



链滴

Java 集合源码学习

作者: [Gakkiyomi2019](#)

原文链接: <https://ld246.com/article/1616562897341>

来源网站: [链滴](#)

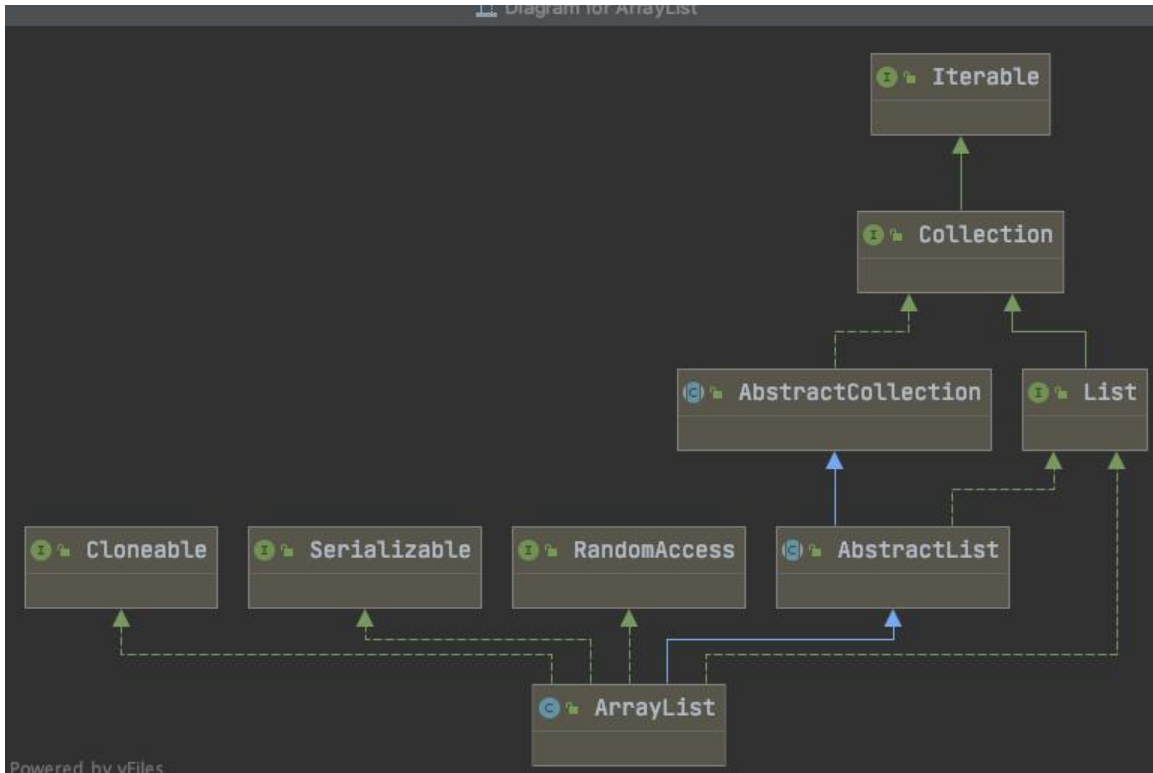
许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

Java集合源码学习

集中记录java中的集合源码以供随时索引和查询。水平有限，请大家勘正。

ArrayList

继承关系图



- 实现了 **Cloneable**(空接口)代表可以调用object.clone方法
- 实现 **Serializable**(空接口)接口 代表可以进行序列化和反序列化
- 实现了 **RandomAccess**(空接口) 代表实现了高速随机访问，如果标注了这个接口，则推荐使用for循环遍历，没有则更推荐迭代器
- 继承自 **AbstractList**抽象类,继承一些list的通用方法
- 实现List接口

默认容量

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

底层数组共享吗？

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

```

.....

/**
 * 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 = {};

.....

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);
    }
}

/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
}

```

- 如果初始化initialCapacity为0的list 则底层共用一个空数组实例 `EMPTY_ELEMENTDATA`
- 如果使用无参构造方法也使用了同一个空数组实例 `DEFAULTCAPACITY_EMPTY_ELEMENTDATA`

```

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

```

ArrayList可以从同一个集合的底层array进行构造。这似乎和go的slice一样，两个list共用一个底层数组，但是继续往下看则会发现。

toArray 会生成新数组，所以除了上面总结的两种共用底层数组的情况，ArrayList不会共用一个底层数组。

```
/**
 * Returns an array containing all of the elements in this list
 * in proper sequence (from first to last element).
 *
 * <p>The returned array will be "safe" in that no references to it are
 * maintained by this list. (In other words, this method must allocate
 * a new array). The caller is thus free to modify the returned array.
 *
 * <p>This method acts as bridge between array-based and collection-based
 * APIs.
 *
 * @return an array containing all of the elements in this list in
 *         proper sequence
 */
public Object[] toArray() {
    return Arrays.copyOf(elementData, size);
}
```

trimToSize()

trimToSize() 方法用于将List中的容量调整为数组中的元素个数。

```
/**
 * Trims the capacity of this <tt>ArrayList</tt> instance to be the
 * list's current size. An application can use this operation to minimize
 * the storage of an <tt>ArrayList</tt> instance.
 */
public void trimToSize() {
    modCount++;
    if (size < elementData.length) {
        elementData = (size == 0)
            ? EMPTY_ELEMENTDATA // 如果size为空 则将底层数组设为共享的空数组
            : Arrays.copyOf(elementData, size); // 如果不为空 则根据size copy一个新数组
    }
}
```

扩容

```
public void ensureCapacity(int minCapacity) {
    int minExpand = (elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
        // 如果没有使用默认空数组 也就是new ArrayList()产生的实例。则期望所有的> 0的扩容量
        ? 0
        // 如果使用了默认空数组 则期望的扩容量要 > 默认的10
        : DEFAULT_CAPACITY;

    if (minCapacity > minExpand) {
        ensureExplicitCapacity(minCapacity);
    }
}
```

```
private void ensureExplicitCapacity(int minCapacity) {
    modCount++; //操作次数加一

    // overflow-conscious code
    if (minCapacity - elementData.length > 0) //如果扩容量大于当前的数组容量就进行扩容
        grow(minCapacity);
}
```

扩容核心代码

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length; //旧的容量
    int newCapacity = oldCapacity + (oldCapacity >> 1); //新容量为旧容量的1.5倍
    if (newCapacity - minCapacity < 0) //如果新容量小于传进来的最小容量
        newCapacity = minCapacity; //将最小容量赋值给新容量
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity); //如果新容量比规定的最大ArraySize还要大
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

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

add(E e)

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1); //如果需要扩容则扩容，并且增加操作次数
    elementData[size++] = e;
    return true;
}
```

add(int index, E element)

```
public void add(int index, E element) {
    rangeCheckForAdd(index); //检查索引越界

    ensureCapacityInternal(size + 1); //如果需要扩容则扩容，并且增加操作次数
    System.arraycopy(elementData, index, elementData, index + 1,
        size - index); //使用system.arraycopy 进行数组复制并且在重新赋值给elementData

    elementData[index] = element;
    size++;
}
```

remove(int index)

```

public E remove(int index) {
    rangeCheck(index); //检查索引越界

    modCount++; //增加操作次数
    E oldValue = elementData(index); //先保留一份备份

    int numMoved = size - index - 1; //获取index 的倒序index
    if (numMoved > 0) //如果>0 则需要将数组index+1到最后的数据 覆盖到 index 到 size-1
        System.arraycopy(elementData, index+1, elementData, index,
            numMoved);
    elementData[--size] = null; // 如果numMoved == 0 相当于直接remove最后一个

    return oldValue; //返回备份
}

```

addAll()

```

public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);

    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacityInternal(size + numNew);

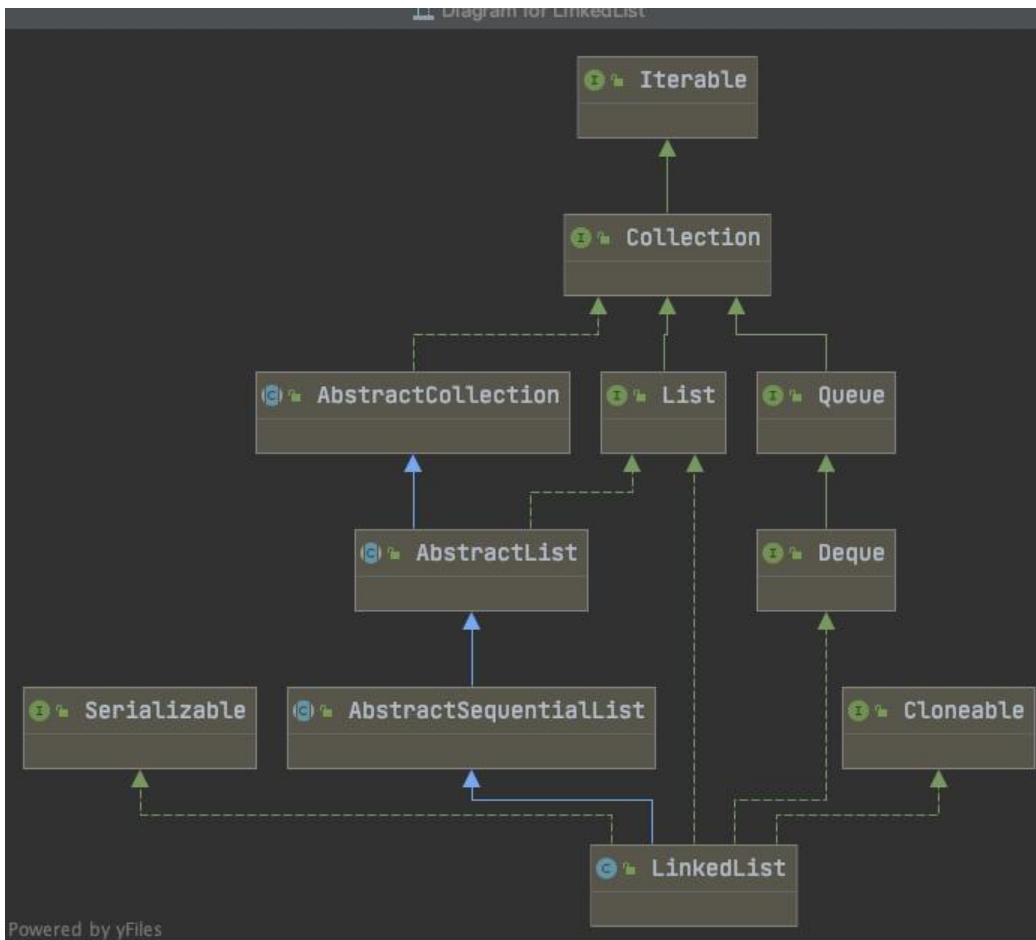
    int numMoved = size - index;
    if (numMoved > 0) //现将index之后的数据往后挪numMoved位置，给插入的数据集腾地方
        System.arraycopy(elementData, index, elementData, index + numNew,
            numMoved);

    System.arraycopy(a, 0, elementData, index, numNew);
    size += numNew;
    return numNew != 0;
}

```

LinkedList

继承关系图



- 实现了 `Cloneable`(空接口)代表可以调用`object.clone`方法
- 实现 `Serializable`(空接口)接口 代表可以进行序列化和反序列化
- 实现了 `Deque`接口 说明`LinkedList` 还是一个双端队列 并且包含了队列和栈的所有功能
- 继承了 `AbstractSequentialList` 只支持迭代器按顺序 访问, 不支持 `RandomAccess`, 所以遍历 `AbstractSequentialList` 的子类, 使用 `for` 循环 `get()` 的效率要 \leq 迭代器遍历:

Deque方法详解

`Deque`既可以用作后进先出的栈, 也可以用作先进先出的队列。

先看一下`Deque`接口中的方法

```

public interface Deque<E> extends Queue<E> {
    // *** Deque methods ***
    void addFirst(E e); //在链表前面插入一个数据
    void addLast(E e); //在链表后面一个数据
    boolean offerFirst(E e); //调用了addFirst 只不过最后返回true
    boolean offerLast(E e); //调用了addLast 只不过最后返回true
    E removeFirst(); //删除并返回链表第一个数据 若数据为空则抛出NoSuchElementException
    E removeLast(); //删除并返回链表最后一个数据 若数据为空则抛出NoSuchElementException
    E pollFirst(); //删除并返回链表第一个数据 若数据为null则返回null
    E pollLast(); //删除并返回链表最后一个数据 若数据为null则返回null
    E getFirst(); //返回链表第一个数据 若数据为空则抛出NoSuchElementException
    E getLast(); //返回链表最后一个数据 若数据为空则抛出NoSuchElementException
    E peekFirst(); //返回链表第一个数据 若数据为空则返回null
}

```

```

    E peekLast(); //返回链表最后一个数据 若数据为空则返回null
    boolean removeFirstOccurrence(Object o); //删除链表中指定元素的第一次出现（从头部遍历
    列表时尾）。如果不包含该元素，则不变。
    boolean removeLastOccurrence(Object o); //删除链表中指定元素的最后一次出现（从头部遍
    列表时尾）。如果不包含该元素，则不变。

    // *** Queue methods ***
    boolean add(E e); //在链表后面追加一个数据
    boolean offer(E e); //在链表后面追加一个数据
    E remove(); //删除链表第一个数据 若数据为空则抛出NoSuchElementException
    E poll(); //删除链表第一个数据 若数据为空则返回null
    E element(); //调用getFirst 返回链表第一个数据 若数据为空则抛出NoSuchElementException
    E peek(); //返回链表第一个数据 若数据为空则返回null

    // *** Stack methods ***
    void push(E e); //在链表前面插入一个数据
    E pop(); //返回链表第一个数据 若数据为空则返回null

    // *** Collection methods ***
    boolean remove(Object o);
    boolean contains(Object o);
    public int size();
    Iterator<E> iterator();
    Iterator<E> descendingIterator();
}

```

核心数据结构-双向链表

```

public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable{
/**
 * Pointer to first node.
 * Invariant: (first == null && last == null) ||
 *             (first.prev == null && first.item != null)
 */
    transient Node<E> first; //头节点

/**
 * Pointer to last node.
 * Invariant: (first == null && last == null) ||
 *             (last.next == null && last.item != null)
 */
    transient Node<E> last; //尾节点

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

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

```



```
}  
}  
}
```

link相关方法 添加新的节点到链表

```
/**  
 * Links e as first element.  
 */  
private void linkFirst(E e) { //将一个数据链接到链表的头部  
    final Node<E> f = first;  
    final Node<E> newNode = new Node<>(null, e, f);  
    first = newNode;  
    if (f == null) //如果链表头为空 则将头尾都设置成newNode  
        last = newNode;  
    else //否则将f的前驱节点设备newNode  
        f.prev = newNode;  
    size++; //链表节点数+1  
    modCount++; //操作次数+1  
}  
  
/**  
 * Links e as last element.  
 */  
void linkLast(E e) { //和上面的逻辑一样  
    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++;  
}  
  
/**  
 * Inserts element e before non-null Node succ.  
 */  
void linkBefore(E e, Node<E> succ) {  
    // assert succ != null;  
    final Node<E> pred = succ.prev; //获取succ的前驱节点  
    final Node<E> newNode = new Node<>(pred, e, succ); //构造newNode 前驱节点为pred  
    后驱节点为succ  
    succ.prev = newNode; //将succ的前驱节点设置为newNode  
    if (pred == null)  
        first = newNode; //如果succ没有前驱节点, 则将newNode设置为头节点  
    else  
        pred.next = newNode; //将pred的后驱节点设置为newNode  
    size++;  
    modCount++;  
}
```

unlink相关方法 链表里删除节点

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

/**
 * Unlinks non-null last node l.
 */
private E unlinkLast(Node<E> l) {
    // assert l == last && l != null;
    final E element = l.item;
    final Node<E> prev = l.prev;
    l.item = null;
    l.prev = null; // help GC
    last = prev;
    if (prev == null)
        first = null;
    else
        prev.next = null;
    size--;
    modCount++;
    return element;
}

/**
 * Unlinks non-null node x.
 */
E unlink(Node<E> x) { //删除x节点 并将x的前驱和后驱节点相连
    // assert x != null;
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;

    if (prev == null) { //如果没有前驱节点 说明这个节点是头节点
        first = next; //将此节点的后驱节点赋值给头节点
    } else {
        prev.next = next; //将前驱节点的next节点 连接到 后驱节点
        x.prev = null; //clear
    }
}
```

```

    }

    if (next == null) {
        last = prev; ///将此节点的前驱节点赋值给尾节点
    } else {
        next.prev = prev; ///将后驱节点的前驱节点 连接到前驱节点
        x.next = null; ///clear
    }

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

```

ArrayDeque

`ArrayDeque`是`Deque`接口的一个实现，使用了可变数组，所以没有容量上的限制。

同时，`ArrayDeque`是线程不安全的，在没有外部同步的情况下，不能再多线程环境下使用。

`ArrayDeque`是`Deque`的实现类，可以作为栈来使用，效率高于`Stack`；

也可以作为队列来使用，效率高于`LinkedList`。

```

public class ArrayDeque<E> extends AbstractCollection<E>
    implements Deque<E>, Cloneable, Serializable
{
    /**存储双端队列的元素的数组。
     *双端队列的容量是该数组的长度,始终是2的幂。
     *数组永远不能成为已满，除了在addX方法中短暂存在的地方
     *装满后立即调整大小（请参阅doubleCapacity），从而避免头和尾缠绕在一起以使它们相等
     *我们还保证所有不包含双端队列元素的数组单元始终为空。
     */
    transient Object[] elements;

    /**
     * The index of the element at the head of the deque (which is the
     * element that would be removed by remove() or pop()); or an
     * arbitrary number equal to tail if the deque is empty.
     */
    transient int head; //头节点index

    /**
     * The index at which the next element would be added to the tail
     * of the deque (via addLast(E), add(E), or push(E)).
     */
    transient int tail; //尾节点index

    /**
     * The minimum capacity that we'll use for a newly created deque.
     * Must be a power of 2.
     */
    private static final int MIN_INITIAL_CAPACITY = 8;
}

```

```
}
```

add操作

上面例子使用的add方法，其实内部使用了addLast方法，addLast也就添加数据到双向队列尾端：

```
public void addLast(E e) {
    if (e == null)
        throw new NullPointerException();
    elements[tail] = e; // 根据尾索引，添加到尾端
    if ( (tail = (tail + 1) & (elements.length - 1)) == head) // 尾索引+1，如果尾索引和头索引重复
    , 说明数组满了，进行扩容
        doubleCapacity();
}
```

addFirst方法跟addLast方法相反，添加数据到双向队列头端：

```
public void addFirst(E e) {
    if (e == null)
        throw new NullPointerException();
    elements[head = (head - 1) & (elements.length - 1)] = e; // 根据头索引，添加到头端，头索引
1
    if (head == tail) // 如果头索引和尾索引重复了，说明数组满了，进行扩容
        doubleCapacity();
}
```

remove操作

remove操作分别removeFirst和removeLast，removeLast代码如下：

```
public E removeLast() {
    E x = pollLast(); // 调用pollLast方法
    if (x == null)
        throw new NoSuchElementException();
    return x;
}

public E pollLast() {
    int t = (tail - 1) & (elements.length - 1); // 尾索引 -1
    @SuppressWarnings("unchecked")
    E result = (E) elements[t]; // 根据尾索引，得到尾元素
    if (result == null)
        return null;
    elements[t] = null; // 尾元素置空
    tail = t;
    return result;
}
```

removeFirst方法原理一样，remove头元素。头索引 +1

扩容

ArrayDeque的扩容会把数组容量扩大2倍，同时还会重置头索引和尾索引，头索引置为0，尾索引置原容量的值。

比如容量为8，扩容为16，头索引变成0，尾索引变成8。

扩容代码如下：

```
private void doubleCapacity() {
    assert head == tail;
    int p = head;
    int n = elements.length;
    int r = n - 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; // 头索引重置
    tail = n; // 尾索引重置
}
```

LinkedBlockingDeque

使用双向链表和ReentrantLock实现的双向的阻塞队列

```
*/
    transient Node<E> first; //头节点

    /**
     * Pointer to last node.
     * Invariant: (first == null && last == null) ||
     *             (last.next == null && last.item != null)
     */
    transient Node<E> last; //尾节点

    /** Number of items in the deque */
    private transient int count; //队列里的元素数量
    /** Maximum number of items in the deque */

    private final int capacity; //队列容量

    /** Main lock guarding all access */
    final ReentrantLock lock = new ReentrantLock(); //:lock:

    /** Condition for waiting takes */
    private final Condition notEmpty = lock.newCondition(); //通过Condition来阻塞当队列为空
    来take元素的线程

    /** Condition for waiting puts */
    private final Condition notFull = lock.newCondition(); //通过Condition来阻塞当队列满了时来
    ut元素的线程
```

核心代码

```
private boolean linkFirst(Node<E> node) {
    // assert lock.isHeldByCurrentThread();
    if (count >= capacity) //如果队列已满
        return false;
    Node<E> f = first;
    node.next = f;
    first = node;
    if (last == null)
        last = node;
    else
        f.prev = node;
    ++count;
    notEmpty.signal(); //加入了节点，唤醒某个等待take的线程
    return true;
}

public E takeLast() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        E x;
        while ( (x = unlinkLast()) == null) //如果队列为空，则进入等待
            notEmpty.await();
        return x;
    } finally {
        lock.unlock();
    }
}

private E unlinkLast() {
    // assert lock.isHeldByCurrentThread();
    Node<E> l = last;
    if (l == null) //如果队列为空，则返回null
        return null;
    Node<E> p = l.prev;
    E item = l.item;
    l.item = null;
    l.prev = l; // help GC
    last = p;
    if (p == null)
        first = null;
    else
        p.next = null;
    --count;
    notFull.signal(); //如果正常删掉节点 则唤醒某个等待put的线程
    return item;
}

public void putFirst(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    Node<E> node = new Node<E>(e);
    final ReentrantLock lock = this.lock;
```

```

lock.lock();
try {
    while (!linkFirst(node))
        notFull.await(); //如果队列已满 则让出锁, 进入等待
} finally {
    lock.unlock();
}
}

```

CopyOnWriteArrayList

CopyOnWriteArrayList是ArrayList的线程安全版本, 从他的名字可以推测, CopyOnWriteArrayLis是在有写操作的时候会copy一份数据, 然后写完再设置成新的数据。CopyOnWriteArrayList适用于多写少的并发场景

add

```

public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock(); //写数据之前上:lock:
    try {
        Object[] elements = getArray(); //获取当前的数组
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1); //新创建一个数组, 拷贝旧数
        并且长度+1
        newElements[len] = e; //追加e
        setArray(newElements); //将新数组设为当前数组
        return true;
    } finally {
        lock.unlock(); //解:lock:
    }
}

```

remove

```

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; //倒序index
        if (numMoved == 0)
            setArray(Arrays.copyOf(elements, len - 1));
        else {
            Object[] newElements = new Object[len - 1];
            System.arraycopy(elements, 0, newElements, 0, index); //将elements的0到index-1 (
            闭右开) 赋给newElenments
            System.arraycopy(elements, index + 1, newElements, index, //然后将index+1后面的
            据 赋给newElements的index到numMoved
            numMoved);
        }
    }
}

```

```
        setArray(newElements);
    }
    return oldValue;
} finally {
    lock.unlock();
}
}
```

get

```
public E get(int index) {
    return get(getArray(), index); // 从数组中取值 不加lock:
}
```

HashSet

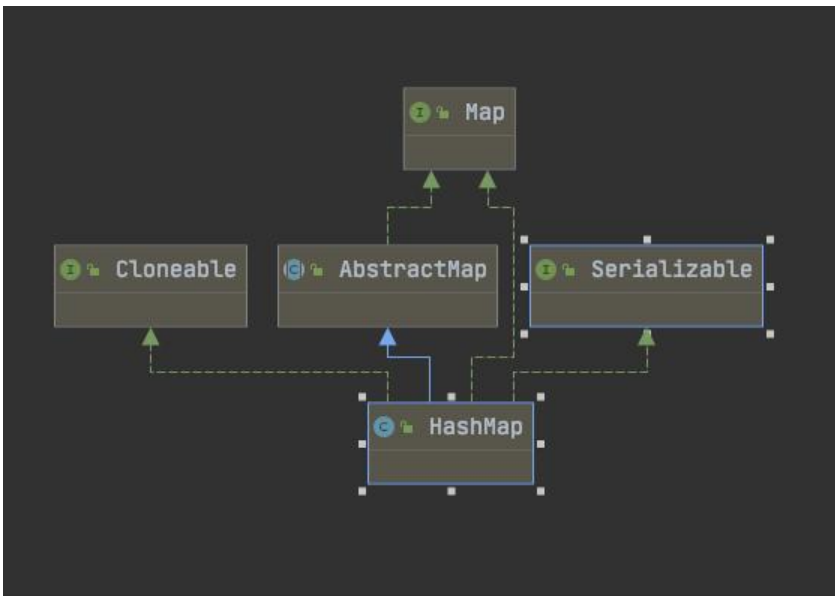
基于hashmap, 用map的key做值, map的value是 new了一个Object对象

TreeSet

基于treemap

HashMap

继承关系图



默认的配置参数

```
/**
 * 默认的容量 16 (必须是2的幂)
 */
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; //最大容量 2的30次方

/**
 * The load factor used when none specified in constructor.
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f; //默认的负载因子0.75

/**
 * 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; //链表转化为树的阈值 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; //取消树化的阈值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;

/**
 * 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; //hash表

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

```

Node

hashmap中真正的数据存储结构

```

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

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;
}

```

get(Object key)

```

public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

/**
 * Implements Map.get and related methods.
 *
 * @param hash hash for key
 * @param key the key
 * @return the node, or null if none
 */
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;

//如果table == null 或者 table.length==0 或者在hash表里找不到元素 则返回null hash公式 (n-1)
hash 获取此hash值在表里的索引
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        if (first.hash == hash &&
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first; // 如果 first的hash值等于传进来的hash 并且key相等 则first就是要找的node
        if ((e = first.next) != null) { //如果first 不是要找的值 则寻找它连着的next
            if (first instanceof TreeNode) //如果first 是红黑树节点 则调用getNode方法
                return ((TreeNode<K,V>)first).getNode(hash, key);
            do { //first 是普通的链表节点 则遍历链表找到key对应的value
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}

```

```

/**
 * Calls find for root node.
 */
final TreeNode<K,V> getTreeNode(int h, Object k) {
    return ((parent != null) ? root() : this).find(h, k, null); //调用find方法查询树中的节点
}

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.hash > h 就将 p的左节点赋给p
            p = pl;
        else if (ph < h) //p.hash < h 就将 p的右节点赋给p
            p = pr;
        else if ((pk = p.key) == k || (k != null && k.equals(pk)))
            return p; //如果 p的key与k相等 则返回p
        else if (pl == null)
            p = pr; //如果没有左节点 就将右节点赋值给p
        else if (pr == null)
            p = pl; //如果没有右节点 就将左节点赋值给p
        else if ((kc != null ||
            //判断是否实现comparable接口 如果没实现则返回null 实现了则返回key的class
            (kc = comparableClassFor(k)) != null) &&
            //根据具体类实现的comparable接口的compareTo方法判断k 与当前pk的大小关系
            (dir = compareComparables(kc, k, pk)) != 0)
            p = (dir < 0) ? pl : pr; //根据大小关系 将左右节点赋给p
        else if ((q = pr.find(h, k, kc)) != null) //递归
            return q;
        else
            p = pl; //如果从右孩子节点递归查找后仍未找到, 那么从左孩子节点进行下一轮循环
    } while (p != null);
    return null;
}

```

put(K key,V value)

```

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

/**
 * Implements Map.put and related methods.
 *
 * @param hash hash for key
 * @param key the key
 * @param value the value to put
 * @param onlyIfAbsent if true, don't change existing value
 * @param evict if false, the table is in creation mode.
 * @return previous value, or null if none
 */

```

```

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
              boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0) //如果tab.length !=0 则赋值给n
        n = (tab = resize()).length; //如果table == null 则将resize之后的tab.length 赋给n
    if ((p = tab[i = (n - 1) & hash]) == null) //如果当前hash值在表里面没有存在 则存到当前位置
        tab[i] = newNode(hash, key, value, null);
    else { //hash表里面的位置被占据了
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p; // key相同
        else if (p instanceof TreeNode) //发生hash冲突 并且目前存储结构是树
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else { //发生hash冲突 并且目前存储结构是链表
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    //如果hash表中的冲突位置的节点没有next, 则将put进来的key,value 链接到此节点
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash); //如果当前链表的长度超过了8则转换成红黑树
                    break;
                }
                if (e.hash == hash && //如果在链表中找到了key相同的, 则跳出, 覆盖它的value
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e; //链表循环next
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value; //替换新值
            afterNodeAccess(e);
            return oldValue; //返回旧值
        }
    }
    ++modCount;
    if (++size > threshold)
        resize(); //如果当前的 hash表的长度已经超过了当前 hash 需要扩容的长度, 重新扩容,条件
    //haspmap 中存放的数据超过了临界值(经过测试),而不是数组中被使用的下标
    afterNodeInsertion(evict);
    return null;
}

```

resize()

```

final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length; //获取当前hash表的长度
    int oldThr = threshold; //获取当前的扩容阈值
    int newCap, newThr = 0; //新的大小和扩容阈值
    if (oldCap > 0) { //hashmap已经初始化过了

```

```

    if (oldCap >= MAXIMUM_CAPACITY) { //如果旧的hash表长度大于等于最大限定长度 则
进行扩容
        threshold = Integer.MAX_VALUE;
        return oldTab;
    }
    // 如果数组元素个数在正常范围内, 那么新的数组容量为老的数组容量的2倍 (左移1位相当于
以2)
    // 如果扩容之后的新容量小于最大容量 并且 老的数组容量大于等于默认初始化容量 (16)
那么新数组的扩容阈值设置为老阈值的2倍。
    else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
        oldCap >= DEFAULT_INITIAL_CAPACITY)
        newThr = oldThr << 1;
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr; //如果老数组的扩容阈值大于0, 那么设置新数组的容量为该阈值这一步
就意味着构造该map的时候, 指定了初始化容量。
    else { // 第一次初始化resize 使用默认参数进
        newCap = DEFAULT_INITIAL_CAPACITY; //默认容量
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY); //默认扩容阈值
默认容量*负载因子0.75
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor; //新长度*负载因子
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
            (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr; //将新阈值 赋给threshold 覆盖老的阈值
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap]; //根据新容量创建hash表
    table = newTab; //将新的hash表赋给table
    if (oldTab != null) { //如果原来的hash表里有值 则进行元素的转移
        for (int j = 0; j < oldCap; ++j) { //遍历旧的hash表
            Node<K,V> e;
            if ((e = oldTab[j]) != null) { //如果j处有值则赋值给e
                oldTab[j] = null; //将旧表j处的数据清空
                if (e.next == null) //如果e后没有连接节点
                    newTab[e.hash & (newCap - 1)] = e; //直接hash到新表位置
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap); //如果当前是树存储 则调用spli
方法
            }
            else { // preserve order
                Node<K,V> loHead = null, loTail = null;
                Node<K,V> hiHead = null, hiTail = null;
                Node<K,V> next;
                do {
                    next = e.next;
                    //拿元素的hash值 和 老数组的长度 做与运算
                    // 数组的长度一定是2的N次方 (例如16), 如果hash值和该长度做与运算, 结果
0, 就说明该hash值小于数组长度 (例如hash值为7),
                    // 那么该hash值再和新数组的长度取摸的话mod值也不会放生变化, 所以该元素
在新数组的位置和在老数组的位置是相同的, 所以该元素可以放置在低位链表中。
                    if ((e.hash & oldCap) == 0) {
                        if (loTail == null) //如果链表没有尾节点
                            loHead = e; //就将e赋给链表的头节点

```

```

        else
            loTail.next = e; //挂到链表尾部
            loTail = e; //然后将尾节点设为e
        }
        else { //说明hash值大于数组长度 所以需要放到高位链表中
            if (hiTail == null) //同上
                hiHead = e;
            else
                hiTail.next = e;
            hiTail = e;
        }
    } while ((e = next) != null);
    if (loTail != null) {
        loTail.next = null;
        newTab[j] = loHead; // 低位的元素组成的链表还是放置在原来的位置
    }
    if (hiTail != null) {
        hiTail.next = null; // 高位的元素组成的链表放置的位置只是在原有位置上偏移了
        newTab[j + oldCap] = hiHead;
    }
}
}
}
}
}
return newTab;
}

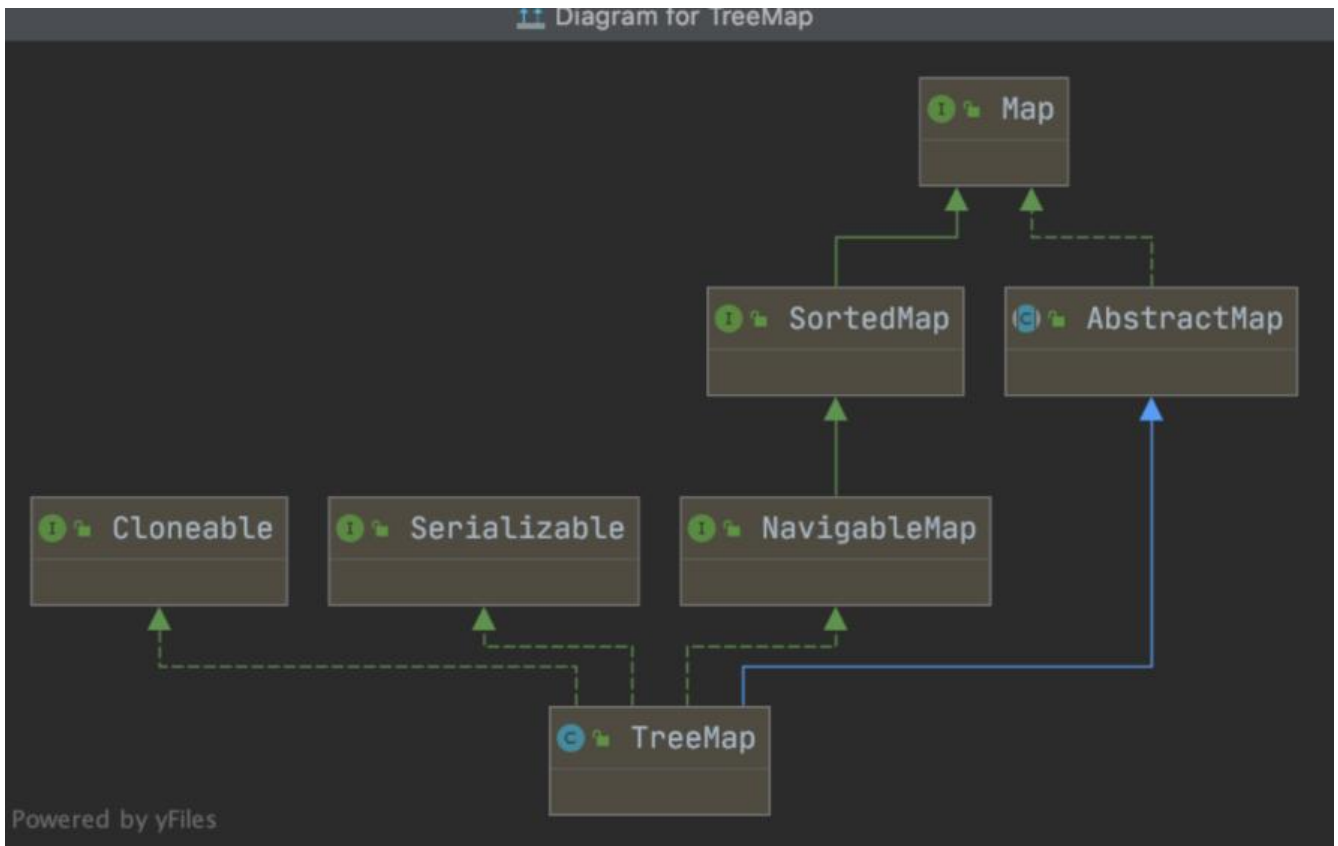
```

数组的长度个位置

TreeMap

基于红黑树，实现了有序的map

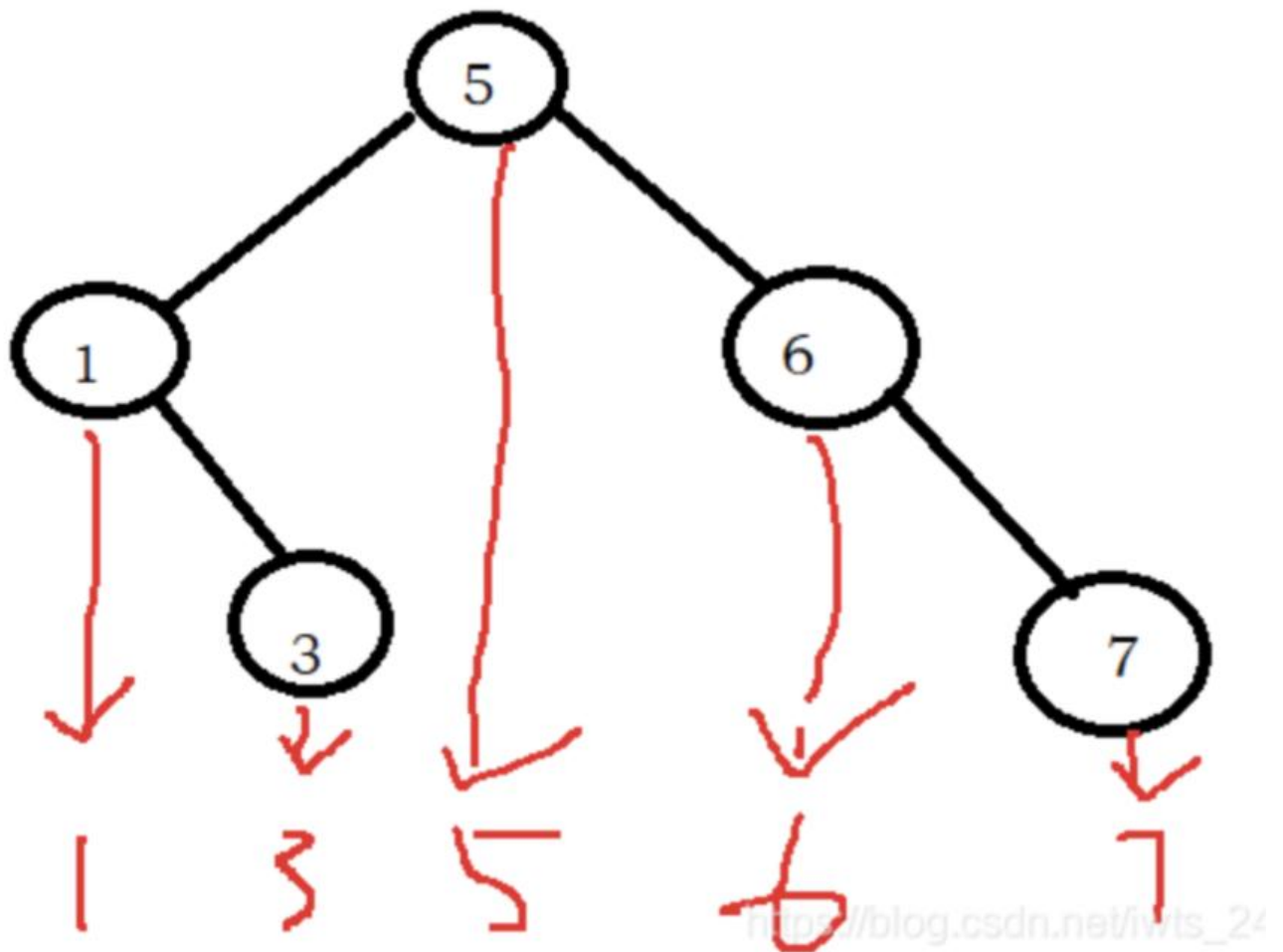
继承关系图



```
//甚至我们可以自己传一个比较器进去 来作为有序的依赖
public TreeMap(Comparator<? super K> comparator) {
    this.comparator = comparator;
}
```

由于treemap是红黑树实现，而红黑树又是二叉树的一种，也就是红黑树也是满足二叉树的一些性质比较重要的：

一个树的节点，如果有左节点和右节点，那么它一定大于左节点，小于右节点。



如上图

获取前继和后继节点

/**

* 返回指定节点的后继节点，如果没有则返回空

怎么理解这个后继呢？

我们认为，将BST“压平”，就是一个有序数组。而所谓的后继节点，就是指在“数组状态”下按顺序遍历的情况，可以理解为迭代。严格地说，有点类似严格ceil方法：寻找大于该节点的所有结点中，值最小的结点。可以理解为树的中序遍历，但是如果需要遍历才能得到一个节点的后继节点，这效率很差，所以有如下定则：

a. 空节点，没有后继

b. 有右子树的节点，后继就是右子树的“最左节点”（下图1）

c. 无右子树的节点，后继就是该节点所在左子树的第一个祖先节点（下图2）

*/

```

static <K,V> TreeMap.Entry<K,V> successor(Entry<K,V> t) {
    if (t == null) //如果输入的节点为空，则返回空
        return null;
    else if (t.right != null) { //有右子树的节点，后继就是右子树的“最左节点”
        Entry<K,V> p = t.right;
        while (p.left != null) //循环找到最左节点
            p = p.left;
        return p;
    } else { //无右子树的节点，后继就是该节点所在左子树的第一个祖先节点
  
```

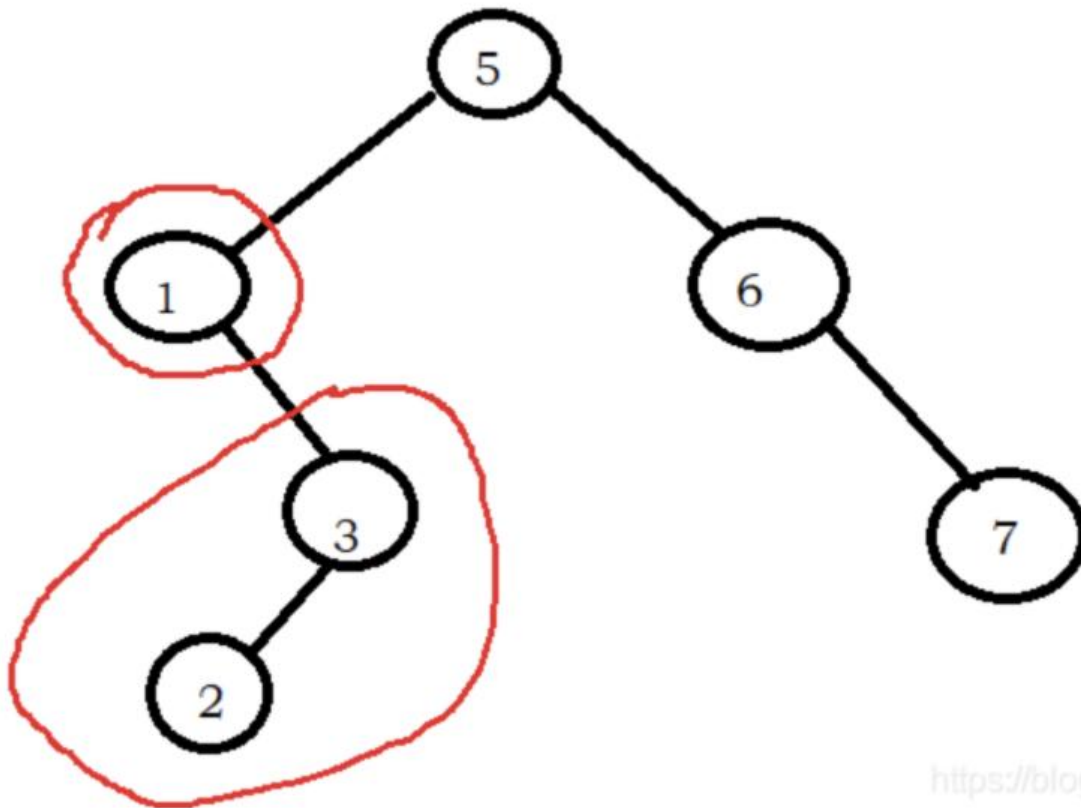


```

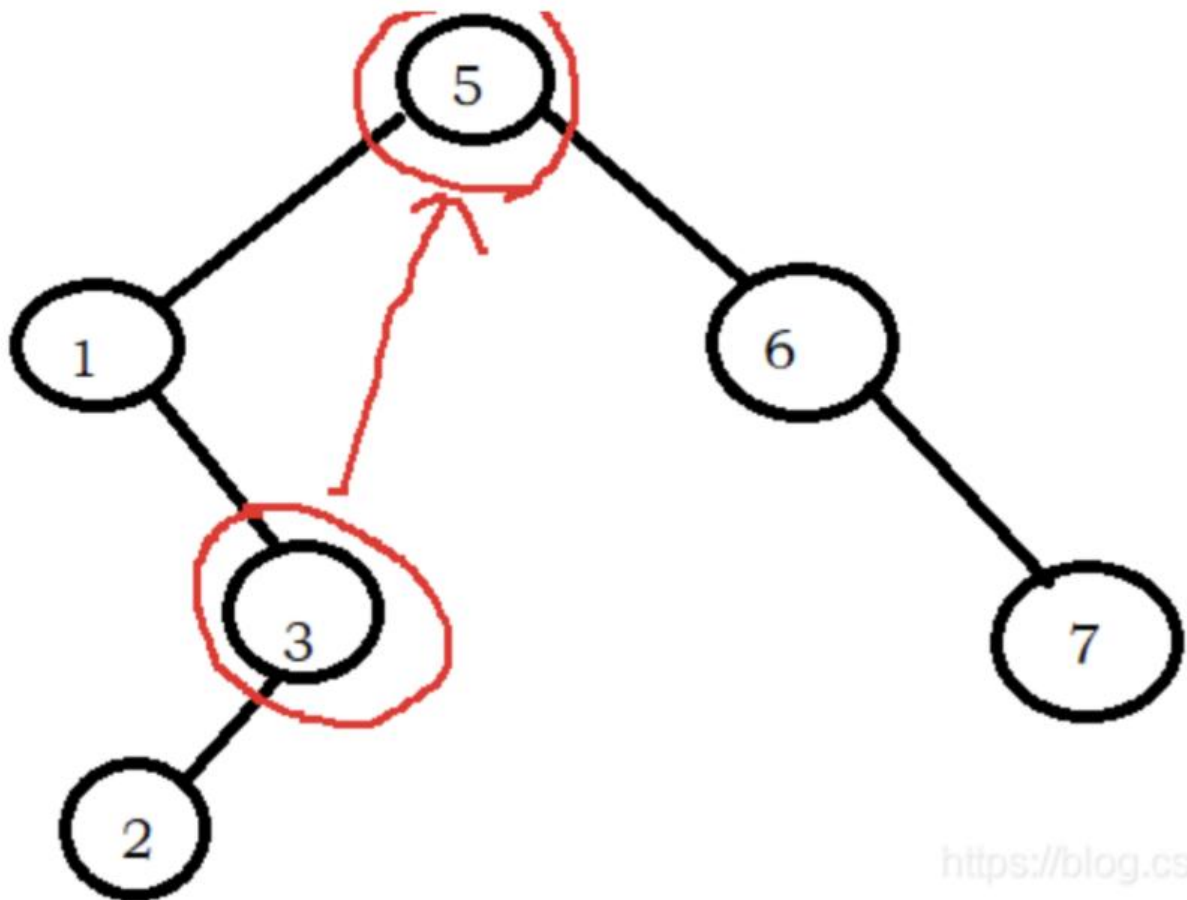
    Entry<K,V> p = t.parent;
    Entry<K,V> ch = t;
    while (p != null && ch == p.right) {
        ch = p;
        p = p.parent; //zha
    }
    return p;
}
}

/**
 * 前继节点
 */
static <K,V> Entry<K,V> predecessor(Entry<K,V> t) {
    if (t == null)
        return null;
    else if (t.left != null) { //有左子树的节点，前继就是左子树的“最右节点”
        Entry<K,V> p = t.left;
        while (p.right != null)
            p = p.right;
        return p;
    } else { //无左子树的节点，前继就是该节点所在右子树的第一个祖先节点
        Entry<K,V> p = t.parent;
        Entry<K,V> ch = t;
        while (p != null && ch == p.left) {
            ch = p;
            p = p.parent;
        }
        return p;
    }
}
}

```



<https://blog.csdn.net/>



<https://blog.csdn.net/>

get/remove

```

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

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

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

final Entry<K,V> getEntry(Object key) {
    // Offload comparator-based version for sake of performance
    if (comparator != null)
        return getEntryUsingComparator(key);
    if (key == null) //不允许null key
        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); //根据comparable接口来比较
        if (cmp < 0)
            p = p.left;
        else if (cmp > 0)
            p = p.right;
        else //当两个key经过比较并相等时就返回
            return p;
    }
    return null;
}

```

put

```

public V put(K key, V value) {
    Entry<K,V> t = root;
    if (t == null) {
        compare(key, key); //如果初始化map时传入了compareable比较器, 则使用map的, 若没
        则使用key的compareable
        root = new Entry<>(key, value, null);
        size = 1;
        modCount++; //操作数加一
        return null; //返回以前的值
    }
    int cmp;
    Entry<K,V> parent;
    // split comparator and comparable paths
    Comparator<? super K> cpr = comparator; //使用自身传入的比较器
    if (cpr != null) { //如果比较器不为null
        do {

```

```

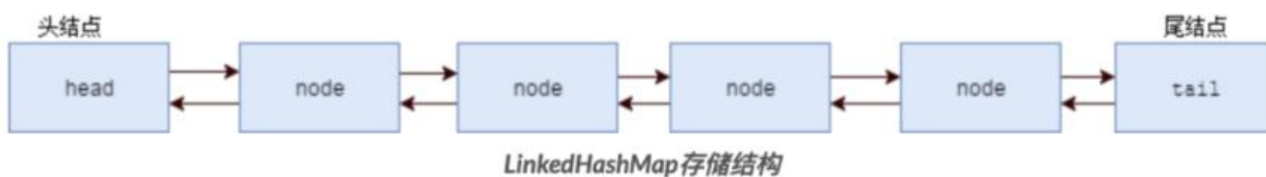
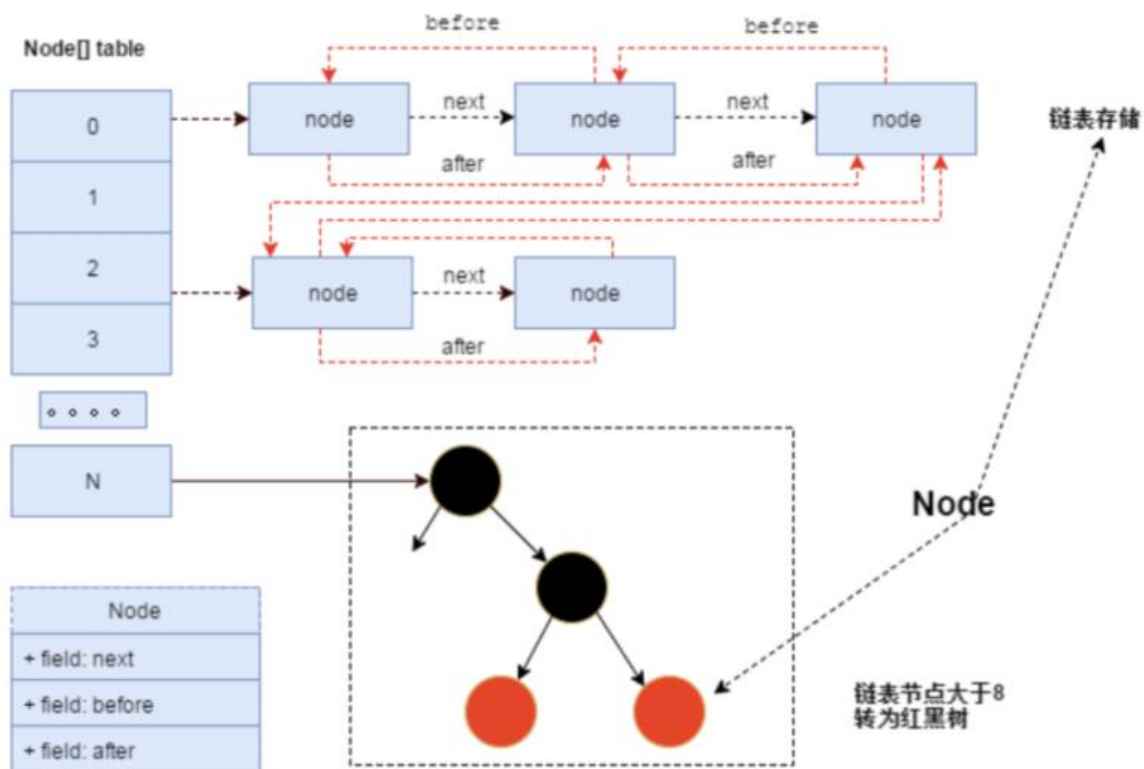
    parent = t;
    cmp = cpr.compare(key, t.key);
    if (cmp < 0) // 如果比较key比t的key小, 则寻找t的左子树
        t = t.left;
    else if (cmp > 0) // 如果比较key比t的key大, 则寻找t的右子树
        t = t.right;
    else
        return t.setValue(value); //如果有相等的key, 则覆盖value
    } while (t != null);
}
else { //如果比较器为空
    if (key == null)
        throw new NullPointerException();
    @SuppressWarnings("unchecked")
    Comparable<? super K> k = (Comparable<? super K>) key; //使用key自己的比较器
    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); //如果比较后发现key没有存在过, 那么
插入 经过多次循环parent已经是需要插入节点的parent了
if (cmp < 0) //根据比较器 选择插入左还是右
    parent.left = e;
else
    parent.right = e;
fixAfterInsertion(e); //插入
size++;
modCount++;
return null;
}

```

LinkedHashMap

LinkedHashMap继承自**HashMap**,主要是使用了双向链表维护了node之间的顺序, 而红黑树本身就有序的

如图

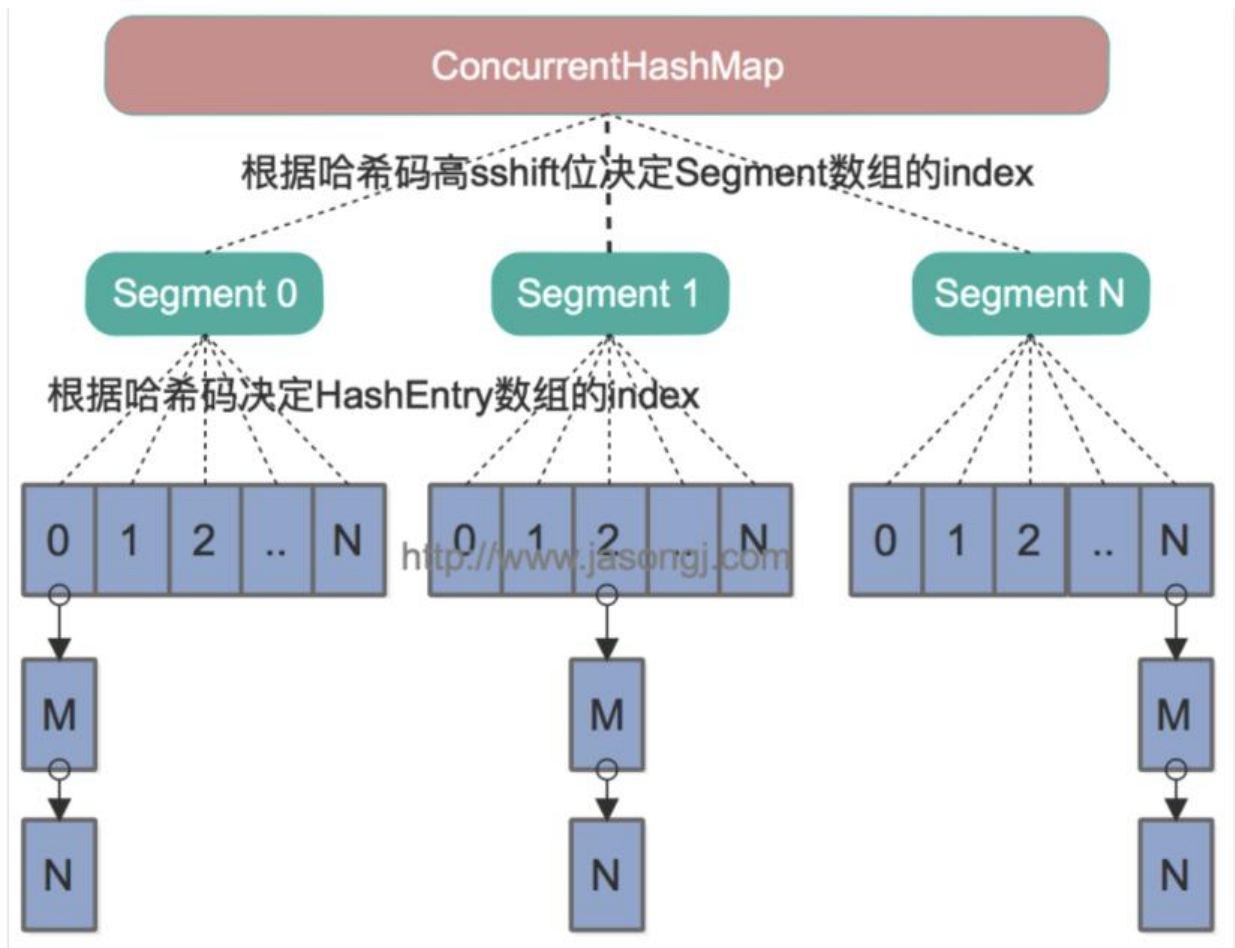


ConcurrentHashMap

Java 7基于分段锁的ConcurrentHashMap

数据结构

Java 7中的ConcurrentHashMap的底层数据结构仍然是数组和链表。与HashMap不同的是，ConcurrentHashMap最外层不是一个大的数组，而是一个Segment的数组。每个Segment包含一个与HashMap数据结构差不多的链表数组。整体数据结构如下图所示。



寻址方式

在读写某个Key时，先取该Key的哈希值。并将哈希值的高N位对Segment个数取模从而得到该Key应属于哪个Segment，接着如同操作HashMap一样操作这个Segment。为了保证不同的值均匀分布到不同的Segment，需要通过如下方法计算哈希值。

```
private int hash(Object k) {
    int h = hashSeed;
    if ((0 != h) && (k instanceof String)) {
        return sun.misc.Hashing.stringHash32((String) k);
    }
    h ^= k.hashCode();
    h += (h << 15) ^ 0xffffcd7d;
    h ^= (h >>> 10);
    h += (h << 3);
    h ^= (h >>> 6);
    h += (h << 2) + (h << 14);
    return h ^ (h >>> 16);
}
```

同样为了提高取模运算效率，通过如下计算，ssize即为大于concurrencyLevel的最小的2的N次方，时segmentMask为 $2^N - 1$ 。这一点跟上文中计算数组长度的方法一致。对于某一个Key的哈希值，需要向右移segmentShift位以取高sshift位，再与segmentMask取与操作即可得到它在Segment数组上的索引。

```

int sshift = 0;
int ssize = 1;
while (ssize < concurrencyLevel) {
    ++sshift;
    ssize <<= 1;
}
this.segmentShift = 32 - sshift;
this.segmentMask = ssize - 1;
Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];

```

同步方式

Segment继承自ReentrantLock，所以我们可以很方便的对每一个Segment上锁。

对于读操作，获取Key所在的Segment时，需要保证可见性。具体实现上可以使用volatile关键字，可使用锁。但使用锁开销太大，而使用volatile时每次写操作都会让所有CPU内缓存无效，也有一定开销。ConcurrentHashMap使用如下方法保证可见性，取得最新的Segment。

```
Segment<K,V> s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)
```

获取Segment中的HashEntry时也使用了类似方法

```
HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
(tab, (((long)((tab.length - 1) & h)) << TSHIFT) + TBASE)
```

对于写操作，并不要求同时获取所有Segment的锁，因为那样相当于锁住了整个Map。它会先获取该Key-Value对所在的Segment的锁，获取成功后就可以像操作一个普通的HashMap一样操作该Segment，并保证该Segment的安全性。

同时由于其它Segment的锁并未被获取，因此理论上可支持concurrencyLevel（等于Segment的个数）个线程安全的并发读写。

获取锁时，并不直接使用lock来获取，因为该方法获取锁失败时会挂起（参考[可重入锁](#)）。事实上，使用了自旋锁，如果tryLock获取锁失败，说明锁被其它线程占用，此时通过循环再次以tryLock的方式申请锁。如果在循环过程中该Key所对应的链表头被修改，则重置retry次数。如果retry次数超过一定次数，则使用lock方法申请锁。

这里使用自旋锁是因为自旋锁的效率比较高，但是它消耗CPU资源比较多，因此在自旋次数超过阈值切换为互斥锁。

size操作

put、remove和get操作只需要关心一个Segment，而size操作需要遍历所有的Segment才能算出整个Map的大小。一个简单的方案是，先锁住所有Segment，计算完后再解锁。但这样做，在做size操作时，不仅无法对Map进行写操作，同时也无法进行读操作，不利于对Map的并行操作。

为更好支持并发操作，ConcurrentHashMap会在不上锁的前提下逐个Segment计算3次size，如果某邻两次计算获取的所有Segment的更新次数（每个Segment都与HashMap一样通过modCount跟踪自己的修改次数，Segment每修改一次其modCount加一）相等，说明这两次计算过程中无更新操作则这两次计算出的总size相等，可直接作为最终结果返回。如果这三次计算过程中Map有更新，则对每个Segment加锁重新计算Size。该计算方法代码如下

```
public int size() {
    final Segment<K,V>[] segments = this.segments;
```

```

int size;
boolean overflow; // true if size overflows 32 bits
long sum; // sum of modCounts
long last = 0L; // previous sum
int retries = -1; // first iteration isn't retry
try {
    for (;;) {
        if (retries++ == RETRIES_BEFORE_LOCK) {
            for (int j = 0; j < segments.length; ++j)
                ensureSegment(j).lock(); // force creation
        }
        sum = 0L;
        size = 0;
        overflow = false;
        for (int j = 0; j < segments.length; ++j) {
            Segment<K,V> seg = segmentAt(segments, j);
            if (seg != null) {
                sum += seg.modCount;
                int c = seg.count;
                if (c < 0 || (size += c) < 0)
                    overflow = true;
            }
        }
        if (sum == last)
            break;
        last = sum;
    }
} finally {
    if (retries > RETRIES_BEFORE_LOCK) {
        for (int j = 0; j < segments.length; ++j)
            segmentAt(segments, j).unlock();
    }
}
return overflow ? Integer.MAX_VALUE : size;
}

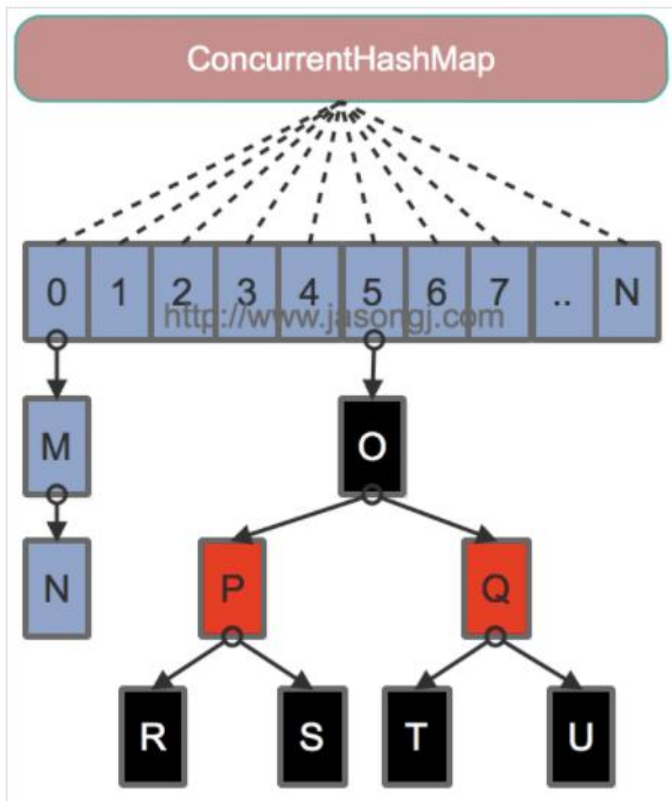
```

Java 8基于CAS的ConcurrentHashMap

数据结构

与java8的hashmap一致

摒弃了分段锁的方案，而是直接使用一个大的数组。同时为了提高哈希碰撞下的寻址性能，Java 8在表长度超过一定阈值（8）时将链表（寻址时间复杂度为 $O(N)$ ）转换为红黑树（寻址时间复杂度为 $O(\log(N))$ ）。其数据结构如下图所示



寻址方式

Java 8的ConcurrentHashMap同样是通过Key的哈希值与数组长度取模确定该Key在数组中的索引。样为了避免不太好的Key的hashCode设计，它通过如下方法计算得到Key的最终哈希值。不同的是，Java 8的ConcurrentHashMap作者认为引入红黑树后，即使哈希冲突比较严重，寻址效率也足够高，以作者并未在哈希值的计算上做过多设计，只是将Key的hashCode值与其高16位作异或并保证最高为0（从而保证最终结果为正整数）。

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

同步方式

对于put操作，如果Key对应的数组元素为null，则通过CAS操作将其设置为当前值。如果Key对应的组元素（也即链表表头或者树的根元素）不为null，则对该元素使用synchronized关键字申请锁，然后进行操作。如果该put操作使得当前链表长度超过一定阈值，则将该链表转换为树，从而提高寻址效率。

对于读操作，由于数组被volatile关键字修饰，因此不用担心数组的可见性问题。同时每个元素是一个Node实例（Java 7中每个元素是一个HashEntry），它的Key值和hash值都由final修饰，不可变更，须关心它们被修改后的可见性问题。而其Value及对下一个元素的引用由volatile修饰，可见性也有保

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

```
}
```

对于Key对应的数组元素的可见性，由Unsafe的getObjectVolatile方法保证。

```
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);  
}
```

size操作

put方法和remove方法都会通过addCount方法维护Map的size。size方法通过sumCount获取由addCount方法维护的Map的size。