTable=null
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                // 发起扩容
                transfer(tab, null);
            s = sumCount();
        }
    }
}
```

```
private final void fullAddCount(long x, boolean wasUncontended) {
    int h;
    if ((h = ThreadLocalRandom.getProbe()) == 0) {
        ThreadLocalRandom.localInit();      // force initialization
        h = ThreadLocalRandom.getProbe();
        wasUncontended = true;
    }
    boolean collide = false;                // True if last slot nonempty
    for (;;) {
        CounterCell[] as; CounterCell a; int n; long v;
        if ((as = counterCells) != null && (n = as.length) > 0) {
            if ((a = as[(n - 1) & h]) == null) {
                if (cellsBusy == 0) {            // Try to attach new Cell
                    CounterCell r = new CounterCell(x); // Optimistic create
                    if (cellsBusy == 0 &&
                        U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
                        boolean created = false;
                        try {               // Recheck under lock
                            CounterCell[] rs; int m, j;
                            if ((rs = counterCells) != null &&
                                (m = rs.length) > 0 &&
                                rs[j = (m - 1) & h] == null) {
                                rs[j] = r;
                                created = true;
                            }
                        } finally {
                            cellsBusy = 0;
                        }
                        if (created)
                            break;
                        continue;            // Slot is now non-empty
                    }
                }
                collide = false;
            }
            else if (!wasUncontended)         // CAS already known to fail
                wasUncontended = true;        // Continue after rehash
            else if (U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))
                break;
            else if (counterCells != as || n >= NCPU)
                collide = false;             // At max size or stale
            else if (!collide)
                collide = true;
            else if (cellsBusy == 0 &&
                     U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
                try {
                    if (counterCells == as) {// Expand table unless stale
                        CounterCell[] rs = new CounterCell[n << 1];
                        for (int i = 0; i < n; ++i)
                            rs[i] = as[i];
                        counterCells = rs;
                    }
                } finally {
                    cellsBusy = 0;
                }
                collide = false;
                continue;                    // Retry with expanded table
            }
            h = ThreadLocalRandom.advanceProbe(h);
        }
        else if (cellsBusy == 0 && counterCells == as &&
                 U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
            boolean init = false;
            try {                           // Initialize table
```

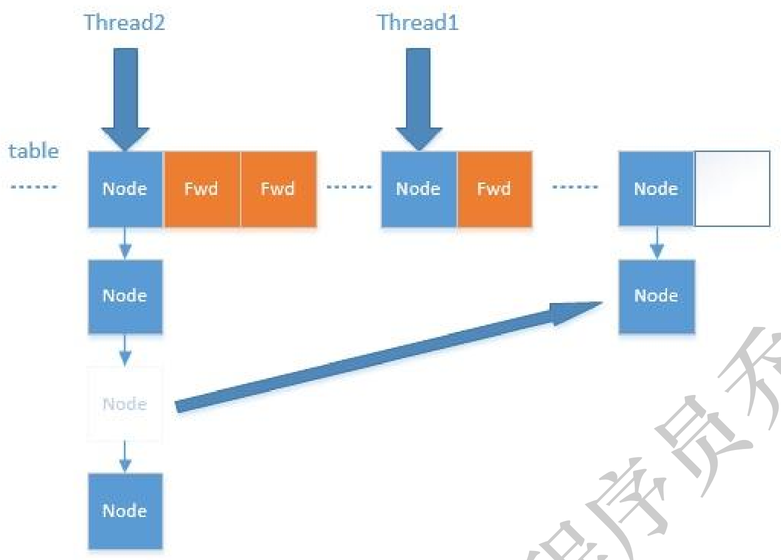
```
                if (counterCells == as) {
                    CounterCell[] rs = new CounterCell[2];
                    rs[h & 1] = new CounterCell(x);
                    counterCells = rs;
                    init = true;
                }
            } finally {
                cellsBusy = 0;
            }
            if (init)
                break;
        }
        else if (U.compareAndSwapLong(this, BASECOUNT, v = baseCount, v + x))
            break;                          // Fall back on using base
    }
}
```

transfer

当 table 容量不足的时候，即 table 的元素数量达到容量阈值 sizeCtl，需要对 table 进行扩容。整个扩容分为两部分：

1）构建一个 nextTable，大小为 table 的两倍。

2）把 table 的数据复制到 nextTable 中。



这两个过程在单线程下实现很简单，但是 ConcurrentHashMap 是支持并发插入的，扩容操作自然也会有并发的出现，这种情况下，第二步可以支持节点的并发复制，这样性能自然提升不少，但实现的复杂度也上升了一个台阶。

先看第一步，构建 nextTable，毫无疑问，这个过程只能有单个线程进行 nextTable 的初始化。通过 Unsafe.compareAndSwapInt 修改 sizeCtl 值，保证只有一个线程能够初始化 nextTable，扩容后的数组长度为原来的两倍。

节点从 table 移动到 nextTable，大体思想是遍历、复制的过程。

1）首先根据运算得到需要遍历的次数 i，然后利用 tabAt 方法获得 i 位置的元素 f，初始化一个 ForwardingNode 实例 fwd。

2）如果 f==null，则在 table 中的 i 位置放入 fwd，这个过程是采用 Unsafe.compareAndSwapObjectf 方法实现的，很巧妙的实现了节点的并发移动。

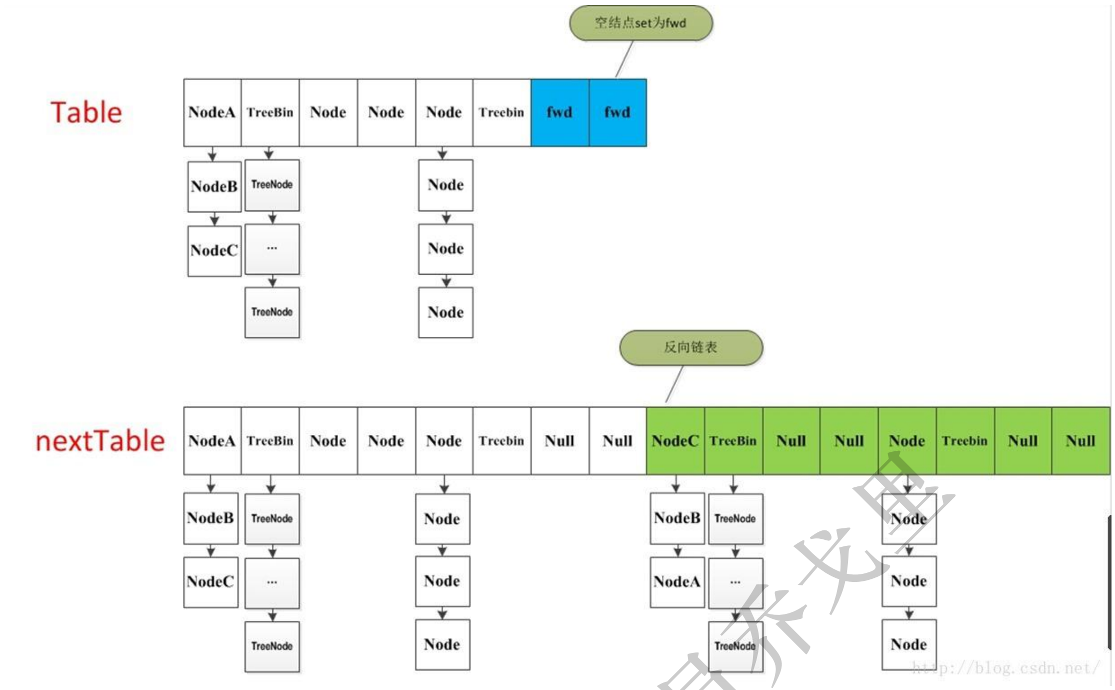
3）如果 f 是链表的头节点，就构造一个反序链表，把他们分别放在 nextTable 的 i 和 i+n 的位置上，移动完成，采用 Unsafe.putObjectVolatile 方法给 table 原位置赋值 fwd。

4）如果 f 是 TreeBin 节点，也做一个反序处理，并判断是否需要 untreeify，把处理的结果分别放在 nextTable 的 i 和 i+n 的位置上，移动完成，同样采用 Unsafe.putObjectVolatile 方法给 table 原位置赋值 fwd。

5）遍历过所有的节点以后就完成了复制工作，把 table 指向 nextTable，并更新 sizeCtl 为新数组大小的 0.75 倍 ，扩容完成。

在多线程环境下，ConcurrentHashMap 用两点来保证正确性：ForwardingNode 和 synchronized。当一个线程遍历到的节点如果是 ForwardingNode，则继续往后遍历，如果不是，则将该节点加锁，防止其他线程进入，完成后设置 ForwardingNode 节点，以便要

其他线程可以看到该节点已经处理过了，如此交叉进行，高效而又安全。



```java
/**
 * Moves and/or copies the nodes in each bin to new table. See
 * above for explanation.
 */
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    int n = tab.length, stride;
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE; // subdivide range
    // 扩容第一步，创建两倍长的数组 nextTable，单线程执行
    // initiating 只能有一个线程进行构造 nextTable，如果别的线程进入发现不为空就不
用构造 nextTable 了
    if (nextTab == null) {            // initiating
        try {
            @SuppressWarnings("unchecked")
            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
            nextTab = nt;
        } catch (Throwable ex) {      // try to cope with OOME
            sizeCtl = Integer.MAX_VALUE;
            return;
        }
        nextTable = nextTab;
        // 原先扩容大小
        transferIndex = n;
    }
    // 扩容第二步，把 table 的数据复制到 nextTable 中，多线程可以同时进行
    int nextn = nextTab.length;
    // 构造一个 ForwardingNode 用于多线程之间的共同扩容情况
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab
    // 遍历每个节点
    for (int i = 0, bound = 0;;) {
        Node<K,V> f; int fh;
        while (advance) {
            int nextIndex, nextBound;
            if (--i >= bound || finishing)
                advance = false;
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1;
                advance = false;
            }
            else if (U.compareAndSwapInt
                     (this, TRANSFERINDEX, nextIndex,
                      nextBound = (nextIndex > stride ?
                                   nextIndex - stride : 0))) {
                bound = nextBound;
                i = nextIndex - 1;
                advance = false;
            }
        }
        // 得到一个 i，i 指向 table 中某一个尚未拷贝的 bucket，下面的代码是对 i 对应
的 bucket 进行拷贝，拷贝完后将 bucket 赋值为 fwd（ForwadingNode）
        //*************************************************************//
        if (i < 0 || i >= n || i + n >= nextn) {
            int sc;
            // 如果原 table 已经复制结束
            if (finishing) {
                nextTable = null;
                table = nextTab;
                // 修改扩容后的阀值，应该是现在容量的 0.75 倍
                sizeCtl = (n << 1) - (n >>> 1);
                return;
            }
            // 采用 CAS 算法更新 SizeCtl，减一，表示有一个新的线程参与到扩容操作
            if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                    return;
                finishing = advance = true;
                i = n; // recheck before commit
            }
        }
        // CAS 算法获取某一个数组的节点，为空就设为 ForwordingNode
        else if ((f = tabAt(tab, i)) == null)
            advance = casTabAt(tab, i, null, fwd);
        // 如果这个节点的 hash 值是 MOVED，就表示这个节点是 ForwordingNode 节点，就
表示这个节点已经被处理过了，直接跳过
        else if ((fh = f.hash) == MOVED)
            advance = true; // already processed
        else {
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    Node<K,V> ln, hn;
                    if (fh >= 0) {
                        //如果这个节点的确是链表节点，则拆分为两个子链表，存储到
nextTable 相应的两个位置
                        int runBit = fh & n;
                        Node<K,V> lastRun = f;
                        for (Node<K,V> p = f.next; p != null; p = p.next) {
                            int b = p.hash & n;
                            if (b != runBit) {
                                runBit = b;
                                lastRun = p;
                            }
                        }
                        if (runBit == 0) {
                            ln = lastRun;
                            hn = null;
                        }
                        else {
                            hn = lastRun;
                            ln = null;
                        }
                        //
                        for (Node<K,V> p = f; p != lastRun; p = p.next) {
                            int ph = p.hash; K pk = p.key; V pv = p.val;
                            if ((ph & n) == 0)
                                ln = new Node<K,V>(ph, pk, pv, ln);
                            else
                                hn = new Node<K,V>(ph, pk, pv, hn);
                        }
                        // CAS 存储在 nextTable 的 i 位置上
                        setTabAt(nextTab, i, ln);
                        // CAS 存储在 nextTable 的 i+n 位置上
                        setTabAt(nextTab, i + n, hn);
                        // CAS 在原 table 的 i 处设置 forwordingNode 节点，表示这个
这个节点已经处理完毕
                        setTabAt(tab, i, fwd);
                        advance = true;
                    }
                    // 如果这个节点是红黑树，则拆分为两颗子树，保存到 nextTable 相
应的两个位置
                    else if (f instanceof TreeBin) {
                        TreeBin<K,V> t = (TreeBin<K,V>)f;
                        TreeNode<K,V> lo = null, loTail = null;
                        TreeNode<K,V> hi = null, hiTail = null;
                        int lc = 0, hc = 0;
                        for (Node<K,V> e = t.first; e != null; e = e.next) {
                            int h = e.hash;
                            TreeNode<K,V> p = new TreeNode<K,V>
                                (h, e.key, e.val, null, null);
                            if ((h & n) == 0) {
                                if ((p.prev = loTail) == null)
                                    lo = p;
                                else
                                    loTail.next = p;
                                loTail = p;
                                ++lc;
                            }
                            else {
                                if ((p.prev = hiTail) == null)
                                    hi = p;
                                else
                                    hiTail.next = p;
                                hiTail = p;
                                ++hc;
                            }
                        }
                        // 如果拆分后的树的节点数量已经少于 6 个就需要重新转化为链表
                        ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
                            (hc != 0) ? new TreeBin<K,V>(lo) : t;
                        hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
                            (lc != 0) ? new TreeBin<K,V>(hi) : t;
                        // CAS 存储在 nextTable 的 i 位置上
                        setTabAt(nextTab, i, ln);
                        // CAS 存储在 nextTable 的 i+n 位置上
                        setTabAt(nextTab, i + n, hn);
                        // CAS 在原 table 的 i 处设置 forwordingNode 节点，表示这个
这个节点已经处理完毕
```

```
                        setTabAt(tab, i, fwd);
                        advance = true;
                    }
                }
            }
        }
    }
}
```

helpTransfer

```
final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
    Node<K,V>[] nextTab; int sc;
    if (tab != null && (f instanceof ForwardingNode) &&
        (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
        int rs = resizeStamp(tab.length);
        while (nextTab == nextTable && table == tab &&
               (sc = sizeCtl) < 0) {
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                sc == rs + MAX_RESIZERS || transferIndex <= 0)
                break;
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                // 调用扩容方法
                transfer(tab, nextTab);
                break;
            }
        }
        return nextTab;
    }
    return table;
}
```

putIfAbsent

```
public V putIfAbsent(K key, V value) {
    return putVal(key, value, true);
}
```

get（无锁）

```java
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    int h = spread(key.hashCode());
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        if ((eh = e.hash) == h) {
            // bucket 中第一个结点就是我们要找的，直接返回
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        else if (eh < 0)
            // bucket 中第一个结点是红黑树根，则调用 find 方法去找
            return (p = e.find(h, key)) != null ? p.val : null;
        // bucket 中第一个结点是链表，则遍历链表查找
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

untreeify（无锁）

```java
static <K,V> Node<K,V> untreeify(Node<K,V> b) {
    Node<K,V> hd = null, tl = null;
    for (Node<K,V> q = b; q != null; q = q.next) {
        Node<K,V> p = new Node<K,V>(q.hash, q.key, q.val, null);
        if (tl == null)
            hd = p;
        else
            tl.next = p;
        tl = p;
    }
    return hd;
}
```

remove（有锁）

```java
public boolean remove(Object key, Object value) {
    if (key == null)
        throw new NullPointerException();
    return value != null && replaceNode(key, null, value) != null;
}
```

```java
/**
 * Implementation for the four public remove/replace methods:
 * Replaces node value with v, conditional upon match of cv if
 * non-null.  If resulting value is null, delete.
 */
final V replaceNode(Object key, V value, Object cv) {
    int hash = spread(key.hashCode());
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0 ||
            (f = tabAt(tab, i = (n - 1) & hash)) == null)
            break;
        else if ((fh = f.hash) == MOVED)
            // 如果已经被移动，那么就帮助进行扩容
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            boolean validated = false;
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    // 如果是链表，则删除链表中的节点
                    if (fh >= 0) {
                        validated = true;
                        for (Node<K,V> e = f, pred = null;;) {
                            K ek;
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                V ev = e.val;
                                if (cv == null || cv == ev ||
                                    (ev != null && cv.equals(ev))) {
                                    oldVal = ev;
                                    if (value != null)
                                        e.val = value;
                                    else if (pred != null)
                                        pred.next = e.next;
                                    else
                                        setTabAt(tab, i, e.next);
                                }
                                break;
                            }
                            pred = e;
                            if ((e = e.next) == null)
                                break;
                        }
                    }
                    // 如果是红黑树，则从红黑树中删除结点
                    else if (f instanceof TreeBin) {
                        validated = true;
                        TreeBin<K,V> t = (TreeBin<K,V>)f;
                        TreeNode<K,V> r, p;
                        if ((r = t.root) != null &&
                            (p = r.findTreeNode(hash, key, null)) != null) {
                            V pv = p.val;
                            if (cv == null || cv == pv ||
                                (pv != null && cv.equals(pv))) {
                                oldVal = pv;
                                if (value != null)
                                    p.val = value;
                                else if (t.removeTreeNode(p))
                                    setTabAt(tab, i, untreeify(t.first));
                            }
                        }
                    }
                }
            }
            if (validated) {
                if (oldVal != null) {
                    if (value == null)
                        addCount(-1L, -1);
                    return oldVal;
                }
                break;
            }
        }
    }
    return null;
}
```

containsKey

```
public boolean containsKey(Object key) {
    return get(key) != null;
}
```

containsValue

```
public boolean containsValue(Object value) {
    if (value == null)
        throw new NullPointerException();
    Node<K,V>[] t;
    if ((t = table) != null) {
        Traverser<K,V> it = new Traverser<K,V>(t, t.length, 0, t.length);
        for (Node<K,V> p; (p = it.advance()) != null; ) {
            V v;
            if ((v = p.val) == value || (v != null && value.equals(v)))
                return true;
        }
    }
    return false;
}
```

```java
static class Traverser<K,V> {
    Node<K,V>[] tab;        // current table; updated if resized
    Node<K,V> next;         // the next entry to use
    TableStack<K,V> stack, spare; // to save/restore on ForwardingNodes
    int index;              // index of bin to use next
    int baseIndex;          // current index of initial table
    int baseLimit;          // index bound for initial table
    final int baseSize;     // initial table size

    Traverser(Node<K,V>[] tab, int size, int index, int limit) {
        this.tab = tab;
        this.baseSize = size;
        this.baseIndex = this.index = index;
        this.baseLimit = limit;
        this.next = null;
    }

    /**
     * Advances if possible, returning next valid node, or null if none.
     */
    final Node<K,V> advance() {
        Node<K,V> e;
        if ((e = next) != null)
            e = e.next;
        for (;;) {
            Node<K,V>[] t; int i, n;  // must use locals in checks
            if (e != null)
                return next = e;
            if (baseIndex >= baseLimit || (t = tab) == null ||
                (n = t.length) <= (i = index) || i < 0)
                return next = null;
            if ((e = tabAt(t, i)) != null && e.hash < 0) {
                if (e instanceof ForwardingNode) {
                    tab = ((ForwardingNode<K,V>)e).nextTable;
                    e = null;
                    pushState(t, i, n);
                    continue;
                }
                else if (e instanceof TreeBin)
                    e = ((TreeBin<K,V>)e).first;
                else
                    e = null;
            }
            if (stack != null)
```

```
                recoverState(n);
            else if ((index = i + baseSize) >= n)
                index = ++baseIndex; // visit upper slots if present
        }
    }

    /**
     * Saves traversal state upon encountering a forwarding node.
     */
    private void pushState(Node<K,V>[] t, int i, int n) {
        TableStack<K,V> s = spare;  // reuse if possible
        if (s != null)
            spare = s.next;
        else
            s = new TableStack<K,V>();
        s.tab = t;
        s.length = n;
        s.index = i;
        s.next = stack;
        stack = s;
    }

    /**
     * Possibly pops traversal state.
     *
     * @param n length of current table
     */
    private void recoverState(int n) {
        TableStack<K,V> s; int len;
        while ((s = stack) != null && (index += (len = s.length)) >= n) {
            n = len;
            index = s.index;
            tab = s.tab;
            s.tab = null;
            TableStack<K,V> next = s.next;
            s.next = spare; // save for reuse
            stack = next;
            spare = s;
        }
        if (s == null && (index += baseSize) >= n)
            index = ++baseIndex;
    }
}
```

遍历

```java
public Set<Map.Entry<K,V>> entrySet() {
    EntrySetView<K,V> es;
    return (es = entrySet) != null ? es : (entrySet = new EntrySetView<K,V>(this));
}
```

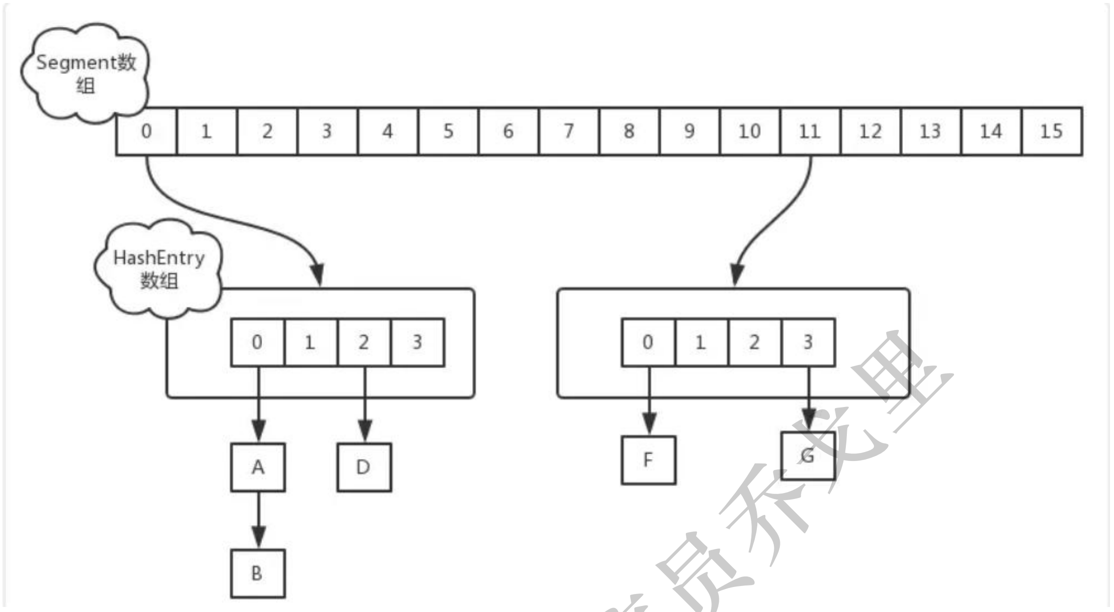
```java
public Iterator<Map.Entry<K,V>> iterator() {
    ConcurrentHashMap<K,V> m = map;
    Node<K,V>[] t;
    int f = (t = m.table) == null ? 0 : t.length;
    return new EntryIterator<K,V>(t, f, 0, f, m);
}
```

```java
static class BaseIterator<K,V> extends Traverser<K,V> {
    final ConcurrentHashMap<K,V> map;
    Node<K,V> lastReturned;
    BaseIterator(Node<K,V>[] tab, int size, int index, int limit,
                 ConcurrentHashMap<K,V> map) {
        super(tab, size, index, limit);
        this.map = map;
        advance();
    }

    public final boolean hasNext() { return next != null; }
    public final boolean hasMoreElements() { return next != null; }

    public final void remove() {
        Node<K,V> p;
        if ((p = lastReturned) == null)
            throw new IllegalStateException();
        lastReturned = null;
        map.replaceNode(p.key, null, null);
    }
}
```

```java
static final class EntryIterator<K,V> extends BaseIterator<K,V>
    implements Iterator<Map.Entry<K,V>> {
    EntryIterator(Node<K,V>[] tab, int index, int size, int limit,
                  ConcurrentHashMap<K,V> map) {
        super(tab, index, size, limit, map);
    }

    public final Map.Entry<K,V> next() {
        Node<K,V> p;
        if ((p = next) == null)
            throw new NoSuchElementException();
        K k = p.key;
        V v = p.val;
        lastReturned = p;
        advance();
        return new MapEntry<K,V>(k, v, map);
    }
}
```

1.7 分段锁实现

采用 Segment + HashEntry 的方式进行实现



put

当执行 put 方法插入数据时，根据 key 的 hash 值，在 Segment 数组中找到相应的位置，如果相应位置的 Segment 还未初始化，则通过 CAS 进行赋值，接着执行 Segment 对象的 put 方法通过加锁机制插入数据，实现如下：

场景：线程 A 和线程 B 同时执行相同 Segment 对象的 put 方法

1、线程 A 执行 tryLock()方法成功获取锁，则把 HashEntry 对象插入到相应的位置；

2、线程 B 获取锁失败，则执行 scanAndLockForPut()方法，在 scanAndLockForPut 方法中，会通过重复执行 tryLock()方法尝试获取锁，在多处理器环境下，重复次数为 64，单处理器重复次数为 1，当执行 tryLock()方法的次数超过上限时，则执行 lock()方法挂起线程 B；

3、当线程 A 执行完插入操作时，会通过 unlock()方法释放锁，接着唤醒线程 B 继续执行；

size

因为 ConcurrentHashMap是可以并发插入数据的,所以在准确计算元素时存在一定的难度,一般的思路是统计每个 Segment 对象中的元素个数，然后进行累加，但是这种方式计算出来的结果并不一样的准确的，因为在计算后面几个 Segment 的元素个数时，已经计算过的 Segment 同时可能有数据的插入或则删除。
先采用不加锁的方式，连续计算元素的个数，最多计算 3 次： 1、如果前后两次计算结果相

同，则说明计算出来的元素个数是准确的； 2、如果前后两次计算结果都不同，则给每个 Segment 进行加锁，再计算一次元素的个数；

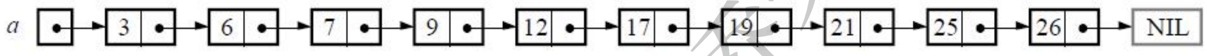
ConcurrentSkipListMap

ConcurrentSkipListMap 有几个 ConcurrentHashMap 不能比拟的优点：

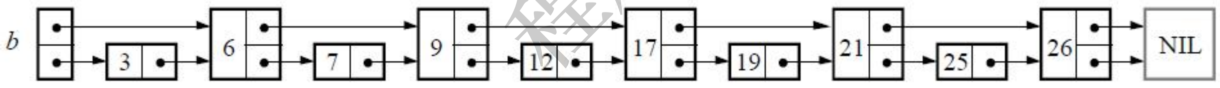
1、ConcurrentSkipListMap 的 key 是有序的。

2、ConcurrentSkipListMap 支持更高的并发。ConcurrentSkipListMap 的存取时间是 log（N），和线程数几乎无关。也就是说在数据量一定的情况下，并发的线程越多，ConcurrentSkipListMap 越能体现出他的优势。

SkipList 跳表：
跳表是平衡树的一种替代的数据结构，但是和红黑树不相同的是，跳表对于树的平衡的实现是基于一种随机化的算法的，这样也就是说跳表的插入和删除的工作是比较简单的。
下面来研究一下跳表的核心思想：
先从链表开始，如果是一个简单的链表，那么我们知道在链表中查找一个元素 I 的话，需要将整个链表遍历一次。



如果是说链表是排序的，并且节点中还存储了指向前面第二个节点的指针的话，那么在查找一个节点时，仅仅需要遍历 N/2 个节点即可。



这基本上就是跳表的核心思想，其实也是一种通过“空间来换取时间”的一个算法，通过在每个节点中增加了向前的指针，从而提升查找的效率。

Map 实现类之间的区别

HashMap 与 ConcurrentHashMap 区别

1）前者允许 key 或 value 为 null，后者不允许
2）前者不是线程安全的，后者是

HashMap、TreeMap 与 LinkedHashMap 区别

1）HashMap 遍历时，取得数据的顺序是完全随机的；
TreeMap 可以按照自然顺序或 Comparator 排序；
LinkedHashMap 可以按照插入顺序或访问顺序排序，且 get 的效率（O(1)）比 TreeMap（O(logn)）更高。
2）HashMap 底层基于哈希表，数组+链表/红黑树；
TreeMap 底层基于红黑树
LinkedHashMap 底层基于 HashMap 与环形双向链表
3）就 get 和 put 效率而言，HashMap 是最高的，LinkedHashMap 次之，TreeMap 最次。

HashMap 与 Hashtable 区别

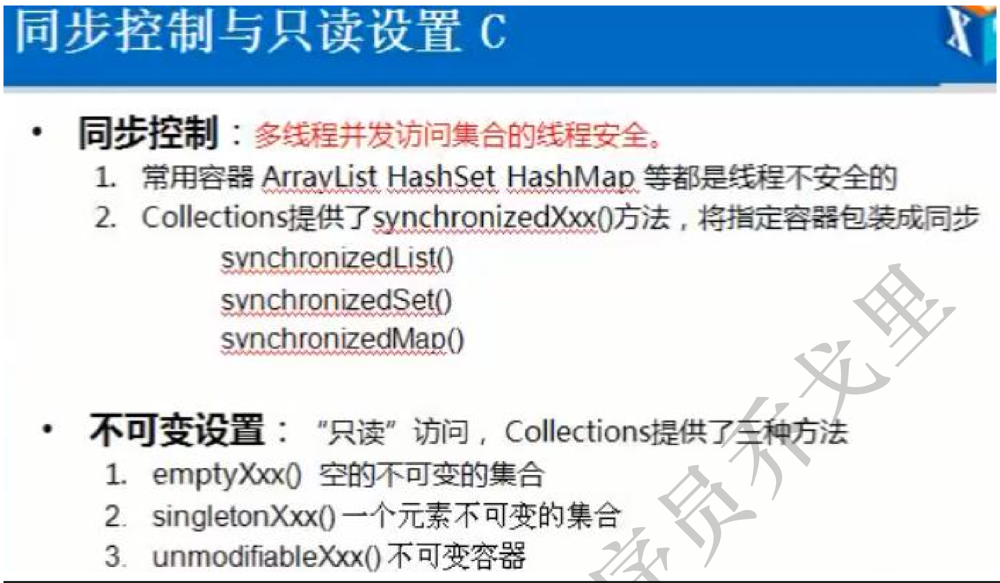
1. 扩容策略：Hashtable 在不指定容量的情况下的默认容量为 11，而 HashMap 为 16，Hashtable 不要求底层数组的容量一定要为 2 的整数次幂（*2+1），而 HashMap 则要求一定为 2 的整数次幂（*2）。
2. 允许 null：Hashtable 中 key 和 value 都不允许为 null，而 HashMap 中 key 和 value 都允许为 null（key 只能有一个为 null，而 value 则可以有多个为 null）。
3. 线程安全：前者不是线程安全的，后者是；

ConcurrentHashMap、Collections.synchronizedMap 与 Hashtable 的异同

它们都是同步 Map，但三者实现同步的机制不同；后两者都是简单地在方法上加 synchronized 实现的，锁的粒度较大；前者是基于 CAS 和 synchronized 实现的，锁的粒度较小，大部分都是 lock-free 无锁实现同步的。
ConcurrentHashMap 还提供了 putIfAbsent 同步方法。

Collections

同步集合包装



同步控制与只读设置 C

- **同步控制**：多线程并发访问集合的线程安全。
  1. 常用容器 ArrayList HashSet HashMap 等都是线程不安全的
  2. Collections提供了synchronizedXxx()方法，将指定容器包装成同步
     - synchronizedList()
     - synchronizedSet()
     - synchronizedMap()
- **不可变设置**：“只读”访问，Collections提供了三种方法
  1. emptyXxx() 空的不可变的集合
  2. singletonXxx() 一个元素不可变的集合
  3. unmodifiableXxx() 不可变容器

```java
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {
    return new SynchronizedMap<>(m);
}
```

```java
private static class SynchronizedMap<K,V>
    implements Map<K,V>, Serializable {
    private static final long serialVersionUID = 1978198479659022715L;

    private final Map<K,V> m;     // Backing Map
    final Object      mutex;        // Object on which to synchronize

    SynchronizedMap(Map<K,V> m) {
        this.m = Objects.requireNonNull(m);
        mutex = this;
    }

    SynchronizedMap(Map<K,V> m, Object mutex) {
        this.m = m;
        this.mutex = mutex;
    }

    public int size() {
        synchronized (mutex) {return m.size();}
    }
    public boolean isEmpty() {
        synchronized (mutex) {return m.isEmpty();}
    }
    public boolean containsKey(Object key) {
        synchronized (mutex) {return m.containsKey(key);}
    }
    public boolean containsValue(Object value) {
        synchronized (mutex) {return m.containsValue(value);}
    }
    public V get(Object key) {
        synchronized (mutex) {return m.get(key);}
    }

    public V put(K key, V value) {
        synchronized (mutex) {return m.put(key, value);}
    }
    public V remove(Object key) {
        synchronized (mutex) {return m.remove(key);}
    }
    public void putAll(Map<? extends K, ? extends V> map) {
        synchronized (mutex) {m.putAll(map);}
    }
    public void clear() {
        synchronized (mutex) {m.clear();}
    }

    private transient Set<K> keySet;
    private transient Set<Map.Entry<K,V>> entrySet;
    private transient Collection<V> values;

    public Set<K> keySet() {
        synchronized (mutex) {
            if (keySet==null)
                keySet = new SynchronizedSet<>(m.keySet(), mutex);
            return keySet;
        }
    }

    public Set<Map.Entry<K,V>> entrySet() {
        synchronized (mutex) {
            if (entrySet==null)
                entrySet = new SynchronizedSet<>(m.entrySet(), mutex);
            return entrySet;
        }
    }
```

```java
    public Collection<V> values() {
        synchronized (mutex) {
            if (values==null)
                values = new SynchronizedCollection<>(m.values(), mutex);
            return values;
        }
    }

    public boolean equals(Object o) {
        if (this == o)
            return true;
        synchronized (mutex) {return m.equals(o);}
    }
    public int hashCode() {
        synchronized (mutex) {return m.hashCode();}
    }
    public String toString() {
        synchronized (mutex) {return m.toString();}
    }

    // Override default methods in Map
    @Override
    public V getOrDefault(Object k, V defaultValue) {
        synchronized (mutex) {return m.getOrDefault(k, defaultValue);}
    }
    @Override
    public void forEach(BiConsumer<? super K, ? super V> action) {
        synchronized (mutex) {m.forEach(action);}
    }
    @Override
    public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)
{
        synchronized (mutex) {m.replaceAll(function);}
    }
    @Override
    public V putIfAbsent(K key, V value) {
        synchronized (mutex) {return m.putIfAbsent(key, value);}
    }
    @Override
    public boolean remove(Object key, Object value) {
        synchronized (mutex) {return m.remove(key, value);}
    }
    @Override
    public boolean replace(K key, V oldValue, V newValue) {
        synchronized (mutex) {return m.replace(key, oldValue, newValue);}
    }
    @Override
    public V replace(K key, V value) {
        synchronized (mutex) {return m.replace(key, value);}
    }
    @Override
    public V computeIfAbsent(K key,
            Function<? super K, ? extends V> mappingFunction) {
        synchronized (mutex) {return m.computeIfAbsent(key, mappingFunction);}
    }
    @Override
    public V computeIfPresent(K key,
            BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
        synchronized (mutex) {return m.computeIfPresent(key,
remappingFunction);}
    }
    @Override
    public V compute(K key,
            BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
        synchronized (mutex) {return m.compute(key, remappingFunction);}
    }
    @Override
    public V merge(K key, V value,
            BiFunction<? super V, ? super V, ? extends V> remappingFunction) {
        synchronized (mutex) {return m.merge(key, value, remappingFunction);}
    }

    private void writeObject(ObjectOutputStream s) throws IOException {
        synchronized (mutex) {s.defaultWriteObject();}
    }
}
```

不可变集合包装

```
public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m)
{
    return new UnmodifiableMap<>(m);
}
```

```
private static class UnmodifiableMap<K,V> implements Map<K,V>, Serializable {
    private static final long serialVersionUID = -1034234728574286014L;
```

```java
private final Map<? extends K, ? extends V> m;

UnmodifiableMap(Map<? extends K, ? extends V> m) {
    if (m==null)
        throw new NullPointerException();
    this.m = m;
}

public int size()                          {return m.size();}
public boolean isEmpty()                    {return m.isEmpty();}
public boolean containsKey(Object key)   {return m.containsKey(key);}
public boolean containsValue(Object val) {return m.containsValue(val);}
public V get(Object key)                    {return m.get(key);}

public V put(K key, V value) {
    throw new UnsupportedOperationException();
}
public V remove(Object key) {
    throw new UnsupportedOperationException();
}
public void putAll(Map<? extends K, ? extends V> m) {
    throw new UnsupportedOperationException();
}
public void clear() {
    throw new UnsupportedOperationException();
}

private transient Set<K> keySet;
private transient Set<Map.Entry<K,V>> entrySet;
private transient Collection<V> values;

public Set<K> keySet() {
    if (keySet==null)
        keySet = unmodifiableSet(m.keySet());
    return keySet;
}

public Set<Map.Entry<K,V>> entrySet() {
    if (entrySet==null)
        entrySet = new UnmodifiableEntrySet<>(m.entrySet());
    return entrySet;
}
```

```
    public Collection<V> values() {
        if (values==null)
            values = unmodifiableCollection(m.values());
        return values;
    }

    public boolean equals(Object o) {return o == this || m.equals(o);}
    public int hashCode()          {return m.hashCode();}
    public String toString()       {return m.toString();}

    // Override default methods in Map
    @Override
    @SuppressWarnings("unchecked")
    public V getOrDefault(Object k, V defaultValue) {
        // Safe cast as we don't change the value
        return ((Map<K, V>)m).getOrDefault(k, defaultValue);
    }

    @Override
    public void forEach(BiConsumer<? super K, ? super V> action) {
        m.forEach(action);
    }

    @Override
    public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)
{
        throw new UnsupportedOperationException();
    }

    @Override
    public V putIfAbsent(K key, V value) {
        throw new UnsupportedOperationException();
    }

    @Override
    public boolean remove(Object key, Object value) {
        throw new UnsupportedOperationException();
    }

    @Override
    public boolean replace(K key, V oldValue, V newValue) {
        throw new UnsupportedOperationException();
    }
    @Override
    public V replace(K key, V value) {
        throw new UnsupportedOperationException();
    }

    @Override
    public V computeIfAbsent(K key, Function<? super K, ? extends V>
mappingFunction) {
        throw new UnsupportedOperationException();
    }

    @Override
    public V computeIfPresent(K key,
            BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
        throw new UnsupportedOperationException();
    }

    @Override
    public V compute(K key,
            BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
        throw new UnsupportedOperationException();
    }

    @Override
    public V merge(K key, V value,
            BiFunction<? super V, ? super V, ? extends V> remappingFunction) {
        throw new UnsupportedOperationException();
    }
```

空集合包装

```
public static final <K,V> Map<K,V> emptyMap() {
    return (Map<K,V>) EMPTY_MAP;
}
```

```
public static final Map EMPTY_MAP = new EmptyMap<>();
```

```
private static class EmptyMap<K,V>
    extends AbstractMap<K,V>
    implements Serializable
{
    private static final long serialVersionUID = 6428348081105594320L;

    public int size()                          {return 0;}
    public boolean isEmpty()                   {return true;}
    public boolean containsKey(Object key)     {return false;}
    public boolean containsValue(Object value) {return false;}
    public V get(Object key)                   {return null;}
    public Set<K> keySet()                     {return emptySet();}
    public Collection<V> values()              {return emptySet();}
    public Set<Map.Entry<K,V>> entrySet()      {return emptySet();}

    public boolean equals(Object o) {
        return (o instanceof Map) && ((Map<?,?>)o).isEmpty();
    }

    public int hashCode()                      {return 0;}

    // Override default methods in Map
    @Override
    @SuppressWarnings("unchecked")
    public V getOrDefault(Object k, V defaultValue) {
        return defaultValue;
    }

    @Override
    public void forEach(BiConsumer<? super K, ? super V> action) {
        Objects.requireNonNull(action);
    }

    @Override
    public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)
{
        Objects.requireNonNull(function);
    }

    @Override
    public V putIfAbsent(K key, V value) {
        throw new UnsupportedOperationException();
    }

    @Override
    public boolean remove(Object key, Object value) {
        throw new UnsupportedOperationException();
    }

    @Override
```

```java
    public boolean replace(K key, V oldValue, V newValue) {
        throw new UnsupportedOperationException();
    }

    @Override
    public V replace(K key, V value) {
        throw new UnsupportedOperationException();
    }

    @Override
    public V computeIfAbsent(K key,
            Function<? super K, ? extends V> mappingFunction) {
        throw new UnsupportedOperationException();
    }

    @Override
    public V computeIfPresent(K key,
            BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
        throw new UnsupportedOperationException();
    }

    @Override
    public V compute(K key,
            BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
        throw new UnsupportedOperationException();
    }

    @Override
    public V merge(K key, V value,
            BiFunction<? super V, ? super V, ? extends V> remappingFunction) {
        throw new UnsupportedOperationException();
    }

    // Preserves singleton property
    private Object readResolve() {
        return EMPTY_MAP;
    }
}
```

Collections.sort

```java
public static <T extends Comparable<? super T>> void sort(List<T> list) {
    list.sort(null);
}
```

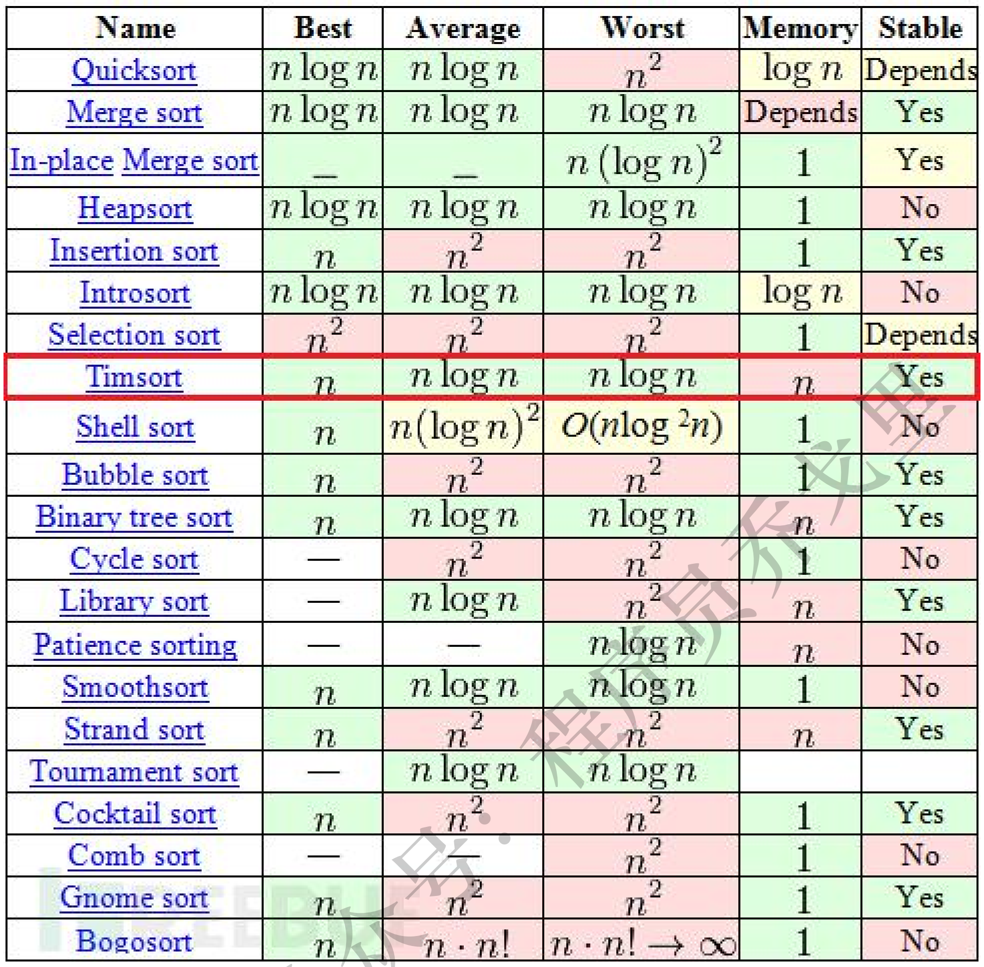
List#sort

```java
default void sort(Comparator<? super E> c) {
    Object[] a = this.toArray();
    Arrays.sort(a, (Comparator) c);
    ListIterator<E> i = this.listIterator();
    for (Object e : a) {
        i.next();
        i.set((E) e);
    }
}
```

```java
public static <T> void sort(T[] a, Comparator<? super T> c) {
    if (c == null) {
        sort(a);
    } else {
        if (LegacyMergeSort.userRequested)
            legacyMergeSort(a, c);
        else
            TimSort.sort(a, 0, a.length, c, null, 0, 0);
    }
}
```

```java
public static void sort(Object[] a) {
    if (LegacyMergeSort.userRequested)
        legacyMergeSort(a);
    else
        ComparableTimSort.sort(a, 0, a.length, null, 0, 0);
}
```

1.7（TimSort）



Name	Best	Average	Worst	Memory	Stable
Quicksort	$n\log n$	$n\log n$	n^2	$\log n$	Depends
Merge sort	$n\log n$	$n\log n$	$n\log n$	Depends	Yes
In-place Merge sort	—	—	$n\,(\log n)^2$	1	Yes
Heapsort	$n\log n$	$n\log n$	$n\log n$	1	No
Insertion sort	n	n^2	n^2	1	Yes
Introsort	$n\log n$	$n\log n$	$n\log n$	$\log n$	No
Selection sort	n^2	n^2	n^2	1	Depends
Timsort	n	$n\log n$	$n\log n$	n	Yes
Shell sort	n	$n(\log n)^2$	$O(n\log^2 n)$	1	No
Bubble sort	n	n^2	n^2	1	Yes
Binary tree sort	n	$n\log n$	$n\log n$	n	Yes
Cycle sort	—	n^2	n^2	1	No
Library sort	—	$n\log n$	n^2	n	Yes
Patience sorting	—	—	$n\log n$	n	No
Smoothsort	n	$n\log n$	$n\log n$	1	No
Strand sort	n	n^2	n^2	n	Yes
Tournament sort	—	$n\log n$	$n\log n$		
Cocktail sort	n	n^2	n^2	1	Yes
Comb sort	—	—	n^2	1	No
Gnome sort	n	n^2	n^2	1	Yes
Bogosort	n	$n\cdot n!$	$n\cdot n!\to\infty$	1	No

结合了归并排序和插入排序的混合算法，它基于一个简单的事实，实际中大部分数据都是部分有序（升序或降序）的。

TimSort 算法为了减少对升序部分的回溯和对降序部分的性能倒退，将输入按其升序和降序特点进行了分区。排序的输入的单位不是一个个单独的数字，而是一个个的块-分区。其中**每一个分区叫一个 run。针对这些 run 序列，每次拿一个 run 出来按规则进行合并。每次合并会将两个 run 合并成一个 run。**合并的结果保存到栈中。合并直到消耗掉所有的 run，这时将栈上剩余的 run 合并到只剩一个 run 为止。这时这个仅剩的 run 便是排好序的结果。

综上述过程，Timsort 算法的过程包括

（0）如果数组长度小于某个值，直接用二分插入排序算法

（1）找到各个 run，并入栈

（2）按规则合并 run

```
/**
 * Sorts the given range, using the given workspace array slice
 * for temp storage when possible. This method is designed to be
 * invoked from public methods (in class Arrays) after performing
```

```java
 * any necessary array bounds checks and expanding parameters into
 * the required forms.
 *
 * @param a the array to be sorted
 * @param lo the index of the first element, inclusive, to be sorted
 * @param hi the index of the last element, exclusive, to be sorted
 * @param work a workspace array (slice)
 * @param workBase origin of usable space in work array
 * @param workLen usable size of work array
 * @since 1.8
 */
static void sort(Object[] a, int lo, int hi, Object[] work, int workBase, int
workLen) {
    assert a != null && lo >= 0 && lo <= hi && hi <= a.length;

    int nRemaining  = hi - lo;
    if (nRemaining < 2)
        return;  // Arrays of size 0 and 1 are always sorted

    // If array is small, do a "mini-TimSort" with no merges
    if (nRemaining < MIN_MERGE) {
        int initRunLen = countRunAndMakeAscending(a, lo, hi);
        binarySort(a, lo, hi, lo + initRunLen);
        return;
    }

    /**
     * March over the array once, left to right, finding natural runs,
     * extending short natural runs to minRun elements, and merging runs
     * to maintain stack invariant.
     */
    ComparableTimSort ts = new ComparableTimSort(a, work, workBase, workLen);
    int minRun = minRunLength(nRemaining);
    do {
        // Identify next run
        int runLen = countRunAndMakeAscending(a, lo, hi);

        // If run is short, extend to min(minRun, nRemaining)
        if (runLen < minRun) {
            int force = nRemaining <= minRun ? nRemaining : minRun;
            binarySort(a, lo, lo + force, lo + runLen);
            runLen = force;
        }
        // Push run onto pending-run stack, and maybe merge
        ts.pushRun(lo, runLen);
        ts.mergeCollapse();

        // Advance to find next run
        lo += runLen;
        nRemaining -= runLen;
    } while (nRemaining != 0);

    // Merge all remaining runs to complete sort
    assert lo == hi;
    ts.mergeForceCollapse();
    assert ts.stackSize == 1;
}
```

```
/**
 * Creates a TimSort instance to maintain the state of an ongoing sort.
 *
 * @param a the array to be sorted
 * @param work a workspace array (slice)
 * @param workBase origin of usable space in work array
 * @param workLen usable size of work array
 */
private ComparableTimSort(Object[] a, Object[] work, int workBase, int workLen)
{
    this.a = a;

    // Allocate temp storage (which may be increased later if necessary)
    int len = a.length;
    int tlen = (len < 2 * INITIAL_TMP_STORAGE_LENGTH) ?
        len >>> 1 : INITIAL_TMP_STORAGE_LENGTH;
    if (work == null || workLen < tlen || workBase + tlen > work.length) {
        tmp = new Object[tlen];
        tmpBase = 0;
        tmpLen = tlen;
    }
    else {
        tmp = work;
        tmpBase = workBase;
        tmpLen = workLen;
    }

    /*
     * Allocate runs-to-be-merged stack (which cannot be expanded).  The
     * stack length requirements are described in listsort.txt.  The C
     * version always uses the same stack length (85), but this was
     * measured to be too expensive when sorting "mid-sized" arrays (e.g.,
     * 100 elements) in Java.  Therefore, we use smaller (but sufficiently
     * large) stack lengths for smaller arrays.  The "magic numbers" in the
     * computation below must be changed if MIN_MERGE is decreased.  See
     * the MIN_MERGE declaration above for more information.
     * The maximum value of 49 allows for an array up to length
     * Integer.MAX_VALUE-4, if array is filled by the worst case stack size
     * increasing scenario. More explanations are given in section 4 of:
     * http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf
     */
    int stackLen = (len <    120  ?  5 :
                   len <   1542  ? 10 :
                   len < 119151  ? 24 : 49);
    runBase = new int[stackLen];
    runLen = new int[stackLen];
}
```

1.6 （MergeSort）

```
private static void legacyMergeSort(Object[] a) {
    Object[] aux = a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

```
private static void mergeSort(Object[] src,
                              Object[] dest,
                              int low,
                              int high,
                              int off) {
    int length = high - low;
    // 7
    // Insertion sort on smallest arrays
    if (length < INSERTIONSORT_THRESHOLD) {
        for (int i=low; i<high; i++)
            for (int j=i; j>low &&
                     ((Comparable) dest[j-1]).compareTo(dest[j])>0; j--)
                swap(dest, j, j-1);
        return;
    }

    // Recursively sort halves of dest into src
```

```java
    int destLow  = low;
    int destHigh = high;
    low  += off;
    high += off;
    int mid = (low + high) >>> 1;
    mergeSort(dest, src, low, mid, -off);
    mergeSort(dest, src, mid, high, -off);

    // If list is already sorted, just copy from src to dest.  This is an
    // optimization that results in faster sorts for nearly ordered lists.
    if (((Comparable)src[mid-1]).compareTo(src[mid]) <= 0) {
        System.arraycopy(src, low, dest, destLow, length);
        return;
    }

    // Merge sorted halves (now in src) into dest
    for(int i = destLow, p = low, q = mid; i < destHigh; i++) {
        if (q >= high || p < mid && ((Comparable)src[p]).compareTo(src[q])<=0)
            dest[i] = src[p++];
        else
            dest[i] = src[q++];
    }
}
```

Fail-Fast

在 ArrayList,LinkedList,HashMap 等等的内部实现增，删，改中我们总能看到 modCount 的身影，modCount 字面意思就是修改次数，但为什么要记录 modCount 的修改次数呢？
所有使用 modCount 属性集合的都是线程不安全的。
在一个迭代器初始的时候会赋予它调用这个迭代器的对象的 modCount，在迭代器遍历的过程中，一旦发现这个对象的 modCount 和迭代器中存储的 modCount 不一样那就抛异常。

它是 java 集合的一种错误检测机制，当多个线程对集合进行结构上的改变的操作时，有可能会产生 fail-fast。

例如 ：假设存在两个线程（线程 1、线程 2），线程 1 通过 Iterator 在遍历集合 A 中的元素，在某个时候线程 2 修改了集合 A 的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生 fail-fast 机制。

原因：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变 modCount 的值。

每当迭代器使用 hashNext()/next() 遍历下一个元素之前, 都会检测 modCount 变量是否为 expectedmodCount 值，是的话就返回遍历；否则抛出异常，终止遍历。

解决办法：使用线程安全的集合